

The meta theory of the Incredible Proof Machine

Joachim Breitner Denis Lohner

May 26, 2024

The Incredible Proof Machine is an interactive visual theorem prover which represents proofs as port graphs. We model this proof representation in Isabelle, and prove that it is just as powerful as natural deduction.

Contents

1	Introduction	2
2	Auxiliary theories	4
2.1	Entailment	4
2.2	Indexed_FSet	4
2.3	Rose_Tree	6
2.3.1	The rose tree data type	6
2.3.2	The set of paths in a rose tree	6
2.3.3	Indexing into a rose tree	6
3	Abstract formulas, rules and tasks	8
3.1	Abstract_Formula	8
3.2	Abstract_Rules	10
4	Incredible Proof Graphs	13
4.1	Incredible_Signatures	13
4.2	Incredible_Deduction	14
4.3	Abstract_Rules_To_Incredible	23
5	Natural Deduction	27
5.1	Natural_Deduction	27
6	Correctness	29
6.1	Incredible_Correctness	29
7	Completeness	39
7.1	Incredible_Trees	39
7.2	Build_Incredible_Tree	50
7.3	Incredible_Completeness	53

8	Instantiations	66
8.1	Propositional_Formulas	66
8.2	Incredible_Propositional	67
8.3	Incredible_Propositional_Tasks	69
8.3.1	Task 1.1	69
8.3.2	Task 2.11	71
8.4	Predicate_Formulas	75
8.5	Incredible_Predicate	79
8.6	Incredible_Predicate_Tasks	81

1 Introduction

The Incredible Proof Machine (<http://incredible.pm>) is an educational tool that allows the user to prove theorems just by dragging proof blocks (corresponding to proof rules) onto a canvas, and connecting them correctly.

In the ITP 2016 paper [Bre16] the first author formally describes the shape of these graphs, as port graphs, and gives the necessary conditions for when we consider such a graph a valid proof graph. The present Isabelle formalization implements these definitions in Isabelle, and furthermore proves that such proof graphs are just as powerful as natural deduction.

All this happens with regard to an abstract set of formulas (theory *Abstract_Formula*) and an abstract set of logic rules (theory *Abstract_Rules*) and can thus be instantiated with various logics.

This formalization covers the following aspects:

- We formalize the definition of port graphs, proof graphs and the conditions for such a proof graph to be a valid graph (theory *Incredible_Deduction*).
- We provide a formal description of natural deduction (theory *Natural_Deduction*), which connects to the existing theories in the AFP entry Abstract Completeness [BPT14].
- For every proof graph, we construct a corresponding natural deduction derivation tree (theory *Incredible_Correctness*).
- Conversely, if we have a natural deduction derivation tree, we can construct a proof graph thereof (theory *Incredible_Completeness*).

This is the much harder direction, mostly because the freshness side condition for locally fixed constants (such as in the introduction rule for the universal quantifier) is a local check in natural deduction, but a global check in proofs graphs, and thus some elaborate renaming has to occur (*globalize* in *Incredible_Trees*).

- To explain our abstract locales, and ensure that the assumptions are consistent, we provide example instantiations for them.

It does not cover the unification procedure and expects that a suitable instantiation is already given. It also does not cover the creation and use of custom blocks, which abstract over proofs and thus correspond to lemmas in Isabelle.

Acknowledgements

We would like to thank Andreas Lochbihler for helpful comments.

References

- [BPT14] Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel, *Abstract completeness*, Archive of Formal Proofs (2014), http://isa-afp.org/entries/Abstract_Completeness.shtml, Formal proof development.
- [Bre16] Joachim Breitner, *Visual theorem proving with the Incredible Proof Machine*, ITP, 2016.

2 Auxiliary theories

2.1 Entailment

```
theory Entailment
imports Main HOL-Library.FSet
begin

type-synonym 'form entailment = ('form fset × 'form)

abbreviation entails :: 'form fset ⇒ 'form ⇒ 'form entailment (infix † 50)
  where a † c ≡ (a, c)

fun add-ctxt :: 'form fset ⇒ 'form entailment ⇒ 'form entailment where
  add-ctxt Δ (Γ † c) = (Γ |∪| Δ † c)

end
```

2.2 Indexed_FSet

```
theory Indexed-FSet
imports
  HOL-Library.FSet
begin
```

It is convenient to address the members of a finite set by a natural number, and also to convert a finite set to a list.

```
context includes fset.lifting
begin
lift-definition fset-from-list :: 'a list => 'a fset is set by (rule finite-set)
lemma mem-fset-from-list[simp]: x |∈| fset-from-list l ↔ x ∈ set l by transfer rule
lemma fimage-fset-from-list[simp]: f |'| fset-from-list l = fset-from-list (map f l) by transfer auto
lemma fset-fset-from-list[simp]: fset (fset-from-list l) = set l by transfer auto
lemmas fset-simps[simp] = set-simps[Transfer.transferred]
lemma size-fset-from-list[simp]: distinct l ⇒ size (fset-from-list l) = length l
  by (induction l) auto

definition list-of-fset :: 'a fset ⇒ 'a list where
  list-of-fset s = (SOME l. fset-from-list l = s ∧ distinct l)

lemma fset-from-list-of-fset[simp]: fset-from-list (list-of-fset s) = s
  and distinct-list-of-fset[simp]: distinct (list-of-fset s)
  unfolding atomize-conj list-of-fset-def
  by (transfer, rule someI-ex, rule finite-distinct-list)

lemma length-list-of-fset[simp]: length (list-of-fset s) = size s
  by (metis distinct-list-of-fset fset-from-list-of-fset size-fset-from-list)

lemma nth-list-of-fset-mem[simp]: i < size s ⇒ list-of-fset s ! i |∈| s
  by (metis fset-from-list-of-fset length-list-of-fset mem-fset-from-list nth-mem)

inductive indexed-fmember :: 'a ⇒ nat ⇒ 'a fset ⇒ bool (- |∈| - [50,50,50] 50) where
  i < size s ⇒ list-of-fset s ! i |∈|i s

lemma indexed-fmember-is-fmember: x |∈|i s ⇒ x |∈| s
proof (induction rule: indexed-fmember.induct)
```

case (1 i s)
 hence $i < \text{length } (\text{list-of-fset } s)$ **by** (*metis length-list-of-fset*)
 hence $\text{list-of-fset } s ! i \in \text{set } (\text{list-of-fset } s)$ **by** (*rule nth-mem*)
 thus $\text{list-of-fset } s ! i \in | s$ **by** (*metis mem-fset-from-list fset-from-list-of-fset*)
qed

lemma *fmember-is-indexed-fmember*:

assumes $x \in | s$
shows $\exists i. x \in |_i s$

proof–

from *assms*
have $x \in \text{set } (\text{list-of-fset } s)$ **using** *mem-fset-from-list* **by** *fastforce*
then obtain i **where** $i < \text{length } (\text{list-of-fset } s)$ **and** $x = \text{list-of-fset } s ! i$ **by** (*metis in-set-conv-nth*)
hence $x \in |_i s$ **by** (*simp add: indexed-fmember.simps*)
thus *?thesis..*

qed

lemma *indexed-fmember-unique*: $x \in |_i s \implies y \in |_j s \implies x = y \longleftrightarrow i = j$

by (*metis distinct-list-of-fset indexed-fmember.cases length-list-of-fset nth-eq-iff-index-eq*)

definition *indexed-members* :: $'a \text{ fset} \Rightarrow (\text{nat} \times 'a) \text{ list}$ **where**

indexed-members $s = \text{zip } [0..<\text{size } s]$ (*list-of-fset* s)

lemma *mem-set-indexed-members*:

$(i, x) \in \text{set } (\text{indexed-members } s) \longleftrightarrow x \in |_i s$
unfolding *indexed-members-def indexed-fmember.simps*
by (*force simp add: set-zip*)

lemma *mem-set-indexed-members'[simp]*:

$t \in \text{set } (\text{indexed-members } s) \longleftrightarrow \text{snd } t \in |_{\text{fst } t} s$
by (*cases t, simp add: mem-set-indexed-members*)

definition *fnth* (**infixl** $||$ 100) **where**

$s || n = \text{list-of-fset } s ! n$

lemma *fnth-indexed-fmember*: $i < \text{size } s \implies s || i \in |_i s$

unfolding *fnth-def* **by** (*rule indexed-fmember.intros*)

lemma *indexed-fmember-fnth*: $x \in |_i s \longleftrightarrow (s || i = x \wedge i < \text{size } s)$

unfolding *fnth-def* **by** (*metis indexed-fmember.simps*)

end

definition *fidx* :: $'a \text{ fset} \Rightarrow 'a \Rightarrow \text{nat}$ **where**

fidx $s \ x = (\text{SOME } i. x \in |_i s)$

lemma *fidx-eq[simp]*: $x \in |_i s \implies \text{fidx } s \ x = i$

unfolding *fidx-def*

by (*rule someI2*)(*auto simp add: indexed-fmember-fnth fnth-def nth-eq-iff-index-eq*)

lemma *fidx-inj[simp]*: $x \in | s \implies y \in | s \implies \text{fidx } s \ x = \text{fidx } s \ y \longleftrightarrow x = y$

by (*auto dest!: fmember-is-indexed-fmember simp add: indexed-fmember-unique*)

lemma *inj-on-fidx*: *inj-on* (*fidx vertices*) (*fset vertices*)

by (*rule inj-onI*) *simp*

end

2.3 Rose_Tree

```
theory Rose_Tree
imports Main HOL-Library.Sublist
begin
```

For theory *Incredible-Trees* we need rose trees; this theory contains the generally useful part of that development.

