Technical University of Berlin
Faculty IV (Electrical Engineering and Computer Science)
Institute of Telecommunication Systems
Communication and Operating Systems, Prof. Heiß

Approaches for Analysing and Comparing Packet Filtering in Firewalls

Diploma Thesis

author:	Lars Frantzen
matriculation number	178329
email:	lf@cs.tu-berlin.de
date:	May 15, 2003

Die selbständige und eigenhändige Anfertigung versichert an Eides Statt:

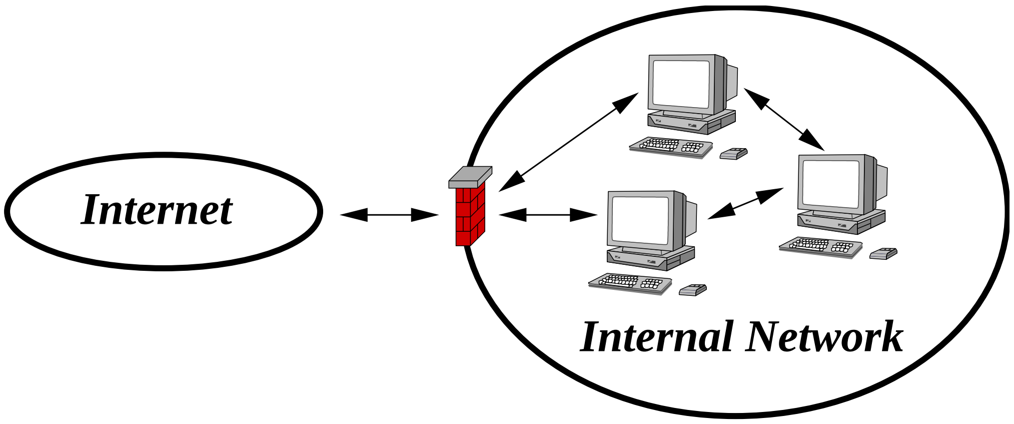
Nijmegen (Niederlande)	Datum	Lars Frantzen

Contents

Chapter 1

Introduction

When a local network is connected to a public network it is exposed to a broad variety of threats. Hackers are seeking continuously to infiltrate a weakly secured machine. Nowadays hacking has become an accessible technique to everyone because of existing tools which exploit known security holes without the need of any expertise by the “user”. The public Internet plays a vital role in all (rich) societies and hence protection from undesirable guests is a necessity. Firewalls play an important role in securing local networks from the other parts of the Internet. They are usually located at a single-purpose router or at a general-purpose computer.



Data transmitted over the Internet is split into *IP-packets*. When any file is sent from one place to another, the Transmission Control Protocol (TCP) layer of TCP/IP divides the file into pieces of an efficient size for routing. Each of these packets is separately numbered and includes the Internet address of its destination together with additional header information. The individual packets of a given connection may travel different routes through the Internet. When they have all arrived, they are reassembled into the original file by the TCP layer at the receiving end.

A firewall basically has to judge if an incoming or outgoing packet is allowed to pass by or has to be denied. Therefore the administrator decides which packets should be allowed and which should not and thereupon configures the firewall accordingly to his/her decisions. This is done using a list of *filter-rules*. Unfortunately there is a tendency of these rule-lists to forfeit perceivability by growing with the time because new rules are added or old ones are changed. As long as the same maintainer keeps control over the firewalls he administrates (which includes commenting, documenting,. . .) everything might be under control but when a new maintainer is confronted with such a grown set of lists he most likely wont be able to grasp the implemented policy. Also changes applied to a long list of rules are very hard to test for desired behaviour because their interconnection with the already existing rules is elusive. Therefore there is a great need for tools which allow performing analysis here. In this work we will examine data structures and algorithms which enable analytical tasks in an efficient, i.e. implemantable way.

Not theoretical but pragmatical considerations have motivated the present syntax in which system administrators configure firewalls. Unfortunately this syntax is not suited to perform serious analysis. Even simple questions like *are two firewalls doing the same?* can not be answered. Approaches to analyse and compare firewalls are subject to the succeeding chapters. In this context new syntactical descriptions of firewalls which allow to reason about them will be presented.

Currently just a few approaches exist to analyse firewalls and they are all still under development. In this work we will outline an existing attempt based on boolean logic and introduce a new concept based upon computational geometry. Implementation issues will be clarified using efficient known data structures and algorithms. Additionally a new algorithm will be presented which can be applied to a broad range of geometry-related topics. It has some promising features which make it an interesting alternative to other known algorithms which are already implemented in firewall analysis tools.

Most of the actual attempts are based on logic and supporting data structures like binary decision diagrams (BDDs). We will give all these approaches a common theoretical background using set- and category theory. This will show how all approaches derive from the same abstract operations and algorithms.

This work lays the fundament to implement a geometry-based solution and to compare it with other ones. It will be seen that a comparison based on theoretical bounds is not practical because all approaches highly depend on heuristics. Therefore only real-life implementations may judge and compare here in the future.

Outline

In **chapter 2** the basic notions and notations will be formally defined. We start investigating the topic by founding it on set-theory. This will not yield implementable algorithms but it clarifies the tasks we deal with in a general way without the bounds of a specialised data structure. Based on this the two main directions to proceed are introduced: logic and geometry.

Chapter 3 gives a structural view on the topic using category theory. All concepts will be modelled as a category and their equivalence (in a categorical sense) will be proven. This shows the interconnections between them and gives them a common mathematical structure. The aim of this chapter is to increase the understanding of the objects we deal with and to proof the structural correctness of the geometrical and logical approaches.

Chapter 4 discusses implementation issues of the geometrical approach. We will introduce and develop supporting data structures and algorithms for storing and modifying so called *hyperrectangles*.

Chapter 5 gives a summary on the topics we have dealt with and an outlook.

Acknowledgements

I would like to thank my supervisors Hans-Ulrich Heiß and Arthur Lochstampfer for their scientific support. Special thanks go to Scott Hazelhurst and Pasi Eronen for supplying me with additional information regarding their work. A very special thanks to Uwe Wolter for his support during the last years.

I would also like to thank Hartmut Ehrig, Bernd Mahr, Klaus Robering, Sebastian John, Gunnar Schröter, Tina Wieczorek and Dan Tidhar for the instructive years we have worked together.

Without several free tools this work would not have been possible, e.g. LaTeX, xpdf, Emacs, CUDD and Xfig.

Chapter 2

Basic notions and approaches

In this chapter the introduced concepts and tasks will be exemplified and defined concretely. Firstly packet filters are presented followed by a demonstration of the analytical problems which come along with their syntactical descriptions (rule-lists). The next part of the chapter examines possible solution approaches from a formal point of view. Set theory is used to model patterns of data structures and algorithms which will be applied to concrete implementations in the remainder of the chapter.

2.1 Packet filtering

Packet filtering is a basic firewall technology. Based on a list of *packet filtering rules* a packet filter decides for every IP-packet it handles if it is **permitted**, i.e. routed to its destination, or **denied**. Packet filtering can roughly be divided into two classes:

- **Basic packet filtering**
  The filter-device examines primarily the data given in the headers of the Internet- and Transport Layer. Usually this means an IP-header encapsulating a TCP- UDP- or ICMP-packet. In addition, the device has knowledge of the interface the packet came in from or wants to go out through.

- **Stateful or dynamic packet filtering**
  In this case the filter-device keeps track of certain states of a transaction. For example it is possible to define rules like:
  Let incoming UDP packets through only if they are responses to outgoing UDP packets you have seen.
  Advanced packet filters are even able to look deeply in protocol-specific data residing at the Application Layer, e.g. usernames of ftp-connections.

We will concentrate in this work on the analysis of basic packet filtering in IPv4. The most common header attributes which are of interest for packet filtering and their domains are shown in table 2.1. `MAXIFACE` represents the maximal number of interfaces addressed by the

Table 2.1: IP-Packets with respect to Packet Filtering

Attribute	**Domain**
Interface	$\{\text{iface}_i \mid 1 \leq i \leq \texttt{MAXIFACE}\}$
Direction	{In, Out}
Source Address	$\{n \mid 0 \leq n \leq 2^{32} - 1\}$
Destination Address	$\{n \mid 0 \leq n \leq 2^{32} - 1\}$
Protocol	{TCP, UDP, ICMP}
Source Port	$\{n \mid 0 \leq n \leq 2^{16} - 1\}$
Destination Port	$\{n \mid 0 \leq n \leq 2^{16} - 1\}$
`ACK` Bit Set	{Any, Yes}

Table 2.2: Example Rule-List

Rule	**Source IP**	**Dest.IP**	**Action**
A	10.*.*.*	172.16.6.*	Permit
B	10.1.99.*	172.16.*.*	Deny
C	Any	Any	Deny

firewall. Depending on the *Transport Layer Protocol* not all attributes are relevant. The table shows that about 2^{100} different packet headers are possible here. This makes clear that there is a great necessity for a formalism that can talk about all this packets efficiently. A list of *filter-rules* is used to define the sets of permitted and denied packets. Table 2.2 shows an example (taken from [24]). Note that in IPv4, IP-addresses are represented as 4 segments of 8 bit each. Rule C corresponds to a *default-deny policy.* Usually this is the default policy implemented in a firewall, rule C could then be omitted. We will always assume here that such a default-policy exists as a rule, i.e. the list handles every possible packet.

Independent of a concrete packet filter syntax, all of them provide constraints on the domains of the attributes. For example `Source IP = 192.168.*.*` is a wildcard-like notation for the constraint ($\texttt{SourceIP} \geq 192.168.0.0) \wedge (\texttt{SourceIP} \leq 192.168.255.255$). For every packet the firewall performs a *lookup*, i.e. going top-down through the list until a rule matches the packet header. Then it takes the action specified by that rule. A good practical introduction to firewalls is [24].

2.2 Analytical tasks

Firewall based research can roughly be divided into three main areas: *enhancing lookup-performance*, *analysis through testing* and *analysis through verification*. Here we concentrate on the latter. That means that we want to analyse a single firewall and compare several ones by proving certain properties. Additionally we are interested in computing new rule-lists out of given ones. These properties and operations should allow for answering questions like:

- regarding a single rule-list:
    - ▶ are there redundant rules?
    - ▶ can the rules be optimised (fewer, ordered, ...)?
- regarding several rule-lists $\mathtt{L}_i$:
    - ▶ accepts respectively denies $\mathtt{L}_1$ the same packets as $\mathtt{L}_2$?
    - ▶ are all packets accepted by $\mathtt{L}_1$ also accepted by $\mathtt{L}_2$?
    - ▶ which packets are accepted by both $\mathtt{L}_1$ and $\mathtt{L}_2$?
    - ▶ which packets are accepted by $\mathtt{L}_2$ but not by $\mathtt{L}_1$?

The firewall administrator can then check and analyse the firewall he is confronted with. Especially if it is a rule-list written by a different person it is quite hard to see what exactly the firewall does. He can also compare several firewalls or test if modifications he applies have the desired effect.

The main problem with filter-rules is that rules may “overlap”, i.e. two rules talk partly about the same set of IP-packets. If one of them is a *Permit*- and the other a *Deny*-rule it is not clear what should be done with the packets in the intersection. That is why a total order needs to be defined on the rules. This is achieved by going top-down through a list of rules as introduced. Because of the importance of the rule-order, rules can not be seen separately - preceding rules may overlap and therefore reduce the effect of a successor.

We visualise this with the example given in table 2.2. If we interpret the `Source IP` and `Dest.IP` Domains as axes spanning a plane we can see the three rules as rectangles. In figure 2.1 the successive application of the rules in order A, B, C is shown. It can be seen that all rules overlap pairwise. Rule B is redundant in this order because it is followed by rule C which also is a *Deny*-rule that includes, respectively totally overlaps rule B. The black rectangle in the lower right part of the figure represents the resulting permitted IP-packets. If we apply the rules in order B, A, C the resulting permitted IP-packets differ. This is visualised in figure 2.2. We have seen that rule B is redundant in the A, B, C-scenario. That means that removing rule B from rule-list A, B, C yields a (semantically) equal list. It follows that semantical equality does not lead to syntactical equality. In other words filter-rules are no *canonical* structures which is another problematic property. These two properties of rule-lists make them unsuited for our needs. In this work we will

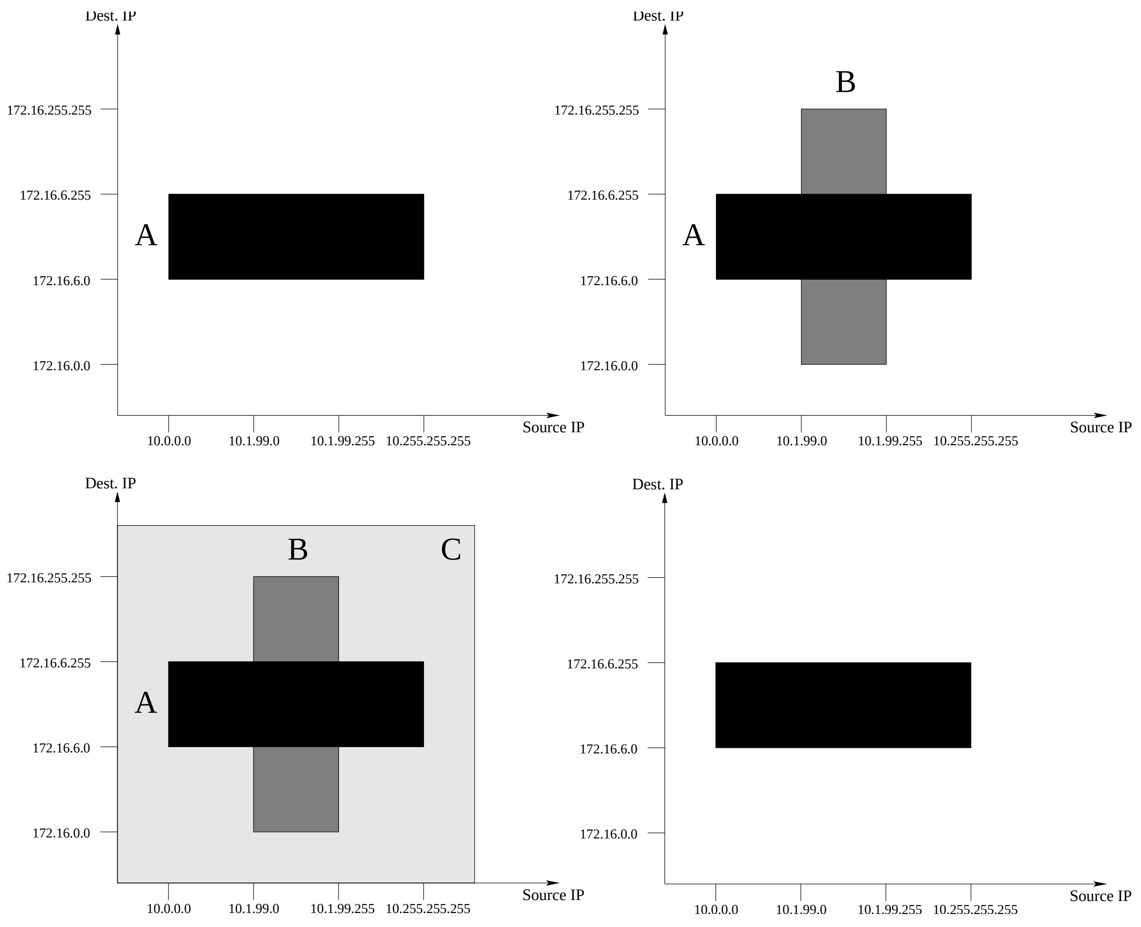


Figure 2.1: Rules Applied in Order A,B,C

study new syntactical representations of rule-lists which allow us to check the properties and apply the operations we want. To do so we first have to transform the rule-list to such a new syntax and make sure that the semantics, i.e. the permitted and denied IP-packets, retain. Additionally it should be possible to transform the new syntactical representation back to a rule-list. Because they are not canonical we can not expect to get the syntactically equal rule-list after transforming to a new syntax and back. Only the semantics will retain. The following diagram shows these connections:

$$\begin{tikzcd}
\text{Rule} - \text{List } R_1 \arrow[d, Rightarrow] \arrow[drr, hook] & & \\
\text{New Syntax} \arrow[d, Rightarrow] \arrow[rr, hook] \arrow[u, phantom, "=" right] & & \textit{Semantics } S \\
\text{Rule} - \text{List } R_2 \arrow[urr, hook] \arrow[u, phantom, "=" right] & &
\end{tikzcd}$$

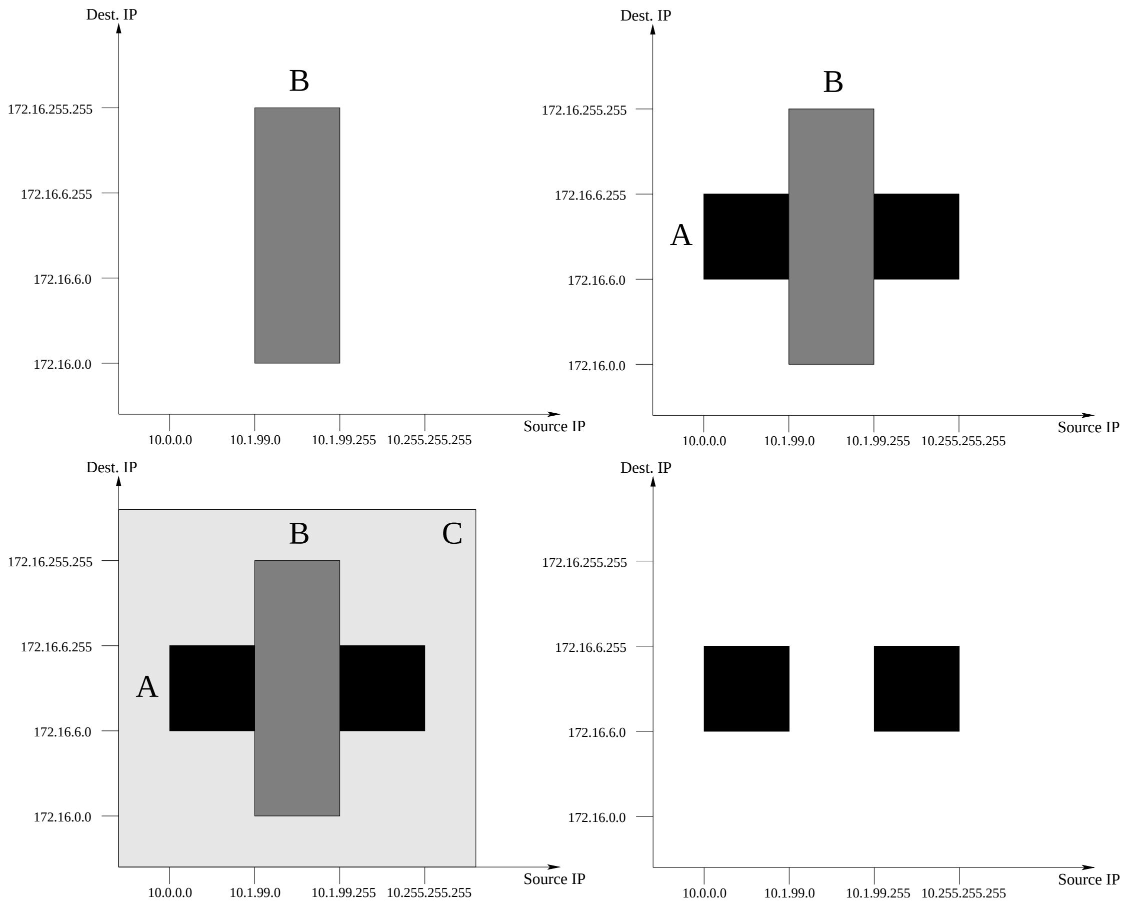


Figure 2.2: Rules Applied in Order B,A,C

More than the two actions permit/deny are conceivable, e.g. a *Reject*-rule could deny a packet and additionally inform the sender about the rejection. We will also study generalisations here.

Work on analysis through verification techniques is relatively rare. Just a few approaches exist and they are still under development. Almost all of them are either logic- or geometry-based. That is why we first give all approaches a common structural foundation based on set- and category theory. Then two concrete implementations will be introduced, a logical approach which was mainly investigated by Scott Hazelhurst ([13], [12], [3]) and a geometrical approach which was partly studied by the author.

Besides studying the concrete implementations we will examine how they derive from the common structural foundation. We want to give the *big picture* which helps to understand all the approaches and their interconnections from a higher structural level.

2.3 A foundation based on sets

For every possible IP-packet a firewall has to decide if it should be permitted or denied. We have seen that a great amount of different IP-headers can occur. Filter-rules are used to classify these packets. Because packet filters distinguish IP-packets only regarding their header information we will use IP-packet and IP-header synonymously in the following[1]. We have also motivated that rule-lists are unsuited for further analysis. New approaches will be introduced with respect to a set-based foundation.

