A Model of Configuration Languages for Routing Protocols

Philip J. Taylor
Computer Laboratory
University of Cambridge
Cambridge, UK
philip.taylor@cl.cam.ac.uk

Timothy G. Griffin
Computer Laboratory
University of Cambridge
Cambridge, UK
timothy.griffin@cl.cam.ac.uk

ABSTRACT

The emergence of programmable routers brings opportunities to design and implement new routing protocols with expressive policy, that better meet the needs of network operators than the current range of protocols. This also introduces significant challenges in designing and understanding protocols. Algebraic routing provides a useful but highly abstract model of networks as weighted graphs, ignoring the complex distributed configuration and computation aspects of practical routing protocols. We present an algebraic model of router configuration languages, reducing the gap between the routing theory and real implementations, as the basis for a language that can be used to specify the operation of routing protocols.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols—*routing protocols*; G.2.2 [**Discrete Mathematics**]: Graph Theory

General Terms

Languages, Theory, Verification

Keywords

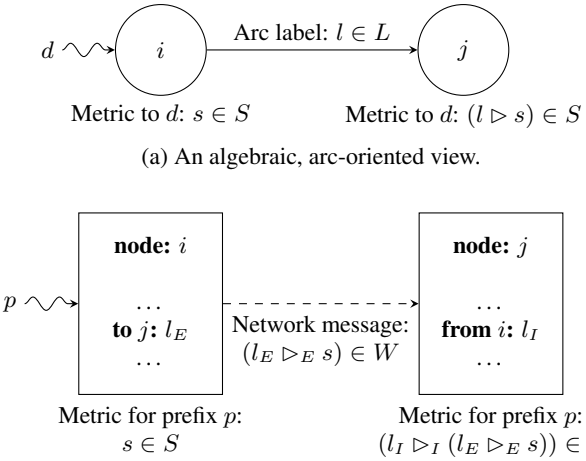
Routing Protocol Configuration, Routing Algebra

1. INTRODUCTION

Programmable routers provide an opportunity for customisation of control-plane routing functionality, supporting research into innovative new routing protocols and rapid development of improvements to currently-deployed protocols. But this flexibility comes with significant challenges in designing protocols: the difficulty of ensuring the distributed algorithms will actually compute a correct set of routes without subtle unexpected errors, and the sheer effort needed to specify and implement new protocols.

We believe these issues can be addressed by adding a level of abstraction separating the high-level decisions of what a routing protocol should compute, and how it should be configured, from the low-level implementation details of how it performs the necessary computations. In particular, our goal is to develop a language that can be used to write concise specifications of routing protocols and can be automatically compiled into executable code, greatly easing the process of developing and testing protocol modifications. It is important that the language has a solid theoretical basis, so we can easily model the operation of the resulting routing protocols and guarantee they will converge to the correct routing solution.

PRESTO'09, August 21, 2009, Barcelona, Spain.



(a) An algebraic, arc-oriented view.

(b) A node-oriented view.

Figure 1: An illustration of the central problem.

We advocate an approach based on *algebraic* models of routing [6, 10, 11, 12], which enforce a clear distinction between what problem is being solved and how solutions are computed. This allows issues of correctness and design trade-offs to be addressed in a more general and tractable manner than current practice. However, there are many ways in which the gap between the abstract algebraic view and existing Internet routing protocols can seem so wide as to be unbridgeable. We hope these difficulties can be overcome without sacrificing correctness and without radically deviating from the way existing protocols work.

This paper addresses one aspect of this gap, illustrated in Figure 1. The algebraic model of routing is based on directed graphs, and in Figure 1(a) we illustrate a single arc from node i to node j in such a model with label (or weight) l. In this paper we will use Sobrinho-like algebras [11, 12] of the form $(S,\ L,\ \preceq,\ \triangleright)$, where S is a set of *metrics* (describing attributes of paths), $\preceq$ is an order over S (for selecting the 'best' paths based on their metrics), L is a set of arc *labels*, and $\triangleright$ is a function that applies a label to a metric to produce a new metric (i.e. $\triangleright \in L \times S \rightarrow S$). If we imagine a path from destination d to node i that is associated with metric $s \in S$, as illustrated in Figure 1(a), and $l \in L$ is the label associ-

$$\underline{\text{Types}}$$

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

$$\underline{\text{Functions}}$$

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

$$\underline{\text{Consistency}}$$

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

Figure 2: The components of a simple network configuration language.

ated with the arc from i to j, then the path that continues on to node j will be associated with the metric $l \rhd s$ (using infix notation).

The arc-oriented algebraic approach is very different from the node-oriented manner in which current routing protocols are configured, illustrated in Figure 1(b). Here, the arcs do not define the network; instead they arise from the configured adjacencies implied by each node's configuration. In addition, there may not be an obvious relationship between these configurations and some abstract arc weight they might represent. This is especially true for protocols as complex as BGP.

