Deterministic Discrete Modeling

Formal Semantics of Firewalls in Isabelle/HOL

Cornelius Diekmann, M.Sc.

Dr. Heiko Niedermayer
Prof. Dr.-Ing. Georg Carle

Lehrstuhl für Netzarchitekturen und Netzdienste
Institut für Informatik
Technische Universität München

Version: April 29, 2014

Agenda

Slides: 63

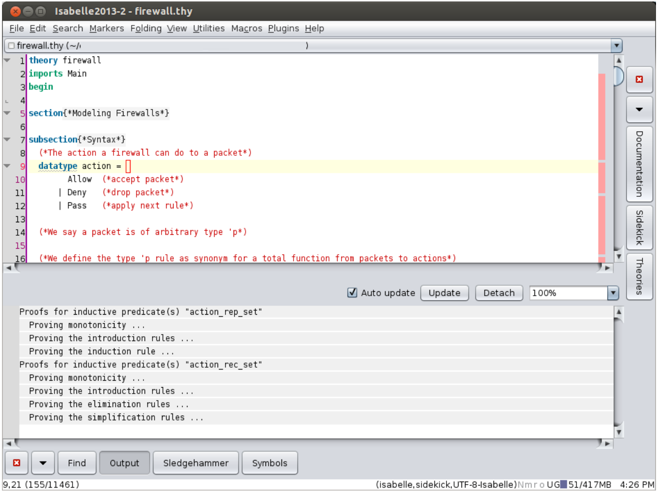
About Isabelle/HOL

- Generic proof assistant
- For
  - interactive theorem proving
  - in higher-order logics (HOL)

http://isabelle.in.tum.de/

```
$ wget http://isabelle.in.tum.de/dist/Isabelle2013-2_linux.tar.gz
$ sha1sum Isabelle2013-2_linux.tar.gz
cb8dca7fdc9090b7640121ca6b575a714a62a492 Isabelle2013-2_linux.tar.gz
$ tar -xzf Isabelle2013-2_linux.tar.gz
$ ./Isabelle2013-2/bin/isabelle jedit &
```

About Isabelle/HOL



Note

Note: The theory files are relevant for the exam! We will soon leave the slide set and interactively work with Isabelle/HOL.

About this Chapter

- Please make sure you have a laptop and Isabelle/HOL with you (alternatively, find a fellow student who does)
- There is a firewall2.thy on the website.

  This is the accompanying exercise.

  It contains several TODOs. Try to solve them after each lecture.

Propositional Logic, First Order Logic

- Propositional Logic

  $(a \longrightarrow b) \vee (b \longrightarrow a)$

- First Order Logic (FOL)

  FOL = Propositional Logic + Quantifiers

- Example FOL

  $\forall a.\ \exists b.\ (a \longrightarrow b) \vee (b \longrightarrow a)$

  $(a \longrightarrow b) \vee (b \longrightarrow a)$

HOL – Higher-Order Logic

- Higher-Order Logic (HOL)

  HOL = Functional Programming + Logic

- Example FOL

  $(a \longrightarrow b) \vee (b \longrightarrow a)$

- Example Lambda Calculus

  $(\lambda x.\ x + 4)\ 38$

- Example Transitive Closure of a Relation R

  $(a, b) \in R \wedge (b, c) \in R \longrightarrow (a, c) \in R^+$

Isabelle/HOL

- Isabelle/HOL: A Proof Assistant for Higher-Order Logic

  We model and reason in HOL

- Example FOL

  $(a \longrightarrow b) \vee (b \longrightarrow a)$

- In Isabelle/HOL

  `lemma` “$(a \longrightarrow b) \vee (b \longrightarrow a)$” `by auto`

- Lambda Calculus

  $(\lambda x.\ x + 4)\ 38$

- In Isabelle/HOL

  `lemma` “$(\lambda x.\ x + 4)\ 38 = 42$” `by simp`

Isabelle/HOL Examples

- Example

  `lemma` "$(a \longrightarrow b) \vee (b \longrightarrow a)$" `by auto`

- Terms are written in ""
- `lemma` starts a lemma
- it can be proven automatically `by auto`
- other automated proof methods
  - `simp` – the simplifier
  - `auto` – simplification, logic, sets, may return unprovable goals
  - `blast` – complete for FOL
  - ...