The basic objects we are dealing with are IP-packets, filter-rules and rule-lists. Let IP be the set of all possible IP-packet headers regarding an IPv4 packet filter. We already know that $|IP| \simeq 2^{100}$, the exact size in a concrete firewall implementation does not matter here. A packet p is in this sense just an element of IP. A single filter-rule describes a set of permitted or denied packets. So it represents a subset of IP connected with an action (permit or deny). A complete rule-list tells the firewall for every packet the desired action. So we can see such a list as a function $filter$:

$$filter : IP \to \{permit,\ deny\}$$

Because every function represents a *sorting of the domain by a property* (see [15]) we can also interpret a rule-list as two sets $Permit$ and $Deny$ which are a *partition*[2] of IP. Summarised we get:

Object	**Semantics**
packet	$p \in IP$
rule	$r \subseteq IP$
rule-list	$Permit \subseteq IP,\ Deny \subseteq IP$

Seeing the semantics of packet filters, i.e. rule-lists, from this point of view there is no hint pointing at procedural top-down semantics. This raises hope to find alternative structures. The first thing we have to do is to get rid of the procedural semantics. Let us see how this works with sets.

2.3.1 Transforming rule-lists to sets

We want to extract the semantics of a rule-list, that means we want to calculate the two sets $Permit$ and $Deny$. In other words we are transforming the procedural semantics into a simple set-based one.

The basic idea of the following algorithm[3] `SetSemTD` is that a packet is permitted by a rule-list iff it is permitted by a *Permit*-rule and not denied by a preceding *Deny*-rule. Analogously a packet is denied by the list iff it is denied by a *Deny*-rule and not permitted by a preceding *Permit*-rule. The algorithm gets as input a rule-list `L` and returns

[1] Formally we are defining equivalence-classes of IP-packets with the same relevant header information.
[2] see Appendix A
[3] A simple pseudocode is used throughout this work.

a set *Permit* of accepted IP-packets and a set *Deny* of rejected ones. It goes top-down through the list. `size(L)` is the number of rules in the list, `L(i)` is the rule number `i`. `type(L(i))` returns `permit` if the rule `i` of list `L` talks about permitted packets or `deny` otherwise. $[\![\texttt{L(i)}]\!]$ stands for the set of IP-packets the rule `L(i)` talks about (a subset of IP).

SetSemTD(List L)

```
$Permit$, $Deny$:  Set := $\emptyset$
FOR i := 1 TO size(L) DO {
   IF type(L(i)) = permit DO
     $Permit$ = $Permit \cup ([\![\texttt{L(i)}]\!] \setminus Deny)$
   IF type(L(i)) = deny DO
     $Deny$ = $Deny \cup ([\![\texttt{L(i)}]\!] \setminus Permit)$
   i := i+1
}
RETURN ($Permit$, $Deny$)
```

$Permit = IP \setminus Deny$ holds because *Permit* and *Deny* are a partition of IP. Therefore it is sufficient to figure out either the set *Permit* or the set *Deny*. We will choose *Permit* because it is usually much smaller than *Deny* with respect to a default-deny policy. Now the algorithm can be simplified to a bottom-up version `SetSemBU`:

SetSemBU(List L)

```
$Permit$:  Set := $\emptyset$
FOR i := size(L) DOWNTO 1 DO {
   IF type(L(i)) = permit DO
     $Permit$ = $Permit \cup [\![\texttt{L(i)}]\!]$
   IF type(L(i)) = deny DO
     $Permit$ = $Permit \setminus [\![\texttt{L(i)}]\!]$
   i := i-1
}
RETURN $Permit$
```

Finally a third recursive version `SetSemREC` is given. The basic idea is here that if the first rule of a list is a *Permit*-rule a packet is permitted by the list iff it is permitted by this rule or permitted by the rest of the list. Dealing with a first *Deny*-rule a packet is permitted iff it not matches this rule but is permitted by the rest of the list. `tail(L)` is the list `L` without the (actual) first rule `L(1)`:

SetSemREC(List L)

```
IF size(L) = 0 RETURN $\emptyset$
ELSE {
  IF type(L(1)) = permit DO
    RETURN SetSemREC(tail(L)) $\cup$ $[\![$L(i)$]\!]$
  IF type(L(1)) = deny DO
    RETURN SetSemREC(tail(L)) $\setminus$ $[\![$L(i)$]\!]$
}
```

All approaches to analyse firewalls use one of these algorithms applied to their data structure to get rid of the procedural semantics.

2.3.2 Performing analysis

In the semantical world of sets analysis is straightforward. From now on we denote with $[\![\mathtt{L}]\!]$ the set of permitted packets by rule-list L, i.e. $[\![\mathtt{L}]\!] =_{def}$ SetSemBU(L) = SetSemREC(L). We can now apply simple set operations to perform analytical tasks. To check if a given IP-packet p is permitted by a firewall with rule-list L we test if $p \in [\![\mathtt{L}]\!]$. Semantical equality of rule-lists is simply an equality on sets and so on. We get the following basic operations:

Analytical Task	**Semantics**
is packet p accepted by list L?	$p \in [\![\mathtt{L}]\!]$
are lists $\mathtt{L}_1$ and $\mathtt{L}_2$ equivalent?	$[\![\mathtt{L}_1]\!] = [\![\mathtt{L}_2]\!]$
are all packets accepted by $\mathtt{L}_1$ also accepted by $\mathtt{L}_2$?	$[\![\mathtt{L}_1]\!] \subseteq [\![\mathtt{L}_2]\!]$
which packets are accepted at least by $\mathtt{L}_1$ or $\mathtt{L}_2$?	$[\![\mathtt{L}_1]\!] \cup [\![\mathtt{L}_2]\!]$
which packets are accepted by both $\mathtt{L}_1$ and $\mathtt{L}_2$?	$[\![\mathtt{L}_1]\!] \cap [\![\mathtt{L}_2]\!]$
which packets are accepted by $\mathtt{L}_1$ but not by $\mathtt{L}_2$?	$[\![\mathtt{L}_1]\!] \setminus [\![\mathtt{L}_2]\!]$

Based on these operations more complex questions can be answered, e.g. *which packets are accepted by either* $\mathtt{L}_1$ *or* $\mathtt{L}_2$*, but not by both?* corresponds to $([\![\mathtt{L}_1]\!] \cup [\![\mathtt{L}_2]\!]) \setminus ([\![\mathtt{L}_1]\!] \cap [\![\mathtt{L}_2]\!])$.

Generalisation to multiple actions

Usually packet filters only have two possible actions, permit and deny. But as introduced in chapter 1 more actions are conceivable, e.g. *Reject*-rules. To analyse such filters the algorithms and operations have to be generalised. Now we deal with n possible actions $A_1, \dots, A_n$. Therefore such a rule-list with multiple actions represents a function $filter_M$:

$$filter_M : IP \to \{A_1, \dots, A_n\}$$

The sets $A_1, \dots, A_n$ are again a partition of IP. Summarised we get here:

Object	**Semantics**
packet	$p \in IP$
rule	$r \subseteq IP$
rule-list	$A_1 \subseteq IP, \ \dots, \ A_n \subseteq IP$

Adapting the Algorithms Now we give the generalised versions of the `SetSem` algorithms. `type(L(i))` gives here the number of the action taken by rule `L(i)`, i.e. `type(L(i))` $\in \{1, \ldots, n\}$.

```
**SetSemTD**$_M$**(List L)**
$A_1, \ldots, A_n$:  Set := $\emptyset$
FOR i := 1 TO size(L) DO {
   SWITCH type(L(i)) {
     CASE 1:   $A_1$ = $A_1 \cup ([\![\texttt{L(i)}]\!] \setminus \bigcup_{j \in \{2..n\}} A_j)$
     ...
     CASE n:   $A_n$ = $A_n \cup ([\![\texttt{L(i)}]\!] \setminus \bigcup_{j \in \{1..n-1\}} A_j)$
   }
   i := i+1
}
RETURN $(A_1, \ldots, A_n)$
```

```
**SetSemBU**$_M$**(List L)**
$A_1, \ldots, A_n$:  Set := $\emptyset$
FOR i := size(L) DOWNTO 1 DO {
   j = type(L(i))
   $A_j$ = $A_j \cup [\![\texttt{L(i)}]\!]$
   FOR ALL $A_k$,  $k \neq j$ DO
     $A_k$ = $A_k \setminus [\![\texttt{L(i)}]\!]$
   i := i-1
}
RETURN $(A_1, \ldots, A_n)$
```

```
**SetSemREC**$_M$**(List L)**
$A_1, \ldots, A_n$:  Set := $\emptyset$
$B_1, \ldots, B_n$:  Set := $\emptyset$
IF size(L) = 0 RETURN $(A_1, \ldots, A_n)$
ELSE {
   $(B_1, \ldots, B_n)$ = SetSemREC$_M$(tail(L))
   j = type(L(i))
   $A_j = B_j \cup [\![\texttt{L(i)}]\!]$
   FOR ALL $A_k$,  $k \neq j$ DO
     $A_k = B_k \setminus [\![\texttt{L(i)}]\!]$
   RETURN $(A_1, \ldots, A_n)$
}
```

Note that the algorithms can be easily adapted to figure out just a choice of the sets A_i.

Adapting the Analysis We set $[\![\texttt{L}]\!] =_{def} ([\![\texttt{L}]\!]_1, \ldots, [\![\texttt{L}]\!]_n) =_{def} \texttt{SetSemTD}_M(\texttt{L}) = \texttt{SetSemBU}_M(\texttt{L}) = \texttt{SetSemREC}_M(\texttt{L})$. Now more specific questions are necessary because the action of interest has to be specified.

Analytical Task	**Semantics**
is action i taken for packet p?	$p \in [\![\texttt{L}]\!]_i$
are $\texttt{L}_1$ and $\texttt{L}_2$ equivalent w.r.t. to action i?	$[\![\texttt{L}_1]\!]_i = [\![\texttt{L}_2]\!]_i$
is for all packets for which action i is taken by $\texttt{L}_1$ also action i taken by $\texttt{L}_2$?	$[\![\texttt{L}_1]\!]_i \subseteq [\![\texttt{L}_2]\!]_i$
for which packets is action i taken by at least $\texttt{L}_1$ or $\texttt{L}_2$?	$[\![\texttt{L}_1]\!]_i \cup [\![\texttt{L}_2]\!]_i$
for which packets is action i taken by both $\texttt{L}_1$ and $\texttt{L}_2$?	$[\![\texttt{L}_1]\!]_i \cap [\![\texttt{L}_2]\!]_i$
for which packets is action i taken by $\texttt{L}_1$ but not by $\texttt{L}_2$?	$[\![\texttt{L}_1]\!]_i \setminus [\![\texttt{L}_2]\!]_i$

All these operations only take care about one specific action. That is why none of them results in a new partition of IP, i.e. a rule-list. If we want to gain new lists out of given ones we have to operate on them as a whole, i.e. taking care about all actions at once. Just performing the above operations separately on every single action will not succeed, e.g. a pairwise uniting $[\![\texttt{L}_1]\!]_i \cup [\![\texttt{L}_2]\!]_i$ for all i may lead to non-disjoint sets. Pairwise intersecting or subtracting may lead to sets which do not cover IP anymore. In other words these operations do not preserve partitions. More complex operators have to be defined. The union should map an action i to a packet if one of the united lists does so. But because several lists generally perform several actions on a packet it is unclear which action should be taken. One possible solution is to establish priorities on the actions and then unite the corresponding sets $[\![\texttt{L}]\!]_i$ with respect to these priorities. That means that always the action with the highest priority will be chosen for the resulting rule-list. This is well defined with the notion of the *ordered union* in appendix A and written here $[\![\texttt{L}_1]\!] \cup_< [\![\texttt{L}_2]\!]$[4].

As said before pairwise intersecting and subtracting is also not sufficient for a well-defined definition. To make the function $filter_M$ total a default action i has to be defined which is taken in all unclear cases. Priorities are not necessary here. We write such an intersection and subtraction with default action as $[\![\texttt{L}_1]\!] \cap_i [\![\texttt{L}_2]\!]$, respectively $[\![\texttt{L}_1]\!] \setminus_i [\![\texttt{L}_2]\!]$ and get the following operations regarding lists seen as a whole:

Analytical Task	**Semantics**
are lists $\texttt{L}_1$ and $\texttt{L}_2$ equivalent?	$[\![\texttt{L}_1]\!] = [\![\texttt{L}_2]\!]$
ordered union	$[\![\texttt{L}_1]\!] \cup_< [\![\texttt{L}_2]\!]$
intersection with default action i	$[\![\texttt{L}_1]\!] \cap_i [\![\texttt{L}_2]\!]$
subtracting with default action i	$[\![\texttt{L}_1]\!] \setminus_i [\![\texttt{L}_2]\!]$

[4]Note that the union of the *Permit*-sets in the case of just two possible actions *Permit* and *Deny* as introduced in subsection 2.3.2 is a special case of the ordered union with *Deny* < *Permit*.

Redundant rules

Given a rule `L(i)` with $\texttt{type}(\texttt{L(i)}) = j$. We call `L(i)` *redundant*, written $\mathfrak{R}(\texttt{L(i)})$, iff $[\![\texttt{L}]\!]_j = [\![\texttt{L} \setminus \texttt{L(i)}]\!]_j$ where $\texttt{L} \setminus \texttt{L(i)}$ is `L` without rule `L(i)`. Three cases can occur. It is possible that rule `L(i)` describes a subset preceding rules $P \subseteq \{\texttt{L(1)}, \dots, \texttt{L(i}-\texttt{1)}\}$. Or it describes a subset of succeeding rules $S \subseteq \{\texttt{L(i}+\texttt{1)}, \dots, \texttt{L(size(L))}\}$ having same action j where additionally holds that all rules in between `L(i)` and the last of S, written $\texttt{L(last(}S\texttt{))}$[5], are not *cut down* by `L(i)`. Thirdly a combination of both is possible. First we define what it means that a rule `L(m)` is cut down by `L(i)`, written $\texttt{L(i)} \triangleright \texttt{L(m)}$:

$$\begin{aligned} \texttt{L(i)} \triangleright \texttt{L(m)} \quad \Leftrightarrow_{def} \quad & i < m \wedge \texttt{type}(\texttt{L(i)}) \neq \texttt{type}(\texttt{L(m)}) \wedge \\ & [\![\texttt{L(i)}]\!] \cap [\![\texttt{L(m)}]\!] \not\subseteq \bigcup_{k \in]i..m[} [\![\texttt{L(k)}]\!] \end{aligned}$$

Now we can define the redundancy formally as:

$$\begin{aligned} \mathfrak{R}(\texttt{L(i)}) \quad \Leftrightarrow \quad & \exists S \subseteq \{\texttt{L(i}+\texttt{1)}, \dots, \texttt{L(size(L))}\} \text{ with } \forall R \in S : \text{type}(R) = j \text{ and} \\ & \forall \texttt{L(k)} \text{ with } k \in]i..\texttt{last}(S)[\text{ holds } \texttt{L(i)} \not\triangleright \texttt{L(k)} \text{ such that} \\ & [\![\texttt{L(i)}]\!] \subseteq \bigcup_{k \in [1..i[} [\![\texttt{L(k)}]\!] \cup \bigcup_{R \in S} [\![R]\!] \end{aligned}$$

Analysis — summary

We have seen that analysis with multiple actions is less straightforward than in the two-action case. More complex operations have to be applied because it is difficult to analyse rule-lists seen as a whole. Of course all that can not be implemented directly because the involved sets are too big for being taken as a data structure. But as announced before the basic ideas and algorithms will reappear in all approaches. That is why the foundation on sets serves as a guideline to develop structures and algorithms which allow performing analysis with a feasible complexity.

2.4 Approaching uncorrelated and canonical structures

As introduced in chapter 1 analysis with rule-lists is difficult for two main reasons:

1. order of the overlapping rules matters
2. syntactically different rule-lists can have the same meaning, i.e. semantics

Because of the first reason and with respect to [21] we call rule-lists *correlated* structures. That is a data structure which consists of interdependent parts. The `SetSem`-algorithms extract the meaning of a rule-list into a set, obviously an *uncorrelated* structure because

[5] $\texttt{i} = \texttt{last}(S) \Leftrightarrow \texttt{L(i)} \in S \wedge \forall \texttt{L(j)} \in S \setminus \{L(i)\} : j < i$

elements of sets are independent entities. Analysis based on sets showed up in an easy and straightforward manner because of this property. What is needed to implement the analytical operations are uncorrelated data structures which can represent such a set efficiently.

Sets have another useful property which helps analysing, they are unique objects. No two sets with the same elements can be different. That means for our data structures that in opposite to rule-lists syntactical difference must lead to semantical difference. Structures having the latter property are called *canonical* structures. One advantage of having such structures is that checking semantical equality can be done simply by comparing the syntax.

Imagine for example that a packet filter only allows to define non-overlapping *Permit*-rules. It accepts all packets which are permitted by these rules and implicitly denies all other. Such rule-lists are no real lists anymore because the order of non-overlapping rules does not matter. Therefore we have gained an *uncorrelated* structure, but because syntactical different non-overlapping lists can still have the same semantics the structure is not canonical. Summarised we are searching for data structures with the following properties:

- implementable with a feasible complexity
- uncorrelated, therefore no procedural semantics
- canonical
- allow performing the analytical operations
- allow back-translation to rule-lists

A general introduction into two main approaches is given in the following. We will take again the case of two possible actions *permit* and *deny* as a starting point and generalise this later.

2.4.1 A geometrical approach

In chapter 1 we have interpreted the `Source IP` and `Dest.IP` Domains as axes spanning a plane. As a result every rule which only cares about these two attributes could be seen as a rectangle. If we deal with more attributes we have to generalise this idea to rectangles with more than two dimensions. Such geometrical objects are called *hyperrectangles.* They are defined by the product of axis-intervals. Strictly speaking we deal in the two-dimensional space with points, horizontal and vertical line segments and rectangles whose sides are axis-parallel. Such objects are also called *orthogonal objects* or *iso-oriented objects* (see [19], VIII, 5.). If we deal with d dimensions we can define a hyperrectangle R as follows:

$$R = \prod_{j=1}^{d} [l_j,\ r_j] \text{ where } l_j \leq r_j\ \forall j$$

Now every rule `L(i)` in a rule-list corresponds to such a hyperrectangle in a d-dimensional space (where d is the number of attributes used by the list) connected with a *permit* or *deny* action, denoted $\langle$`L(i)`$\rangle$. A whole rule-list represents therefore a list of such hyperrectangles and a packet is a d-dimensional point.

Applying this to the example given in chapter 1, table 2.2, we get a list of three hyperrectangles A, B and C. Because the rules deal with two attributes we have two axes, `Source IP`:[0.0.0.0 - 255.255.255.255] and `Dest.IP`: [0.0.0.0 - 255.255.255.255]. Together we get:

Rule	**Source IP**	×	**Dest.IP**	**Action**
A	[10.0.0.0, 10.255.255.255]	×	[172.16.6.0, 172.16.6.255]	Permit
B	[10.1.99.0, 10.1.99.255]	×	[172.16.0.0, 172.16.255.255]	Deny
C	[0.0.0.0, 255.255.255.255]	×	[0.0.0.0, 255.255.255.255]	Deny

Let p be the packet with `Source IP` = 10.1.99.42 and `Dest.IP` = 172.16.1.2. Interpreted as a two-dimensional point we write $\langle p\rangle = (10.1.99.42,\ 172.16.1.2)$. To decide if this packet is accepted or denied we go again top-down through the list and check if the corresponding point lies within the actual rectangle. Because $(10.1.99.42,\ 172.16.1.2) \in [10.1.99.0, 10.1.99.255] \times [172.16.0.0, 172.16.255.255]$ rule B matches and the packet is denied.

Of course the unwanted properties are still present, the structure is correlated and not canonical. The geometrical interpretation would only be useful if we manage now to get rid of these properties by geometrical transformations.