How might this gap be bridged? We make the simplifying assumption that from a complex router configuration file we can extract an *export label* ($l_E \in L_E$) on the i-end of the arc and an *import label* ($l_I \in L_I$) on the j-end of the arc. (This ignores complexities associated with *route maps*, which we discuss in Section 4.) Figure 2 presents our approach, which we will call a *configuration language* – an algebraic language that can represent the configuration of a network, along with defining the computations performed on the network's configuration data by the routing protocol. Two distinct functions, $\rhd_E$ and $\rhd_I$, are used to represent the computation that occurs in a BGP-like vectoring protocol with the export and import of a metric in a route advertisement.

This gives us a model that is closer to the node-oriented view of routing, and we can now build a bridge back to the original algebraic model: a configuration language includes a routing algebra $(S,\ L,\ \preceq,\ \rhd)$, and we want the node-oriented protocol to compute the same solution as this arc-oriented algebra. We therefore have an operation $\odot$ that can reconstruct a label from adjacent configurations, and we insist on *consistency*: the equation

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

must be valid for all label and metric values. In other words, we can reconstruct a label $l = l_I \odot l_E$ from bits of configuration data at each end of a link, and the metric $s' = l \rhd s$ will always be the same as first computing $w = l_E \rhd_E s$ and then $s' = l_I \rhd_I w$. The type of w is W, which corresponds to the data type that is shared over the wire by the vectoring protocol; this may not be equal to the algebra's metric type S.

In this paper we will not concern ourselves with algebraic properties of routing algebras that are associated with correct usage of various algorithms. This topic is treated in the algebraic routing literature cited above.

In this paper we propose a method of constructing configuration languages. It is based on the metarouting approach to constructing routing algebras [7]. Metarouting uses a language for describing routing algebras from which code can be extracted and automatically checked for correctness. A short review of metarouting is provided in Section 2. In Section 3 we extend the metarouting grammar with simple constructs that place sub-configurations at one or the other side of an adjacency. We are then able to extend the semantics of the language so that the types and functions of Figure 2 can be automatically constructed together with a proof of consistency. In Section 4 we discuss related work and directions for further research.

2. METAROUTING

This section introduces a concrete syntax for writing routing algebras in the arc-oriented model of Figure 1(a), along with its algebraic semantics. Since this is a language used for writing routing languages (or routing algebras), we call it a *metalanguage*. The metalanguage presented here is a small fragment of the language we are currently implementing [2], chosen to illustrate the main features of our approach.

2.1 Routing Algebras

As described in Section 1, in this paper routing algebras have the form $(S,\ L,\ \preceq,\ \rhd)$. For example, the basic integer shortest-paths routing algebra has the form $(\mathbb{N}^\infty,\ \mathbb{N}^\infty,\ \leq,\ +)$. Each route has a metric that is of type $\mathbb{N}^\infty$, either a positive integer or infinity. This ∞ metric indicates no data can be forwarded along the path, and is a way of modelling routes that have been filtered out. Metrics are compared using the standard integer $\leq$ operator. Each arc weight is similarly a positive integer or infinity, and a route's metric is combined with the arc weight using integer addition.

Some algebras can be expressed by a finite table. The customer–provider–peer relationships and constraints described in [5] can be modelled algebraically [7, 11] with

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

where $\preceq$ is defined by the order $C \prec R \prec P \prec \infty$, and $\rhd_{cpp}$ is defined by the table

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

For example, a route received from a peer has metric R. An arc from a customer (to a provider) has label c. Since $c \rhd_{cpp} R = \infty$, the route will be filtered out. It also gives $p \rhd_{cpp} R = P$, so the same route sent across an arc from provider to customer will be accepted and given new metric P to indicate it came from a provider.

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

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

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

The *lexicographic product* constructor combines two routing algebras. For example, the lexicographic product of two integer-shortest-paths algebras is an algebra whose metrics and arc labels are pairs of integers. Given $A = (S_A,\ L_A,\ \preceq_A,\ \triangleright_A)$ and $B = (S_B,\ L_B,\ \preceq_B,\ \triangleright_B)$, we define $A \vec{\times} B = (S,\ L,\ \preceq,\ \triangleright)$ where

$$\begin{array}{rcll} S & = & (S_A - \{\infty\}) \times (S_B - \{\infty\}) \cup \{\infty\} & \\ L & = & L_A \times L_B & \\ \preceq & = & \preceq_A \vec{\times} \preceq_B & \text{(lexicographic order)} \\ \triangleright & = & \triangleright_A \times \triangleright_B & \end{array}$$