`lemma` "$(a \longrightarrow b) \vee (b \longrightarrow a)$" `by simp`

`lemma` "$(a \longrightarrow b) \vee (b \longrightarrow a)$" `by blast`

`lemma` "$(a \longrightarrow b) \vee (b \longrightarrow a)$" `by fastforce`

Isabelle/HOL Examples

- About the transitive closure

  `lemma` "$(a,b) \in R \wedge (b,c) \in R \Longrightarrow (a,c) \in R^{+}$" `by auto`

- About the reflexive transitive closure

  `lemma` "$(a,a) \in R^{*}$" `by simp`

- FOL is modeled in HOL
  - HOL implication: $\Longrightarrow$
  - FOL implication: $\longrightarrow$

  `lemma` "$(a,b) \in R \wedge (b,c) \in R \longrightarrow (a,c) \in R^{+}$" `by auto`

Isabelle/HOL

- Implications associate to the right

  $A \Longrightarrow B \Longrightarrow C$ means $A \Longrightarrow (B \Longrightarrow C)$
- This is equal to

  $A \wedge B \Longrightarrow C$
- or

  `lemma assumes` A `and` B `shows` C

- Homework: prove

$$(a \longrightarrow b \longrightarrow c) \longleftrightarrow (a \wedge b \longrightarrow c)$$

  on paper and in Isabelle/HOL

Isabelle/HOL Notation

- FOL all quantifier: $\forall$
- HOL all quantifier: $\bigwedge$
- Example
  - `lemma ‘‘`$\forall x.\ x$`’’`
    note: obviously wrong, counterexample `x = False`
  - `apply(rule)`

    produces subgoal $\bigwedge x.\ x$

    From FOL to HOL
- “free” variable (can be instantiated): `?x`
- Example
  - `lemma x: ‘‘a = a’’ by simp`
  - `thm x` gives you `?a = ?a`
  - That is, you can instantiate lemma `x` for arbitrary `?a`
  - Example: `thm a[of "foo"]` is `foo = foo`
- Concluding Example: Inspect `allI`

Types & Functional Programming

Does the set of all sets contain itself?

Does the set of all sets contain itself?

- If not: It is obviously not the set of all sets! ⚡

Does the set of all sets contain itself?

- If not: It is obviously not the set of all sets! ↯
- If it does: Let A be the subset that contains all sets that do not contain themselves.
  - Is $A \in A$?
  - If not: A is obviously not the set of all sets that do not contain themselves! ↯
  - If it does: A contains a set that contains itself! ↯

Types

- Why is HOL typed?
- An example of untyped mathematics:

Does the set of all sets contain itself?

- If not: It is obviously not the set of all sets! ⚡
- If it does: Let A be the subset that contains all sets that do not contain themselves.
  - Is $A \in A$?
  - If not: A is obviously not the set of all sets that do not contain themselves! ⚡
  - If it does: A contains a set that contains itself! ⚡

Types

- Typed sets
  - nat set e.g., $1 \in \{1,2\}$
  - nat set set e.g., $\{1,2\} \in \{\{1,2\},\{\}\}$
  - nat set set set e.g., $\{\{1,2\},\{\}\} \in \{\{\{1,2\},\{\}\},\{\{1,2,3,4\}\}\}$
- $\in$ is of type $\underbrace{\mathit{nat}}_{1}\ \underbrace{\Rightarrow}_{\in}\ \underbrace{\mathit{nat\ set}}_{\{1,2\}} \underbrace{\Rightarrow \mathit{bool}}_{\mathit{is\ true}}$

- The concept of *set of all sets* is not well-typed
- $\mathcal{M}$ = *the set of all sets*
- What is the type of $\mathcal{M}$?
  - There is no (finite) type for $\mathcal{M}$!
- Even if there were a type for $\mathcal{M}$,
  $\mathcal{M} \in \mathcal{M}$ is an obvious type error

- HOL is typed!