Splitting and normalising hyperrectangles

Let us first concentrate on the "overlap problem". We have solved this in the set scenario by applying one of the `SetSem`-algorithms. They give us now a good pattern for a geometrical solution. If we take again a look at the `SetSemBU` variant we see two main operations, the union ($\cup$) and the subtraction ($\setminus$) of sets. If we denote the geometrical counterparts with $\oplus$ and $\ominus$ the algorithm looks like this:

```
GeoInterBU(List L)
Permit: Set of Hyperrectangles := ∅
FOR i := size(L) DOWNTO 1 DO {
   IF type(L(i)) = permit DO
     Permit = Permit ⊕ ⟨L(i)⟩
   IF type(L(i)) = deny DO
     Permit = Permit ⊖ ⟨L(i)⟩
   i := i-1
}
RETURN Permit
```

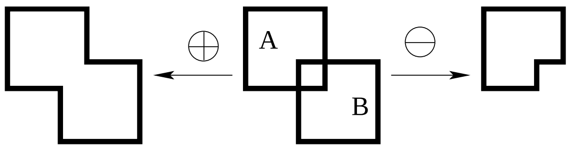


Figure 2.3: Naive Applying of $A \oplus B$ and $A \ominus B$

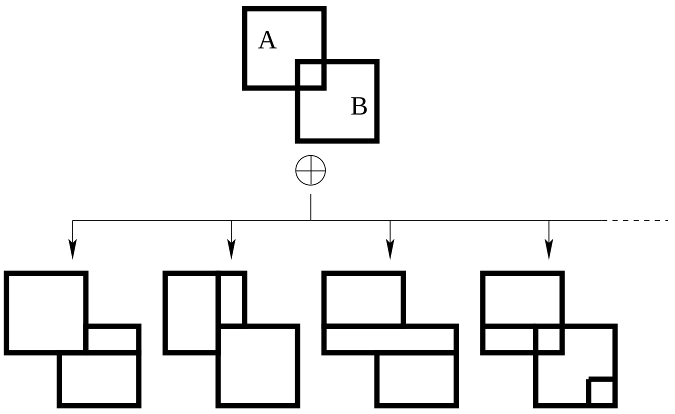


Figure 2.4: Some Possible Results for $A \oplus B$

Now it has to be clarified what exactly the operations $\oplus$ and $\ominus$ should do. We want to gain a set *Permit* of hyperrectangles which include exactly the points, i.e. packets, of $[\![\mathtt{L}]\!]$. In the $\oplus$-case a union should be undertaken. The $\ominus$ operator has to subtract a given hyperrectangle from a set of given ones. Figure 2.3 shows a simple two-dimensional example with two rectangles. It can be seen that uniting and intersecting hyperrectangles does generally not lead to hyperrectangles anymore[6]. This is problematic for two main reasons:

- efficient data structures and algorithms for hyperrectangles can not be applied
- back-translation to rule-lists is not straightforward

That is why we want the operators $\oplus$ and $\ominus$ to result again in a set of hyperrectangles. But we have to make sure that they do not overlap to gain an uncorrelated structure. Generally there are many ways to do so. Figure 2.4 shows some possible results. Of course there are a lot more candidates for a solution which may differ in the amount of resulting rectangles. Figure 2.5 gives examples for the $\ominus$ operator. We will study concrete implementations in chapter 4. For now we assume having efficient algorithms for such $\oplus$ and $\ominus$ operators which are suited for any dimensions. Now it is possible to use the `GeoInterBU`-algorithm

[6]In the $\oplus$-case the resulting object is called the *contour* of the union. Efficient algorithms exist which solve the so called *contour problem*. See [19], 5.2.2.

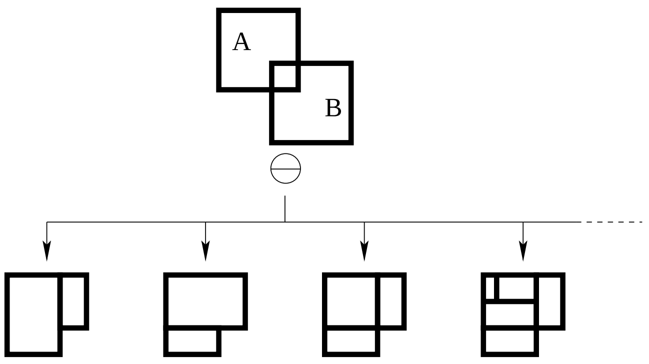


Figure 2.5: Some Possible Results for $A \ominus B$

to transform a rule-list into a set *Permit* of non-intersecting hyperrectangles which exactly include the $[\![\mathtt{L}]\!]$ packets.

We have decorrelated the hyperrectangles and therefore solved the "overlap problem". But what shall we do when we want to compare two such *Permit*-sets? Imagine a list $\mathtt{L}_1$ with two *Permit*-rules A, B and a list $\mathtt{L}_2$ with three *Permit*-rules A, B, C. Assume further that both lists are semantically equal, i.e. $[\![\mathtt{L}_1]\!] = [\![\mathtt{L}_2]\!]$. The problem is that we can not be sure that applying the `GeoInterBU`-algorithm will result in two (syntactically) equal *Permit*-sets. In other words our structure is not canonical. A second transformation has to be done which *normalises* the resulting *Permit*-sets in a canonical way to make semantically equal sets also syntactically equal. Let us denote the canonical normalisation-operator with $\bigtriangledown$. This is exemplified in figure 2.6. Again there are many candidates for such a *normalform*. One canonical has to be fixed and computed. If we call the normalisation algorithm `GeoNorm` we have to compute `GeoNorm(GeoInterBU(List L))` to get a decorrelated and canonical set of hyperrectangles. We denote this set by $\langle \mathtt{L} \rangle$. In chapter 4 we will see that is possible to do both decorrelation and canonical normalisation in a single transformation. Summarised we have:

Object	**Interpretation**
packet	d-dimensional point
rule	hyperrectangle
rule-list	set of normalised, non-intersecting hyperrectangles

Performing analysis

We have already seen how it can be checked if a packet p matches a hyperrectangles R, written $\langle p \rangle \in R$. To see if p is accepted by a list `L` we just have to check if the packet matches one of the rectangles in $\langle \mathtt{L} \rangle$:

$$\langle p \rangle \in_g \langle \mathtt{L} \rangle \Leftrightarrow_{def} \exists R \in \langle \mathtt{L} \rangle : \langle p \rangle \in R$$

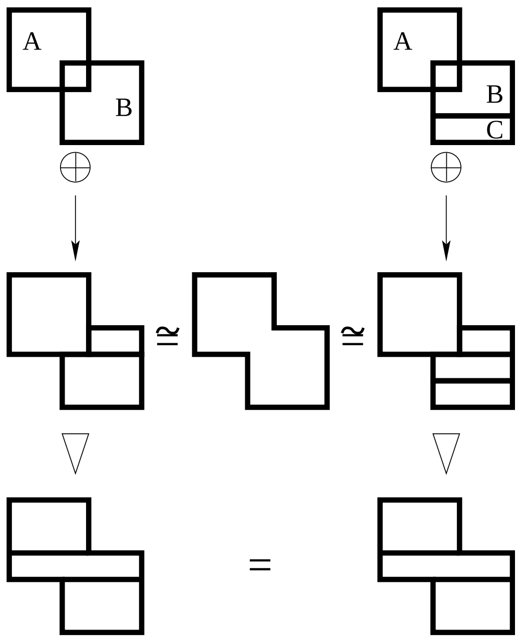


Figure 2.6: Normalisation of Hyperrectangles

Because ⟨L⟩ is uncorrelated we can check this now parallel for every R and do not have to take care of any order. Canonical normalisation gives us additionally the comfort of checking semantical equality of lists simply by checking equality on sets of hyperrectangles. Let us study the other operations.

∘ **Union** (∪) We want to gain a set of non-intersecting hyperrectangles which include all the packets from $[\![\mathtt{L}_1]\!] \cup [\![\mathtt{L}_2]\!]$. Because we have already an implementation of the $\oplus$ operator we can adjust `GeoInterBU` to do so. It is called by `GeoUnion(`$\langle \mathtt{L}_1 \rangle$`, `$\langle \mathtt{L}_2 \rangle$`)`.

GeoUnion(Set of Hyperrectangles H_1, H_2)

```
$Permit$:  Set of Hyperrectangles := $H_1$
FOR ALL $R \in H_2$ DO
    $Permit$ = $Permit \oplus R$
RETURN $Permit$
```

Additionally applying `GeoNorm` yields the wanted set, written

$$\langle \mathtt{L}_1 \rangle \cup_g \langle \mathtt{L}_2 \rangle =_{def} \mathtt{GeoNorm}(\mathtt{GeoUnion}(\langle \mathtt{L}_1 \rangle, \langle \mathtt{L}_2 \rangle))$$

∘ **Subset** (⊆) Obviously $[\![\mathtt{L}_1]\!] \subseteq [\![\mathtt{L}_2]\!]$ holds geometrically iff the hyperrectangles in $\langle \mathtt{L}_2 \rangle$ cover all the points from $\langle \mathtt{L}_1 \rangle$. Figure 2.7 shows an example of a rectangle $Q \in \langle \mathtt{L}_1 \rangle$ which

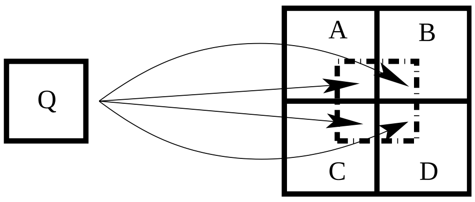


Figure 2.7: Rectangle $Q \in \langle \mathtt{L}_1 \rangle$ covered by $\{A, B, C, D\} \in 2^{\langle \mathtt{L}_2 \rangle}$

is covered by four rectangles $\{A, B, C, D\} \in 2^{\langle \mathtt{L}_2 \rangle}$. Implementing the subset-check like this would be very inefficient because for every $Q \in \langle \mathtt{L}_1 \rangle$ the corresponding set of covering hyperrectangles has to be computed. A very fast subset check would be possible if the normalisation-algorithm `GeoNorm` would split $\langle \mathtt{L}_1 \rangle$ and $\langle \mathtt{L}_2 \rangle$ in a way that $[\![\mathtt{L}_1]\!] \subseteq [\![\mathtt{L}_2]\!] \Leftrightarrow \forall Q \in \langle \mathtt{L}_1 \rangle\ \exists R \in \langle \mathtt{L}_2 \rangle : Q = R$. But to do so is rather inefficient too because `GeoNorm` would need to know both sets $\langle \mathtt{L}_1 \rangle$, $\langle \mathtt{L}_2 \rangle$ and normalise them simultaneously. Furthermore the amount of resulting rectangles would grow tremendously. Unfortunately the same is true for a `GeoNorm` implementation where $[\![\mathtt{L}_1]\!] \subseteq [\![\mathtt{L}_2]\!] \Leftrightarrow \forall Q \in \langle \mathtt{L}_1 \rangle\ \exists R \in \langle \mathtt{L}_2 \rangle : Q \subseteq R$ holds. We take a closer look on normalising hyperrectangles in chapter 4. A different way must be found for the subset check. Fortunately we can ask again set-theory for help. We know that for two sets A and B

$$A \subseteq B \Leftrightarrow A \cup B = B$$

holds. We have to check if the geometrical counterpart

$$[\![\mathtt{L}_1]\!] \subseteq [\![\mathtt{L}_2]\!] \Leftrightarrow \langle \mathtt{L}_1 \rangle \cup_g \langle \mathtt{L}_2 \rangle = \langle \mathtt{L}_2 \rangle$$

holds too. It does because we deal with a canonical structure, otherwise the equality $\langle \mathtt{L}_1 \rangle \cup_g \langle \mathtt{L}_2 \rangle = \langle \mathtt{L}_2 \rangle$ would not be guaranteed. Figure 2.8 demonstrates this. Rectangle $Q \in \langle \mathtt{L}_1 \rangle$ is covered by rectangles A and $B \in \langle \mathtt{L}_2 \rangle$. The geometrical union ($\cup_g$) first unites the rectangles A, B and Q to a new set of non-intersecting rectangles via $\mathtt{GeoInter}_{\cup}$. But the normalisation via `GeoNorm` finally produces a result which is equal to the original rectangles A and B. So it can be defined:

$$\langle \mathtt{L}_1 \rangle \subseteq_g \langle \mathtt{L}_2 \rangle \Leftrightarrow_{def} \langle \mathtt{L}_1 \rangle \cup_g \langle \mathtt{L}_2 \rangle = \langle \mathtt{L}_2 \rangle$$

∘ **Subtraction** (\\) We want to gain hyperrectangles which represent the set $[\![\mathtt{L}_1]\!] \setminus [\![\mathtt{L}_2]\!]$, i.e. a set of all packets which are permitted by $\mathtt{L}_1$ but not by $\mathtt{L}_2$. Similar to the union-case we can use the implementation of the $\ominus$ operator and adjust the `GeoInterBU`-algorithm:

```
GeoMinus(Set of Hyperrectangles $H_1$, $H_2$)
$Permit$:  Set of Hyperrectangles := $H_1$
FOR ALL $R \in H_2$ DO
```

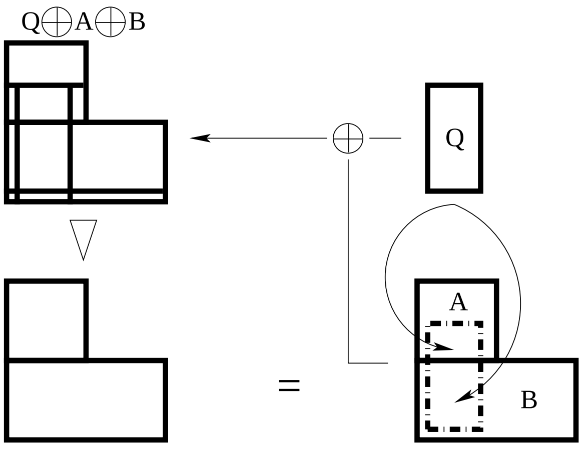


Figure 2.8: Subset-Check $\{Q\} \subseteq_g \{A, B\}$

```
    $Permit$ = $Permit \ominus R$
RETURN $Permit$
```

Additionally applying `GeoNorm` yields again the wanted set, written

$$\langle \mathrm{L}_1 \rangle \setminus_g \langle \mathrm{L}_2 \rangle =_{def} \texttt{GeoNorm}(\texttt{GeoMinus}(\langle \mathrm{L}_1 \rangle, \langle \mathrm{L}_2 \rangle))$$

∘ **Intersection (∩)** Set-theory helps again to gain all packets which are in $[\![\mathrm{L}_1]\!] \cap [\![\mathrm{L}_2]\!]$ because we know that for two sets A and B

$$A \cap B = A \setminus (A \setminus B)$$

holds. We use our already defined $\setminus_g$ operator to define here:

$$\langle \mathrm{L}_1 \rangle \cap_g \langle \mathrm{L}_2 \rangle =_{def} \langle \mathrm{L}_1 \rangle \setminus_g (\langle \mathrm{L}_1 \rangle \setminus_g \langle \mathrm{L}_2 \rangle)$$

Now we have developed geometrical counterparts which implement all the basic operations as introduced in subsection 2.3.2:

Semantical Task	**Interpretation**
$p \in [\![\mathrm{L}]\!]$	$\langle p \rangle \in_g \langle \mathrm{L} \rangle$
$[\![\mathrm{L}_1]\!] = [\![\mathrm{L}_2]\!]$	$\langle \mathrm{L}_1 \rangle = \langle \mathrm{L}_2 \rangle$
$[\![\mathrm{L}_1]\!] \subseteq [\![\mathrm{L}_2]\!]$	$\langle \mathrm{L}_1 \rangle \subseteq_g \langle \mathrm{L}_2 \rangle$
$[\![\mathrm{L}_1]\!] \cup [\![\mathrm{L}_2]\!]$	$\langle \mathrm{L}_1 \rangle \cup_g \langle \mathrm{L}_2 \rangle$
$[\![\mathrm{L}_1]\!] \cap [\![\mathrm{L}_2]\!]$	$\langle \mathrm{L}_1 \rangle \cap_g \langle \mathrm{L}_2 \rangle$
$[\![\mathrm{L}_1]\!] \setminus [\![\mathrm{L}_2]\!]$	$\langle \mathrm{L}_1 \rangle \setminus_g \langle \mathrm{L}_2 \rangle$

Generalisation to multiple actions

We have clarified what the $\oplus$ and $\ominus$ operators have to do. Based on this we have defined the geometrical interpretations of the basic set operations. Using this we can easily adopt everything we said about multiple actions in subsection 2.3.2. Taking again the `SetSemBU`$_M$ variant as a pattern we get:

GeoInterBU$_M$(List L)

```
$A_1, \ldots, A_n$:  Set of Hyperrectangles := $\emptyset$
FOR i := size(L) DOWNTO 1 DO {
   j = type(L(i))
   $A_j$ = $A_j \oplus \langle$L(i)$\rangle$
   FOR ALL $A_k$, $k \neq j$ DO
     $A_k$ = $A_k \ominus \langle$L(i)$\rangle$
   i := i-1
}
RETURN $(A_1, \ldots, A_n)$
```

Exchanging now the set operations with their geometrical counterparts yields the interpretations for all analytical tasks. For example answering *for which packets is action i taken by both* $\mathtt{L}_1$ *and* $\mathtt{L}_2$*?* corresponds now to $\langle \mathtt{L}_1 \rangle_i \cap_g \langle \mathtt{L}_2 \rangle_i$. The definition of the ordered union can be directly translated the same way and to add a default action i we simply define a hyperrectangle R which spans the whole set IP and perform $\langle \mathtt{L} \rangle_i = \langle \mathtt{L} \rangle_i \cup_g (\{R\} \setminus_g \bigcup_{j=1}^{n} \langle \mathtt{L} \rangle_j)$.

Translating back to rule-lists

We are only dealing with sets hyperrectangles. It depends on the concrete expressive power of a packet filter formalism if a single hyperrectangles fits to a single rule or if it has to be split into several parts. Nevertheless it is straightforward to transform such a set back to a rule-list. Because the hyperrectangles only represent *Permit*-sets the order of the rules can be chosen arbitrarily.

Analysis without normalisation

We could have also used the subtraction-operation $\setminus_g$ to check if $[\![\mathtt{L}_1]\!] \subseteq [\![\mathtt{L}_2]\!]$ holds. We know that for sets we have $A \subseteq B \Leftrightarrow A \setminus B = \emptyset$. Therefore we can define:

$$\langle \mathtt{L}_1 \rangle \subseteq_g \langle \mathtt{L}_2 \rangle \Leftrightarrow_{def} \langle \mathtt{L}_1 \rangle \setminus_g \langle \mathtt{L}_2 \rangle = \emptyset$$

Based on this implementation we can also check if $[\![\mathtt{L}_1]\!] = [\![\mathtt{L}_2]\!]$ holds:

$$\langle \mathtt{L}_1 \rangle = \langle \mathtt{L}_2 \rangle \Leftrightarrow_{def} \langle \mathtt{L}_1 \rangle \subseteq_g \langle \mathtt{L}_2 \rangle \wedge \langle \mathtt{L}_2 \rangle \subseteq_g \langle \mathtt{L}_1 \rangle$$

Note that we have redefined the two operations which needed canonical normalisation to work properly. Furthermore we can allow now that hyperrectangles overlap, i.e. $\oplus = \cup$.

Note that this still leads to an uncorrelated structure because we are only dealing with overlapping hyperrectangles representing *Permit*-sets, their order does not matter. But normalisation may be useful when dealing with more advanced tasks. It depends on concrete implementation issues if normalisation is useful respectively necessary or not.

Geometrical approach — summary

An uncorrelated and canonical structure has been found – sets of non-intersecting hyperrectangles. Three basic algorithms are necessary: the addition of a hyperrectangle $\oplus$, the subtraction $\ominus$ and the canonical normalisation of a set of hyperrectangles via `GeoNorm`. Based on them all analytical tasks can be carried out. We will study in chapter 4 how these algorithms can be implemented.

2.4.2 A logical approach

Scott Hazelhurst [13], [12], [3] has developed an approach based on boolean logic. We give here just a short introduction to the basic ideas.

A rule-list was introduced in chapter 1 as a list of constraints about the domains of the used attributes. Using `SIP` and `DIP` as abbreviations for `Source IP` and `Dest.IP`, rule A of example 2.2 can be seen as the constraint $\mathfrak{C}(A) =$

$$((\texttt{SIP} \geq 10.0.0.0) \wedge (\texttt{SIP} \leq 10.255.255.255) \wedge (\texttt{DIP} \geq 172.16.6.0) \wedge (\texttt{DIP} \leq 172.16.6.255))$$

This constraint describes a condition a packet has to match to be permitted by rule A. Going top-down through the list a *Permit*- or *Deny*-rule can occur. In the first case a packet is accepted iff it matches the rule or if it is accepted by the rest of the list. In the second case the packet is accepted iff it not matches the rule and is accepted by the rest of the list. Therefore we can see the whole example as the proposition:

$$\mathfrak{C}(A) \vee (\neg\mathfrak{C}(B) \wedge \neg\mathfrak{C}(C))$$

Because $\mathfrak{C}(C)$ is a default-deny rule it matches every packet and is therefore logically a tautology, written $\top$. The negation of a tautology is a contradiction, written $\bot$. The proposition can be simplified:

$$\begin{aligned}
\mathfrak{C}(A) \vee (\neg\mathfrak{C}(B) \wedge \neg\mathfrak{C}(C)) &\equiv \mathfrak{C}(A) \vee (\neg\mathfrak{C}(B) \wedge \neg\top) \\
&\equiv \mathfrak{C}(A) \vee (\neg\mathfrak{C}(B) \wedge \bot) \\
&\equiv \mathfrak{C}(A) \vee \bot \\
&\equiv \mathfrak{C}(A)
\end{aligned}$$

If we apply the rules in order B, A, C we gain this proposition:

$$\neg\mathfrak{C}(B) \wedge (\mathfrak{C}(A) \vee \neg\mathfrak{C}(C)) \equiv \neg\mathfrak{C}(B) \wedge (\mathfrak{C}(A) \vee \neg\top) \equiv \neg\mathfrak{C}(B) \wedge \mathfrak{C}(A)$$

Obviously these propositions are no real propositional formulas because instead of boolean expressions we have constraints representing a rule. But fortunately these constraints can be easily transformed into boolean expression.