As a technicality, the operation $\triangleright_A \times \triangleright_B$ is defined so that the infinity metrics are removed from the component algebras, a single infinity is added to the combined type, and $\triangleright$ is defined so that an infinity in a single component propagates outwards to make the whole product become infinity:

$$(l_A,\ l_B)\ \ \triangleright_A \times \triangleright_B\ \ (s_A,\ s_B)$$
$$= \begin{cases} \infty & \text{if } l_A \triangleright_A s_A = \infty \\ \infty & \text{if } l_B \triangleright_B s_B = \infty \\ (l_A \triangleright_A s_A,\ l_B \triangleright_B s_B) & \text{otherwise} \end{cases}$$

The lexicographic order defined by $\vec{\times}$ means that metrics are compared by examining the first component first, using the second component to tie-break if the first components are equal. The route selection process in BGP, ignoring `MED`s, can be viewed as implementing a lexicographic choice function.

The *function union* constructor provides a way to combine different kinds of arc into a single routing algebra. The metric type and comparison operator must be the same in both of the component algebras A and B, and are not changed. Given the algebras $A = (S,\ L_A,\ \preceq,\ \triangleright_A)$ and $B = (S,\ L_B,\ \preceq,\ \triangleright_B)$, we construct $A \uplus B = (S,\ L_A \uplus L_B,\ \preceq,\ \triangleright_A \uplus \triangleright_B)$, where the labels are a *disjoint union* of L_A and L_B :

$$L_A \uplus L_B \quad = \quad \{\mathrm{inl}(l) \mid l \in L_A\} \cup \{\mathrm{inr}(l) \mid l \in L_B\}.$$

We use the tags inl and inr to distinguish elements in the 'left' and 'right' components of the disjoint union. The operation $\triangleright_A \uplus \triangleright_B$ performs a case split on the label to determine which of the original algebras' operations to apply, $\triangleright_A$ or $\triangleright_B$:

$$\begin{array}{rcl} \mathrm{inl}(l)\ \ \triangleright_A \uplus \triangleright_B\ \ s & = & l \triangleright_A s, \\ \mathrm{inr}(l)\ \ \triangleright_A \uplus \triangleright_B\ \ s & = & l \triangleright_B s. \end{array}$$

We now define two constructors that take only a single component routing algebra, and then modify its behaviour. First, the functions right and left are defined as

$$\begin{array}{rcl} l\ \mathrm{right}\ s & = & s, \\ l\ \mathrm{left}\ s & = & l. \end{array}$$

We can *lift* these operations to routing algebras (parameterised on a component algebra) as follows.

$$\begin{array}{rcl} \mathbf{right}(S,\ L,\ \preceq,\ \triangleright) & = & (S,\ \mathbf{1},\ \preceq,\ \mathrm{right}), \\ \mathbf{left}(S,\ L,\ \preceq,\ \triangleright) & = & (S,\ S,\ \preceq,\ \mathrm{left}). \end{array}$$

Here $\mathbf{1}$ is the *unit set* containing only a single element, $\mathbf{1} = \{\bot\}$. In **right** this is used because there is no choice of label on an arc; the label is ignored entirely and the metric never changes. The metric is therefore controlled entirely by the origination point of the route. The metric type and order are copied unchanged from the component algebra. **left** also copies the metric type and order, but arcs are labelled with the metric type S. When a route is sent across an arc, its metric is entirely forgotten and replaced with the label value. This implements a 'local preference' policy.

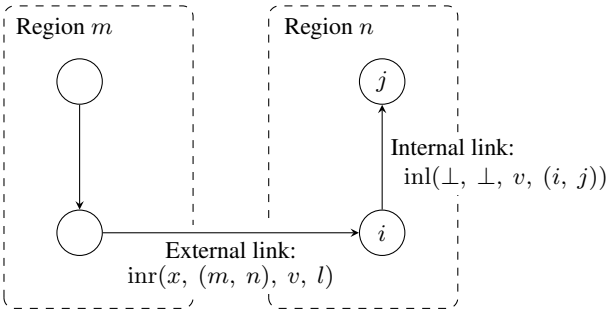


Figure 3: A network with link weights from the scoped product algebra.

2.2 A Mini-Metalanguage

The syntax for our small metalanguage is as follows:

base ::= `sp`
| `cpp`
| `paths`

exp ::= *base*
| `lex_product <`*name*`:`*exp*`,` *name*`:`*exp*`>`
| `function_union <`*name*`:`*exp*`,` *name*`:`*exp*`>`
| `right` *exp*
| `left` *exp*

We now need to define the *semantics* of the metalanguage. This is a mapping from the syntax onto our mathematical model of a routing algebra. First, we say

$$[\![\texttt{sp}]\!] = (\mathbb{N}^\infty,\ \mathbb{N}^\infty,\ \leq,\ +)$$

to declare that the semantics of `sp` is the shortest-paths routing algebra described earlier. Similarly, $[\![\texttt{cpp}]\!]$ and $[\![\texttt{paths}]\!]$ are the other two basic routing algebras described.