Polymorphic Types

- Arbitrary types *'a*, *'b*, ...
- $\in$ is of type $'a \Rightarrow {'a}\ set \Rightarrow bool$

- For example
  - $False \in \{True, False\}$, bool set
  - $(a \longrightarrow b) \in \{(a \longrightarrow b), x\}$, bool set
  - $``Hello'' \in \{``Hello'', ``World''\}$, string set
  - $apple \in \{apple, banana\}$, $'a$ set,
    apple and *banana* are of arbitrary type
- `value` "$\{True,\ False\} :: bool\ set$"

Functional Programming

Demo: Introduction.thy

```
fun find_fives :: "nat list ⇒ nat list" where
 "find_fives [] = []"
|"find_fives (x#xs) =
    (if x = 5 then 5#find_fives xs else find_fives xs)"
```

```
value "find_fives [1,3,5,8,5,2,5]"
```

```
lemma "find_fives [1,3,5,8,5,2,5] = [5,5,5]" by simp
```

Library function `filter`

```
filter :: "('a ⇒ bool) ⇒ 'a list ⇒ 'a list"
```

```
lemma "find_fives l = filter (λx. x = 5) l"
 by(induction l, simp_all)
```

Total Functions

- A *total* function is function that is defined for all input values
  - For computer scientists:

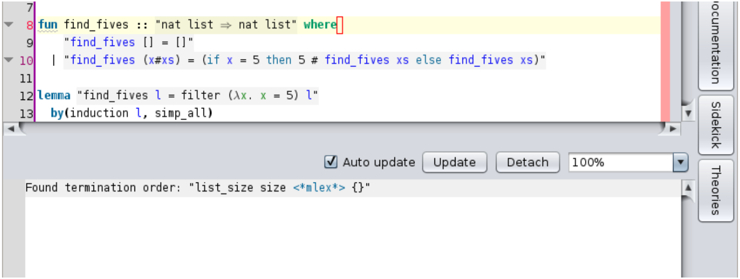
    A total function must always terminate!
- Written $'a \Rightarrow\ 'b$
- HOL is a total logic!
- Why total functions?
- Assume f of type $nat \Rightarrow nat$.
  $f\ n = \texttt{while(true)}\{\}\ \texttt{return 0}$

  1. $f\ n = \mathrm{does-not-terminate}$
  2. $\mathrm{does-not-terminate} = \mathrm{does-not-terminate} + 1$
  3. from (1) and (2): $f\ n = (f\ n) + 1$
  4. subtracting $f\ n$ on both sides: $0 = 0 + 1$
  5. $0 = 1$

- HOL is total and every function needs a proof for that!

Total Functions

Usually, the *total*-proof is for free.



Here, the size of the list is used for the automatic proof.

1. The function obviously terminates for the empty list
2. If the list consists of $x :: xs$, all subsequent calls to *find_fives* only get *xs*.

Wrap-Up

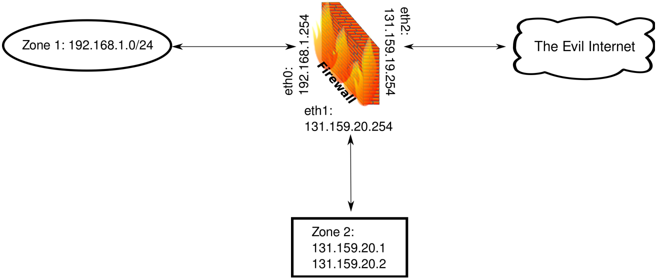
- HOL = Functional Programming + Logic
- Strong Typing
- Total functions
- Proofs!