2.3.1 The rose tree data type

```
datatype 'a rose-tree = RNode (root: 'a) (children: 'a rose-tree list)
```

2.3.2 The set of paths in a rose tree

Too bad that **inductive-set** does not allow for varying parameters...

```
inductive it_pathsP :: 'a rose-tree  $\Rightarrow$  nat list  $\Rightarrow$  bool where
  it_paths-Nil: it_pathsP t []
| it_paths-Cons:  $i < \text{length } (\text{children } t) \Longrightarrow \text{children } t ! i = t' \Longrightarrow \text{it\_pathsP } t' \text{ is} \Longrightarrow \text{it\_pathsP } t (i\#\text{is})$ 
```

```
inductive-cases it_pathP-ConsE: it_pathsP t (i\#is)
```

```
inductive-cases it_pathP-RNodeE: it_pathsP (RNode r ants) is
```

```
definition it_paths:: 'a rose-tree  $\Rightarrow$  nat list set where
  it_paths t = Collect (it_pathsP t)
```

```
lemma it_paths-eq [pred-set-conv]: it_pathsP t = ( $\lambda x. x \in \text{it\_paths } t$ )
by(simp add: it_paths-def)
```

```
lemmas it_paths-intros [intro?] = it_pathsP.intros[to-set]
lemmas it_paths-induct [consumes 1, induct set: it_paths] = it_pathsP.induct[to-set]
lemmas it_paths-cases [consumes 1, cases set: it_paths] = it_pathsP.cases[to-set]
lemmas it_paths-ConsE = it_pathP-ConsE[to-set]
lemmas it_paths-RNodeE = it_pathP-RNodeE[to-set]
lemmas it_paths-simps = it_pathsP.simps[to-set]
```

```
lemmas it_paths-intros(1)[simp]
```

```
lemma it_paths-RNode-Nil[simp]: it_paths (RNode r []) = {[]}
by (auto elim: it_paths-cases)
```

```
lemma it_paths-Union: it_paths t  $\subseteq \text{insert } [] (\text{Union } (((\lambda (i,t). ((\#) i) \text{'it\_paths } t) \text{'set } (\text{List.enumerate } (0::\text{nat})$ 
( $\text{children } t$ ))))))
  apply (rule)
  apply (erule it_paths-cases)
  apply (auto intro!: bexI simp add: in-set-enumerate-eq)
done
```

```
lemma finite-it_paths[simp]: finite (it_paths t)
by (induction t) (auto intro!: finite-subset[OF it_paths-Union] simp add: in-set-enumerate-eq)
```

2.3.3 Indexing into a rose tree

```
fun tree-at :: 'a rose-tree  $\Rightarrow$  nat list  $\Rightarrow$  'a rose-tree where
  tree-at t [] = t
```

| $tree-at\ t\ (i\#\!is) = tree-at\ (children\ t\ !\ i)\ is$

lemma *it-paths-SnocE[elim-format]*:

assumes $is\ @\ [i] \in it-paths\ t$

shows $is \in it-paths\ t \wedge i < length\ (children\ (tree-at\ t\ is))$

using *assms*

by (*induction is arbitrary: t*)(*auto intro!: it-paths-intros elim!: it-paths-ConsE*)

lemma *it-paths-strict-prefix*:

assumes $is \in it-paths\ t$

assumes *strict-prefix is' is*

shows $is' \in it-paths\ t$

proof–

from *assms*(2)

obtain is'' **where** $is = is' @ is''$ **using** *strict-prefixE'* **by** *blast*

from *assms*(1)[*unfolded this*]

show *?thesis*

by(*induction is' arbitrary: t*) (*auto elim!: it-paths-ConsE intro!: it-paths-intros*)

qed

lemma *it-paths-prefix*:

assumes $is \in it-paths\ t$

assumes *prefix is' is*

shows $is' \in it-paths\ t$

using *assms it-paths-strict-prefix strict-prefixI* **by** *fastforce*

lemma *it-paths-butlast*:

assumes $is \in it-paths\ t$

shows *butlast is* $\in it-paths\ t$

using *assms prefixeq-butlast* **by** (*rule it-paths-prefix*)

lemma *it-path-SnocI*:

assumes $is \in it-paths\ t$

assumes $i < length\ (children\ (tree-at\ t\ is))$

shows $is\ @\ [i] \in it-paths\ t$

using *assms*

by (*induction t arbitrary: is i*)

(*auto 4 4 elim!: it-paths-RNodeE intro: it-paths-intros*)

end

3 Abstract formulas, rules and tasks

3.1 Abstract_Formula

theory *Abstract-Formula*

imports

Main

HOL-Library.FSet

HOL-Library.Stream

Indexed-FSet

begin

The following locale describes an abstract interface for a set of formulas, without fixing the concret shape, or set of variables.

The variables mentioned in this locale are only the *locally fixed constants* occurring in formulas, e.g. in the introduction rule for the universal quantifier. Normal variables are not something we care about at this point; they are handled completely abstractly by the abstract notion of a substitution.

locale *Abstract-Formulas* =

— Variables can be renamed injectively

fixes *freshenLC* :: *nat* \Rightarrow *'var* \Rightarrow *'var*

— A variable-changing function can be mapped over a formula

fixes *renameLCs* :: (*'var* \Rightarrow *'var*) \Rightarrow (*'form* \Rightarrow *'form*)

— The set of variables occurring in a formula

fixes *lconsts* :: *'form* \Rightarrow *'var set*

— A closed formula has no variables, and substitutions do not affect it.

fixes *closed* :: *'form* \Rightarrow *bool*

— A substitution can be applied to a formula.

fixes *subst* :: *'subst* \Rightarrow *'form* \Rightarrow *'form*

— The set of variables occurring (in the image) of a substitution.

fixes *subst-lconsts* :: *'subst* \Rightarrow *'var set*

— A variable-changing function can be mapped over a substitution

fixes *subst-renameLCs* :: (*'var* \Rightarrow *'var*) \Rightarrow (*'subst* \Rightarrow *'subst*)

— A most generic formula, can be substituted to anything.

fixes *anyP* :: *'form*

assumes *freshenLC-eq-iff[simp]*: *freshenLC* *a v* = *freshenLC* *a' v'* \iff *a* = *a'* \wedge *v* = *v'*

assumes *lconsts-renameLCs*: *lconsts* (*renameLCs* *p f*) = *p* ' *lconsts* *f*

assumes *rename-closed*: *lconsts* *f* = {} \implies *renameLCs* *p f* = *f*

assumes *subst-closed*: *closed* *f* \implies *subst* *s f* = *f*

assumes *closed-no-lconsts*: *closed* *f* \implies *lconsts* *f* = {}

assumes *fv-subst*: *lconsts* (*subst* *s f*) \subseteq *lconsts* *f* \cup *subst-lconsts* *s*

assumes *rename-rename*: *renameLCs* *p1* (*renameLCs* *p2 f*) = *renameLCs* (*p1* \circ *p2*) *f*

assumes *rename-subst*: *renameLCs* *p* (*subst* *s f*) = *subst* (*subst-renameLCs* *p s*) (*renameLCs* *p f*)

assumes *renameLCs-cong*: ($\bigwedge x. x \in \text{lconsts } f \implies f1\ x = f2\ x$) \implies *renameLCs* *f1 f* = *renameLCs* *f2 f*

assumes *subst-renameLCs-cong*: ($\bigwedge x. x \in \text{subst-lconsts } s \implies f1\ x = f2\ x$) \implies *subst-renameLCs* *f1 s* = *subst-renameLCs* *f2 s*

assumes *subst-lconsts-subst-renameLCs*: *subst-lconsts* (*subst-renameLCs* *p s*) = *p* ' *subst-lconsts* *s*

assumes *lconsts-anyP*: *lconsts* *anyP* = {}

assumes *empty-subst*: $\exists s. (\forall f. \text{subst } s\ f = f) \wedge \text{subst-lconsts } s = \{\}$

assumes *anyP-is-any*: $\exists s. \text{subst } s\ \text{anyP} = f$

begin

definition *freshen* :: *nat* \Rightarrow *'form* \Rightarrow *'form* **where**

freshen *n* = *renameLCs* (*freshenLC* *n*)

definition *empty-subst* :: *'subst* **where**

$empty\text{-subst} = (SOME\ s.\ (\forall\ f.\ subst\ s\ f = f) \wedge subst\text{-lconsts}\ s = \{\})$

lemma *empty-subst-spec*:

$(\forall\ f.\ subst\ empty\text{-subst}\ f = f) \wedge subst\text{-lconsts}\ empty\text{-subst} = \{\}$

unfolding *empty-subst-def* **using** *empty-subst* **by** (rule *someI-ex*)

lemma *subst-empty-subst[simp]*: $subst\ empty\text{-subst}\ f = f$

by (metis *empty-subst-spec*)

lemma *subst-lconsts-empty-subst[simp]*: $subst\text{-lconsts}\ empty\text{-subst} = \{\}$

by (metis *empty-subst-spec*)

lemma *lconsts-freshen*: $lconsts\ (freshen\ a\ f) = freshenLC\ a\ \text{'}\ lconsts\ f$

unfolding *freshen-def* **by** (rule *lconsts-renameLCs*)

lemma *freshen-closed*: $lconsts\ f = \{\} \implies freshen\ a\ f = f$

unfolding *freshen-def* **by** (rule *rename-closed*)

lemma *closed-eq*:

assumes *closed f1*

assumes *closed f2*

shows $subst\ s1\ (freshen\ a1\ f1) = subst\ s2\ (freshen\ a2\ f2) \longleftrightarrow f1 = f2$

using *assms*

by (auto *simp add: closed-no-lconsts freshen-def lconsts-freshen subst-closed rename-closed*)

lemma *freshenLC-range-eq-iff[simp]*: $freshenLC\ a\ v \in range\ (freshenLC\ a') \longleftrightarrow a = a'$

by *auto*

definition *rename* :: $'var\ set \Rightarrow nat \Rightarrow nat \Rightarrow ('var \Rightarrow 'var) \Rightarrow ('var \Rightarrow 'var)$ **where**

$rename\ V\ from\ to\ f\ x = (if\ x \in freshenLC\ from\ \text{'}\ V\ then\ freshenLC\ to\ (inv\ (freshenLC\ from)\ x)\ else\ f\ x)$

lemma *inj-freshenLC[simp]*: $inj\ (freshenLC\ i)$

by (rule *injI*) *simp*

lemma *rename-freshen[simp]*: $x \in V \implies rename\ V\ i\ (isidx\ is)\ f\ (freshenLC\ i\ x) = freshenLC\ (isidx\ is)$

x

unfolding *rename-def* **by** *simp*

lemma *range-rename*: $range\ (rename\ V\ from\ to\ f) \subseteq freshenLC\ to\ \text{'}\ V \cup range\ f$

by (auto *simp add: rename-def split: if-splits*)

lemma *rename-noop*:

$x \notin freshenLC\ from\ \text{'}\ V \implies rename\ V\ from\ to\ f\ x = f\ x$

by (auto *simp add: rename-def split: if-splits*)

lemma *rename-rename-noop*:

$freshenLC\ from\ \text{'}\ V \cap lconsts\ form = \{\} \implies renameLCs\ (rename\ V\ from\ to\ f)\ form = renameLCs\ f$

form

by (intro *renameLCs-cong rename-noop*) *auto*

lemma *rename-subst-noop*:

$freshenLC\ from\ \text{'}\ V \cap subst\text{-lconsts}\ s = \{\} \implies subst\text{-renameLCs}\ (rename\ V\ from\ to\ f)\ s =$

$subst\text{-renameLCs}\ f\ s$

by (intro *subst-renameLCs-cong rename-noop*) *auto*

end

end

3.2 Abstract_Rules

```

theory Abstract-Rules
imports
  Abstract-Formula
begin

```

Next, we can define a logic, by giving a set of rules.

In order to connect to the AFP entry Abstract Completeness, the set of rules is a stream; the only relevant effect of this is that the set is guaranteed to be non-empty and at most countable. This has no further significance in our development.

Each antecedent of a rule consists of

- a set of fresh variables
- a set of hypotheses that may be used in proving the conclusion of the antecedent and
- the conclusion of the antecedent.

Our rules allow for multiple conclusions (but must have at least one).

In order to prove the completeness (but incidentally not to prove correctness) of the incredible proof graphs, there are some extra conditions about the fresh variables in a rule.

- These need to be disjoint for different antecedents.
- They need to list all local variables occurring in either the hypothesis and the conclusion.
- The conclusions of a rule must not contain any local variables.

```

datatype ('form, 'var) antecedent =
  Antecedent (a-hyps: 'form fset) (a-conc: 'form) (a-fresh: 'var set)

```

```

abbreviation plain-ant :: 'form  $\Rightarrow$  ('form, 'var) antecedent
where plain-ant f  $\equiv$  Antecedent {||} f {}

```

```

locale Abstract-Rules =
  Abstract-Formulas freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP
for freshenLC :: nat  $\Rightarrow$  'var  $\Rightarrow$  'var
and renameLCs :: ('var  $\Rightarrow$  'var)  $\Rightarrow$  ('form  $\Rightarrow$  'form)
and lconsts :: 'form  $\Rightarrow$  'var set
and closed :: 'form  $\Rightarrow$  bool
and subst :: 'subst  $\Rightarrow$  'form  $\Rightarrow$  'form
and subst-lconsts :: 'subst  $\Rightarrow$  'var set
and subst-renameLCs :: ('var  $\Rightarrow$  'var)  $\Rightarrow$  ('subst  $\Rightarrow$  'subst)
and anyP :: 'form +

```

```

fixes antecedent :: 'rule  $\Rightarrow$  ('form, 'var) antecedent list
and consequent :: 'rule  $\Rightarrow$  'form list
and rules :: 'rule stream

```

```

assumes no-empty-conclusions:  $\forall xs \in sset\ rules. consequent\ xs \neq []$ 

```

```

assumes no-local-consts-in-consequences:  $\forall xs \in sset\ rules. \bigcup (lconsts\ ' (set\ (consequent\ xs))) = \{\}$ 

```

```

assumes no-multiple-local-consts:

```

```

 $\bigwedge r\ i\ i'. r \in sset\ rules \implies$ 
   $i < length\ (antecedent\ r) \implies$ 
   $i' < length\ (antecedent\ r) \implies$ 

```

$a\text{-fresh } (antecedent\ r\ !\ i) \cap a\text{-fresh } (antecedent\ r\ !\ i') = \{\} \vee i = i'$

assumes *all-local-consts-listed*:
 $\bigwedge r\ p.\ r \in \text{sset rules} \implies p \in \text{set } (antecedent\ r) \implies$
 $\text{lconsts } (a\text{-conc } p) \cup (\bigcup (\text{lconsts } 'fset\ (a\text{-hypos } p))) \subseteq a\text{-fresh } p$

begin
definition *f-antecedent* :: $'rule \Rightarrow ('form, 'var)\ antecedent\ fset$
where *f-antecedent* $r = \text{fset-from-list } (antecedent\ r)$
definition *f-consequent* $r = \text{fset-from-list } (consequent\ r)$
end

Finally, an abstract task specifies what a specific proof should prove. In particular, it gives a set of assumptions that may be used, and lists the conclusions that need to be proven.

Both assumptions and conclusions are closed expressions that may not be changed by substitutions.

locale *Abstract-Task* =
Abstract-Rules freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP antecedent consequent rules
for *freshenLC* :: $\text{nat} \Rightarrow 'var \Rightarrow 'var$
and *renameLCs* :: $('var \Rightarrow 'var) \Rightarrow ('form \Rightarrow 'form)$
and *lconsts* :: $'form \Rightarrow 'var\ \text{set}$
and *closed* :: $'form \Rightarrow \text{bool}$
and *subst* :: $'subst \Rightarrow 'form \Rightarrow 'form$
and *subst-lconsts* :: $'subst \Rightarrow 'var\ \text{set}$
and *subst-renameLCs* :: $('var \Rightarrow 'var) \Rightarrow ('subst \Rightarrow 'subst)$
and *anyP* :: $'form$
and *antecedent* :: $'rule \Rightarrow ('form, 'var)\ \text{antecedent list}$
and *consequent* :: $'rule \Rightarrow 'form\ \text{list}$
and *rules* :: $'rule\ \text{stream} +$

fixes *assumptions* :: $'form\ \text{list}$
fixes *conclusions* :: $'form\ \text{list}$
assumes *assumptions-closed*: $\bigwedge a.\ a \in \text{set } \text{assumptions} \implies \text{closed } a$
assumes *conclusions-closed*: $\bigwedge c.\ c \in \text{set } \text{conclusions} \implies \text{closed } c$