We define the semantics of expressions in a recursive way, making use of the semantics of their subexpressions.

$$\begin{array}{rcl} [\![\texttt{lex\_product} < \texttt{n} : \texttt{e}_1,\ \texttt{m} : \texttt{e}_2 >]\!] & = & [\![\texttt{e}_1]\!] \vec{\times} [\![\texttt{e}_2]\!] \\ [\![\texttt{function\_union} < \texttt{n} : \texttt{e}_1,\ \texttt{m} : \texttt{e}_2 >]\!] & = & [\![\texttt{e}_1]\!] \uplus [\![\texttt{e}_2]\!] \\ [\![\texttt{right e}]\!] & = & \mathbf{right}([\![\texttt{e}]\!]) \\ [\![\texttt{left e}]\!] & = & \mathbf{left}([\![\texttt{e}]\!]) \end{array}$$

Note that the names present in the syntax are discarded in the semantics – they help us talk about components of a routing algebra, but have no effect on the algebraic semantics. (We intend to incorporate names into the semantics in future work.)

2.3 Scoped product example

To illustrate the utility of this mini-metalanguage we introduce a *scoped product* [8] example. The defined routing algebra models a single-level region system, conceptually similar to BGP ASes. The metric is split into two parts: the first is used for external (inter-region) routing, and the second is used for internal (intra-region) routing. All arcs in the network are labelled as either internal or external. Figure 3 shows a simple network separated into regions with internal and external links (the labels will be explained below).

This routing algebra would be compiled to produce a *single* routing protocol; but comparing to existing standard routing protocols, it will give similar functionality to a *combination* of an EGP (such as BGP, for the inter-region component) and an IGP (such as RIP, for the intra-region component).

The example can be written in our syntax as:

```
function_union
<
  internal:
    lex_product
    <
      ecomm: right cpp,
      epath: right paths,
      idist: sp,
      ipath: paths
    >,
  external:
    lex_product
    <
      ecomm: cpp,
      epath: paths,
      idist: left sp,
      ipath: left paths
    >
>
```

In this example we use n-ary `lex_product` as shorthand for nested binary products. First we will consider the component named `internal` in the `function_union`. This is a lexicographic product of four components. The first two components are of the form `right e`, and referring back to our definition of $[\![\texttt{right e}]\!]$ we see the metric type is the same as $[\![\texttt{e}]\!]$'s metric; but the label type is the unit set, and the new $\triangleright$ ignores the label and passes the metric through unchanged. We use this for the *external* components of the metric, so that they pass through the interior of a region (i.e. across `internal` arcs) unchanged, and can be used for inter-region routing once they come out of the region again.

Ignoring the `right` constructors for the moment, the first of these external components, `ecomm`, is used to implement the inter-region commercial customer–provider–peer relationships with `cpp`. Since it is the first component in the lexicographic product, it is the first to be compared when selecting between multiple routes, and will therefore enforce the constraint that customer routes are preferred over any other routes.

This is followed by a `paths` component, to record the identifiers of the regions the route passes through. `paths` was defined with shorter lists being preferred over longer lists, meaning that this routing algebra will prefer routes that have passed through fewer regions. Since duplicate list entries are disallowed, this also prevents routing loops from one region back into itself.

The next two components of the lexicographic product are providing the internal, intra-region part of the metric. We use `sp` for a simple integer distance measure, similar to RIP. To avoid the counting-to-infinity problems of distance vector protocols, without imposing a small maximum value on distances, we add another `paths` component (`ipath`) to keep track of the router identifiers used by the path *inside* the current region.

The `external` part of the `function_union` has exactly the same metric type as the `internal` part, which is an important aspect of our model: routes are not tagged as external or internal, only arcs are. Unlike `internal`, it doesn't use `right` for the external components `ecomm` and `epath`, so their original label types and $\triangleright$ functions will be used. However, it uses `left` for the internal components, which means their values will be forgotten and replaced when routes cross over external arcs between regions; internal routing information never leaks outside a region.

Given these definitions, we can derive the semantics of this example routing algebra by applying the rules to its syntax. The metric type S is the product

$$(\{C,\ R,\ P\} \times \text{lists} \times \mathbb{N} \times \text{lists}) \cup \{\infty\}.$$

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

$$\text{inl}(\mathbf{1} \times \mathbf{1} \times \mathbb{N}^\infty \times (\text{id} \times \text{id})), \text{ or}$$
$$\text{inr}(\{C,\ R,\ P,\ \infty\} \times (\text{id} \times \text{id}) \times \mathbb{N}^\infty \times (\text{id} \times \text{id}))$$

where id is the router or region identifier type.