Transforming rule-lists into boolean expressions

The basic idea is to code the binary representation of a number as a boolean formula. For example the decimal number 172_{10} is binary 10101100_2. $\lceil \log_2 172 \rceil = 8$ bits are needed, therefore eight conjuncted boolean variables $\{x_7, \ldots, x_0\}$ can represent this number. The corresponding formula is denoted as $\lbag 172_{10} \rbag$:

$$\lbag 172_{10} \rbag = \lbag 10101100_2 \rbag \quad = \quad x_7 \wedge \neg x_6 \wedge x_5 \wedge \neg x_4 \wedge x_3 \wedge x_2 \wedge \neg x_1 \wedge \neg x_0$$

In IPv4 IP-addresses consist of four segments with eight bit each. 32 variables are needed here to encode each of the `Source IP` and `Dest.IP` attributes. The same way we can encode the `Source Port` and `Dest.Port` with 16 variables each. Allowing three different protocol types `TCP, UDP, ICMP` two variables are needed and so on. Concentrating again on the example 2.2 we deal with two attributes, `Source IP` and `Dest.IP`. 64 variables are needed here: $\{s_{31}, \ldots, s_0, d_{31}, \ldots, d_0\}$. To encode a packet p_1 with `Source IP` $= 10.1.99.42$ and `Dest.IP` $= 172.16.1.2$ we conjunct the formulas representing the single attributes:

$$\lbag p_1 \rbag = \lbag 10.1.99.42_{IP} \rbag \wedge \lbag 172.16.1.2_{IP} \rbag$$

The whole formula $\lbag p_1 \rbag$ now looks like this:

$$\begin{array}{rcl}
\lbag 10_{10} \rbag = \lbag 00001010_2 \rbag & \Rrightarrow & \neg s_{31} \wedge \neg s_{30} \wedge \neg s_{29} \wedge \neg s_{28} \wedge s_{27} \wedge \neg s_{26} \wedge s_{25} \wedge \neg s_{24} \wedge \\
\lbag 1_{10} \rbag = \lbag 00000001_2 \rbag & \Rrightarrow & \neg s_{23} \wedge \neg s_{22} \wedge \neg s_{21} \wedge \neg s_{20} \wedge \neg s_{19} \wedge \neg s_{18} \wedge \neg s_{17} \wedge s_{16} \wedge \\
\lbag 99_{10} \rbag = \lbag 01100011_2 \rbag & \Rrightarrow & \neg s_{15} \wedge s_{14} \wedge s_{13} \wedge \neg s_{12} \wedge \neg s_{11} \wedge \neg s_{10} \wedge s_9 \wedge s_8 \wedge \\
\lbag 42_{10} \rbag = \lbag 00101010_2 \rbag & \Rrightarrow & \neg s_7 \wedge \neg s_6 \wedge s_5 \wedge \neg s_4 \wedge s_3 \wedge \neg s_2 \wedge s_1 \wedge \neg s_0 \wedge \\
\lbag 172_{10} \rbag = \lbag 10101100_2 \rbag & \Rrightarrow & d_{31} \wedge \neg d_{30} \wedge d_{29} \wedge \neg d_{28} \wedge d_{27} \wedge d_{26} \wedge \neg d_{25} \wedge \neg d_{24} \wedge \\
\lbag 16_{10} \rbag = \lbag 00010000_2 \rbag & \Rrightarrow & \neg d_{23} \wedge \neg d_{22} \wedge \neg d_{21} \wedge d_{20} \wedge \neg d_{19} \wedge \neg d_{18} \wedge \neg d_{17} \wedge \neg d_{16} \wedge \\
\lbag 1_{10} \rbag = \lbag 00000001_2 \rbag & \Rrightarrow & \neg d_{15} \wedge \neg d_{14} \wedge \neg d_{13} \wedge \neg d_{12} \wedge \neg d_{11} \wedge \neg d_{10} \wedge \neg d_9 \wedge d_8 \wedge \\
\lbag 2_{10} \rbag = \lbag 00000010_2 \rbag & \Rrightarrow & \neg d_7 \wedge \neg d_6 \wedge \neg d_5 \wedge \neg d_4 \wedge \neg d_3 \wedge \neg d_2 \wedge d_1 \wedge \neg d_0
\end{array}$$

The next step is to encode a single rule as a formula. To do so it has to be clarified how constraints like ($\texttt{SIP} \geq 10.0.0.0) \wedge (\texttt{SIP} \leq 10.255.255.255$) can be converted. It says that every packet with 10 as first segment of its `Source IP` matches, the other segments play no role. To check this only the variables $\{s_{31}, \ldots, s_{24}\}$ which represent the first segment have to match. That is why we omit all other variables and convert this constraint into a formula which only describes this first segment:

$$\lbag (\texttt{SIP} \geq 10.0.0.0) \wedge (\texttt{SIP} \leq 10.255.255.255) \rbag = \neg s_{31} \wedge \neg s_{30} \wedge \neg s_{29} \wedge \neg s_{28} \wedge s_{27} \wedge \neg s_{26} \wedge s_{25} \wedge \neg s_{24}$$

The same way all constraint can be converted. Again we conjunct them to gain a formula representing a whole rule, e.g. $\lbag A \rbag =$

$$\lbag (\texttt{SIP} \geq 10.0.0.0) \wedge (\texttt{SIP} \leq 10.255.255.255) \rbag \wedge \lbag (\texttt{DIP} \geq 172.16.6.0) \wedge (\texttt{DIP} \leq 172.16.6.255) \rbag$$

A packet p matches a rule `L(i)` now iff $\wr\texttt{L(i)}\int$ derives from $\wr p \int$, written $\wr p \int \vdash \wr\texttt{L(i)}\int$. Obviously $\wr p_1 \int \nvdash \wr A \int$ and $\wr p_1 \int \vdash \wr B \int$ holds.

Now that a whole rule can be transformed into a boolean formula we can use our initial idea to transform a whole list. The `SetSemREC`-algorithm implements exactly that idea but we use again the easier non-recursive `SetSemBU`-algorithm to demonstrate the conversion. Boolean algebra gives directly the logical connectives for the set operations, the union becomes a disjunction and because $A \setminus B = A \cap \overline{B}$ we get:

```
LogicInterBU(List L)
$Permit$:  Boolean Formula := $\bot$
FOR i := size(L) DOWNTO 1 DO {
  IF type(L(i)) = permit DO
    $Permit = \underline{(}Permit \lor \wr\texttt{L(i)}\int\underline{)}$
  IF type(L(i)) = deny DO
    $Permit$ = $\underline{(}Permit \land \neg\wr\texttt{L(i)}\int\underline{)}$
  i := i-1
}
RETURN $Permit$
```

Note that the underlined brackets are part of the syntax of the resulting formula.

Normalising boolean formulas

We have managed to transform a whole rule-list into a boolean formula. But because syntactically different formulas may be semantically equivalent ($\equiv$) we also need to normalise them in a canonical way. There are some known normalforms, e.g. *conjunctive normal forms* (see e.g. [14]). But they have two properties which make them useless for our needs. First they are no canonical normalforms, e.g. $(p \land \neg p) \lor (q \land \neg q) \equiv (p \land \neg p)$. Moreover transforming a formula into its conjunctive normal form can blow up the result exponentially. Fortunately there exist a data structure which represents boolean formulas in a canonical and usually compact manner: *Binary Decision Diagrams* (BDDs), originally developed by Bryant (see [6]). For now it is enough to know that there exists a canonical representation which additionally allows to apply all basic logical operations (which result again in a canonical representation). Using this structure and its operations in the `LogicInterBU`-algorithm we gain a canonical formula representing the permitted packets of a list `L`, denoted $\wr\texttt{L}\int$. Summarised we have:

Object	**Interpretation**
packet	canonical boolean formula (representing a truth-assignment)
rule	canonical boolean formula
rule-list	canonical boolean formula

Performing analysis

We have seen how to check if a packet p matches a rule. We can test the same way if the packet is permitted by list `L`:

$$p \in [\![\mathtt{L}]\!] \Leftrightarrow \wr p \wr \vdash \wr \mathtt{L} \wr$$

Because we use canonical representations semantical equality of lists leads to equivalent representations. Union corresponds to disjunction and intersection to conjunction. Set-subtraction can be modelled by conjunction and negation. The only interesting case is the subset-relation. We want to check if $[\![\mathtt{L}_1]\!] \subseteq [\![\mathtt{L}_2]\!]$. $\wr \mathtt{L}_1 \wr$ models all formulas which represent packets from $[\![\mathtt{L}_1]\!]$. $\wr \mathtt{L}_2 \wr$ does the same for $[\![\mathtt{L}_2]\!]$. That is why we just have to check if $\wr \mathtt{L}_2 \wr$ derives from $\wr \mathtt{L}_1 \wr$:

$$[\![\mathtt{L}_1]\!] \subseteq [\![\mathtt{L}_2]\!] \Leftrightarrow \wr \mathtt{L}_1 \wr \vdash \wr \mathtt{L}_2 \wr$$

Putting it all together:

Semantical Task	**Interpretation**
$p \in [\![\mathtt{L}]\!]$	$\wr p \wr \vdash \wr \mathtt{L} \wr$
$[\![\mathtt{L}_1]\!] = [\![\mathtt{L}_2]\!]$	$\wr \mathtt{L}_1 \wr = \wr \mathtt{L}_2 \wr$
$[\![\mathtt{L}_1]\!] \subseteq [\![\mathtt{L}_2]\!]$	$\wr \mathtt{L}_1 \wr \vdash \wr \mathtt{L}_2 \wr$
$[\![\mathtt{L}_1]\!] \cup [\![\mathtt{L}_2]\!]$	$\wr \mathtt{L}_1 \wr \lor \wr \mathtt{L}_2 \wr$
$[\![\mathtt{L}_1]\!] \cap [\![\mathtt{L}_2]\!]$	$\wr \mathtt{L}_1 \wr \land \wr \mathtt{L}_2 \wr$
$[\![\mathtt{L}_1]\!] \setminus [\![\mathtt{L}_2]\!]$	$\wr \mathtt{L}_1 \wr \land \neg \wr \mathtt{L}_2 \wr$

Generalisation to multiple actions

Taking the `SetSemBU`$_M$ as a pattern we get:

```
LogicInterBU$_M$(List L)
$A_1, \ldots, A_n$:  Boolean Formula := $\bot$
FOR i := size(L) DOWNTO 1 DO {
   j = type(L(i))
   $A_j$ = $\underline{(}A_j \lor \wr$L(i)$\wr\underline{)}$
   FOR ALL $A_k$,  $k \neq j$ DO
     $A_k$ = $\underline{(}A_k \land \neg\wr$L(i)$\wr\underline{)}$
   i := i-1
}
RETURN $(A_1, \ldots, A_n)$
```

Again it is straightforward to translate the ordered union. A default policy rule is represented by the formula $\top$. Therefore to add a default action i we perform $\wr \mathtt{L}_i \wr = \wr \mathtt{L}_i \wr \lor \neg \bigvee_{j=1}^{n} \wr \mathtt{L}_j \wr$.

Translating back to rule-lists

Because a whole list is transformed into single canonical boolean formula represented by a BDD it is more complicated to gain back a rule-list here. In [3] a algorithm called *tablerep* is given which transforms a BDD representing a rule-list back into a tabular representation. Roughly speaking the algorithm splits the d-dimensional space into smaller cubes until such a cube completely covers *Permit*-packets. All such cubes represent a resulting rule. The shape and size of the resulting table highly depends on a fixed order of the variables.

Logical approach — summary

With the help of BDDs another decorrelated and canonical structure has been found – boolean formulas. Performing analysis can be done by applying basic logical operations but gaining back rule-lists is more difficult.

2.5 Summary

Three fundamental concepts showed the way to perform analysis: set-theory, geometry and logic. The semantical world of sets laid the foundations for more specific structures. The geometrical approach is flexible and powerful enough to implement all wanted operations. Using hyperrectangles may appear at first side less obvious because a canonical supporting data structure needs to be developed. But BDDs are also tricky structures which are just more familiar. And hyperrectangles do much more relate to a human readable tabular form than BDDs do. In chapter 4 we will see how to implement data structures and algorithms for hyperrectangles.

We have seen that rule-lists represent sets of packets and how they can be transformed into sets of hyperrectangles and boolean formulas. Back-translating to rule-lists gives bidirectional interconnections between all three concepts. From a semantical point of view all concepts are equal. In the next chapter we will formalise and proof this equality with the help of category theory which will give the concepts a common mathematical structure.

The logical approach was developed by Scott Hazelhurst [13], [12], [3]. Adi Attar [2] gives a detailed explanation of Scott's ideas with a focus on performance. Mikkel Christiansen and Emmanuel Fleury [7] use first-order formulas with integer constraints instead of boolean formulas and Interval Decision Diagrams (IDDs) as a supporting data structure. Alain Mayer, Avishai Wool and Elisha Ziskind [17] developed a topology-based Firewall Analysis Engine (Fang). Pasi Eronen and Jukka Zitting [11] use an expert system to analyse rule-lists.

Chapter 3

A categorical foundation

Every category consists of *objects* and so called *morphisms* between the objects. For example sets as objects and functions as morphisms form a category. Other examples are topological spaces with continuous functions or algebras with homomorphisms. The connections between whole categories is given by *functors*. Formal definitions for all used notions in this chapter are given in Appendix A. We will only deal here with very simple categories and basic notions. We will use the notations of [10]. Other introductions to category theory are e.g. in [16] and [1].

In chapter 2 we have dealt with three types of objects: sets of packets, sets of hyperrectangles and boolean formulas. The most fundamental operation firewalls perform is lookup, i.e. answering the question *is packet p accepted by list* `L`*?* In sets this was modelled by the $\in$-relation. We could have also used the $\subseteq$-relation because $e \in A \Leftrightarrow \{e\} \subseteq A$. The same holds for sets of hyperrectangles because every packet is a special case of a hyperrectangles – a point. Therefore $\in_g$ is a special case of $\subseteq_g$. And with formulas we have directly modelled the lookup via derivation: $\wr p \int \vdash \wr L \int$ instead of interpreting a packet as a truth-assignment. In other words a set consists of elements, a hyperrectangle consists of points and a formula represents the truth-assignments which evaluate it to true. The relations $\subseteq$, $\subseteq_g$ and $\vdash$ are in this sense a *is part of*-relation. All of them are reflexive and transitive (a preorder, $\subseteq$ and $\subseteq_g$ are even partial orders). That is why they are suited to serve as morphisms.

For every concept a category will be constructed. Because preorders are used to define if there is a morphism between two objects such categories are called *preorder-categories*. We will see how the universal categorical constructions *initial object*, *terminal object*, *product*, *coproduct* and *exponent* show up in every single category. Analytical tasks will reappear in basic categorical terms. For example the product will exactly answer the question *which packets are accepted by both* $\mathtt{L}_1$ *and* $\mathtt{L}_2$*?*

Finally we will model how to get from one category to another via functors. This will show that all three categories are absolutely equal from a structural point of view. Such equal objects are called *isomorphic* objects[1].

Proofs in preorder-categories are usually quite simple because there is at most one morphism between objects. That is why not every proof will be given in detail. We will take again the case of two possible actions *permit* and *deny* as a starting point and give some ideas for generalisation later.

3.1 Categories with respect to firewall analysis

Because we are dealing with strict mathematical models of our structures every used notion has to be formalised. We assume that the filter-rules deal with d attributes $T_1..T_d$. The domain of an attribute is denoted by $dom(T_i)$ and we assume that:

$$\forall i \in \{1..d\}\ \exists p_i \in \mathbb{N} : dom(T_i) = [0, (2^{p_i} - 1)]$$

Note that even IP-protocols are specified in the range $[0, (2^8 - 1]$ according to RFC1340. Now we can define the set IP as follows:

$$IP =_{def} \prod_{i=1}^{d} dom(T_i)$$

3.1.1 The category of IP-sets: $\mathbf{2^{IP}}$

The category $\mathbf{2^{IP}}$ is defined as follows:

$$\begin{aligned} Ob_{\mathbf{2^{IP}}} &= 2^{IP} \\ Mor_{\mathbf{2^{IP}}}(A,B) &= \begin{cases} \{*\} & \text{if} A \subseteq B \\ \emptyset & \text{else} \end{cases} \end{aligned}$$

That means that there is exactly one morphism from A to B (called "$*$") iff $A \subseteq B$. Because $\subseteq$ is a partial order the category-axioms hold.

Constructions

Initial object $\emptyset$ is the initial object because for all sets A holds $\emptyset \subseteq A$.

Terminal object Because we deal with the powerset of IP all objects of $\mathbf{2^{IP}}$ are a subset of IP. Therefore IP is the terminal object.

[1]Categories itself can be seen as objects in the category of categories with functors as morphisms.

Product The product of two objects A and B, written $A \times B$, is the intersection:

$$A \times B = A \cap B$$

If there is an object X which is a subset of A and a subset of B then it follows immediately that X is also a subset of $A \cap B$. That is all which has to be proven here. Note that a product consists of the product-object and two morphisms, the *projections*. Because there is only one possible morphism between objects these morphisms will not be mentioned specially here.

Coproduct The coproduct, also called sum, of two objects A and B, written $A + B$, is the union:

$$A + B = A \cup B$$

If there is an object X with $A \subseteq X$ and $B \subseteq X$ then it follows immediately that also $A \cup B \subseteq X$ holds.

Exponent For every two objects $A, B \in Ob_{\mathbf{2^{IP}}}$ there is an exponent, written B^A:

$$B^A = \overline{A} \cup B$$

Note that $\overline{A} = IP \backslash A$. Because $A \times B^A = A \cap (\overline{A} \cup B) = A \cap B \subseteq B$ the exponent-morphism $\epsilon_{A,B} : A \times B^A \to B$ always exists. Given an object C and a morphism $f : A \times C \to B$ we know that $A \times C = A \cap C \subseteq B$. We have to show that $C \subseteq \overline{A} \cup B$. Given an element $e \in C$ we have two cases. If $e \in A$ we know that $e \in A \cap C$ and because $A \cap C \subseteq B$ we get $e \in B$ which induces that $e \in \overline{A} \cup B$. If $e \notin A$ we have $e \in \overline{A}$ and therefore in every case $e \in \overline{A} \cup B$ holds.

Because the category $\mathbf{2^{IP}}$ has a terminal object, products and exponents the category is *cartesian closed*.

Summary

Construction	$\mathbf{2^{IP}}$
initial object 0	$\emptyset$
terminal object 1	IP
product $A \times B$	$A \cap B$
coproduct $A + B$	$A \cup B$
exponent B^A	$\overline{A} \cup B$

The categorical counterpart of a packet $p \in IP$, denoted $\mathcal{C}(p)$, is represented in $\mathbf{2^{IP}}$ by the singleton set $\{p\} \in Ob_{\mathbf{2^{IP}}}$. A rule-list can describe any possible subset of IP, therefore every object could be the *Permit*-set of a list `L`, denoted $\mathcal{C}(\mathtt{L})$[2].

[2]Note that $[\![\mathtt{L}]\!] = \mathcal{C}(\mathtt{L})$ w.r.t. $\mathbf{2^{IP}}$.

