Defining and implementing a new routing protocol is difficult!

- The space is large
- The proofs are difficult
- Correctness conditions hard to get right

Could the design process be partially automated?
A simple grammar for a mini-metalanguage

Our mini-metalanguage will describe routing algebras

- \((S, \oplus, F \subseteq S \rightarrow S)\)
- \(\oplus\) is commutative, idempotent, and has identity \(\alpha\).

base ::= sp
| bw
| rel

algebra ::= term
| right term
| left term
| lex_product term ... term
| function_union term ... term

term ::= base
| (algebra)

The Semantics

For category base

- \([sp]^B = (\mathbb{N} \cup \{\infty\}, \min, F_+)\)
- \([bw]^B = (\mathbb{N} \cup \{\infty\}, \max, F_{\min})\)
- \([rel]^B = ([0, 1], \max, F_\times)\)

For category term

- \([b]^T = [b]^B\)
- \([[a]]^T = [a]^A\)
The Semantics

For category \textit{algebra}

- \([ t ]^A = [ t ]^B\)
- \([\text{right } t]^A = (S, \oplus, \{i\})\)
  - where \([ t ]^T = (S, \oplus, F)\)
- \([\text{left } t]^A = (S, \oplus, K(S))\)
  - where \([ t ]^T = (S, \oplus, F)\)

\[ \text{lex}_\text{product } t \] \[ A \]
\[ (S, \oplus, F) \times (T, \odot, G) = (S \times T, \oplus \times \odot, F \times G) \]
  - where \([ t ]^T = (S, \oplus, F)\)
  - and \([ t' ]^T = (T, \odot, G)\)

\[ \text{lex}_\text{product } t \ldots t'' \] \[ A \]
\[ (S, \oplus, F) \times (T, \odot, G) = (S \times T, \oplus \times \odot, F \times G) \]
  - where \([ t ]^T = (S, \oplus, F)\)
  - and \([ \text{lex}_\text{product } t' \ldots t'' ]^A = (T, \odot, G)\)
The Semantics

- $[\text{function} \_\text{union} \ t]^A = [t]^T$
- $[\text{function} \_\text{union} \ t \ t']^A = (S, \oplus, F) +_m (S, \oplus, G) = (S, \oplus, F \cup G)$
  - where $[t]^T = (S, \oplus, F)$
  - and $[t']^T = (S, \oplus, G)$
- $[\text{function} \_\text{union} \ t \ t' \ldots t'']^A = (S, \oplus, F) +_m (S, \oplus, G) = (S, \oplus, F \cup G)$
  - where $[t]^T = (S, \oplus, F)$
  - and $[\text{function} \_\text{union} \ t' \ldots t'']^A = (S, \oplus, G)$

Some interesting properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>$\forall a, b \in S \forall f \in F : f(a \oplus b) = f(a) \oplus f(b)$</td>
</tr>
<tr>
<td>C</td>
<td>$\forall a, b \in S \forall f \in F - {\omega} : f(a) = f(b) \implies a = b$</td>
</tr>
<tr>
<td>K</td>
<td>$\forall a, b \in S \forall f \in F : f(a) = f(b)$</td>
</tr>
<tr>
<td>I</td>
<td>$\forall a \in S \forall f \in F : a \neq \alpha \implies a &lt;_L f(a)$</td>
</tr>
<tr>
<td>ND</td>
<td>$\forall a \in S \forall f \in F : a \leq_L f(a)$</td>
</tr>
</tbody>
</table>
We know a few rules ... 

(some of the) rules needed for global optimality

\[
\begin{align*}
M(\text{right}(S)) \\
M(\text{left}(S)) \\
C(\text{right}(S)) \\
K(\text{left}(S))M(S \not\rightarrow T) & \iff M(S) \land M(T) \land (C(S) \lor K(T)) \\
M(S +_m T) & \iff M(S) \land M(T)
\end{align*}
\]

... and a few more rules

(some of the) rules needed for local optimality (and for loop-freedom in next-hop forwarding)

\[
\begin{align*}
I(S \not\rightarrow T) & \iff I(S) \lor (\text{ND}(S) \land I(T)) \\
\text{ND}(S \not\rightarrow T) & \iff I(S) \lor (\text{ND}(S) \land \text{ND}(T)) \\
I(S +_m T) & \iff I(S) \land I(T) \\
\text{ND}(S +_m T) & \iff \text{ND}(S) \land \text{ND}(T)
\end{align*}
\]
We can turn rules into bottom-up methods

Example: The \( \iff \) rule

\[
M(S 
\times
 T) \iff M(S) \land M(T) \land (C(S) \lor K(T))
\]

becomes a bottom-up method for deriving property \( M \) or property \( \neg M \) for any expression

\[
e = \text{lex_product } t_1 \; t_2
\]

<table>
<thead>
<tr>
<th>if derive properties for ( t_1 )</th>
<th>and derive properties for ( t_2 )</th>
<th>then derive property for ( e )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M, ; C )</td>
<td>( M )</td>
<td>( M )</td>
</tr>
<tr>
<td>( M )</td>
<td>( M, ; K )</td>
<td>( M )</td>
</tr>
<tr>
<td>( \neg M )</td>
<td>( \neg M )</td>
<td>( \neg M )</td>
</tr>
<tr>
<td>( \neg C )</td>
<td>( \neg K )</td>
<td>( \neg M )</td>
</tr>
</tbody>
</table>

Magic

We know everything about our base algebras

<table>
<thead>
<tr>
<th></th>
<th>M</th>
<th>C</th>
<th>K</th>
<th>I</th>
<th>ND</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>bw</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>rel</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Now, for each algebra expression \( a \) defined by our mini-metalanguage and each property \( P \), we can determine in a bottom-up manner whether

\[
P([a]^A)
\]

or

\[
\neg P([a]^A)
\]

holds.

No proofs required at algebra specification time!
A few examples

<table>
<thead>
<tr>
<th>lex_product sp bw</th>
<th>M</th>
<th>C</th>
<th>K</th>
<th>I</th>
<th>ND</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>lex_product sp sp</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>lex_product bw sp</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>lex_product rel bw</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>lex_product rel bw sp</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

BGP-like Example

```plaintext
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 > >
```
import on internal
\[
\text{inl}(\perp, \perp, v, (i, j)) \triangleright (ec, ep, d, ip) = (ec, ep, v + d, (i, j) \triangleright_{\text{paths}} l)
\]

import on external
\[
\text{inr}(x, (m, n), v, l) \triangleright (ec, ep, d, ip) = (x \triangleright_{\text{cpp}} ec, (m, n) \triangleright_{\text{paths}} ep, v, l)
\]