begin
definition *ass-forms* **where** *ass-forms* = $\text{fset-from-list } \text{assumptions}$
definition *conc-forms* **where** *conc-forms* = $\text{fset-from-list } \text{conclusions}$

lemma *mem-ass-forms[simp]*: $a \in | \text{ass-forms} \iff a \in \text{set } \text{assumptions}$
by (*auto simp add: ass-forms-def*)

lemma *mem-conc-forms[simp]*: $a \in | \text{conc-forms} \iff a \in \text{set } \text{conclusions}$
by (*auto simp add: conc-forms-def*)

lemma *subst-freshen-assumptions[simp]*:
assumes $pf \in \text{set } \text{assumptions}$
shows $\text{subst } s\ (\text{freshen } a\ pf) = pf$
using *assms assumptions-closed*
by (*simp add: closed-no-lconsts freshen-def rename-closed subst-closed*)

lemma *subst-freshen-conclusions[simp]*:
assumes $pf \in \text{set } \text{conclusions}$
shows $\text{subst } s\ (\text{freshen } a\ pf) = pf$
using *assms conclusions-closed*
by (*simp add: closed-no-lconsts freshen-def rename-closed subst-closed*)

lemma *subst-freshen-in-ass-formsI*:
assumes $pf \in \text{set } \text{assumptions}$

shows *subst s (freshen a pf) |∈| ass-forms*
using *assms* **by** *simp*

lemma *subst-freshen-in-conc-formsI:*
assumes *pf ∈ set conclusions*
shows *subst s (freshen a pf) |∈| conc-forms*
using *assms* **by** *simp*

end

end

4 Incredible Proof Graphs

4.1 Incredible_Signatures

```
theory Incredible-Signatures
imports
  Main
  HOL-Library.FSet
  HOL-Library.Stream
  Abstract-Formula
begin
```

This theory contains the definition for proof graph signatures, in the variants

- Plain port graph
- Port graph with local hypotheses
- Labeled port graph
- Port graph with local constants

```
locale Port-Graph-Signature =
  fixes nodes :: 'node stream
  fixes inPorts :: 'node  $\Rightarrow$  'inPort fset
  fixes outPorts :: 'node  $\Rightarrow$  'outPort fset

locale Port-Graph-Signature-Scoped =
  Port-Graph-Signature +
  fixes hyps :: 'node  $\Rightarrow$  'outPort  $\rightarrow$  'inPort
  assumes hyps-correct: hyps n p1 = Some p2  $\implies$  p1  $\in$  outPorts n  $\wedge$  p2  $\in$  inPorts n
begin
  inductive-set hyps-for' :: 'node  $\Rightarrow$  'inPort  $\Rightarrow$  'outPort set for n p
    where hyps n h = Some p  $\implies$  h  $\in$  hyps-for' n p

  lemma hyps-for'-subset: hyps-for' n p  $\subseteq$  fset (outPorts n)
    using hyps-correct by (meson hyps-for'.cases subsetI)

  context includes fset.lifting
  begin
  lift-definition hyps-for :: 'node  $\Rightarrow$  'inPort  $\Rightarrow$  'outPort fset is hyps-for'
    by (meson finite-fset hyps-for'-subset rev-finite-subset)
  lemma hyps-for-simp[simp]: h  $\in$  hyps-for n p  $\longleftrightarrow$  hyps n h = Some p
    by transfer (simp add: hyps-for'.simps)
  lemma hyps-for-simp'[simp]: h  $\in$  fset (hyps-for n p)  $\longleftrightarrow$  hyps n h = Some p
    by transfer (simp add: hyps-for'.simps)
  lemma hyps-for-collect: fset (hyps-for n p) = {h . hyps n h = Some p}
    by auto
  end
  lemma hyps-for-subset: hyps-for n p  $\subseteq$  outPorts n
    using hyps-for'-subset
    by (fastforce simp add: hyps-for.rep-eq simp del: hyps-for-simp hyps-for-simp')
end

locale Labeled-Signature =
  Port-Graph-Signature-Scoped +
  fixes labelsIn :: 'node  $\Rightarrow$  'inPort  $\Rightarrow$  'form
  fixes labelsOut :: 'node  $\Rightarrow$  'outPort  $\Rightarrow$  'form
```

```

locale Port-Graph-Signature-Scoped-Vars =
  Port-Graph-Signature nodes inPorts outPorts +
  Abstract-Formulas freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP
for nodes :: 'node stream and inPorts :: 'node  $\Rightarrow$  'inPort fset and outPorts :: 'node  $\Rightarrow$  'outPort fset
and freshenLC :: nat  $\Rightarrow$  'var  $\Rightarrow$  'var
  and renameLCs :: ('var  $\Rightarrow$  'var)  $\Rightarrow$  'form  $\Rightarrow$  'form
  and lconsts :: 'form  $\Rightarrow$  'var set
  and closed :: 'form  $\Rightarrow$  bool
  and subst :: 'subst  $\Rightarrow$  'form  $\Rightarrow$  'form
  and subst-lconsts :: 'subst  $\Rightarrow$  'var set
  and subst-renameLCs :: ('var  $\Rightarrow$  'var)  $\Rightarrow$  ('subst  $\Rightarrow$  'subst)
  and anyP :: 'form +

  fixes local-vars :: 'node  $\Rightarrow$  'inPort  $\Rightarrow$  'var set

end

```

4.2 Incredible_Deduction

```

theory Incredible-Deduction
imports
  Main
  HOL-Library.FSet
  HOL-Library.Stream
  Incredible-Signatures
  HOL-Eisbach.Eisbach
begin

```

This theory contains the definition for actual proof graphs, and their various possible properties.

The following locale first defines graphs, without edges.

```

locale Vertex-Graph =
  Port-Graph-Signature nodes inPorts outPorts
  for nodes :: 'node stream
  and inPorts :: 'node  $\Rightarrow$  'inPort fset
  and outPorts :: 'node  $\Rightarrow$  'outPort fset +
  fixes vertices :: 'v fset
  fixes nodeOf :: 'v  $\Rightarrow$  'node
begin
  fun valid-out-port where valid-out-port (v,p)  $\longleftrightarrow$  v | $\in$ | vertices  $\wedge$  p | $\in$ | outPorts (nodeOf v)
  fun valid-in-port where valid-in-port (v,p)  $\longleftrightarrow$  v | $\in$ | vertices  $\wedge$  p | $\in$ | inPorts (nodeOf v)

  fun terminal-node where
    terminal-node n  $\longleftrightarrow$  outPorts n = {}
  fun terminal-vertex where
    terminal-vertex v  $\longleftrightarrow$  v | $\in$ | vertices  $\wedge$  terminal-node (nodeOf v)
end

```

And now we add the edges. This allows us to define paths and scopes.

```

type-synonym ('v, 'outPort, 'inPort) edge = (('v  $\times$  'outPort)  $\times$  ('v  $\times$  'inPort))

```

```

locale Pre-Port-Graph =
  Vertex-Graph nodes inPorts outPorts vertices nodeOf

```

```

for nodes :: 'node stream
and inPorts :: 'node ⇒ 'inPort fset
and outPorts :: 'node ⇒ 'outPort fset
and vertices :: 'v fset
and nodeOf :: 'v ⇒ 'node +
fixes edges :: ('v, 'outPort, 'inPort) edge set
begin
fun edge-begin :: (('v × 'outPort) × ('v × 'inPort)) ⇒ 'v where
  edge-begin ((v1,p1),(v2,p2)) = v1
fun edge-end :: (('v × 'outPort) × ('v × 'inPort)) ⇒ 'v where
  edge-end ((v1,p1),(v2,p2)) = v2

lemma edge-begin-tup: edge-begin x = fst (fst x) by (metis edge-begin.simps prod.collapse)
lemma edge-end-tup: edge-end x = fst (snd x) by (metis edge-end.simps prod.collapse)

inductive path :: 'v ⇒ 'v ⇒ ('v, 'outPort, 'inPort) edge list ⇒ bool where
  path-empty: path v v [] |
  path-cons: e ∈ edges ⇒ path (edge-end e) v' pth ⇒ path (edge-begin e) v' (e#pth)

inductive-simps path-cons-simp': path v v' (e#pth)
inductive-simps path-empty-simp[simp]: path v v' []
lemma path-cons-simp: path v v' (e # pth) ⟷ fst (fst e) = v ∧ e ∈ edges ∧ path (fst (snd e)) v' pth
by(auto simp add: path-cons-simp', metis prod.collapse)

lemma path-appendI: path v v' pth1 ⇒ path v v'' pth2 ⇒ path v v'' (pth1@pth2)
by (induction pth1 arbitrary: v) (auto simp add: path-cons-simp)

lemma path-split: path v v' (pth1@[e]@pth2) ⟷ path v (edge-end e) (pth1@[e]) ∧ path (edge-end e) v'
pth2
by (induction pth1 arbitrary: v) (auto simp add: path-cons-simp edge-end-tup intro: path-empty)

lemma path-split2: path v v' (pth1@(e#pth2)) ⟷ path v (edge-begin e) pth1 ∧ path (edge-begin e) v'
(e#pth2)
by (induction pth1 arbitrary: v) (auto simp add: path-cons-simp edge-begin-tup intro: path-empty)

lemma path-snoc: path v v' (pth1@[e]) ⟷ e ∈ edges ∧ path v (edge-begin e) pth1 ∧ edge-end e = v'
by (auto simp add: path-split2 path-cons-simp edge-end-tup edge-begin-tup)

inductive-set scope :: 'v × 'inPort ⇒ 'v set for ps where
  v |e| vertices ⇒ (∧ pth v'. path v v' pth ⇒ terminal-vertex v' ⇒ ps ∈ snd ' set pth)
  ⇒ v ∈ scope ps

lemma scope-find:
assumes v ∈ scope ps
assumes terminal-vertex v'
assumes path v v' pth
shows ps ∈ snd ' set pth
using assms by (auto simp add: scope.simps)

lemma snd-set-split:
assumes ps ∈ snd ' set pth
obtains pth1 pth2 e where pth = pth1@[e]@pth2 and snd e = ps and ps ∉ snd ' set pth1
using assms
proof (atomize-elim, induction pth)
  case Nil thus ?case by simp
next
  case (Cons e pth)