Finally, the $\triangleright$ function "extracted" by the semantics is presented in Figure 4(a). Line (1) corresponds to labels on internal links in Figure 3, while line (2) deals with labels on inter-region arcs.

The rules are sufficiently precise that we can implement a compiler for this metalanguage [2], taking the syntax as input and automatically generating optimised code that implements the algebra's types and functions. This code can be linked into a generic routing protocol implementation, resulting in a new routing protocol based on the algebra specified in the metalanguage.

However, some generic routing protocol implementations (for example a modified version of BGP where the route attributes have been replaced with hooks for a routing algebra) are much closer to the node-oriented view of figure 1 than the algebraic view. As well as the conceptual gap between these two views, there is this implementation gap that also needs to be bridged. The next section extends the metalanguage so that an implementation in the node-oriented view can be extracted automatically.

3. THE EXTENSION

We now extend the mini-metalanguage so that we can extract consistent configuration languages as described in Figure 2. We add only two new constructors to the language, but we substantially modify the semantic function $[\![\ldots]\!]$. An expression e will now map to a pair, $[\![\texttt{e}]\!]^\circ = (A,\ C)$, where $A = (S,\ L,\ \preceq,\ \triangleright)$ is the routing algebra as before, and C is a tuple $(L_I,\ L_E,\ W,\ \triangleright_I,\ \triangleright_E,\ \odot)$ capturing the other components of a configuration language described in Figure 2.

We want to guarantee consistency by construction. That is, we want the following theorem to hold:

THEOREM 1. *If e is an expression in the extended metalanguage and $[\![\texttt{e}]\!]^\circ = (A,\ C)$ with*

$$\begin{aligned} A &= (S,\ L,\ \preceq,\ \triangleright), \\ C &= (L_I,\ L_E,\ W,\ \triangleright_I,\ \triangleright_E,\ \odot), \end{aligned}$$

then the consistency condition holds. That is for every $l_I \in L_I$, $l_E \in L_E$ and $s \in S$ we have

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

In the definition of $[\![\texttt{e}]\!]^\circ$ we need a default configuration behaviour for any expression that does not use the new constructors. We choose, somewhat arbitrarily, to put configuration and computation on the "importing side" of an arc. If e does not use any new constructors, then we want the semantic function to return

$$[\![\texttt{e}]\!]^\circ = ([\![\texttt{e}]\!],\ C_{\text{import\_only}}(S,\ L,\ \triangleright)),$$

where $[\![\texttt{e}]\!] = (S,\ L,\ \preceq,\ \triangleright)$ and

$$C_{\text{import\_only}}(S,\ L,\ \triangleright) = (L,\ \mathbf{1},\ S,\ \triangleright,\ \text{right},\ \text{left}).$$

To see that $C_{\text{import\_only}}(S,\ L,\ \triangleright)$ is consistent with $(S,\ L,\ \preceq,\ \triangleright)$ suppose that $l_I = l$ and $l_E = \bot$. Then $(l_I \odot l_E) \triangleright s = (l \text{ left } \bot) \triangleright s = l \triangleright s = l \triangleright (\bot \text{ right } s) = l_I \triangleright_I (l_E \triangleright_E s)$.

The following tuples are also used in our semantics.

C	L_I	L_E	W	$\triangleright_I$	$\triangleright_E$	$\odot$
$C_{\text{export}}(S,\ L,\ \triangleright)$	$\mathbf{1}$	L	S	right	$\triangleright$	right
$C_{\text{export\_import}}(S,\ \otimes)$	S	S	S	$\otimes$	$\otimes$	$\otimes$
$C_{\text{right}}(S)$	$\mathbf{1}$	$\mathbf{1}$	S	right	right	right
$C_{\text{left}}(S)$	S	$\mathbf{1}$	$\mathbf{1}$	left	left	left

$$
\begin{array}{lrcll}
(1) & \mathrm{inl}(\bot,\ \bot,\ v,\ (i,\ j)) & \rhd & (ec,\ ep,\ d,\ ip) & = & (ec,\ ep,\ v+d,\ (i,\ j) \rhd_{paths} l) \\
(2) & \mathrm{inr}(x,\ (m,\ n),\ v, l) & \rhd & (ec,\ ep,\ d,\ ip) & = & (x \rhd_{cpp} ec,\ (m,\ n) \rhd_{paths} ep,\ v,\ l)
\end{array}
$$

(a) The arc-oriented operation $\rhd$ for the scoped-product example.