Modeling Firewalls

Firewalls – Introduction

- Access control on the network level
- Often positioned between a trusted and a less trusted network
  - E.g. between internal network and Internet

Example:



Firewalls

- Decides per packet what to do
  - Allow – Forward the packet
  - Deny – Drop the packet
- A firewall is configured by a ruleset. Example:

#	Src IP	Dst IP	Proto	Src Port	Dst Port	Action
A	Alice	Gabby	TCP	*	80	Allow
B	Alice	8.8.8.8	UDP	*	53	Allow
C	*	*	*	*	*	Deny

Textual explanation on next slide (walls of text)

Firewalls

- Rules are processed consecutively for every packet
  Assume a packet arrives
  1. Is the source Alice and the destination Gabby and the protocol TCP and the destination port 80 (HTTP)?
     If yes, the firewall allows the packet.
     If not, the next rule is processed.
  2. Is the source Alice and the destination 8.8.8.8 (google DNS server) and the protocol UDP and the destination port 53 (DNS)?
     If yes, the firewall allows the packet.
     If not, the next rule is processed.
  3. The firewall discards the packet.
     This default strategy is called blacklisting. Everything which is not explicitly allowed is denied.

#	Src IP	Dst IP	Proto	Src Port	Dst Port	Action
A	Alice	Gabby	TCP	*	80	Allow
B	Alice	8.8.8.8	UDP	*	53	Allow
C	*	*	*	*	*	Deny

Firewalls

- For simplicity, our example ignores
  - Stateful firewalls
  - Answer packets.
    Alice can send out HTTP and DNS request, but the firewall will never allow the answer to reach Alice.
- We only focus on the packet header.

#	Src IP	Dst IP	Proto	Src Port	Dst Port	Action
A	Alice	Gabby	TCP	*	80	Allow
B	Alice	8.8.8.8	UDP	*	53	Allow
C	*	*	*	*	*	Deny

Modeling Firewalls – Representing Rules

Demo: firewall.thy

Step 1: Rules Basics

- Per Rule Action
  - `Allow`
  - `Deny`
  - `Pass` — Rule does not apply, apply next rule

```
datatype action = Allow | Deny | Pass
```

- The Type of a Rule
  - A rule processes an packet of arbitrary type `'p`
  - A rule defines an action for that packet
  - A rule is a total function from packets to actions

```
type_synonym 'p rule = 'p ⇒ action
```

Representing Rules – Example 1

- Let a packet consist of a tuple of arbitrary type `'a` $\times$ `'b`
- Let `'a` be the source and `'b` be the destination address
- We can write a deny-all rule as

  λ `(src,dst).  Deny`
- The type is `('a` $\times$ `'b) rule`

Representing Rules – Example 2

- Let a packet consist of a tuple strings `string` × `string`
- The first entry is the source and the second the destination address
- We can write a rule which allows packets from *Alice* to *Bob* and denies all others

  λ (src, dst). `if src = ''Alice''` $\wedge$ `dst = ''Bob'' then Allow else Deny`
- The type is `(string` × `string) rule`
- For a packet `(''Alice'', ''Carl'')`
- We can show:
  `(`λ `(src,dst). if src = ''Alice''` $\wedge$ `dst = ''Bob'' then Allow else Deny) (''Alice'', ''Carl'') = Deny`

  Applying the packet `(''Alice'', ''Carl'')` to the rule results in `Deny`

Modeling Firewalls – Representing Rules

Step 2: A Firewalls Rule Set

Probably, rule list would be a better terminology, we will soon prove why.

- The firewall has a 'table' of rules
- For packets of type `'p`, we develop `datatype 'p ruleset`

Modeling Firewalls – Representing Rules

`datatype 'p ruleset`

- Rules apply sequentially
- If no rule matches, a default rule must apply
- Default rules
  - `DefaultAllow` – Blacklisting
    Everything not prohibited is allowed by default
  - `DefaultDeny` – Whitelisting
    Everything not allowed is denied by default
- A `Rule` consists of a rule and its successor rules
  - `Rule "'p rule" "'p ruleset"`

```
datatype 'p ruleset = DefaultAllow
                    | DefaultDeny
                    | Rule "'p rule" "'p ruleset"
```

Rulesets – Example 1

- Let a packet consist of arbitrary type `'a`
- We can write a rule set with a Allow-All rule, a Deny-All rule, and a default allow strategy

  `Rule (λp. Allow) (Rule (λp. Deny) DefaultAllow)`
- The type is `'p ruleset`

`Rule` Notation

- Instead of `Rule`, we will use the infix operator §
- Example:
  - `Rule x y` is written as `x § y`