```

```

show ?case
proof(cases snd e = ps)
  case True
    hence e # pth = [] @ [e] @ pth ∧ snd e = ps ∧ ps ∉ snd ' set [] by auto
    thus ?thesis by (intro exI)
  next
    case False
    with Cons(2)
    have ps ∈ snd ' set pth by auto
    from Cons.IH[OF this this]
    obtain pth1 e' pth2 where pth = pth1 @ [e'] @ pth2 ∧ snd e' = ps ∧ ps ∉ snd ' set pth1 by auto
    with False
    have e#pth = (e# pth1) @ [e'] @ pth2 ∧ snd e' = ps ∧ ps ∉ snd ' set (e#pth1) by auto
    thus ?thesis by blast
  qed
qed

```

lemma scope-split:

```

assumes v ∈ scope ps
assumes path v v' pth
assumes terminal-vertex v'
obtains pth1 e pth2
  where pth = (pth1@[e])@pth2 and path v (fst ps) (pth1@[e]) and path (fst ps) v' pth2 and snd e = ps
and ps ∉ snd ' set pth1
proof-
  from assms
  have ps ∈ snd ' set pth by (auto simp add: scope.simps)
  then obtain pth1 pth2 e where pth = pth1@[e]@pth2 and snd e = ps and ps ∉ snd ' set pth1 by (rule
snd-set-split)

```

from ⟨path - -⟩ **and** ⟨pth = pth1@[e]@pth2⟩

have path v (edge-end e) (pth1@[e]) **and** path (edge-end e) v' pth2 **by** (metis path-split)+

show thesis

proof(rule that)

show pth = (pth1@[e])@pth2 **using** ⟨pth= -⟩ **by** simp

show path v (fst ps) (pth1@[e]) **using** ⟨path v (edge-end e) (pth1@[e])⟩ ⟨snd e = ps⟩ **by** (simp add: edge-end-tup)

show path (fst ps) v' pth2 **using** ⟨path (edge-end e) v' pth2⟩ ⟨snd e = ps⟩ **by** (simp add: edge-end-tup)

show ps ∉ snd ' set pth1 **by** fact

show snd e = ps **by** fact

qed

qed

end

This adds well-formedness conditions to the edges and vertices.

locale Port-Graph = Pre-Port-Graph +

assumes valid-nodes: nodeOf ' fset vertices ⊆ sset nodes

assumes valid-edges: ∀ (ps1,ps2) ∈ edges. valid-out-port ps1 ∧ valid-in-port ps2

begin

lemma snd-set-path-verties: path v v' pth ⇒ fst ' snd ' set pth ⊆ fset vertices

apply (induction rule: path.induct)

apply auto

apply (metis valid-in-port.elims(2) edge-end.simps case-prodD valid-edges)

done

lemma fst-set-path-verties: path v v' pth ⇒ fst ' fst ' set pth ⊆ fset vertices

apply (induction rule: path.induct)


```

apply auto
apply (metis valid-out-port.elims(2) edge-begin.simps case-prodD valid-edges)
done
end

```

A pruned graph is one where every node has a path to a terminal node (which will be the conclusions).

```

locale Pruned-Port-Graph = Port-Graph +
  assumes pruned:  $\bigwedge v. v \in \text{vertices} \implies (\exists \text{pth } v'. \text{path } v \text{ } v' \text{ pth} \wedge \text{terminal-vertex } v')$ 
begin
  lemma scopes-not-refl:
    assumes  $v \in \text{vertices}$ 
    shows  $v \notin \text{scope } (v,p)$ 
  proof(rule notI)
    assume  $v \in \text{scope } (v,p)$ 

    from pruned[OF assms]
    obtain pth t where terminal-vertex t and path v t pth and least:  $\forall \text{pth}'. \text{path } v \text{ } t \text{ pth}' \implies \text{length } \text{pth} \leq \text{length } \text{pth}'$ 
    by atomize-elim (auto simp del: terminal-vertex.simps elim: ex-has-least-nat)

    from scope-split[OF  $\langle v \in \text{scope } (v,p) \rangle \langle \text{path } v \text{ } t \text{ pth} \rangle \langle \text{terminal-vertex } t \rangle$ ]
    obtain pth1 e pth2 where  $\text{pth} = (\text{pth1} @ [e]) @ \text{pth2}$  path v t pth2 by (metis fst-conv)

    from this(2) least
    have  $\text{length } \text{pth} \leq \text{length } \text{pth2}$  by auto
    with  $\langle \text{pth} = \rightarrow \rangle$ 
    show False by auto
qed

```

This lemma can be found in [Bre16], but it is otherwise inconsequential.

```

lemma scopes-nest:
  fixes ps1 ps2
  shows  $\text{scope } \text{ps1} \subseteq \text{scope } \text{ps2} \vee \text{scope } \text{ps2} \subseteq \text{scope } \text{ps1} \vee \text{scope } \text{ps1} \cap \text{scope } \text{ps2} = \{\}$ 
proof(cases ps1 = ps2)
  assume  $\text{ps1} \neq \text{ps2}$ 
  {
  fix v
  assume  $v \in \text{scope } \text{ps1} \cap \text{scope } \text{ps2}$ 
  hence  $v \in \text{vertices}$  using scope.simps by auto
  then obtain pth t where path v t pth and terminal-vertex t using pruned by blast

  from  $\langle \text{path } v \text{ } t \text{ pth} \rangle$  and  $\langle \text{terminal-vertex } t \rangle$  and  $\langle v \in \text{scope } \text{ps1} \cap \text{scope } \text{ps2} \rangle$ 
  obtain pth1a e1 pth1b where  $\text{pth} = (\text{pth1a} @ [e1]) @ \text{pth1b}$  and path v (fst ps1) (pth1a@[e1]) and  $\text{snd } e1 = \text{ps1}$  and  $\text{ps1} \notin \text{snd } e1$  set pth1a
    by (auto elim: scope-split)

  from  $\langle \text{path } v \text{ } t \text{ pth} \rangle$  and  $\langle \text{terminal-vertex } t \rangle$  and  $\langle v \in \text{scope } \text{ps1} \cap \text{scope } \text{ps2} \rangle$ 
  obtain pth2a e2 pth2b where  $\text{pth} = (\text{pth2a} @ [e2]) @ \text{pth2b}$  and path v (fst ps2) (pth2a@[e2]) and  $\text{snd } e2 = \text{ps2}$  and  $\text{ps2} \notin \text{snd } e2$  set pth2a
    by (auto elim: scope-split)

  from  $\langle \text{pth} = (\text{pth1a} @ [e1]) @ \text{pth1b} \rangle$   $\langle \text{pth} = (\text{pth2a} @ [e2]) @ \text{pth2b} \rangle$ 
  have  $\text{set } \text{pth1a} \subseteq \text{set } \text{pth2a} \vee \text{set } \text{pth2a} \subseteq \text{set } \text{pth1a}$  by (auto simp add: append-eq-append-conv2)
  hence  $\text{scope } \text{ps1} \subseteq \text{scope } \text{ps2} \vee \text{scope } \text{ps2} \subseteq \text{scope } \text{ps1}$ 
  proof
    assume  $\text{set } \text{pth1a} \subseteq \text{set } \text{pth2a}$  with  $\langle \text{ps2} \notin \rightarrow \rangle$ 