$$
\begin{array}{lrcll}
(3) & \mathrm{inl}(\bot,\ \bot,\ v,\ \bot)) & \rhd_E & (ec,\ ep,\ d,\ ip) & = & \mathrm{inl}(ec,\ ep,\ v+d,\ ip) \\
(4) & \mathrm{inr}(x,\ (m,\ n),\ \bot,\ \bot) & \rhd_E & (x \rhd_{cpp} ec,\ ep,\ d,\ ip) & = & \mathrm{inr}(x \rhd ec,\ (m,\ n) \rhd_{paths} ep,\ \bot,\ \bot) \\
(5) & \mathrm{inl}(\bot,\ \bot,\ v,\ (i,\ j)) & \rhd_I & \mathrm{inl}(ec,\ ep,\ d,\ ip) & = & (ec,\ ep,\ v+d,\ (i,\ j) \rhd_{paths} ip) \\
(6) & \mathrm{inr}(\bot,\ \bot,\ v,\ l) & \rhd_I & \mathrm{inr}(ec,\ ep,\ \bot,\ \bot) & = & (ec,\ ep,\ v,\ l) \\
(7) & \mathrm{inl}(\bot,\ \bot,\ v,\ (i,\ j)) & \rhd_I & \mathrm{inr}(ec,\ ep,\ \bot,\ \bot) & = & \infty \\
(8) & \mathrm{inr}(\bot,\ \bot,\ v,\ l) & \rhd_I & \mathrm{inl}(ec,\ ep,\ d,\ ip) & = & \infty
\end{array}
$$

(b) The node-oriented operations for the scoped-product example.

Figure 4: Functions extracted from the scoped-product examples. Note that the $\bot$-values could easily be optimized away when implementing these types.

We can then show that

$$
\begin{array}{rl}
C_{\mathrm{export}}(S,\ L,\ \rhd) & \text{is consistent with } (S,\ L,\ \preceq,\ \rhd) \\
C_{\mathrm{export\_import}}(S,\ \otimes) & \text{is consistent with } (S,\ S,\ \preceq,\ \otimes) \\
C_{\mathrm{right}}(S) & \text{is consistent with } (S,\ \mathbf{1},\ \preceq,\ \mathrm{right}) \\
C_{\mathrm{left}}(S) & \text{is consistent with } (S,\ S,\ \preceq,\ \mathrm{left}).
\end{array}
$$

The operations of $C_{\mathrm{export}}(S,\ L,\ \rhd)$ shift computation to the "export side" of an arc. The operations of $C_{\mathrm{export\_import}}(S,\ \otimes)$ allow both import and export computations, as long as the corresponding operation $\otimes$ is associative. The operations of $C_{\mathrm{right}}(S)$ cause a metric s to be copied over an arc, while the operations of $C_{\mathrm{left}}(S)$ cause the origination of a new value s on import.

For the semantics of lexicographic product and function union, we need to compose tuples of configuration operations,

$$
\begin{array}{rcl}
C_1 & = & (L_I^1,\ L_E^1,\ W^1,\ \rhd_I^1,\ \rhd_E^1,\ \odot^1), \\
C_2 & = & (L_I^2,\ L_E^2,\ W^2,\ \rhd_I^2,\ \rhd_E^2,\ \odot^2).
\end{array}
$$

This is accomplished by defining $C_1 \times C_2$ and $C_1 \uplus C_2$ as follows.

	$C_1 \times C_2$	$C_1 \uplus C_2$
L_I	$L_I^1 \times L_I^2$	$L_I^1 \uplus L_I^2$
L_E	$L_E^1 \times L_E^2$	$L_E^1 \uplus L_E^2$
W	$W^1 \times W^2$	$W^1 \uplus W^2$
$\rhd_I$	$\rhd_I^1 \times \rhd_I^2$	$\rhd_I^1 +_\infty \rhd_I^2$
$\rhd_E$	$\rhd_E^1 \times \rhd_E^2$	$\rhd_E^1 \uplus \rhd_E^2$
$\odot$	$\odot^1 \times \odot^2$	$\odot^1 +_\infty \odot^2$

The operator $+_\infty$ is defined as

$$
\begin{array}{rcl}
\mathrm{inl}(a)\ f +_\infty g\ \mathrm{inl}(b) & = & f(a,\ b), \\
\mathrm{inr}(a)\ f +_\infty g\ \mathrm{inr}(b) & = & g(a,\ b), \\
\mathrm{inl}(a)\ f +_\infty g\ \mathrm{inr}(b) & = & \infty, \\
\mathrm{inr}(a)\ f +_\infty g\ \mathrm{inl}(b) & = & \infty.
\end{array}
$$

It is not too hard to show that if C_1 is consistent with A_1 and C_2 is consistent with A_2, then

$$
\begin{array}{ll}
C_1 \times C_2 & \text{is consistent with } A_1 \vec{\times} A_2, \\
C_1 \uplus C_2 & \text{is consistent with } A_1 \uplus A_2.
\end{array}
$$