3.1.2 The category of IP-hyperrectangles: $[\mathbf{2^{HRect_{IP}}}]$

In chapter 2 we have defined the $\subseteq_g$-relation to check if $[\![\mathtt{L}_1]\!] \subseteq [\![\mathtt{L}_2]\!]$. The implementation of $\subseteq_g$ was based on canonical normalisation to make the approach efficient. Here we do not care about efficiency but structure. That is why we will see sets of hyperrectangles just as sets of points when checking if $[\![\mathtt{L}_1]\!] \subseteq [\![\mathtt{L}_2]\!]$ holds. The advantage is that we can use the normal subset-relation $\subseteq$ on sets of points instead of the tricky $\subseteq_g$ one. Another advantage of a point-wise view is that we can allow hyperrectangles within a *Permit*-set to overlap. First we define the set HRect_{IP}:

$$\mathrm{HRect}_{IP} =_{def} \left\{ \prod_{i=1}^{d} [l_i, r_i] \mid l_i, r_i \in dom(T_i) \text{ with } l_i \leq r_i \right\}$$

A packet, i.e. point $p = (t_1, \ldots, t_d) \in IP$ is now a special case of a hyperrectangle because $\prod_{i=1}^{d}[t_i, t_i] = \{p\}$ and therefore we have $\{p\} \in \mathrm{HRect}_{IP}$. Let S be a set of hyperrectangles, i.e. $S \in 2^{\mathrm{HRect}_{IP}}$. We define the point-wise variant of S, written $\boxdot(S)$, as:

$$\boxdot(S) =_{def} \bigcup_{R \in S} \{\{p\} \mid p \in R\}$$

Note that $\boxdot(S) \in 2^{\mathrm{HRect}_{IP}}$ and that $\boxdot(\emptyset) = \emptyset$. Now we can define an equivalence relation $\overset{\boxdot}{=}$ on sets of hyperrectangles. Let $S_1, S_2 \in 2^{\mathrm{HRect}_{IP}}$. We define:

$$S_1 \overset{\boxdot}{=} S_2 \Leftrightarrow_{def} \boxdot(S_1) = \boxdot(S_2)$$

The objects of our category are equivalence classes of hyperrectangles regarding $\overset{\boxdot}{=}$. We define the category $[\mathbf{2^{HRect_{IP}}}]$ as follows:

$$\begin{array}{rcl} Ob_{[\mathbf{2^{HRect_{IP}}}]} & = & \{[S]_{\overset{\boxdot}{=}} \mid S \in 2^{\mathrm{HRect}_{IP}}\} \\ Mor_{[\mathbf{2^{HRect_{IP}}}]}([A]_{\overset{\boxdot}{=}}, [B]_{\overset{\boxdot}{=}}) & = & \begin{cases} \{*\} & \text{if } \boxdot(A) \subseteq \boxdot(B) \\ \emptyset & \text{else} \end{cases} \end{array}$$

From now on we will just write $[A]$ instead of $[A]_{\overset{\boxdot}{=}}$.

Constructions

Because we have also chosen the $\subseteq$-relation as a morphism-indicator the constructions (and proofs) are very similar to the ones in $\mathbf{2^{IP}}$.

Initial object $[\emptyset] = \emptyset$ is the initial object because for all sets A holds $\emptyset \subseteq A$.

Terminal object The equivalence class which includes sets of hyperrectangles spanning the whole set IP is the terminal object. Therefore we have $[\{IP\}]$ as the terminal object here.

Product The product of two objects $[A]$ and $[B]$ is the intersection of the point-wise representatives:

$$[A] \times [B] = [\boxdot(A) \cap \boxdot(B)]$$

Coproduct The coproduct of two objects $[A]$ and $[B]$ is the point-wise union:

$$[A] + [B] = [\boxdot(A) \cup \boxdot(B)]$$

Exponent For every two objects $[A], [B] \in Ob_{[\mathbf{2}^{\mathbf{HRect_{IP}}}]}$ there is an exponent, written $[B]^{[A]}$:

$$[B]^{[A]} = [\overline{\boxdot(A)} \cup \boxdot(B)]$$

The complement of a set of points $\boxdot(A)$ is defined here again with respect to the set of all possible points in IP:

$$\overline{\boxdot(A)} =_{def} \boxdot(\{IP\}) \setminus \boxdot(A)$$

Because the category $[\mathbf{2}^{\mathbf{HRect_{IP}}}]$ has a terminal object, products and exponents the category is *cartesian closed.*

Summary

Construction	$[\mathbf{2}^{\mathbf{HRect_{IP}}}]$
initial object 0	$[\emptyset]$
terminal object 1	$[\{IP\}]$
product $[A] \times [B]$	$[\boxdot(A) \cap \boxdot(B)]$
coproduct $[A] + [B]$	$[\boxdot(A) \cup \boxdot(B)]$
exponent $[B]^{[A]}$	$[\overline{\boxdot(A)} \cup \boxdot(B)]$

The categorical counterpart $\mathcal{C}(p)$ of a packet $p \in IP$ is represented in $[\mathbf{2}^{\mathbf{HRect_{IP}}}]$ by the singleton set $[\{\{p\}\}] = \{\{\{p\}\}\} \in Ob_{[\mathbf{2}^{\mathbf{HRect_{IP}}}]}$.

3.1.3 The category of IP-formulas: $[\mathbf{Form_{IP}}]$

First we have to know how many boolean variables we need for our formulas. This is easy because we have assumed that every domain is of the form $\{0..(2^{p_i} - 1)\}$. Therefore $\sum_{i=1}^{d} p_i$ variables are needed. The names are defined as follows:

$$\mathfrak{P} =_{def} \bigcup_{i=1}^{d} \{x_{i,1} .. \, x_{i,p_i}\}$$

By Form($\mathfrak{P}$) we denote the set of boolean formulas regarding the variable-set $\mathfrak{P}$. Because syntactically different formulas $\varphi, \psi \in$ Form($\mathfrak{P}$) can be semantically equal, written $\varphi \equiv$

ψ, we use sets of semantically equal formulas as objects. A morphism in the category $[\mathbf{Form_{IP}}]$ is now indicated by the preorder $\vdash$:

$$\begin{aligned} Ob_{[\mathbf{Form_{IP}}]} &= \{[\varphi]_\equiv \mid \varphi \in \mathrm{Form}(\mathfrak{P})\} \\ Mor_{[\mathbf{Form_{IP}}]}([\varphi]_\equiv, [\psi]_\equiv) &= \begin{cases} \{*\} & \text{if } \varphi \vdash \psi \\ \emptyset & \text{else} \end{cases} \end{aligned}$$

From now on we will just write $[\varphi]$ instead of $[\varphi]_\equiv$. By $\perp$ we denote a contradiction and by $\top$ a tautology.

Constructions

Initial object $[\perp]$ is the initial object because for all formulas φ holds $\perp \vdash \varphi$.

Terminal object For all formulas φ holds $\varphi \vdash \top$, therefore $[\top]$ is the terminal object.

Product The product $[\varphi] \times [\psi]$ of two objects $[\varphi]$ and $[\psi]$ is the conjunction:

$$[\varphi] \times [\psi] = [\varphi \wedge \psi]$$

This is correct because if $\chi \vdash \varphi$ and $\chi \vdash \psi$ we also have $\chi \vdash \varphi \wedge \psi$.

Coproduct The coproduct $[\varphi] + [\psi]$ of two objects $[\varphi]$ and $[\psi]$ is the disjunction:

$$[\varphi] + [\psi] = [\varphi \vee \psi]$$

Note that if $\varphi \vdash \chi$ and $\psi \vdash \chi$ we also have $\varphi \vee \psi \vdash \chi$.

Exponent For every two objects $[\varphi], [\psi] \in Ob_{[\mathbf{Form_{IP}}]}$ the implication defines the exponent object, written $[\psi]^{[\varphi]}$:

$$[\psi]^{[\varphi]} = [\varphi \to \psi]$$

The *Modus Ponens* and $\varphi \wedge \chi \vdash \psi \Rightarrow \chi \vdash \varphi \to \psi$ is all which is needed here.

Because the category $[\mathbf{Form_{IP}}]$ has a terminal object, products and exponents the category is *cartesian closed.*

Summary

Construction	$[\mathbf{Form_{IP}}]$
initial object 0	$[\perp]$
terminal object 1	$[\top]$
product $[\varphi] \times [\psi]$	$[\varphi \wedge \psi]$
coproduct $[\varphi] + [\psi]$	$[\varphi \vee \psi]$
exponent $[\psi]^{[\varphi]}$	$[\varphi \to \psi]$

For all variables $x \in \mathfrak{P}$ we define the *literal-set* $\mathcal{L}(x)$ as $\mathcal{L}(x) =_{def} \{x, \neg x\}$. In subsection 2.4.2 we have seen that a packet corresponds to a conjunction of literals representing the bits of a number. The categorical counterpart $\mathcal{C}(p)$ of a packet $p \in IP$ is therefore represented in $[\mathbf{Form_{IP}}]$ by an object like:

$$\left[\bigwedge_{x \in \mathfrak{P}} y \in \mathcal{L}(x)\right]$$

3.1.4 Analysis in categorical terms

We have defined simple categories for every concept. Packets and rule-lists correspond to the objects and there is a morphism between two objects A and B iff A *is part of* B. Semantically equal constructs have been capsuled into a single object to get a bijection between the categories. This will be studied further in the next section. Analytical tasks of subsection 2.3.2 reappeared as simple categorical concepts and constructions. The following table gives a summary:

Analytical Task	**Categorical**
is packet p accepted by list $\mathtt{L}$?	$\exists m : \mathcal{C}(p) \to \mathcal{C}(\mathtt{L})$
are all packets accepted by $\mathtt{L}_1$ also accepted by $\mathtt{L}_2$?	$\exists m : \mathcal{C}(\mathtt{L}_1) \to \mathcal{C}(\mathtt{L}_2)$
which packets are accepted at least by $\mathtt{L}_1$ or $\mathtt{L}_2$?	$\mathcal{C}(\mathtt{L}_1) + \mathcal{C}(\mathtt{L}_2)$
which packets are accepted by both $\mathtt{L}_1$ and $\mathtt{L}_2$?	$\mathcal{C}(\mathtt{L}_1) \times \mathcal{C}(\mathtt{L}_2)$

3.2 Changing the views

In the last section we have seen that the same categorical constructions yield in every category an object representing a certain analytical task. The categories seem to have the same structure applied to different objects. That means that these categories are equal from a structural point of view, they model the same thing in different words, i.e. objects. We can see every category as a certain *view* on the modelled thing. To prove this we have to show that the structure and expressive power of the views is equal for every category. Therefore we need a connection between the objects and morphisms of different categories. Such a connection is given by a *functor* going from one category to another.

3.2.1 From IP-sets to IP-hyperrectangles and back

Every set of hyperrectangles is basically a set of points. Every point is a packet. Therefore every object in $\mathbf{2^{IP}}$ corresponds to one in $[\mathbf{2^{HRect_{IP}}}]$ and vice versa. Because we only deal with preorder-categories a functor has only one choice to map a morphism:

$$F_{Mor}(*) = *$$

That is why we wont mention this specially. The functor $F : \mathbf{2^{IP}} \to [\mathbf{2^{HRect_{IP}}}]$ is defined as follows:

$$\begin{aligned} F_{Ob}(\emptyset) &= [\emptyset] \\ F_{Ob}(\{p_1, \ldots, p_j\}) &= [\{\{p_1\}, \ldots, \{p_j\}\}] \end{aligned}$$

We simply map a set of packets to the corresponding set of points. Note that $F_{Ob}(A) = [S] \Rightarrow \boxdot(S) = S$ holds. Therefore $A \overset{*}{\to} B \Rightarrow F_{Ob}(A) \overset{*}{\to} F_{Ob}(B)$ holds too, i.e. F_{Mor} is well-defined.

We could have also defined a functor $J : [\mathbf{2^{HRect_{IP}}}] \to \mathbf{2^{IP}}$ which maps the point-wise variant of a set of hyperrectangles to the corresponding set of packets. The functors seem to be *inverse*, i.e. $F \circ J = Id_{[\mathbf{2^{HRect_{IP}}}]}$ and $J \circ F = Id_{\mathbf{2^{IP}}}$ which induces that the categories are really structurally equal, i.e. isomorphic, written $[\mathbf{2^{HRect_{IP}}}] \cong \mathbf{2^{IP}}$. Instead of defining the functor I we simply argue that F is bijective which also proves the isomorphism. F is obviously injective. To prove that F is also surjective we have to show that for every $[S] \in Ob_{[\mathbf{2^{HRect_{IP}}}]}$ there is a set $A \in Ob_{\mathbf{2^{IP}}}$ with $F_{Ob}(A) = [S]$. This holds too because we know that $[\boxdot(S)] = [S]$ and F_{Ob} maps sets of packets to sets of points.

3.2.2 From IP-sets to IP-formulas and back

Now we show how to get from a set of packets to an equivalence class of boolean formulas. Every variable in $\mathfrak{P}$ represents a bit. For example the variables $\{x_{2,1}, \ldots, x_{2,p_2}\}$ represent the p_2 bits which are needed to encode a number $n \in dom(T_2) = [0, (2^{p_2} - 1)]$. As introduced a packet $p \in A \in Ob_{\mathbf{2^{IP}}}$ maps therefore to a conjunction of literals where a negative (negated) literal represents a zero and a positive literal a one. We use again the notation $\wr p \int \in \text{Form}(\mathfrak{P})$ of chapter 2 to denote this conjunction. The functor $G : \mathbf{2^{IP}} \to [\mathbf{Form_{IP}}]$ is now defined as:

$$\begin{aligned} G_{Ob}(\emptyset) &= [\bot] \\ G_{Ob}(\{p_1, \ldots, p_j\}) &= [(\wr p_1 \int) \vee \ldots \vee (\wr p_j \int)] \end{aligned}$$

Because for all boolean formulas φ, ψ holds $\varphi \vdash \varphi \vee \psi$ G_{Mor} is well-defined. Again it is obvious that G is injective. To prove that G is surjective we have to show that every boolean formula φ can be normalised to a semantically equal formula of the following form:

$$\bigvee \bigwedge_{x \in \mathfrak{P}} y \in \mathcal{L}(x)$$

Note that we are dealing with a special *disjunctive normal form.* It is easy to prove that this special form always exists. Because most books on propositional logic include this proof we just sketch it here (see e.g. [10]). Given the truth-table of φ every row represents a truth-assignment which can be seen as a conjunction of literals. We just have to disjunct the formulas corresponding to rows which evaluate φ to true to gain a formula of the wanted form which is semantically equal to φ. Because G is bijective the categories are again isomorphic: $[\mathbf{Form_{IP}}] \cong \mathbf{2^{IP}}$.

3.2.3 Conclusions

Because isomorphism is an equivalence-relation we can conclude:

$$\mathbf{2^{IP}} \cong [\mathbf{2^{HRect_{IP}}}] \cong [\mathbf{Form_{IP}}]$$

We have shown that all categories are structurally equal. This has several implications. Category theory guarantees now that we can travel safely from one category to another via the functors. For example applying G to a product-object $A \times B \in Ob_{\mathbf{2^{IP}}}$ yields again a product-object $G_{Ob}(A \times B)$ of $G_{Ob}(A)$ and $G_{Ob}(B)$ in $[\mathbf{Form_{IP}}]$. Imagine for example that $A = \{p_1, p_2\}$ and $B = \{p_2, p_3\}$. The following diagram shows the product-translation:

$$\begin{tikzcd}
A = \{p_1, p_2\} \arrow[r, mapsto, "G_{Ob}"] & {[(\wr p_1 \int) \vee (\wr p_2 \int)]} \\
A \times B = \{p_2\} \arrow[r, mapsto, "G_{Ob}"] \arrow[u, "\subseteq"'] \arrow[d, "\subseteq"] & {[(\wr p_2 \int)] = G_{Ob}(A) \times G_{Ob}(B)} \arrow[u, "\vdash"'] \arrow[d, "\vdash"] \\
B = \{p_2, p_3\} \arrow[r, mapsto, "G_{Ob}"] & {[(\wr p_2 \int) \vee (\wr p_3 \int)]}
\end{tikzcd}$$

At first sight it might surprise that $((\wr p_1 \int) \vee (\wr p_2 \int)) \wedge ((\wr p_2 \int) \vee (\wr p_3 \int)) \equiv \wr p_2 \int$ holds. But notice that every formula $\wr p \int$ represents exactly one truth-assignment which corresponds to a number. Therefore $\wr p \int \equiv \wr p \int \wedge \neg \bigvee \wr q \int,\ q \in IP \setminus \{p\}$ holds. In general every construction and even algorithms using the constructions can be transferred between the views. We have already done this in chapter 2 when adapting the `SetSem`-algorithms.

Another aspect is that we could see $\mathbf{2^{IP}}$ as the *semantical category* and $[\mathbf{2^{HRect_{IP}}}]$, $[\mathbf{Form_{IP}}]$ as *implementation categories*. From this point of view the semantical category acts as a *specification of the semantical structure*. Showing that the implementation categories are isomorphic to it is like proving that they are *correct with respect to the semantical structure*.

Generalisation to multiple actions is more complicated because we are dealing with partitions which separate IP into more than two parts. This yields set-families $(S_i)_{i \in \{A_1, \ldots, A_n\}}$ as objects for the semantical category. Because there is no simple lookup-concept anymore we can not generalise this to gain a morphism-indicator. In other words we have no *is part of*-operator when dealing with these partitions. One solution to define morphisms here is given in [23],5.1 by Uwe Wolter. The basic idea is roughly that there is a morphism between A and B iff A *makes more elements of* IP *indistinguishable than* B. We wont go into detail here and keep this as future work.

3.3 Summary

All approaches talk about the same – IP-packets. Every rule-list represents a set of permitted packets. That is why such sets become the objects of the semantical category $\mathbf{2^{IP}}$. The basic lookup-concept can be generalised to yield morphisms between the objects. The geometrical and logical approach can be modelled as implementation categories which are isomorphic to the semantical one and therefore correct with respect to the semantical structure. Simple categorical notions and constructions correspond to analytical tasks we have studied in chapter 2. We have seen that it is more appropriate to talk e.g. of the product of two rule-lists instead of their intersection which is a set-fixed notion. Categorical constructions are general but exact descriptions which appear as concrete constructions in a given category.

The geometrical approach has the advantage that it can describe large subsets of IP by spanning hyperrectangles. This is why rule-lists use this idea to make firewall-setup easy. But because points are a special case of hyperrectangles there is no loss of expressive power, i.e. every subset of IP can be described by a set of hyperrectangles. The logical approach is based on numbering of the $\prod_{i=1}^{d} 2^{p_i} = 2^{\sum_{i=1}^{d} p_i}$ elements of IP. To do so $\sum_{i=1}^{d} p_i$ boolean variables are needed which represent bits. A set corresponds then to the numbers of its elements. From a structural point of view it does not matter if an element is represented by a point in an appropriate space or by a number described by a formula. That is why we could prove the structural correctness.

Chapter 4

Implementing the geometrical approach

In chapter 2 we have seen how sets of hyperrectangles can be used to perform analysis of rule-lists. In this chapter we will introduce efficient data structures and algorithms for these geometrical objects. The discipline we are dealing with here is known as *computational geometry*. In the next section 4.1 well-known data structures are summarised which can be used to perform intersection-tests of hyperrectangles. A new algorithm which removes intersections and performs canonical normalisation simultaneously is presented in section 4.2.

4.1 Data-structures for hyperrectangles

Four tree-like structures are relevant: *balanced binary search trees*, *range trees*, *priority search trees* and *segment trees*. All of them deal with points or intervals. Because hyperrectangles are interval products we can use these structures for storing and querying. Everything in this section can be found more detailed in [4], chapter 5 and 10.

4.1.1 Binary search trees

A node $\mathfrak{v}$ of a binary search tree consists of a key $\kappa(\mathfrak{v})$ and links to the left subtree $lc(\mathfrak{v})$ and right subtree $rc(\mathfrak{v})$. Additionally for all nodes $\mathfrak{v}$ must hold that if $\mathfrak{l}$ is a node in the left subtree of $\mathfrak{v}$ then $\kappa(\mathfrak{l}) \leq \kappa(\mathfrak{v})$ and if $\mathfrak{r}$ is a node in the right subtree of $\mathfrak{v}$ then $\kappa(\mathfrak{v}) < \kappa(\mathfrak{r})$. See figure 4.1 for an example. Note that different binary search trees can represent the same set of values. The worst-case running time for most operations is proportional to the height of the tree. That is why several enhanced versions have been developed, e.g. *AVL trees*, *2-3 trees*, *B trees*, *Red-Black trees* or *Splay trees*. They are called *balanced search trees* and have the property that their height is in $\mathcal{O}(\log(n))$.

Such trees can be used to perform one-dimensional range searching. Given a set of one-dimensional points p and a query interval $[\Delta, \Delta']$ we want to report all points which

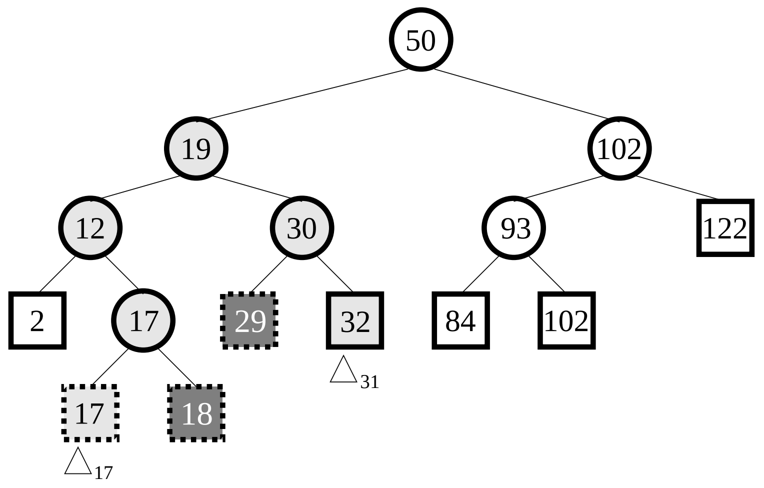


Figure 4.1: Binary Search Tree

lie inside the interval, i.e. $\Delta \leq p \leq \Delta'$. To do so we store the points at the leaves of the tree and see the internal nodes just as splitting values to guide the search. Note that in example 4.1 we have already drawn the leaves as squares. This tree therefore stores the set of points $\{2, 17, 18, 29, 32, 84, 102, 122\}$. We will see later how to construct such a tree when dealing with range trees. To find all points in a query range like $[17, 31]$ we search for 17 and 31 in the tree until we reach a leaf. Let Δ_{17} and Δ_{31} be these two leaves where the searches end. The points in the interval $[17, 31]$ are stored in the leaves between Δ_{17} and Δ_{31} plus possibly the points stored at Δ_{17} and Δ_{31} itself. Regarding the example we have $\kappa(\Delta_{17}) = 17$ and $\kappa(\Delta_{31}) = 32$. Therefore the resulting set of points is $\{17, 18, 29\}$ (drawn dotted).