```

```

have ps2 ∉ snd ‘ set (pth1a@[e1]) using ⟨ps1 ≠ ps2⟩ ⟨snd e1 = ps1⟩ by auto

have scope ps1 ⊆ scope ps2
proof
  fix v'
  assume v' ∈ scope ps1
  hence v' ∈ vertices using scope.simps by auto
  thus v' ∈ scope ps2
  proof(rule scope.intros)
    fix pth' t'
    assume path v' t' pth' and terminal-vertex t'
    with ⟨v' ∈ scope ps1⟩
    obtain pth3a e3 pth3b where pth' = (pth3a@[e3])@pth3b and path (fst ps1) t' pth3b
      by (auto elim: scope-split)

    have path v t' ((pth1a@[e1]) @ pth3b) using ⟨path v (fst ps1) (pth1a@[e1])⟩ and ⟨path (fst ps1) t'
pth3b⟩
      by (rule path-appendI)
    with ⟨terminal-vertex t'⟩ ⟨v ∈ scope ps1 ∩ scope ps2⟩
    have ps2 ∈ snd ‘ set ((pth1a@[e1]) @ pth3b) by (meson IntD2 scope.cases)
    hence ps2 ∈ snd ‘ set pth3b using ⟨ps2 ∉ snd ‘ set (pth1a@[e1])⟩ by auto
    thus ps2 ∈ snd ‘ set pth' using ⟨pth' =-⟩ by auto
  qed
qed
thus ?thesis by simp
next
assume set pth2a ⊆ set pth1a with ⟨ps1 ∉ -⟩
have ps1 ∉ snd ‘ set (pth2a@[e2]) using ⟨ps1 ≠ ps2⟩ ⟨snd e2 = ps2⟩ by auto

have scope ps2 ⊆ scope ps1
proof
  fix v'
  assume v' ∈ scope ps2
  hence v' ∈ vertices using scope.simps by auto
  thus v' ∈ scope ps1
  proof(rule scope.intros)
    fix pth' t'
    assume path v' t' pth' and terminal-vertex t'
    with ⟨v' ∈ scope ps2⟩
    obtain pth3a e3 pth3b where pth' = (pth3a@[e3])@pth3b and path (fst ps2) t' pth3b
      by (auto elim: scope-split)

    have path v t' ((pth2a@[e2]) @ pth3b) using ⟨path v (fst ps2) (pth2a@[e2])⟩ and ⟨path (fst ps2) t'
pth3b⟩
      by (rule path-appendI)
    with ⟨terminal-vertex t'⟩ ⟨v ∈ scope ps1 ∩ scope ps2⟩
    have ps1 ∈ snd ‘ set ((pth2a@[e2]) @ pth3b) by (meson IntD1 scope.cases)
    hence ps1 ∈ snd ‘ set pth3b using ⟨ps1 ∉ snd ‘ set (pth2a@[e2])⟩ by auto
    thus ps1 ∈ snd ‘ set pth' using ⟨pth' =-⟩ by auto
  qed
qed
thus ?thesis by simp
qed
}
thus ?thesis by blast
qed simp
end

```

A well-scoped graph is one where a port marked to be a local hypothesis is only connected to the corresponding input port, either directly or via a path. It must not be, however, that there is a path from such a hypothesis to a terminal node that does not pass by the dedicated input port; this is expressed via scopes.

locale *Scoped-Graph* = *Port-Graph* + *Port-Graph-Signature-Scoped*

locale *Well-Scoped-Graph* = *Scoped-Graph* +

assumes *well-scoped*: $((v_1, p_1), (v_2, p_2)) \in \text{edges} \implies \text{hyps} (\text{nodeOf } v_1) p_1 = \text{Some } p' \implies (v_2, p_2) = (v_1, p') \vee v_2 \in \text{scope } (v_1, p')$

context *Scoped-Graph*

begin

definition *hyps-free* **where**

hyps-free *pth* = $(\forall v_1 p_1 v_2 p_2. ((v_1, p_1), (v_2, p_2)) \in \text{set } pth \longrightarrow \text{hyps} (\text{nodeOf } v_1) p_1 = \text{None})$

lemma *hyps-free-Nil*[*simp*]: *hyps-free* [] **by** (*simp* *add*: *hyps-free-def*)

lemma *hyps-free-Cons*[*simp*]: *hyps-free* (*e*#*pth*) \longleftrightarrow *hyps-free* *pth* \wedge *hyps* (*nodeOf* (*fst* (*fst* *e*))) (*snd* (*fst* *e*)) = *None*

by (*auto* *simp* *add*: *hyps-free-def*) (*metis* *prod.collapse*)

lemma *path-vertices-shift*:

assumes *path* *v* *v'* *pth*

shows *map* *fst* (*map* *fst* *pth*)@[*v*] = *v*#*map* *fst* (*map* *snd* *pth*)

using *assms* **by** *induction* *auto*

inductive *terminal-path* **where**

terminal-path-empty: *terminal-vertex* *v* \implies *terminal-path* *v* *v* [] |

terminal-path-cons: $((v_1, p_1), (v_2, p_2)) \in \text{edges} \implies \text{terminal-path } v_2 v' pth \implies \text{hyps} (\text{nodeOf } v_1) p_1 = \text{None} \implies \text{terminal-path } v_1 v' (((v_1, p_1), (v_2, p_2))\#pth)$

lemma *terminal-path-is-path*:

assumes *terminal-path* *v* *v'* *pth*

shows *path* *v* *v'* *pth*

using *assms* **by** *induction* (*auto* *simp* *add*: *path-cons-simp*)

lemma *terminal-path-is-hyps-free*:

assumes *terminal-path* *v* *v'* *pth*

shows *hyps-free* *pth*

using *assms* **by** *induction* (*auto* *simp* *add*: *hyps-free-def*)

lemma *terminal-path-end-is-terminal*:

assumes *terminal-path* *v* *v'* *pth*

shows *terminal-vertex* *v'*

using *assms* **by** *induction*

lemma *terminal-pathI*:

assumes *path* *v* *v'* *pth*

assumes *hyps-free* *pth*

assumes *terminal-vertex* *v'*

shows *terminal-path* *v* *v'* *pth*

using *assms*

by *induction* (*auto* *intro*: *terminal-path.intros*)

end

An acyclic graph is one where there are no non-trivial cyclic paths (disregarding edges that are local

hypotheses – these are naturally and benignly cyclic).

```

locale Acyclic-Graph = Scoped-Graph +
  assumes hyps-free-acyclic:  $path\ v\ v'\ pth \implies hyps\text{-}free\ pth \implies pth = []$ 
begin
lemma hyps-free-vertices-distinct:
  assumes terminal-path  $v\ v'\ pth$ 
  shows distinct ( $map\ fst\ (map\ fst\ pth)@[v']$ )
using assms
proof(induction  $v\ v'\ pth$ )
  case terminal-path-empty
  show ?case by simp
next
  case (terminal-path-cons  $v_1\ p_1\ v_2\ p_2\ v'\ pth$ )
  note terminal-path-cons.IH
  moreover
  have  $v_1 \notin fst\ 'fst\ 'set\ pth$ 
  proof
    assume  $v_1 \in fst\ 'fst\ 'set\ pth$ 
    then obtain  $pth1\ e'\ pth2$  where  $pth = pth1@[e']@pth2$  and  $v_1 = fst\ (fst\ e')$ 
    apply (atomize-elim)
    apply (induction  $pth$ )
    apply (solves simp)
    apply (auto)
    apply (solves  $\langle rule\ exI[where\ x = []];\ simp \rangle$ )
    apply (metis Cons-eq-appendI image-eqI prod.sel(1))
    done
  with terminal-path-is-path[OF  $\langle terminal\text{-}path\ v_2\ v'\ pth \rangle$ ]
  have  $path\ v_2\ v_1\ pth1$  by (simp add: path-split2 edge-begin-tup)
  with  $\langle ((v_1, p_1), (v_2, p_2)) \in - \rangle$ 
  have  $path\ v_1\ v_1\ (((v_1, p_1), (v_2, p_2)) \# pth1)$  by (simp add: path-cons-simp)
  moreover
  from terminal-path-is-hyps-free[OF  $\langle terminal\text{-}path\ v_2\ v'\ pth \rangle$ ]
     $\langle hyps\ (nodeOf\ v_1)\ p_1 = None \rangle$ 
     $\langle pth = pth1@[e']@pth2 \rangle$ 
  have  $hyps\text{-}free\ (((v_1, p_1), (v_2, p_2)) \# pth1)$ 
    by (auto simp add: hyps-free-def)
  ultimately
  show False using hyps-free-acyclic by blast
qed
moreover
have  $v_1 \neq v'$ 
  using hyps-free-acyclic path-cons terminal-path-cons.hyps(1) terminal-path-cons.hyps(2) terminal-path-cons.hyps(3)
terminal-path-is-hyps-free terminal-path-is-path by fastforce
  ultimately
  show ?case by (auto simp add: comp-def)
qed

```

```

lemma hyps-free-vertices-distinct':
  assumes terminal-path  $v\ v'\ pth$ 
  shows distinct ( $v \# map\ fst\ (map\ snd\ pth)$ )
  using hyps-free-vertices-distinct[OF assms]
  unfolding path-vertices-shift[OF terminal-path-is-path[OF assms]]
  .