The extended grammar for the metalanguage is as follows.

exp ::= *base*
| `lex_product` <*name*`:`*exp*, *name*`:`*exp*>
| `function_union` <*name*`:`*exp*, *name*`:`*exp*>
| `right` *exp*
| `left` *exp*
| `export` *exp*
| `export_import` *exp*

$$
\begin{array}{l}
[\![\texttt{lex\_product} < \texttt{n} : \texttt{e}_1,\ \texttt{m} : \texttt{e}_2 >]\!]^\circ = (A_1 \vec{\times} A_2, C_1 \times C_2) \\
\qquad \text{where } \left\{ \begin{array}{rcl} [\![\texttt{e}_1]\!]^\circ & = & (A_1,\ C_1) \\ [\![\texttt{e}_2]\!]^\circ & = & (A_2,\ C_2) \end{array} \right. \\
[\![\texttt{function\_union} < \texttt{n} : \texttt{e}_1,\ \texttt{m} : \texttt{e}_2 >]\!]^\circ = (A_1 \uplus A_2, C_1 \uplus C_2) \\
\qquad \text{where } \left\{ \begin{array}{rcl} [\![\texttt{e}_1]\!]^\circ & = & (A_1,\ C_1) \\ [\![\texttt{e}_2]\!]^\circ & = & (A_2,\ C_2) \end{array} \right. \\
[\![\texttt{right e}]\!]^\circ = (\mathbf{right}(A),\ C_{\mathrm{right}}(S)) \\
\qquad \text{where } \left\{ \begin{array}{rcl} [\![\texttt{e}]\!]^\circ & = & (A,\ C) \\ A & = & (S,\ L,\ \preceq,\ \rhd) \end{array} \right. \\
[\![\texttt{left e}]\!]^\circ = (\mathbf{left}(A),\ C_{\mathrm{left}}(S)) \\
\qquad \text{where } \left\{ \begin{array}{rcl} [\![\texttt{e}]\!]^\circ & = & (A,\ C) \\ A & = & (S,\ L,\ \preceq,\ \rhd) \end{array} \right. \\
[\![\texttt{export e}]\!]^\circ = ((S,\ L,\ \preceq,\ \rhd),\ C_{\mathrm{export}}(S,\ L,\ \rhd)) \\
\qquad \text{where } \left\{ \begin{array}{rcl} [\![\texttt{e}]\!]^\circ & = & (A,\ C) \\ A & = & (S,\ L,\ \preceq,\ \rhd) \end{array} \right. \\
[\![\texttt{export\_import e}]\!]^\circ = (A,\ C_{\mathrm{export\_import}}(S,\ \otimes)) \\
\qquad \text{where } \left\{ \begin{array}{rcl} [\![\texttt{e}]\!]^\circ & = & (A,\ C) \\ A & = & (S,\ S,\ \preceq,\ \otimes) \end{array} \right.
\end{array}
$$

Figure 5: Semantics of extended mini-metalanguage.

The intended meaning of the expression `export` e is to override the default behavior and shift all computation to the "export side" of an arc. The intended meaning of the expression `export_import` e is to allow policy to be applied at both sides of an arc. This requires that e is associated with a routing algebra of the form $(S,\ S,\ \preceq,\ \otimes)$, where $\otimes$ is an associative operation. If this is not the case then our semantics will return an error (error handling is not explicitly defined here).

The extended semantic function is then defined in Figure 5. The proof of Theorem 1 now follows by a straightforward structural induction of the expression e.

Note that some operations throw away the C component associated with a sub-expression. In these cases we would expect that C is nothing more than the default behavior.

3.1 Example revisited

We now return to the scoped product example of the previous section. We want to split the computation associated with external links in a BGP-like manner. The *ecomm* and *epath* attributes should be computed on export, while the *idist* and *ipath* should originate new values on import into a region.

We accomplish this by using the new operations of the language (in bold):

```
function_union
<
  internal:
    lex_product
    <
      ecomm: right cpp,
      epath: right paths,
      idist: export_import sp,
      ipath: paths
    >,
  external:
    lex_product
    <
      ecomm: export cpp,
      epath: export paths,
      idist: left sp,
      ipath: left paths
    >
>
```

Figure 4 presents the configuration functions "extracted" from this expression by the semantic function. Line (3) corresponds to applying export policy on internal links, line (4) to export policy on external links, line (5) to import policy on internal links, and line (6) to import policy on external links. Lines (7) and (8) deal with configuration errors involving mismatched labels (we will address the problem of generating code to check this at adjacency set-up time in future work).

4. DISCUSSION

Most work in this area has focused on raising the level of abstraction of network configuration by various means of generating routing configuration files [1], checking correctness of such configurations [4], or centralising the computation of routes [3].