Previous slide's example

`Rule (`λ`p. Allow) (Rule (`λ`p. Deny) DefaultAllow)`

Is written as

`(`λ`p. Allow) § (`λ`p. Deny) § DefaultAllow`

Conclusion

- We defined the syntax of
  - `action`
  - `'p rule`
  - `'p ruleset`
- Next, we will develop an example of a concrete ruleset for a concrete `'p`
- Afterwards, the semantics of firewalls are specified.
  - Syntax: How to write things down
  - Semantic: Meaning

Modeling Firewalls – Syntax Example

Demo: firewall.thy

Semantics

We define a function `firewall` to capture the semantics of a firewall.

Step 1) The type

- A firewall operates according to a `ruleset`
- A firewall processes packets accordingly
- The firewall decides what to do with the packet
  It returns an action

Hence, the type is `'p ruleset ⇒ 'p ⇒ action`

Semantics of `firewall`

- The semantics are specified via pattern matching on the ruleset
  - an underscore ‘_’ matches everything
- If the ruleset is only `DefaultAllow`, the firewall allows any packet

  “`firewall DefaultAllow _ = Allow`”
- If the ruleset is only `DefaultDeny`, the firewall denies any packet

  “`firewall DefaultDeny _ = Deny`”
- Most of the time, the ruleset will consists of a current `rule` and and some `next_rules`

  `rule § next_rules`
- `next_rules` can be any `'p ruleset`, either further rules or one of the default rules

Semantics of `firewall`

`firewall (rule § next_rules) p`

- For a packet `p`
- The firewall examines `rule` and tests whether the rule matches

  `rule p`

- If the action is `Allow` or `Deny`, it applies the respective action
- If the action is `Pass`, the `next_rules` must be applied to `p`
  This is a recursive call to `firewall`

Example

Demo: firewall.thy

Termination of `firewall`

- The default cases trivially terminate
- For the case `rule § next_rules`, the size of the ruleset is decreased with every recursive call.
  If `firewall (rule § next_rules) p` is called
  - it either terminates with `Allow` or `Deny`
  - or calls `firewall next_rules p`
- This is an inductive argument.

Add-In: Induction

Induction

- Induction on natural numbers $\mathbb{N}$
- To prove $P\ n$ for $n \in \mathbb{N}$
- Show that P holds for 0
- Assume P holds for n and show that it also holds for $n + 1$

Induction rule in Isabelle/HOL (simplified)

$$P\ 0 \Longrightarrow (\bigwedge n.\ P\ n \Longrightarrow P\ (n+1)) \Longrightarrow P\ x$$

Read as follows

- To show $P\ x$ (the goal right)
- Show $P\ 0$
- and show $\bigwedge n.\ P\ n \Longrightarrow P\ (n+1)$

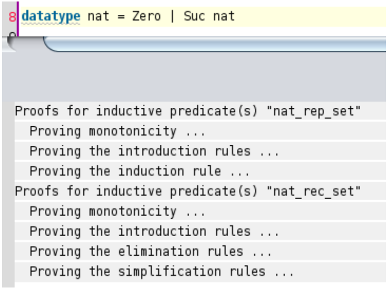
Demo: intro_induction.thy

Induction – Induction Rules

- Where do we get `nat.induct` from?
- Isabelle/HOL proves a lot automatically for us in the background

- How are natural numbers modeled?
  - `Zero` is a natural number
  - The successor `Suc` of a natural number is a natural number

```
datatype nat = Zero | Suc nat
```



Induction – Induction Rules

```
datatype nat = Zero | Suc nat
```

`nat.induct`

$$P\ \mathtt{Zero} \Longrightarrow (\bigwedge n.\ P\ n \Longrightarrow P\ (\mathtt{Suc}\ n)) \Longrightarrow P\ x$$

- Base case: $P\ \mathtt{Zero}$
- Induction step: $(\bigwedge n.\ P\ n \Longrightarrow P\ (\mathtt{Suc}\ n))$

Induction – Firewall Rules

- Firewall rules are modeled similarly
  - `DefaultAllow` is a ruleset
  - `DefaultDeny` is a ruleset
  - A `Rule` consists of a rule and a ruleset

```
datatype 'p ruleset = DefaultAllow
                    | DefaultDeny
                    | Rule "'p rule" "'p ruleset"
```

- Two base cases
  - `DefaultAllow`
  - `DefaultDeny`
- One induction step
  - Assume it holds for some `rules :: 'p ruleset`.
    For any rule `r :: 'p rule`, show that it holds for `Rule r rules`.