```

```

lemma hyps-free-limited:
  assumes terminal-path  $v\ v'\ pth$ 
  shows  $length\ pth \leq fcard\ vertices$ 

```

```

proof–
  have  $\text{length } pth = \text{length } (\text{map } \text{fst } (\text{map } \text{fst } pth))$  by simp
  also
  from hyps-free-vertices-distinct[OF assms]
  have  $\text{distinct } (\text{map } \text{fst } (\text{map } \text{fst } pth))$  by simp
  hence  $\text{length } (\text{map } \text{fst } (\text{map } \text{fst } pth)) = \text{card } (\text{set } (\text{map } \text{fst } (\text{map } \text{fst } pth)))$ 
    by (rule distinct-card[symmetric])
  also have  $\dots \leq \text{card } (\text{fset } \text{vertices})$ 
  proof (rule card-mono[OF finite-fset])
    from assms(1)
    show  $\text{set } (\text{map } \text{fst } (\text{map } \text{fst } pth)) \subseteq \text{fset } \text{vertices}$ 
      by (induction v v' pth) (auto, metis valid-edges case-prodD valid-out-port.simps)
  qed
  also have  $\dots = \text{fcard } \text{vertices}$  by (simp add: fcard.rep-eq)
  finally show ?thesis.
qed

```

lemma *hyps-free-path-not-in-scope*:

```

  assumes terminal-path v t pth
  assumes  $(v', p') \in \text{snd } \text{' set } pth$ 
  shows  $v' \notin \text{scope } (v, p)$ 
proof
  assume  $v' \in \text{scope } (v, p)$ 

  from  $\langle (v', p') \in \text{snd } \text{' set } pth \rangle$ 
  obtain pth1 pth2 e where  $pth = pth1 @ [e] @ pth2$  and  $\text{snd } e = (v', p')$  by (rule snd-set-split)
  from terminal-path-is-path[OF assms(1), unfolded  $\langle pth = - \rangle$ ]  $\langle \text{snd } e = - \rangle$ 
  have  $\text{path } v \ v' (pth1 @ [e])$  and  $\text{path } v' \ t \ pth2$  unfolding path-split by (auto simp add: edge-end-tup)

  from  $\langle v' \in \text{scope } (v, p) \rangle$  terminal-path-end-is-terminal[OF assms(1)]  $\langle \text{path } v' \ t \ pth2 \rangle$ 
  have  $(v, p) \in \text{snd } \text{' set } pth2$  by (rule scope-find)
  then obtain pth2a e' pth2b where  $pth2 = pth2a @ [e'] @ pth2b$  and  $\text{snd } e' = (v, p)$  by (rule snd-set-split)
  from  $\langle \text{path } v' \ t \ pth2 \rangle$  [unfolded  $\langle pth2 = - \rangle$ ]  $\langle \text{snd } e' = - \rangle$ 
  have  $\text{path } v' \ v (pth2a @ [e'])$  and  $\text{path } v \ t \ pth2b$  unfolding path-split by (auto simp add: edge-end-tup)

  from  $\langle \text{path } v \ v' (pth1 @ [e]) \rangle$   $\langle \text{path } v' \ v (pth2a @ [e']) \rangle$ 
  have  $\text{path } v \ v ((pth1 @ [e]) @ (pth2a @ [e']))$  by (rule path-appendI)
  moreover
  from terminal-path-is-hyps-free[OF assms(1)]  $\langle pth = - \rangle$   $\langle pth2 = - \rangle$ 
  have hyps-free  $((pth1 @ [e]) @ (pth2a @ [e']))$  by (auto simp add: hyps-free-def)
  ultimately
  have  $((pth1 @ [e]) @ (pth2a @ [e'])) = []$  by (rule hyps-free-acyclic)
  thus False by simp
qed

```

end

A saturated graph is one where every input port is incident to an edge.

```

locale Saturated-Graph = Port-Graph +
  assumes saturated: valid-in-port  $(v, p) \implies \exists e \in \text{edges} . \text{snd } e = (v, p)$ 

```

These four conditions make up a well-shaped graph.

```

locale Well-Shaped-Graph = Well-Scoped-Graph + Acyclic-Graph + Saturated-Graph + Pruned-Port-Graph

```

Next we demand an instantiation. This consists of a unique natural number per vertex, in order to rename the local constants apart, and furthermore a substitution per block which instantiates the

schematic formulas given in *Labeled-Signature*.

```

locale Instantiation =
  Vertex-Graph nodes - - vertices - +
  Labeled-Signature nodes - - - labelsIn labelsOut +
  Abstract-Formulas freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP
  for nodes :: 'node stream and edges :: ('vertex, 'outPort, 'inPort) edge set and vertices :: 'vertex fset and
labelsIn :: 'node  $\Rightarrow$  'inPort  $\Rightarrow$  'form and labelsOut :: 'node  $\Rightarrow$  'outPort  $\Rightarrow$  'form
  and freshenLC :: nat  $\Rightarrow$  'var  $\Rightarrow$  'var
    and renameLCs :: ('var  $\Rightarrow$  'var)  $\Rightarrow$  'form  $\Rightarrow$  'form
    and lconsts :: 'form  $\Rightarrow$  'var set
    and closed :: 'form  $\Rightarrow$  bool
    and subst :: 'subst  $\Rightarrow$  'form  $\Rightarrow$  'form
    and subst-lconsts :: 'subst  $\Rightarrow$  'var set
    and subst-renameLCs :: ('var  $\Rightarrow$  'var)  $\Rightarrow$  ('subst  $\Rightarrow$  'subst)
    and anyP :: 'form +
  fixes vidx :: 'vertex  $\Rightarrow$  nat
    and inst :: 'vertex  $\Rightarrow$  'subst
  assumes vidx-inj: inj-on vidx (fset vertices)
begin
  definition labelAtIn :: 'vertex  $\Rightarrow$  'inPort  $\Rightarrow$  'form where
    labelAtIn v p = subst (inst v) (freshen (vidx v) (labelsIn (nodeOf v) p))
  definition labelAtOut :: 'vertex  $\Rightarrow$  'outPort  $\Rightarrow$  'form where
    labelAtOut v p = subst (inst v) (freshen (vidx v) (labelsOut (nodeOf v) p))
end

```

A solution is an instantiation where on every edge, both incident ports are labeled with the same formula.

```

locale Solution =
  Instantiation - - - - edges for edges :: (('vertex  $\times$  'outPort)  $\times$  'vertex  $\times$  'inPort) set +
  assumes solved:  $((v_1, p_1), (v_2, p_2)) \in \text{edges} \implies \text{labelAtOut } v_1 \ p_1 = \text{labelAtIn } v_2 \ p_2$ 

```

```

locale Proof-Graph = Well-Shaped-Graph + Solution

```

If we have locally scoped constants, we demand that only blocks in the scope of the corresponding input port may mention such a locally scoped variable in its substitution.

```

locale Well-Scoped-Instantiation =
  Pre-Port-Graph nodes inPorts outPorts vertices nodeOf edges +
  Instantiation inPorts outPorts nodeOf hyps nodes edges vertices labelsIn labelsOut freshenLC renameLCs
lconsts closed subst subst-lconsts subst-renameLCs anyP vidx inst +
  Port-Graph-Signature-Scoped-Vars nodes inPorts outPorts freshenLC renameLCs lconsts closed subst subst-lconsts
subst-renameLCs anyP local-vars
  for freshenLC :: nat  $\Rightarrow$  'var  $\Rightarrow$  'var
    and renameLCs :: ('var  $\Rightarrow$  'var)  $\Rightarrow$  'form  $\Rightarrow$  'form
    and lconsts :: 'form  $\Rightarrow$  'var set
    and closed :: 'form  $\Rightarrow$  bool
    and subst :: 'subst  $\Rightarrow$  'form  $\Rightarrow$  'form
    and subst-lconsts :: 'subst  $\Rightarrow$  'var set
    and subst-renameLCs :: ('var  $\Rightarrow$  'var)  $\Rightarrow$  ('subst  $\Rightarrow$  'subst)
    and anyP :: 'form
    and inPorts :: 'node  $\Rightarrow$  'inPort fset
    and outPorts :: 'node  $\Rightarrow$  'outPort fset
    and nodeOf :: 'vertex  $\Rightarrow$  'node
    and hyps :: 'node  $\Rightarrow$  'outPort  $\Rightarrow$  'inPort option
    and nodes :: 'node stream
    and vertices :: 'vertex fset
    and labelsIn :: 'node  $\Rightarrow$  'inPort  $\Rightarrow$  'form