The algorithm `1DRangeQuery` uses two subroutines. `FindSplitNode` computes the node where the two search paths split (or the leaf where both end). It gets as input a binary search tree $\mathfrak{T}$ and the query interval $[\Delta, \Delta']$:

FindSplitNode(Tree $\mathfrak{T}$, Interval $[\Delta, \Delta']$)

```
$\mathfrak{v}$:  Node := root($\mathfrak{T}$)
WHILE $\mathfrak{v}$ is not a leaf AND ($\Delta' \leq \kappa(\mathfrak{v})$ OR $\Delta > \kappa(\mathfrak{v})$) DO {
   IF ($\Delta' \leq \kappa(\mathfrak{v})$) DO
     $\mathfrak{v} = lc(\mathfrak{v})$ ELSE
     $\mathfrak{v} = rc(\mathfrak{v})$
}
RETURN $\mathfrak{v}$
```

In example 4.1 the split-node $\mathfrak{v}_{split}$ for $[17, 31]$ has $\kappa(\mathfrak{v}_{split}) = 19$. The second subroutine traverses the subtree rooted at a given node and reports the points stored at its leaves. It is called `ReportSubtree`. Now we can define the algorithm `1DRangeQuery`:

```
1DRangeQuery(SearchTree $\mathfrak{T}$, Interval $[\Delta, \Delta']$)
$\mathfrak{v}_{split}$: Node := FindSplitNode($\mathfrak{T}, [\Delta, \Delta']$)
IF $\mathfrak{v}_{split}$ is a leaf DO
   { IF $\kappa(\mathfrak{v}_{split}) \in [\Delta, \Delta']$ report $\kappa(\mathfrak{v}_{split})$ }
ELSE {

     $\mathfrak{v} = lc(\mathfrak{v}_{split})$
     WHILE $\mathfrak{v}$ is not a leaf DO {
       IF $\Delta \leq \kappa(\mathfrak{v})$ DO {
         ReportSubtree(rc($\mathfrak{v}$))
         $\mathfrak{v} = lc(\mathfrak{v})$
       } ELSE
       $\mathfrak{v} = rc(\mathfrak{v})$
     }
     IF $\kappa(\mathfrak{v}) = \Delta$ report $\kappa(\mathfrak{v})$

     $\mathfrak{v} = rc(\mathfrak{v}_{split})$
     WHILE $\mathfrak{v}$ is not a leaf DO {
       IF $\Delta' > \kappa(\mathfrak{v})$ DO {
         ReportSubtree(lc($\mathfrak{v}$))
         $\mathfrak{v} = rc(\mathfrak{v})$
       } ELSE
       $\mathfrak{v} = lc(\mathfrak{v})$
     }
     IF $\kappa(\mathfrak{v}) = \Delta'$ report $\kappa(\mathfrak{v})$
}
RETURN $\mathfrak{v}$
```

In example 4.1 the search path is drawn in light grey whereas the subtrees given to `ReportSubtree` are drawn in dark grey. Summarised we have:

Proposition 4.1 (One-dimensional range queries) *Let P be a set of one-dimensional points. They can be stored in a balanced binary search tree which uses $\mathcal{O}(n)$ storage and has $\mathcal{O}(n \log n)$ construction time. The points in a query range can be reported in time $\mathcal{O}(k + \log n)$ where k is the number of reported points.*

Proofs for this can be found in [4]. For further information about binary search trees see e.g. [8] or [18].

4.1.2 Range trees

Now we are coming to rectangular two-dimensional range queries. The corresponding data structure presented here is called range tree[1]. The basic idea is that a two-dimensional query is a composition of two one-dimensional queries, one on the x-coordinate of the points and one on the y-coordinate. That is why we can use a combination of binary search trees to find all points lying inside a rectangle.

Let P be a set of two-dimensional points. The main tree is a balanced binary search tree $\mathfrak{T}$ built on the x-coordinates of the points in P. In addition to the coordinate we have to store a link to the point itself to know where the coordinate belongs to. At every node $\mathfrak{v}$ of $\mathfrak{T}$ we add a link to an associated balanced binary search tree $\mathfrak{A}(\mathfrak{v})$ storing the y-coordinates of the points located at the leaves of $\mathfrak{v}$. Again we link the y-coordinates with the corresponding points. Note that `ReportSubtree`$(\mathfrak{A}(\text{root}(\mathfrak{T}))) = P$ and that if $\mathfrak{v}$ is a leaf $\mathfrak{A}(\mathfrak{v})$ stores the y-coordinates of the point stored at the leaf.

The following recursive algorithm `Build2DRangeTree` constructs such a range tree out of a given set of points P in the plane:

```
Build2DRangeTree(PointSet $P$)
$\mathfrak{v}_{left}, \mathfrak{v}_{right}$:  RangeTree
Construct a binary search tree $\mathfrak{A}$ on the y-coordinates of the points in $P$.
Store at the leaves a link to the point itself.
IF $P$ contains only one point DO
  Create a leaf $\mathfrak{v}$ storing the x-coordinate of the point.
  Link it with the point itself and with $\mathfrak{A}$.
ELSE {
  Let $x_{mid}$ be the median x-coordinate of the points in $P$.
  Let $P_{left}$ be the subset of $P$ including points having a x-coordinate
  less than or equal to $x_{mid}$.
  $P_{right} = P \setminus P_{left}$
  $\mathfrak{v}_{left}$ = Build2DRangeTree($P_{left}$)
  $\mathfrak{v}_{right}$ = Build2DRangeTree($P_{right}$)
  Create a node $\mathfrak{v}$ with $\kappa(\mathfrak{v}) = x_{mid}$ and link it with $\mathfrak{A}$.
  Set $lc(\mathfrak{v})$ = $\mathfrak{v}_{left}$ and $rc(\mathfrak{v})$ = $\mathfrak{v}_{right}$.
}
RETURN $\mathfrak{v}$
```

The query algorithm called `2DRangeQuery` gets as input a two-dimensional range tree $\mathfrak{T}$ and a query rectangle $[\Delta_1, \Delta_1'] \times [\Delta_2, \Delta_2']$. Note that the only difference to `1DRangeQuery` is that calls to `ReportSubtree` are replaced by calls to `1DRangeQuery`:

[1]We could have also chosen *Kd-Trees* which are worse in query time but better in storage-usage.

```
2DRangeQuery(RangeTree $\mathfrak{T}$, Rectangle $[\Delta_1, \Delta_1'] \times [\Delta_2, \Delta_2']$)
$\mathfrak{v}_{split}$:  Node := FindSplitNode($\mathfrak{T}, [\Delta_1, \Delta_1']$)
IF $\mathfrak{v}_{split}$ is a leaf DO
  Check if the point linked with $\mathfrak{v}_{split}$ must be reported.
ELSE {

    $\mathfrak{v} = lc(\mathfrak{v}_{split})$
    WHILE $\mathfrak{v}$ is not a leaf DO {
      IF $\Delta_1 \leq \kappa(\mathfrak{v})$ DO {
        1DRangeQuery($\mathfrak{A}$(rc($\mathfrak{v}$)),$[\Delta_2, \Delta_2']$)
        $\mathfrak{v} = lc(\mathfrak{v})$
      } ELSE
      $\mathfrak{v} = rc(\mathfrak{v})$
    }
    Check if the point linked with $\mathfrak{v}$ must be reported.

    $\mathfrak{v} = rc(\mathfrak{v}_{split})$
    WHILE $\mathfrak{v}$ is not a leaf DO {
      IF $\Delta_1' > \kappa(\mathfrak{v})$ DO {
        1DRangeQuery($\mathfrak{A}$(lc($\mathfrak{v}$)),$[\Delta_2, \Delta_2']$)
        $\mathfrak{v} = rc(\mathfrak{v})$
      } ELSE
      $\mathfrak{v} = lc(\mathfrak{v})$
    }
    Check if the point linked with $\mathfrak{v}$ must be reported.
}
```

The basic idea used for a two-dimensional query can be generalised to any dimension. That means that we can see a d-dimensional query as a composition of d one-dimensional queries, one on every coordinate of the points. Therefore the construction and querying of two dimensional range trees can be easily adapted. We construct a balanced binary search tree on the first coordinate of the points and use a (d-1)-dimensional range tree as the associated structure.

Using a technique called *fractional cascading* can speed up the algorithms for range trees. These special versions are called *layered range trees*. Together we have then:

Proposition 4.2 (d-dimensional range queries) *Let P be a set of n d-dimensional points with $d \geq 2$. They can be stored in a layered range tree which uses $\mathcal{O}(n \log^{d-1} n)$ storage and has $\mathcal{O}(n \log^{d-1} n)$ construction time. The points in a rectangular query range can be reported in time $\mathcal{O}(\log^{d-1} n + k)$ where k is the number of reported points.*

Proofs for this can be found again in [4]. Note that we need an enormous restriction to make coordinates of points comparable and to compute medians, no two points are allowed to have the same coordinate regarding every single dimension. But we can get rid of this restriction by using so called *composite numbers* for every coordinate. See also [4] for details.

4.1.3 Priority search trees

Sometimes it is sufficient to compute all points inside a query rectangle which is unbounded to one side, e.g. to the left: $]-\infty, \Delta_x] \times [\Delta_y, \Delta'_y]$. Priority search trees are a combination of search trees and heaps. In comparison to range trees they improve the storage bound to $\mathcal{O}(n)$ w.r.t. these unbounded queries. Let us assume again that all points have distinct coordinates. Given a set P of points the plane a recursive $\mathcal{O}(n \log n)$ algorithms builds a priority search tree as follows:

- if $P = \emptyset$ then the priority search tree is an empty leaf
- else let $p_{min} \in P$ be the point having the smallest x-coordinate. Let y_{mid} be the median of the y-coordinates of the remaining points. Let

  $$\begin{aligned} P_{below} &=_{def} \{p \in P \setminus \{p_{min}\} \mid p_y < y_{mid}\} \\ P_{above} &=_{def} \{p \in P \setminus \{p_{min}\} \mid p_y > y_{mid}\} \end{aligned}$$

  The priority search tree consists of a root node $\mathfrak{v}$ with an associated point $\mathfrak{p}(\mathfrak{v}) =_{def} p_{min}$ and $\kappa(\mathfrak{v}) =_{def} y_{mid}$. A priority search tree constructed for P_{below} becomes $lc(\mathfrak{v})$ and one for P_{above} becomes $rc(\mathfrak{v})$.

If the points in P are already sorted on their y-coordinate a bottom-up construction can be done even in linear time. The query-algorithm `QueryPrioSearchTree` uses a subroutine `ReportInSubtree` which gets a root $\mathfrak{v}$ of a priority search tree and a x-coordinate Δ_x as input. It computes all points having their x-coordinate at most Δ_x. By p_x we denote the x-coordinate of a point p.

```
**ReportInSubtree(Node $\mathfrak{v}$, Coordinate $\Delta_x$)**
IF $\mathfrak{v}$ is not a leaf and $(\mathfrak{p}(\mathfrak{v}))_x \leq \Delta_x$ DO {
  report $\mathfrak{p}(\mathfrak{v})$
  ReportInSubtree($lc(\mathfrak{v})$,$\Delta_x$)
  ReportInSubtree($rc(\mathfrak{v})$,$\Delta_x$)
}
```

`QueryPrioSearchTree` gets a priority search tree $\mathfrak{T}$ and query rectangle unbounded to the left. It outputs all points lying inside that rectangle:

```
**QueryPrioSearchTree(PrioSearchTree $\mathfrak{T}$, Rectangle $]-\infty, \Delta_x] \times [\Delta_y, \Delta'_y]$)**
$\mathfrak{v}_{split}$:  Node
Search with $\Delta_y$ and $\Delta'_y$ in $\mathfrak{T}$.
```

```
Let $\mathfrak{v}_{split}$ be the node where the search paths split.
FOR ALL nodes $\mathfrak{v}$ on the search path of $\Delta_y$ or $\Delta'_y$ DO
   IF $\mathfrak{p}(\mathfrak{v}) \in ]-\infty, \Delta_x] \times [\Delta_y, \Delta'_y]$ DO report $\mathfrak{p}(\mathfrak{v})$
FOR ALL nodes $\mathfrak{v}$ on the path of $\Delta_y$ in $lc(\mathfrak{v}_{split})$ DO
   IF the search path goes left at $\mathfrak{v}$ DO ReportInSubtree($rc(\mathfrak{v}), \Delta_x$)
FOR ALL nodes $\mathfrak{v}$ on the path of $\Delta'_y$ in $rc(\mathfrak{v}_{split})$ DO
   IF the search path goes right at $\mathfrak{v}$ DO ReportInSubtree($lc(\mathfrak{v}), \Delta_x$)
```

Finally we have here:

Proposition 4.3 (Unbounded 2-dimensional range queries) *Let P be a set of n 2-dimensional points. They can be stored in a priority search tree which uses $\mathcal{O}(n)$ storage and has $\mathcal{O}(n \log n)$ construction time. The points in a query range of the form $]-\infty, \Delta_x] \times [\Delta_y, \Delta'_y]$ can be reported in time $\mathcal{O}(\log n + k)$ where k is the number of reported points.*

For proofs see [4].

4.1.4 Segment trees

If we are not interested in storing one-dimensional points but intervals and if we want to compute all intervals which contain a query point (the counterpart of what we have called a one-dimensional range query) we can use segment trees for that. They are especially useful when additional information has to be stored with the intervals. Let $I = \{[\Delta_1, \Delta'_1], [\Delta_2, \Delta'_2], \ldots, [\Delta_n, \Delta'_n]\}$ be a set of n intervals and $p_1, p_2, \ldots, p_m$ be the list of distinct interval endpoints with $p_i < p_j \ \forall i, j \in \{1..m\}$. The *elementary intervals* are now defined as:

$$]-\infty, p_1[,\ [p_1, p_1],\ ]p_1, p_2[,\ [p_2, p_2],\ \ldots,]p_{m-1}, p_m[,\ [p_m, p_m],\ ]p_m, +\infty[$$

A segment tree for I is a binary search tree of the following form:

- Its $2m+1$ leaves correspond to the elementary intervals. The leftmost leaf corresponds to the leftmost elementary interval and so on. The elementary interval corresponding to leaf $\mathfrak{v}$ is denoted by $\mathfrak{Int}(\mathfrak{v})$.
- An internal node $\mathfrak{v}$ corresponds to the interval-union of the elementary intervals $\mathfrak{Int}(\mathfrak{v}')$ corresponding to the leaves $\mathfrak{v}'$ of $\mathfrak{v}$.
- Each node and leaf $\mathfrak{v}$ stores additionally a list $\mathfrak{I}(\mathfrak{v}) \subseteq I$ of intervals. Denoting the parent of $\mathfrak{v}$ by $parent(\mathfrak{v})$, $\mathfrak{I}(\mathfrak{v})$ contains the intervals $[\Delta, \Delta'] \in I$ such that $\mathfrak{Int}(\mathfrak{v}) \subseteq [\Delta, \Delta'] \wedge \mathfrak{Int}(parent(\mathfrak{v})) \not\subseteq [\Delta, \Delta']$.

To construct a segment tree the endpoints of the intervals in I are sorted firstly in $\mathcal{O}(\log n + k)$ time. Then a balanced binary search tree on the elementary intervals is constructed where additionally the interval $\mathfrak{Int}(\mathfrak{v})$ is determined for the nodes $\mathfrak{v}$ in a bottom-up manner taking linear time. The list $\mathfrak{I}(\mathfrak{v})$ is constructed by inserting every interval $[\Delta, \Delta'] \in I$ into $\mathfrak{T}$ via the following function `InsertSegmentTree`. It initially gets the root of the segment tree and the interval to insert:

```
InsertSegmentTree(Node $\mathfrak{v}$, Interval $[\Delta, \Delta']$)
IF $\mathfrak{Int}(\mathfrak{v}) \subseteq [\Delta, \Delta']$ DO
   store $[\Delta, \Delta']$ at $\mathfrak{v}$
ELSE {
   IF $\mathfrak{Int}(lc(\mathfrak{v})) \cap [\Delta, \Delta'] \neq \emptyset$ DO
     InsertSegmentTree($lc(\mathfrak{v}), [\Delta, \Delta']$)
   IF $\mathfrak{Int}(rc(\mathfrak{v})) \cap [\Delta, \Delta'] \neq \emptyset$ DO
     InsertSegmentTree($rc(\mathfrak{v}), [\Delta, \Delta']$)
}
```

Now it is straightforward to give the query algorithm `QuerySegmentTree` which gets a root of a segment tree and a query point p as input. It returns all intervals containing p:

```
QuerySegmentTree(Node $\mathfrak{v}$, Point $p$)
Report all intervals in $\mathfrak{I}(\mathfrak{v})$
IF $\mathfrak{v}$ is not a leaf DO {
   IF $p \in \mathfrak{Int}(lc(\mathfrak{v}))$ DO
     QuerySegmentTree($lc(\mathfrak{v}), p$)
   ELSE
     QuerySegmentTree($rc(\mathfrak{v}), p$)
}
```

We summarise here:

Proposition 4.4 (Segment trees) *A segment tree for a set I containing n intervals uses $\mathcal{O}(n \log n)$ storage and can be built in $\mathcal{O}(n \log n)$ time. We can report all intervals that contain a query point in $\mathcal{O}(\log n + k)$ time where k is the number of reported intervals.*

For proofs see [4].

4.1.5 Testing for intersection

To get rid of hyperrectangle-intersections the very first thing we have to know is which hyperrectangles intersect at all. Several techniques exist here, see e.g. [20] for an overview. We want to solve the following problem:

> Let S be a set hyperrectangles. Given a query hyperrectangle $Q = \prod_{j=1}^{d}[l_j,\ r_j]$ report all $R \in S$ with $Q \cap R \neq \emptyset$.

For every $n \leq d$ we define the n-projection R^n of a hyperrectangle R as $R^n =_{def} \prod_{j=0}^{n}[l_j,\ r_j]$. To distinguish the interval borders of two hyperrectangles R and Q we use use the notation $l_i(R)$ and $r_i(R)$ resp. $l_i(Q)$ and $r_i(Q)$. The solution to the intersection problem is based on two obvious facts:

$$R \cap Q \neq \emptyset \quad \Leftrightarrow \quad [l_d(R), r_d(R)] \cap [l_d(Q), r_d(Q)] \neq \emptyset \ \text{ and } \ R^{d-1} \cap Q^{d-1} \neq \emptyset$$

And we have:

$$[l_d(R), r_d(R)] \cap [l_d(Q), r_d(Q)] \neq \emptyset \quad \Leftrightarrow \quad l_d(R) \in [l_d(Q), r_d(Q)] \ \text{ or } \ l_d(Q) \in [l_d(R), r_d(R)]$$

The latter definition shows that the basic operation is checking if a given one-dimensional query point lies within an interval. Edelsbrunner and Maurer [9] have developed a technique called *orthogonal object intersection* which basically uses the data structures as introduced in this section. Furthermore the use of dynamic extensions of these structures allows insertion and deletion operations. We get here:

Proposition 4.5 (Orthogonal object intersection) *To compute all hyperrectangles $R \in S$ which intersect a query hyperrectangle Q can be done in $\mathcal{O}(\log^d n + k)$ time and $\mathcal{O}(n \log^{d-1} n)$ space. k is the number of reported intersecting hyperrectangle. Preprocessing takes $\mathcal{O}(n \log^d n)$ time and Insertion and deletion takes $\mathcal{O}(\log^d n)$ time.*

See also [19] 5.3 for a general description of this technique. We have see seen that efficient structures and algorithms exist to check for intersection. Using their dynamic versions we can already add and delete hyperrectangles to our set *Permit* and perform intersection checks. What is missing is an efficient algorithm which can remove intersections resulting again in a set hyperrectangles.