`ruleset` induction rule

`thm ruleset.induct`

$$P\,\texttt{DefaultAllow} \Longrightarrow P\,\texttt{DefaultDeny} \Longrightarrow$$
$$(\bigwedge r\ rules.\ P\ rules \Longrightarrow P\ (r\ \S\ rules)) \Longrightarrow P\ ruleset$$

For an arbitrary `ruleset`, to prove P `ruleset`

- Prove that P holds for the base cases
  - P `DefaultAllow`
  - P `DefaultDeny`
- And the induction step
  - Assume $P\ rules$
  - Show $P\ (r\ \S\ rules))$
  - $(\bigwedge r\ rules.\ P\ rules \Longrightarrow P\ (r\ \S\ rules))$

Back to `firewall`

Analyzing `firewall`

- For a ruleset `r` and a packet `p`
- `firewall r p` never returns `Pass`
- It either returns `Allow` or `Deny`

```
lemma ‘‘firewall r p = Allow ∨ firewall r p = Deny’’
proof(induction r)
  ...
```

Demo: firewall.thy

Yes, the *.thy is important!

Analyzing Rule Sets

- What does the policy `allow_DNS § allow_HTTP § DefaultDeny` allow?
- Only UDP port 53 or TCP port 80
- Phrasing as a `lemma`
  - A firewall with this rule set
  - A packet `p` applied to it returns `Allow` if and only if
  - `p` is UDP port 53 or `p` is TCP port 80

```
lemma ''firewall (allow_DNS § allow_HTTP § DefaultDeny) p = Allow
                              ⟷
            (proto p = UDP ∧ dst_port p = 53) ∨
            (proto p = TCP ∧ dst_port p = 80)''

by(simp add:  allow_DNS_def allow_HTTP_def)
```

- Why did the previous lemma require a (difficult) induction whereas this lemma could be directly solved by the simplifier?
- The previous lemma argued about arbitrary rule sets whereas this lemma had a known rule set.
- The `firewall` is defined in terms of the rule set.
- For unknown rule sets we (often) need induction over the rule set.
- If the rule set and the packet are given, the proof is often trivial as `firewall` is executable.

Analyzing Rule Sets

- Rule sets may be inefficient or misleading
- For example, rules can *shadow* each other

`deny_UDP § allow_DNS § allow_HTTP § DefaultDeny`

- The rule `allow_DNS` is shadowed by `deny_UDP`
- It can never apply
- We can simplify this ruleset an preserve the semantics!

`deny_UDP § allow_HTTP § DefaultDeny`

- We can prove that the two rulesets are semantically equivalent if `firewall` is equal for both of them
- `lemma ‘‘firewall rules1 = firewall rules2’’`

$\hookleftarrow$

When are two functions equal?

- Two (total) functions f and g are equal iff they return the same result for every input

$$(f = g) = (\forall x.\ f\ x = g\ x) \qquad \texttt{fun\_eq\_iff}$$

Back to firewalls

- Two firewall rule sets `rules1` and `rules2` are semantically equivalent iff `firewall` is equal for them
- `firewall rules1` is equal to `firewall rules2` iff the two functions are equal for all packets