In contrast, we are attempting to integrate the *design* of a configuration language with an underlying algebraic semantics. We are currently implementing the scheme described here in the Metarouting toolkit [2]. This system includes a compiler that translates metalanguage expressions into into C++ code, guided by semantic functions of the kind described in this paper. The goal is to allow routing implementations to be automatically generated from a high-level specification and to automate the tedious theorem proving associated with the correctness of routing protocols.

Other approaches to the design of configuration languages include Nettle [13] and Declarative Routing [9]. Nettle suggests the use of the strongly-typed functional programming language Haskell for specifying routing configurations, while Declarative Routing uses a declarative database query language to express both policy application and routing algorithms.

Router vendors support *route maps* as an important mechanism for encoding policy. In Cisco IOS these consist of sequences of *match* and *set* commands, and can be applied to import and export sides of links. In BGP they can match on the route's *med* value, on its AS path (matching with a regular expression), on its communities, and on its prefix; and they can set the matching route's local preference and *med* attributes, set or add to its communities list, and prepend arbitrary lists to its AS path.

How might we model a route map over an algebra? If we start with the arc-oriented model (Figure 1(a)), we could imagine associating a *function* of type $h \in S \to L$ with a link. We could then define $h \hat{\triangleright} s = h(s) \triangleright s$. That is, the route map determines which label to apply to s by some kind of inspection of s itself. (An implementation of our model could use a language like Haskell to define the route map functions directly, as suggested by Nettle [13].)

Do route maps belong in the configuration language or in the routing algebra itself? The original metarouting paper [7] placed such functions in the algebra. An argument could be made that they belong in the configuration language. The difficulty hinges on which algebraic properties are important for correctness. If we need only the *increasing property*,

$$\forall s \in S,\ l \in L\ :\ s \neq \infty \implies s \prec l \triangleright s,$$

then the introduction of route maps will not impact correctness. This is because we will have

$$\forall s \in S,\ h \in S \to L\ :\ s \neq \infty \implies s \prec h \hat{\triangleright} s.$$

However, if we need the *monotonicity property*,

$$\forall s_1,\ s_2 \in S,\ l \in L\ :\ s_1 \preceq s_2 \implies l \triangleright s_1 \preceq l \triangleright s_2,$$

then the introduction of route maps may well impact correctness. This is because it may be very hard to enforce the condition

$$\forall s_1,\ s_2 \in S,\ h \in S \to L\ :\ s_1 \preceq s_2 \implies h \hat{\triangleright} s_1 \preceq h \hat{\triangleright} s_2.$$

For example, it may be that $h(s_1) = \omega$, where $\omega \triangleright s_1 = \infty$, so that $h \hat{\triangleright} s_1 = \infty$, but that $h \hat{\triangleright} s_2 \neq \infty$.

Acknowledgements

We would like to thank the Engineering and Physical Sciences Research Council (EPSRC) for support (Grant EP/F002718/1). Thanks also to the metarouting group for helpful feedback — Abdul Alim, Arthur Amorim, John Billings, Alexander Gurney, Vilius Naudziunas, and Balraj Singh.

5. REFERENCES

[1] C. Alaettinoglu, C. Villamizar, E. Gerich, D. Kessens, D. Meyer, T. Bates, D. Karrenberg, and M. Terpstra. Routing policy specification language (RPSL), 1999.

[2] J. N. Billings, P. J. Taylor, M. A. Alim, and T. G. Griffin. Equid: A metarouting toolkit. Draft.

[3] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and implementation of a routing control platform. In *NSDI*, 2005.

[4] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *NSDI*, 2005.

[5] L. Gao and J. Rexford. Stable internet routing without global coordination. *IEEE/ACM Transactions on Networking*, pages 681–692, December 2001.

[6] M. Gondran and M. Minoux. *Graphs, Dioids, and Semirings. New Models and Algorithms*. Springer, 2008.

[7] T. G. Griffin and J. L. Sobrinho. Metarouting. In *Proc. ACM SIGCOMM*, August 2005.

[8] A. Gurney and T. G. Griffin. Lexicographic products in metarouting. In *ICNP*, October 2007.

[9] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *Proc. ACM SIGCOMM*, 2005.

[10] J. L. Sobrinho. Algebra and algorithms for QoS path computation and hop-by-hop. *IEEE/ACM Transactions on Networking*, 10(4):541–550, August 2002.

[11] J. L. Sobrinho. Network routing with path vector protocols: Theory and applications. In *Proc. ACM SIGCOMM*, September 2003.

[12] J. L. Sobrinho. An algebraic theory of dynamic network routing. *IEEE/ACM Transactions on Networking*, 13(5):1160–1173, October 2005.

[13] A. Voellmy and P. Hudak. Nettle: A language for configuring routing networks. Draft.