4.2 Removing intersections and canonical normalisation

We introduce a new algorithm which transforms a set S such hyperrectangles into a set S' of non-intersecting ones covering the same space, i.e. $\bigcup R \in S = \bigcup R \in S'$. We have seen in chapter 2 that in general there are many possible results conceivable. That is why the algorithm is based on a total order on the axes and will "slice" the rectangles in a maximised way using this order. Given two sets S_1 and S_2 covering the same space we will get equal results provided that the axes-order is the same. That means that the transformation is canonical. To ease the description we allow here that hyperrectangles touch each other, i.e. their contour may intersect. It is straightforward to adapt the algorithm to the discrete case.

4.2.1 *Ordered Maximal Slicing* algorithm

Let S be a given set of d-dimensional hyperrectangles. First a total order on the d axes has to be defined: $a_1 > a_2 > \ldots > a_d$. The algorithm completes recursively the $[l_i,\ r_i]$

```
OMS(Ordinal: $ord$, Set of hyperrectangles: $S$)
$Act$, $Part$, $Res_{sub}$, $Res$:  Set of hyperrectangles := $\emptyset$
IF $S = \emptyset$ RETURN $\emptyset$ ELSE
IF $ord = 0$ RETURN a set with an empty hyperrectangle ELSE
REPEAT
    sweep the $a_{ord}$-axis $\underline{\text{only taking care of the hyperrectangles in } S}$
    stop when hyperrectangles begin or end at a sweep-point $p$
    add all beginning rectangles to $Act$ and subtract all ending ones
    $Res_{sub}$ = OMS($ord - 1$, $Act$)
    set $l_{ord} = p$ for all hyperrectangles in $Res_{sub}$
    set $r_{ord} = p$ for all hyperrectangles in $Part$
    check for all pairs $(R_1,\ R_2) \in Part \times Res_{sub}$ if they can be glued
    with respect to $a_{ord}$, i.e. $[l_i(R_1),\ r_i(R_1)] = [l_i(R_2),\ r_i(R_2)]\ \forall i < ord$
    if so, add $R_1$ to $Res_{sub}$, subtract $R_1$ from $Part$ and $R_2$ from $Res_{sub}$
    $Res = Res \cup Part$
    $Part = Res_{sub}$
UNTIL sweeping of the $a_{ord}$-axis has finished
RETURN $Res$
```

Figure 4.2: Ordered Maximal Slicing Algorithm

interval-borders, beginning at a_1. It sweeps the projections of the hyperrectangle in every single dimension, i.e. projections to the real line. The initial call is OMS(d, S). The algorithm is given in figure 4.2.

Example

To demonstrate the algorithm we apply it to the set of two-dimensional rectangles shown in figure 4.3. We have two axes, $a_1 = x$, $a_2 = y$, and set $a_1 > a_2$. The algorithm is initiated by the call OMS(2, $\{A, B, C, D, E\}$) and therefore starts sweeping $a_2 = y$ axis. the In the following table the resulting sets after every REPEAT-loop are shown. The recursive calls to OMS(1, $\{?\}$) are not shown explicitly, their results come out the same way. When two hyperrectangles glue they are underlined.

The number of resulting hyperrectangles depends on the axes-order. If we had chosen $a_2 > a_1$ six rectangles would have resulted instead of eight in the $a_1 > a_2$ case. Note that it is also possible that the algorithm even optimises the set, i.e. the resulting set of hyperrectangles is smaller that the original one.

It can be seen that equal calls to `OMS` can recursively occur several times. That is why saving the results of the `OMS`-calls can enormously speed up the running time.

p	Act	Res	$Part$	Res_{sub}
1	B, E	$\emptyset$	$\begin{matrix}[2,6]\times[1,?]\\ \underline{[11,13]}\times[1,?]\end{matrix} \longleftarrow \begin{matrix}[2,6]\\ [11,13]\end{matrix}$	$= \texttt{OMS(1, }\{B,E\})$
			$\overset{glue}{\leftrightsquigarrow}$	
3	B, C, E	$[2,6]\times[1,3]$	$\begin{matrix}[11,13]\times[1,?]\\ [2,10]\times[3,?]\end{matrix} \longleftarrow \begin{matrix}\underline{[11,13]}\\ [2,10]\end{matrix}$	$= \texttt{OMS(1, }\{B,C,E\})$
4	B, C, D, E	$[11,13]\times[1,4]$ $[2,10]\times[3,4]$ $\ldots$	$[1,13]\times[4,?] \longleftarrow [1,13]$	$= \texttt{OMS(1, }\{B,C,D,E\})$
5	B, C, E	$[1,13]\times[4,5]$ $\ldots$	$\begin{matrix}[2,10]\times[5,?]\\ \underline{[11,13]}\times[5,?]\end{matrix} \longleftarrow \begin{matrix}[2,10]\\ [11,13]\end{matrix}$	$= \texttt{OMS(1, }\{B,C,E\})$
			$\overset{glue}{\leftrightsquigarrow}$	
6	B, E	$[2,10]\times[5,6]$ $\ldots$	$\begin{matrix}[11,13]\times[5,?]\\ \underline{[2,6]}\times[6,?]\end{matrix} \longleftarrow \begin{matrix}\underline{[11,13]}\\ [2,6]\end{matrix}$	$= \texttt{OMS(1, }\{B,E\})$
			$\overset{glue}{\leftrightsquigarrow}$	
7	B	$[11,13]\times[5,7]$ $\ldots$	$[2,6]\times[6,?] \longleftarrow \underline{[2,6]}$	$= \texttt{OMS(1, }\{B\})$
8	A, B	$[2,6]\times[6,8]$ $\ldots$	$\underline{[2,8]}\times[8,?] \longleftarrow [2,8]$	$= \texttt{OMS(1, }\{A,B\})$
			$\overset{glue}{\leftrightsquigarrow}$	
9	A	no change	$[2,8]\times[8,?] \longleftarrow \underline{[2,8]}$	$= \texttt{OMS(1, }\{A\})$
10	$\emptyset$	$[2,8]\times[8,10]$ $\ldots$	$\emptyset \longleftarrow$	$\texttt{OMS(1, }\{\emptyset\}) = \emptyset$

Correctness of `OMS`

Let S be a set of d-dimensional hyperrectangles. By $R_1 \boxplus R_2$ we denote that hyperrectangle R_1 at most touches R_2, i.e. $R_1 \cap R_2 = \emptyset$ or their contour intersects. Regarding a single

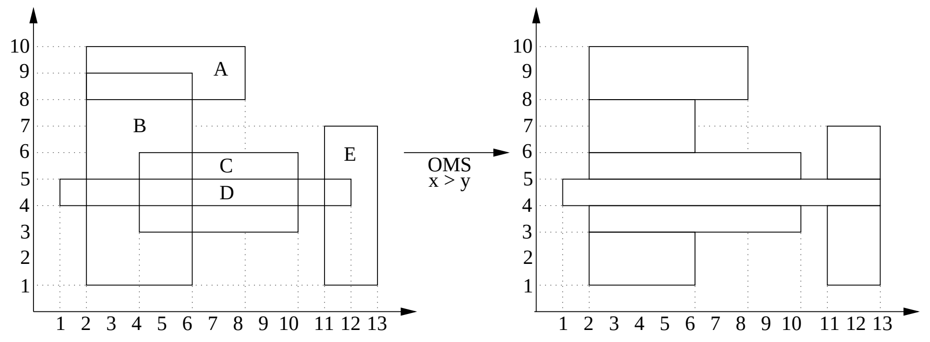


Figure 4.3: Example with $x > y$

call of OMS we have to show that

- $\bigcup_{R \in S} R = \bigcup_{R \in \texttt{OMS}(\texttt{d},S)} R$ (point-wise correctness)
- $\forall R_1, R_2 \in \texttt{OMS}(\texttt{d},S), R_1 \neq R_2 : R_1 \boxplus R_2$ (no overlapping)

For two calls OMS(d,S_1) and OMS(d,S_2) we have to prove the *unique normalisation property*:

$$\bigcup_{R \in S_1} R = \bigcup_{R \in S_2} R \Leftrightarrow \texttt{OMS}(\texttt{d},S_1) = \texttt{OMS}(\texttt{d},S_2)$$

Point-wise Correctness We prove the point-wise correctness for any dimension d and any set S of d-dimensional hyperrectangles by induction on d.

Base case

We have $d = 1$. S is a set of one-dimensional hyperrectangles, i.e. intervals on the real line. For $R \in S$ we have $R = [l_1,\ r_1]$. We have to prove that $\bigcup_{R \in S} R = \bigcup_{R \in \texttt{OMS(1},S)} R$ holds for every S. If $S = \emptyset$ we have $\texttt{OMS(1,}\emptyset\texttt{)} = \emptyset = \bigcup_{R \in \emptyset} R$. If $S \neq \emptyset$ we prove the equality in two steps:

"$\subseteq$" — Given a $R \in S$. We have to show that $R \subseteq \bigcup_{R \in \texttt{OMS(1},S)} R$. Coming to sweep point $l_1(R)$ we have $R \in \mathit{Act}$. We also know that $\mathit{Res}_{sub} = \{R_2\}$ with R_2 being an empty hyperrectangle. If $\mathit{Part} = \emptyset$ OMS sets $l_1(R_2) = l_1(R)$ and we have $\mathit{Part} = \{R_2\}$ before the sweeping continues. If $\mathit{Part} = \{R_1\}$ we know that $l_1(R_1) < l_1(R)$ and that OMS glues R_1 with R_2. The sweeping continues with $\mathit{Part} = \{R_1\}$. In both cases we have $\mathit{Part} = \{R'\}$ with $l_1(R') \leq l_1(R)$ before the sweeping continues. Note that because OMS(0,Act) returns a single (empty) hyperrectangle when $S \neq \emptyset$ we always have $|\mathit{Part}| \leq 1$ in the one-dimensional case. Coming to a sweep point $p < r_1(R)$ where Act changes we know that OMS will glue the empty rectangle obtained by OMS(0,Act) with $R' \in \mathit{Part}$. When sweep point $r_1(R)$ is reached and $\mathit{Act} = \emptyset$ R' ends having $r_1(R') = r_1(R)$. If $\mathit{Act} \neq \emptyset$ the gluing continues. Because eventually Act will be empty R' moves to Res having $r_1(R') \geq r_1(R)$. Together we have $R \subseteq R'$.

"$\supseteq$" — Given a $R \in$ OMS(1,S). We show that $R \subseteq \bigcup_{R \in S} R$. At sweep point $l_1(R)$ we know that $\exists R' \in Act \subseteq S$ with $l_1(R) = l_1(R')$. The sweeping continues with $Part = \{R\}$ and whenever Act changes without becoming empty `OMS` glues the empty rectangle obtained by `OMS(0,`Act`)` with R until sweep point $r_1(R)$ is reached. Here we have $Act = \emptyset$ and R ends. Because Act was not empty from $l_1(R)$ until $r_1(R)$ we know that whole R is covered by hyperrectangle from S.

Inductive step
Having $d > 1$ and given $\bigcup_{R\in S} R = \bigcup_{R\in \texttt{OMS(d-1,}S)} R$ for every set S of (d-1)-dimensional hyperrectangles we have to show that $\bigcup_{R\in S} R = \bigcup_{R\in \texttt{OMS(d,}S)} R$ holds for every set S of d-dimensional ones. If $S = \emptyset$ we have `OMS(d,`$\emptyset$`)` $= \emptyset = \bigcup_{R\in\emptyset} R$. If $S \neq \emptyset$ we prove the equality in two steps:
"$\subseteq$" — Given a $R \in S$. We have to show that $R \subseteq \bigcup_{R\in \texttt{OMS(d,}S)} R$. We know that $R = R^{d-1} \times [l_d(R),\ r_d(R)]$. Coming to sweep point $l_d(R)$ we have $R \in Act$. Using the induction hypothesis we get $R^{d-1} \subseteq \bigcup_{R\in Res_{sub}} R^{d-1} = \bigcup_{R\in \texttt{OMS(d-1,}Act)} R^{d-1}$. That means that there are $R_2 \in Res_{sub}$ with $R^{d-1} \cap R_2^{d-1} \neq \emptyset$ which cover R^{d-1}. If such an R_2 is not glued it gets $l_d(R_2) = l_d(R)$. If R_2 is glued with a $R_1 \in Part$ we know that $R_1^{d-1} = R_2^{d-1}$ holds, therefore we have $R^{d-1} \cap R_2^{d-1} = R^{d-1} \cap R_1^{d-1}$. So finally we have R^{d-1} totally covered by hyperrectangles $R' \in Res_{sub}$ which have $l_d(R') \le l_d(R)$. Res_{sub} becomes $Part$ and the sweeping continues. Coming to a sweep point $p < r_d(R)$ where Act changes two cases can occur with the hyperrectangles R_1 in $Part$ which together cover R up to p. If R_1 is not glued it ends and moves to Res. Then this part $R^{d-1} \cap R_1^{d-1} \times [l_d(R),\ p]$ of R is already part of $\bigcup_{R\in \texttt{OMS(d,}S)} R$. If R_1 is glued with a $R_2 \in$ `OMS(d,`Act`)` this part of R stays saved in the glued hyperrectangle. Because R is still part of Act we know again that all parts of R^{d-1} are covered by the hyperrectangles in Res_{sub} which become $Part$. If `OMS` finally comes to sweep point $r_d(R)$ all parts of $R^{d-1} \times [l_d(R),\ r_d(R)]$ are already saved in Res or part of the hyperrectangles in $Part$. Because all these rectangles $R' \in Part$ eventually end with $r_d(R') \ge r_d(R)$ whole R is finally subset of $\bigcup_{R\in \texttt{OMS(d,}S)} R$.
"$\supseteq$" — Given a $R \in$ `OMS(d,`S`)`. We show that $R \subseteq \bigcup_{R\in S} R$. At sweep point $l_d(R)$ we have $R^{d-1} \in Res_{sub} =$ `OMS(d-1,`Act`)`. The induction hypothesis gives $R^{d-1} \subseteq \bigcup_{R\in Act} R^{d-1}$. Because $Act \subseteq S$ and $\forall R' \in Act : l_d(R') \le l_d(R)$ we already know that $R^{d-1} \times [l_d(R),\ l_d(R)] \subseteq \bigcup_{R\in Act} R \subseteq \bigcup_{R\in S} R$ holds. R moves to $Part$ and the sweeping continues. Coming to a sweep point $p < r_d(R)$ where Act changes R glues with a $R_2 \in Res_{sub} =$ `OMS(d-1,`Act`)`. We know that $R^{d-1} = R_2^{d-1}$ and because the induction hypothesis gives $R_2^{d-1} \subseteq \bigcup_{R\in Act} R^{d-1}$ we get again $R^{d-1} \subseteq \bigcup_{R\in Act} R^{d-1}$. Together we have $R^{d-1} \times [l_d(R),\ p] \subseteq \bigcup_{R\in S} R$. Coming finally to sweep point $r_d(R)$ we know that R is completely covered by the hyperrectangles in S. ■

No Overlapping At first we define that R_1 at most touches R_2:

$$R_1 \boxplus R_2 \ \Leftrightarrow\ \exists i \in \{1, \ldots, d\} : r_i(R_1) \le l_i(R_2) \ \vee\ r_i(R_2) \le l_i(R_1)$$

We immediately get

$$R_1^{n-1} \boxplus R_2^{n-1} \Rightarrow R_1^n \boxplus R_2^n \ \ \forall n > 1 \tag{4.1}$$

We prove that the resulting hyperrectangles do not overlap for any dimension d and any set S of d-dimensional hyperrectangles by induction on d.

Base case

We have $d = 1$. S is a set of one-dimensional hyperrectangles. If $|\texttt{OMS(1,}S\texttt{)}| < 2$ there is nothing to prove. If $|\texttt{OMS(1,}S\texttt{)}| \geq 2$ let $R \in \texttt{OMS(1,}S\texttt{)}$. We show that $R \boxplus R'$ holds for all $R' \in \texttt{OMS(d,}S\texttt{)}$ with $R' \neq R$. We know that $Part = \emptyset$ at sweep point $l_1(R)$ because otherwise `OMS` would glue R with the (only) hyperrectangle in $Part$. Therefore all $R' \in Res$ have $r_1(R') < l_1(R)$. The sweeping continues and `OMS` glues R whenever Act changes (without becoming empty) with the empty hyperrectangle obtained by $\texttt{OMS(0,}Act\texttt{)}$. Coming to sweep point $r_1(R)$ we know that $Act = \emptyset$. R ends and the sweeping continues. All other R' which are added later to Res have $l_1(R') > r_1(R)$.

Inductive step

S is now a set of d-dimensional hyperrectangles. If $|\texttt{OMS(d,}S\texttt{)}| < 2$ there is nothing to prove. If $|\texttt{OMS(d,}S\texttt{)}| \geq 2$ let $R \in \texttt{OMS(d,}S\texttt{)}$. We show that $R \boxplus R'$ holds for all $R' \in \texttt{OMS(d,}S\texttt{)}$ with $R' \neq R$. We know that at sweep point $l_d(R)$ we have $R^{d-1} \in Res_{sub} = \texttt{OMS(d-1,}Act\texttt{)}$. All R' which are already in Res have $r_d(R') < l_d(R)$, therefore we have $R \boxplus R'$ here. For all $R' \in Part$ which do not glue and therefore finish and move to Res we have $r_d(R') = l_d(R)$, $R \boxplus R'$ holds here too. The induction hypothesis gives $R^{d-1} \boxplus R_2^{d-1}$ $\forall R_2 \in Res_{sub}$ with $R_2 \neq R$. If there is a gluing of such a R_2 with a $R_1 \in Part$ we know that $R_1^{d-1} = R_2^{d-1}$, therefore we know that R^{d-1} at most touches the (d-1)-projections of the other hyperrectangles in Res_{sub} which then becomes $Part$. Coming to a sweep point $p < r_d(R)$ where Act changes. For all $R' \in Part$ which do not glue and therefore move to Res we know by (4.1) that $R \boxplus R'$. R glues with a $R_2 \in Res_{sub} = \texttt{OMS(d-1,}Act\texttt{)}$. We know that $R^{d-1} = R_2^{d-1}$ and the induction hypothesis gives $R_2^{d-1} \boxplus R_{2'}^{d-1}$ $\forall R_{2'} \in Res_{sub}$ with $R_2 \neq R_{2'}$. If there is a gluing of such a $R_{2'}$ with a $R_1 \in Part$ we know that $R_1^{d-1} = R_2^{d-1}$ and therefore we know again that R^{d-1} at most touches the (d-1)-projections of the hyperrectangles which become the new $Part$-set. Coming finally to sweep-point $r_d(R)$. R ends and moves to Res. By (4.1) we know again that $R \boxplus R'$ for all $R' \in Part$ which end here too. If an $R_1 \in Part$ glues with a $R_2 \in Res_{sub} = \texttt{OMS(d,}Act\texttt{)}$ we have $R_1^{n-1} = R_2^{n-1}$ and because $R^{n-1} \boxplus R_1^{n-1}$ we also have $R^{n-1} \boxplus R_2^{n-1}$. By (4.1) we know again that R at most touches the glued hyperrectangles when it eventually ends. For all $R_2 \in Res_{sub}$ which do not glue we have $r_d(R) = l_d(R_2)$. All other $R' \in \texttt{OMS(d,}S\texttt{)}$ which are added later to Res will have $l_d(R') > r_d(R)$. Finally for all $R' \in \texttt{OMS(d,}S\texttt{)}$ with $R' \neq R$ we have shown that $R \boxplus R'$ holds. ■

Unique Normalisation Property Because of the point-wise correctness we already know that

$$\bigcup_{R \in S_1} R = \bigcup_{R \in S_2} R \;\Leftarrow\; \texttt{OMS(d,}S_1\texttt{)} = \texttt{OMS(d,}S_2\texttt{)}$$

holds. We prove the opposite direction for any dimension d and any sets S_1, S_2 of d-dimensional hyperrectangles by induction on d.

Base case
We have $d = 1$. $\bigcup_{R\in S_1} R = \bigcup_{R\in S_2} R \Rightarrow$ `OMS`$(1, S_1) =$ `OMS`$(1, S_2)$ follows directly out of the point-wise correctness and the fact that $\forall R_1, R_2 \in$ `OMS(1,`S_1`)` resp. $\forall R_1, R_2 \in$ `OMS(1,`S_2`)` holds $R_1 \cap R_2 = \emptyset$.