- The previous rule sets are semantically equivalent

$$\texttt{deny\_UDP} \S \texttt{allow\_DNS} \S \texttt{allow\_HTTP} \S \texttt{DefaultDeny}$$

$$\texttt{deny\_UDP} \S \texttt{allow\_HTTP} \S \texttt{DefaultDeny}$$

- This can be proven by the simplifier once it is rewritten to

$\forall p.\ \texttt{firewall ?r1}\ p = \texttt{firewall ?r2}\ p$

- This rule set can even be reordered

$$\texttt{allow\_HTTP} \S \texttt{deny\_UDP} \S \texttt{DefaultDeny}$$

Ruleset Reordering

- We say a rule applies if it returns an action which is not `Pass`
- Rules `r1` and `r2` can be reordered if there is no packet such that both rules apply to the packet simultaneously

$$\nexists p.\ \texttt{r1}\ p \neq \texttt{Pass} \quad \wedge \quad \texttt{r2}\ p \neq \texttt{Pass}$$

- This is equal to the following:
- For all packets, at least one of `r1` and `r2` must not apply

$$\forall p.\ \texttt{r1}\ p = \texttt{Pass} \quad \vee \quad \texttt{r2}\ p = \texttt{Pass}$$

- This condition is sufficient to allow reordering of the first rules

```
firewall (r1 § r2 § r3) = firewall (r2 § r1 § r3)
```

- The condition is not necessary (try `quickcheck`)

Rule Sets or Rule Lists?

Rule Sets or Rule Lists

- The `rulesets` we saw bear great resemblance to lists
- `r1` § `r2` § `r3` § `r4` ...
- The order of the elements is important
- Can we represent firewall rules as list?

Lists

- `[1 :: nat, 2, 3, 5] :: nat list`
- recall `find_fives`
- `rev [1 :: int, 2, 3, 4] = [4, 3, 2, 1]`
- `lemma ''rev (rev l) = l'' by simp`
- `[a, b, c] :: 'a list`
- `[allow_DNS, allow_HTTP] :: packet rule list`

Semantics of a firewall with a rule list

- We define the semantics of a firewall whose rules are defined as list

  Type: `'p rule list ⇒ 'p ⇒ action`

- The list is processed sequentially – the rule in the list which matches first is applied
- If a rule returns `Pass`, the next list item is observed
- We use Default-Deny semantics – the empty list corresponds to `Deny`
- Note the similarity
  - `ruleset: rule § next_rules`
  - `rule list: rule # next_rules`

See `firewall_list_defaultDeny`

`ruleset` vs. `rule list`

- How does the `ruleset` and `rule list` correspond?
- we can translate them

  `'p ruleset` $\Rightarrow$ `'p rule list`

- With the Default-Deny semantics
  - `DefaultDeny` corresponds to the empty list `[]`
  - `DefaultAllow` corresponds to the one-element list with an allow-all rule `[(`$\lambda p.$ `Allow)]`
  - the correspondence `rule` § `next_rules` and `rule # next_rules` is defined recursively

See `ruleset_to_rulelist`

`ruleset` to `rule list` – Example

- A `ruleset` can be translated to a `rule list`

```
ruleset_to_rulelist (allow_DNS § allow_HTTP § DefaultDeny)
                 = [allow_DNS, allow_HTTP]
```

- Does this preserve the semantics?
- in this example, yes!

```
firewall (allow_DNS § allow_HTTP § DefaultDeny) =
firewall_list_defaultDeny [allow_DNS, allow_HTTP]
```

- Does this hold in general?

Relating the semantics

- Translating a `ruleset` to a `rule list`, the `firewall` and the `firewall_list_defaultDeny` are equal

```
lemma fw_eq_fwlist:  ‘‘firewall r =
firewall_list_defaultDeny (ruleset_to_rulelist r)"
```

- The proof (idea)
  - First, use `fun_eq_iff`
  - This leaves the subgoal that the two definitions are equal for all packets
  - Then, apply induction over the ruleset `r`

Demo: firewall.thy

Yes, the *.thy is important!

Conclusion

Conclusion

- Basics of modeling in Isabelle/HOL
  - FOL, HOL, λ-calculus, polymorphic types
  - Types and total functions
  - Induction
- Firewalls with `rulesets`
  - Syntax
    - actions, rules, `ruleset`
  - Semantics
    - Proofs about the semantics
    - Termination, never returns Pass, ruleset shadowing, ruleset reordering
- Firewall modeled with lists
- Semantical equality

Thanks for your attention!

Questions?