Inductive step
S_1 and S_2 are sets of d-dimensional hyperrectangles. We know that $\bigcup_{R\in S_1} R = \bigcup_{R\in S_2} R$ holds which induces that $min\left(\bigcup_{R\in S_1}[l_d,\ r_d]\right) = min\left(\bigcup_{R\in S_2}[l_d,\ r_d]\right)$. That means that `OMS`$_1$`(d,`$S_1$`)` and `OMS`$_2$`(d,`$S_2$`)` have the same sweep point p_1 where *Act* changes the first time. We use the index at `OMS`$_{1|2}$ and their variables to distinguish both calls of `OMS`. We assume that $\bigcup_{R\in S_1} R = \bigcup_{R\in S_2} R \Rightarrow$ `OMS(d-1,`S_1`)` $=$ `OMS(d-1,`S_2`)` holds for all sets S_1 and S_2 of (d-1)-dimensional hyperrectangles. Coming to this first sweep point p_1 where `OMS`$_1$ and `OMS`$_2$ stop simultaneously because *Act* changes we know that $\bigcup_{R\in Act_1} R^{n-1} = \bigcup_{R\in Act_2} R^{n-1}$. The induction hypothesis gives `OMS`$_1$`(d-1,`Act_1`)` $=$ `OMS`$_2$`(d-1,`Act_2`)`. Because $Part_1 = Part_2 = \emptyset$ nothing is glued and both calls to `OMS` end up in the same set *Part* before the sweeping continues. If `OMS`$_1$ and `OMS`$_2$ stop again simultaneously because *Act* changes the induction hypothesis guarantees again that both have equal variable-states before the sweeping continues. Coming to a sweep point p where one of them, let us say `OMS`$_1$, stops because its Act_1 changes. By Act_1^{old} we denote the state of Act_1 before p was reached. `OMS`$_2$ does not stop because no $R \in S_2$ begin or end at p. But therefore we know that $\bigcup_{R\in Act_1} R^{n-1} = \bigcup_{R\in Act_1^{old}} R^{n-1}$. The induction hypothesis gives here `OMS`$_1$`(d-1,`Act_1`)` $=$ `OMS`$_1$`(d-1,`Act_2^{old}`)`. That is why we know that for all $R_2 \in Res_{sub} =$ `OMS`$_1$`(d-1,`Act_1`)` there exists exactly one $R_1 \in Part$ with $R_1^{n-1} = R_2^{n-1}$ and vice versa (a bijection). Therefore all these pairs glue and no one ends which means that $Part_2$ does not change. Together we have seen that both calls to `OMS` result in the same set *Res*. ∎

Complexity of `OMS`

Without any optimisation like the mentioned saving of intermediate results of recursive calls the algorithm performs less than $2n^d$ recursive calls with n being the number of initial hyperrectangles. Future research is needed to gain a deeper understanding of the real complexity because the axes-order is a dominant factor and the distribution of the hyperrectangles in the space matters to.

4.3 Summary

We have everything together to implement the $\oplus$-operator of the `GetInterBU` algorithm. We add a new *Permit*-hyperrectangle $\langle$`L(i)`$\rangle$ to the geometrical data structure representing the *Permit*-set via the insert-operation. Then we perform the orthogonal object intersection algorithm to gain a set $I \subseteq Permit$ of intersecting hyperrectangles. Next we apply `OMS` on I to remove the intersections. Finally we exchange the intersection-free result with I and get so a canonical, non-intersecting set *Permit* as wanted. The $\ominus$-operator can also be implemented by using a slightly modified version of the `OMS`-algorithm, we just have to interpret *Deny*-hyperrectangles as interval-gaps on the axes-projections. Therefore we can implement all the algorithms of chapter 2.

Because the size of the result of `OMS` depends highly on the axes-order it is difficult to compare our algorithm with other ones. Future research needs to be done to develop heuristics for finding efficient orders for specific fields like firewall rule-lists. A dynamic adaptation of the order to optimise the result is also conceivable.

Sanchez and Condell [21] introduce their *decorrelation* algorithm which also removes intersections from a given set of hyperrectangles. Their approach is not canonical. Some tools for firewall analysis are based on this technique, e.g. [11].

Chapter 5

Summary and Outlook

Analysing firewalls turned out to be a more complex process than one may expect when considering firewall semantics – sets of permitted and denied IP-packets. A transformation into a new formalism is necessary to perform analytical tasks. Furthermore a back-translation into a tabular representation is essential to exploit the computed results.

A formal foundation based on set-theory was evolved. It laid the general fundament for every approach and constituted a pattern for suitable data structures and algorithms. With the help of category theory the resulting implementable structures could be modelled as objects of categories. Showing the isomorphisms between these categories proved the structural correctness of the implementations.

The classical formalism for describing a packet filter is a rule-list. It has two properties which make it unsuited to perform analysis, it is correlated and not canonical. In this work two main approaches for firewall analysis have been introduced – a geometrical and a logical one which mainly traces back to the work done by Scott Hazelhurst. Both approaches are still in development. Neither experimental nor theoretical results exist which would allow a fair comparison. And both highly depend on heuristic data structures (BDDs) or algorithms (OMS). This suggests a concentration on real-life comparison because worst case complexity of these structures is usually quite bad (see e.g. [5]). But there seems to be an interesting interconnection between the variable order of BDDs and the axes order of the hyperrectangles. The basic observation is that the set of truth-assignments which evaluate a formula φ to true can be interpreted as points in space.

The geometrical approach is a promising new concept for analysing firewall rule-lists. Well-studied data structures and algorithms exist which allow an efficient implementation. In addition there is a direct and simple interconnection between filter-rules and hyperrectangles. Furthermore the `OMS`-algorithm is a real alternative to the decorrelation strategy [21] which is implemented in several tools like [11]. Beside firewall-analysis it can be applied to a broad range of problems, e.g. hardware design [22]. After the theoretical background was given in this work a real-world implementation would finally show the applicability and efficiency of a geometry-based analysis. Such an implementation may also help answering questions like *in which cases is canonical normalisation really* *necessary/useful?*

The basis for a generalisation to multiple actions which applies to every approach was given too. Such a generalisation may become an important topic in the growing complexity and diversity of the Internet protocol-structures and security policies. An extension to stateful packet filtering and dealing with IPv6 are the next big challenges.

Another field of interest is the development of a substitute for the classical rule-list syntax which has a lot of disadvantages as seen throughout the work. Hyperrectangles are basically rules seen geometrically and BDDs are everything but human-friendly. Hence both approaches do not yield a candidate here at first sight.

A recent tendency is to hide structured information in a single protocol, e.g. web services use http as its carrier. It will be seen how firewalls deal with this and how analysis can be performed here. But as things become more and more complex the availability of applicable analysis tools will become vital.

Appendix A

Mathematical notions

Set theory

Definition A.1 (Powerset) 2^H *denotes the* powerset *of set H.*

Definition A.2 (Disjoint Sets) *Two sets A and B are* disjoint *iff $A \cap B = \emptyset$.*

Definition A.3 (Partition) *Let H be a set and $(\eta_i)_{i \in I}$ a set-family such that $\forall i \in I : \eta_i \in 2^H$. $(\eta_i)_{i \in I}$ is a* partition *of H iff*

- $\bigcup_{i \in I} \eta_i = H$,
- $\forall j, k \in I,\ j \neq k :\ \eta_j$ *and* η_k *are disjoint.*

Definition A.4 (Ordered Union) *Let $(\eta_i)_{i \in I}$ and $(\vartheta_i)_{i \in I}$ be set-families over the same index-set I with $|I| = n$. Given a total order $<$ over I with $i_1 < \ldots < i_n < \top$, $\{i_1, .., i_n\} = I$, the* ordered union *$(\xi_i)_{i \in I}$, written $(\xi_i)_{i \in I} = (\eta_i)_{i \in I} \cup_< (\vartheta_i)_{i \in I}$, is recursively defined as follows:*

- $\xi_\top = \emptyset$
- $\xi_{i_j} = (\eta_{i_j} \cup \vartheta_{i_j}) \setminus \bigcup_{k > i_j} \xi_k$

Proposition A.1 (Ordered Union preserves Partitions) *Let $(\eta_i)_{i \in I}$ and $(\vartheta_i)_{i \in I}$ be partitions of a set H. Then $(\xi_i)_{i \in I} = (\eta_i)_{i \in I} \cup_< (\vartheta_i)_{i \in I}$ is also a partition of H.*

Proof:

- *We prove that $\bigcup_{i \in I} \xi_i = H$. Let e be an arbitrary element of H. Because $(\eta_i)_{i \in I}$ is a partition of H we know that there exists a j with $e \in \eta_{i_j}$. It follows that $e \in \eta_{i_j} \cup \vartheta_{i_j}$. Furthermore we know that $\xi_{i_j} = (\eta_{i_j} \cup \vartheta_{i_j}) \setminus \bigcup_{k > i_j} \xi_k$. In the case that also $e \in \bigcup_{k > i_j} \xi_k$ it follows immediately that $e \in \bigcup_{i \in I} \xi_i$. If $e \notin \bigcup_{k > i_j} \xi_k$ it follows that $e \in \xi_{i_j}$ which also induces that $e \in \bigcup_{i \in I} \xi_i$.*

- *We prove that $\forall j, k \in I,\ j \neq k :\ \xi_j$ and ξ_k are disjoint. To do so we prove that every ξ_{i_j} is disjoint to all $\xi_k,\ k > i_j$. This follows immediately out of the fact that $\xi_{i_j} = (\eta_{i_j} \cup \vartheta_{i_j}) \setminus \bigcup_{k>i_j} \xi_k$.* ■

Category theory

Definition A.5 (Category) *A* category $\mathbf{C} = (Ob_{\mathbf{C}}, Mor_{\mathbf{C}}, \circ, id)$ *consists of*

1. *a class $Ob_{\mathbf{C}}$ of objects,*
2. *for every two objects $A, B \in Ob_{\mathbf{C}}$ a set $Mor_{\mathbf{C}}(A, B)$, the morphisms,*
3. *for every three objects $A, B, C \in Ob_{\mathbf{C}}$ an operation*

   $$\circ : Mor_{\mathbf{C}}(A, B) \times Mor_{\mathbf{C}}(B, C) \to Mor_{\mathbf{C}}(A, C)$$

   the composition operator*,*
4. *for every object $A \in Ob_{\mathbf{C}}$ a morphism $id_A \in Mor_{\mathbf{C}}(A, A)$, the* identity*,*

such that the following laws hold:

Associative law *— for all $A, B, C, D \in Ob_{\mathbf{C}}$ and all $f \in Mor_{\mathbf{C}}(A, B)$, $g \in Mor_{\mathbf{C}}(B, C)$ and $h \in Mor_{\mathbf{C}}(C, D)$ holds:*

$$(h \circ g) \circ f = h \circ (g \circ f)$$

Identity law *— For all $A, B \in Ob_{\mathbf{C}}$ and all $f \in Mor_{\mathbf{C}}(A, B)$ holds:*

$$f \circ id_A = f \quad \text{and} \quad id_B \circ f = f$$

If $f \in Mor_{\mathbf{C}}(A, B)$ we write also $f : A \to B$.

Definition A.6 (Isomorphism) *A morphism $i : A \to B$ is called an* isomorphism *iff there is a morphism $j : B \to A$ with $j \circ i = id_A$ and $i \circ j = id_B$. Two objects $A, B \in Ob_{\mathbf{C}}$ are called* isomorphic*, written $A \cong B$, iff there is an isomorphism $i : A \to B$.*

Definition A.7 (Terminal and initial object) *Let $\mathbf{C}$ be a category.*
An object $\mathbf{1} \in Ob_{\mathbf{C}}$ is called terminal *in $\mathbf{C}$ iff for every object $X \in Ob_{\mathbf{C}}$ there exists exactly one morphism $!_X : X \to \mathbf{1}$.*
An object $\mathbf{0} \in Ob_{\mathbf{C}}$ is called initial *in $\mathbf{C}$ iff for every object $X \in Ob_{\mathbf{C}}$ there exists exactly one morphism $?_X : \mathbf{1} \to X$.*
Given two terminal or two initial objects A and B we have $A \cong B$, i.e. terminal and initial object are unique up to isomorphism.

Definition A.8 (Product) *Let $\mathbf{C}$ be a category and $A, B \in Ob_{\mathbf{C}}$.*
A product $(A \times B, \pi_1, \pi_2)$ *of A and B consists of*

- *an object $A \times B$ together with*
- *two morphisms $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$*

such that for every object X together with two morphisms $f : X \to A$ and $g : X \to B$ there is exacly one morphism $\langle f, g\rangle : X \to A \times B$ with $\pi_1 \circ \langle f, g\rangle = f$ and $\pi_2 \circ \langle f, g\rangle = g$. Products are unique up to isomorphism.

Definition A.9 (Cartesian category) *A category $\mathbf{C}$ is called* cartesian *iff it contains a terminal object and a product $(A \times B, \pi_1, \pi_2)$ for every pair $A, B \in Ob_{\mathbf{C}}$.*

Definition A.10 (Product of morphisms) *Let $(A \times B, \pi_1, \pi_2)$ and $(A' \times B', \pi_1', \pi_2')$ be products of A and B resp. A' and B'. Given two maps $f : A \to A'$ and $g : B \to B'$ we gain a morphism $f \times g : A \times B \to A' \times B'$ defined by $f \times g = \langle f \circ \pi_1, g \circ \pi_2\rangle$.*

Definition A.11 (Coproduct) *Let $\mathbf{C}$ be a category and $A, B \in Ob_{\mathbf{C}}$.*
A coproduct *$(A + B, \iota_1, \iota_2)$ of A and B consists of*

- *an object $A + B$ together with*
- *two morphisms $\iota_1 : A \to A + B$ and $\iota_2 : B \to A + B$*

such that for every object X together with two morphisms $f : A \to X$ and $g : B \to X$ there is exactly one morphism $[f, g] : A + B \to X$ with $[f, g] \circ \iota_1 = f$ and $[f, g] \circ \iota_2 = g$. Coproducts are unique up to isomorphism.

Definition A.12 (Exponent) *Let $\mathbf{C}$ be a cartesian category and $A, B \in Ob_{\mathbf{C}}$.*
An exponent *$(B^A, \epsilon_{A,B})$ of A and B consists of*

- *an object B^A together with*
- *a morphism $\epsilon_{A,B} : A \times B^A \to B$*

such that for every object C and all morphisms $f : A \times C \to B$ there exists exacly one morphism $f^\sharp : C \to B^A$ with $\epsilon_{A,B} \circ (id_A \times f^\sharp) = f$.
Exponents are unique up to isomorphism.

Definition A.13 (Cartesian closed category) *A cartesian category $\mathbf{C}$ is called* cartesian closed *iff it contains an exponent $(B^A, \epsilon_{A,B})$ for every pair $A, B \in Ob_{\mathbf{C}}$.*

Definition A.14 (Functor) *Let $\mathbf{C}$ and $\mathbf{D}$ be a categories.*
A functor *$F = (F_{Ob}, F_{Mor}) : \mathbf{C} \to \mathbf{D}$ consists of*

- *an operation $F_{Ob} : Ob_{\mathbf{C}} \to Ob_{\mathbf{D}}$ and*
- *for every two objects $A, B \in Ob_{\mathbf{C}}$ an operation*

$$F_{Mor} : Mor_{\mathbf{C}}(A, B) \to Mor_{\mathbf{D}}(F_{Ob}(A), F_{Ob}(B))$$

such that

- *for all* **C***-morphisms* $f : A \to B$ *and* $g : B \to C$ *holds*
$$F_{Mor}(g \circ^{\mathbf{C}} f) = F_{Mor}(g) \circ^{\mathbf{D}} F_{Mor}(f)$$
- *and for all* $A \in Ob_{\mathbf{C}}$ *holds*
$$F_{Mor}(id^{\mathbf{C}}_A) = id^{\mathbf{D}}_{F_{Ob}(A)} \ .$$

Definition A.15 (Composition of functors) *The* composition $G \circ F$ *of two functors* $F : \mathbf{C} \to \mathbf{D}$ *and* $G : \mathbf{D} \to \mathbf{E}$ *is defined as:*

$$\begin{array}{rcll} (G \circ F)(A) & = & G(F(A)) & (A \in Ob_{\mathbf{C}}) \\ (G \circ F)(f) & = & G(F(f)) & (f : A \to B \text{ in } \mathbf{C}) \end{array}$$

The functor $Id_{\mathbf{C}} : \mathbf{C} \to \mathbf{C}$ *is defined as:*

$$\begin{array}{rcll} Id_{\mathbf{C}}(A) & = & A & (A \in Ob_{\mathbf{C}}) \\ Id_{\mathbf{C}}(f) & = & f & (f : A \to B \text{ in } \mathbf{C}) \end{array}$$

Definition A.16 (Isomorphic categories) *Two categories* **C** *and* **D** *are called isomorphic iff there exist functors* $I : \mathbf{C} \to \mathbf{D}$ *and* $J : \mathbf{D} \to \mathbf{C}$ *such that*

$$J \circ I = Id_C \text{ and } I \circ J = Id_D$$

Bibliography

[1] A. Asperti, G. Longo. *Categories, Types, and Structures. An Introduction to Category Theory for the Working Computer Scientist*. Foundations of Computing. MIT Press 1991.

[2] A. Attar. *Performance Characteristics of BDD-Based Packet Filters*. Technical Report TR-Wits-CS-2002-2, Department of Computer Science, University of the Witwatersrand, Johannesburg, November 2001.

[3] A. Attar, S. Hazelhurst, R. Sinnappan. *Packet Classification Based on Logic Representations*. Draft, Department of Computer Science, University of the Witwatersrand, Johannesburg.

[4] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf. *Computational Geometry. Algorithms and Applications*. 2nd Edition. Springer 2000.

[5] B. Bollig, I. Wegener. *Improving the Variable Ordering of OBDDs is NP-Complete*. IEEE Transactions on Computers, 45(9):993-1002, September 1996.

[6] R. E. Bryant. *Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams*. ACM Computing Surveys, 24(3):293-318, September 1992.

[7] M. Christiansen, E. Fleury. *Using IDDs for Packet Filtering*. BRICS Report Series, RS-02-43, October 2002.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest. *Introduction to Algorithms*. MIT Press 1990.

[9] H. Edelsbrunner, H. A. Maurer. *On the intersection of orthogonal objects*. Information Processing Letters Vol. 13:177-181, 1981.

[10] H. Ehrig, B. Mahr, F. Cornelius, M. Große-Rhode, P. Zeitz. *Mathematisch-strukturelle Grundlagen der Informatik*. 2.Auflage. Springer 2001.

[11] P. Eronen, J. Zitting. *An expert system for analyzing firewall rules*. Proceedings of the 6th Nordic Workshop on Secure IT Systems, 100-107, Copenhagen, November 2001.

[12] S. Hazelhurst. *Algorithms for Analysing Firewall and Router Access Lists.* Technical Report TR-Wits-CS-1999-5, Department of Computer Science, University of the Witwatersrand, Johannesburg, July 1999.

[13] S. Hazelhurst, A. Fatti, A. Henwood. *Binary Decision Diagram Representations of Firewall and Router Access Lists.* Technical Report TR-Wits-CS-1998-3, Department of Computer Science, University of the Witwatersrand, Johannesburg, October 1998.

[14] M. R. A. Huth, M. D. Ryan. *Logic in Computer Science. Modelling and reasoning about systems.* Cambridge University Press 2000.

[15] F. W. Lawvere, S. H. Schanuel. *Conceptual Mathematics. A first introduction to categories.* Cambridge University Press 1998.

[16] A. Martini. *Elements of Basic Category Theory.* Technical Report 96-5, Faculty IV, Technical University of Berlin, 1996.

[17] A. Mayer, A. Wool, E. Ziskind. *Fang: A Firewall Analysis Engine.* Proceeding of the 21st IEEE Symposium on Security and Privacy, Oakland, CA, May 2000.

[18] K. Mehlhorn. *Sorting and Searching.* EATCS Monographs on Theoretical Computer Science 1. Springer 1984.

[19] K. Mehlhorn. *Multi-dimensional Searching and Computational Geometry.* EATCS Monographs on Theoretical Computer Science 3. Springer 1984.

[20] M. D. Petty, A. Mukherjee. *Experimental Comparison of d-Rectangle Intersection Algorithms Applied to HLA Data Distribution.* Fall Simulation Interoperability Workshop (SIW), The Society for Modeling and Simulation International (SCS), IEEE, Orlando, 1997.

[21] L. A. Sanchez, M. N. Condell. *Security Policy Protocol.* Internet-Draft ietf-ipsp-spp-00, `http://www.ietf.org/proceedings/00jul/I-D/ipsp-spp-00.txt`, July 2000.

[22] G. Spiegel. *Bestimmung möglicher Fabrikationsfehler aus dem Schaltungslayout.* Verlag Dr. Kŏvac 1995.

[23] U. Wolter. *On Corelations, Cokernels, and Coequations.* Proceedings of CMCS'2000, Electronic Notes in Theoretical Computer Science Vol. 33, `http://www.elsevier.nl/gej-ng/31/29/23/25/23/58/tcs33016.ps`, March 2000

[24] E. D. Zwicky, S. Cooper, D. B. Chapman. *Building Internet Firewalls.* 2nd Edition. O'Reilly 2000.