```

```

and labelsOut :: 'node ⇒ 'outPort ⇒ 'form
and vidx :: 'vertex ⇒ nat
and inst :: 'vertex ⇒ 'subst
and edges :: ('vertex, 'outPort, 'inPort) edge set
and local-vars :: 'node ⇒ 'inPort ⇒ 'var set +
assumes well-scoped-inst:
  valid-in-port (v,p) ⇒
    var ∈ local-vars (nodeOf v) p ⇒
    v' |∈| vertices ⇒
    freshenLC (vidx v) var ∈ subst-lconsts (inst v') ⇒
    v' ∈ scope (v,p)
begin
  lemma out-of-scope: valid-in-port (v,p) ⇒ v' |∈| vertices ⇒ v' ∉ scope (v,p) ⇒ freshenLC (vidx v) ‘
  local-vars (nodeOf v) p ∩ subst-lconsts (inst v') = {}
  using well-scoped-inst by auto
end

```

The following locale assembles all these conditions.

```

locale Scoped-Proof-Graph =
  Instantiation inPorts outPorts nodeOf hyps nodes edges vertices labelsIn labelsOut freshenLC renameLCs
  lconsts closed subst subst-lconsts subst-renameLCs anyP vidx inst +
  Well-Shaped-Graph nodes inPorts outPorts vertices nodeOf edges hyps +
  Solution inPorts outPorts nodeOf hyps nodes vertices labelsIn labelsOut freshenLC renameLCs lconsts closed
  subst subst-lconsts subst-renameLCs anyP vidx inst edges +
  Well-Scoped-Instantiation freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP
  inPorts outPorts nodeOf hyps nodes vertices labelsIn labelsOut vidx inst edges local-vars
  for freshenLC :: nat ⇒ 'var ⇒ 'var
  and renameLCs :: ('var ⇒ 'var) ⇒ 'form ⇒ 'form
  and lconsts :: 'form ⇒ 'var set
  and closed :: 'form ⇒ bool
  and subst :: 'subst ⇒ 'form ⇒ 'form
  and subst-lconsts :: 'subst ⇒ 'var set
  and subst-renameLCs :: ('var ⇒ 'var) ⇒ ('subst ⇒ 'subst)
  and anyP :: 'form
  and inPorts :: 'node ⇒ 'inPort fset
  and outPorts :: 'node ⇒ 'outPort fset
  and nodeOf :: 'vertex ⇒ 'node
  and hyps :: 'node ⇒ 'outPort ⇒ 'inPort option
  and nodes :: 'node stream
  and vertices :: 'vertex fset
  and labelsIn :: 'node ⇒ 'inPort ⇒ 'form
  and labelsOut :: 'node ⇒ 'outPort ⇒ 'form
  and vidx :: 'vertex ⇒ nat
  and inst :: 'vertex ⇒ 'subst
  and edges :: ('vertex, 'outPort, 'inPort) edge set
  and local-vars :: 'node ⇒ 'inPort ⇒ 'var set

end

```

4.3 Abstract_Rules_To_Incredible

```

theory Abstract-Rules-To-Incredible
imports
  Main
  HOL-Library.FSet
  HOL-Library.Stream

```

Incredible-Deduction
Abstract-Rules

begin

In this theory, the abstract rules given in *Incredible-Proof-Machine.Abstract-Rules* are used to create a proper signature.

Besides the rules given there, we have nodes for assumptions, conclusions, and the helper block.

datatype ('form, 'rule) graph-node = Assumption 'form | Conclusion 'form | Rule 'rule | Helper

type-synonym ('form, 'var) in-port = ('form, 'var) antecedent

type-synonym 'form reg-out-port = 'form

type-synonym 'form hyp = 'form

datatype ('form, 'var) out-port = Reg 'form reg-out-port | Hyp 'form hyp ('form, 'var) in-port

type-synonym ('v, 'form, 'var) edge' = (('v × ('form, 'var) out-port) × ('v × ('form, 'var) in-port))

context Abstract-Task

begin

definition nodes :: ('form, 'rule) graph-node stream **where**

nodes = Helper ## shift (map Assumption assumptions) (shift (map Conclusion conclusions) (smap Rule rules))

lemma Helper-in-nodes[simp]:

Helper ∈ sset nodes **by** (simp add: nodes-def)

lemma Assumption-in-nodes[simp]:

Assumption a ∈ sset nodes \longleftrightarrow a ∈ set assumptions **by** (auto simp add: nodes-def stream.set-map)

lemma Conclusion-in-nodes[simp]:

Conclusion c ∈ sset nodes \longleftrightarrow c ∈ set conclusions **by** (auto simp add: nodes-def stream.set-map)

lemma Rule-in-nodes[simp]:

Rule r ∈ sset nodes \longleftrightarrow r ∈ sset rules **by** (auto simp add: nodes-def stream.set-map)

fun inPorts' :: ('form, 'rule) graph-node \Rightarrow ('form, 'var) in-port list **where**

inPorts' (Rule r) = antecedent r

inPorts' (Assumption r) = []

inPorts' (Conclusion r) = [plain-ant r]

inPorts' Helper = [plain-ant anyP]

fun inPorts :: ('form, 'rule) graph-node \Rightarrow ('form, 'var) in-port fset **where**

inPorts (Rule r) = f-antecedent r

inPorts (Assumption r) = {||}

inPorts (Conclusion r) = { | plain-ant r | }

inPorts Helper = { | plain-ant anyP | }

lemma inPorts-fset-of:

inPorts n = fset-from-list (inPorts' n)

by (cases n rule: inPorts.cases) (auto simp: f-antecedent-def)

definition outPortsRule **where**

outPortsRule r = ffUnion ((λ a. (λ h. Hyp h a) |[†] a-hyps a) |[†] f-antecedent r) |[†] Reg |[†] f-consequent r

lemma Reg-in-outPortsRule[simp]: Reg c |∈| outPortsRule r \longleftrightarrow c |∈| f-consequent r

by (auto simp add: outPortsRule-def ffUnion.rep-eq)

lemma Hyp-in-outPortsRule[simp]: Hyp h c |∈| outPortsRule r \longleftrightarrow c |∈| f-antecedent r ∧ h |∈| a-hyps c

by (auto simp add: outPortsRule-def ffUnion.rep-eq)

fun outPorts **where**

outPorts (Rule r) = outPortsRule r


```

|outPorts (Assumption r) = {|Reg r|}
|outPorts (Conclusion r) = {|}|
|outPorts Helper = {| Reg anyP |}

```

```

fun labelsIn where
  labelsIn - p = a-conc p

```

```

fun labelsOut where
  labelsOut - (Reg p) = p
| labelsOut - (Hyp h c) = h

```

```

fun hyps where
  hyps (Rule r) (Hyp h a) = (if a |∈| f-antecedent r ∧ h |∈| a-hyps a then Some a else None)
| hyps - - = None

```

```

fun local-vars :: ('form, 'rule) graph-node ⇒ ('form, 'var) in-port ⇒ 'var set where
  local-vars - a = a-fresh a

```

sublocale Labeled-Signature nodes inPorts outPorts hyps labelsIn labelsOut

proof(standard,goal-cases)

```

case (1 n p1 p2)
thus ?case by(induction n p1 rule: hyps.induct) (auto split: if-splits)
qed

```

lemma hyps-for-conclusion[simp]: hyps-for (Conclusion n) p = {|}|

using hyps-for-subset **by** auto

lemma hyps-for-Helper[simp]: hyps-for Helper p = {|}|

using hyps-for-subset **by** auto

lemma hyps-for-Rule[simp]: ip |∈| f-antecedent r ⇒ hyps-for (Rule r) ip = (λ h. Hyp h ip) |' a-hyps ip
by (auto elim!: hyps.elims split: if-splits)

end

Finally, a given proof graph solves the task at hand if all the given conclusions are present as conclusion blocks in the graph.

locale Tasked-Proof-Graph =

Abstract-Task freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP antecedent consequent rules assumptions conclusions +

Scoped-Proof-Graph freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP inPorts outPorts nodeOf hyps nodes vertices labelsIn labelsOut vidx inst edges local-vars

```

for freshenLC :: nat ⇒ 'var ⇒ 'var
and renameLCs :: ('var ⇒ 'var) ⇒ 'form ⇒ 'form
and lconsts :: 'form ⇒ 'var set
and closed :: 'form ⇒ bool
and subst :: 'subst ⇒ 'form ⇒ 'form
and subst-lconsts :: 'subst ⇒ 'var set
and subst-renameLCs :: ('var ⇒ 'var) ⇒ ('subst ⇒ 'subst)
and anyP :: 'form

```

and antecedent :: 'rule ⇒ ('form, 'var) antecedent list

and consequent :: 'rule ⇒ 'form list

and rules :: 'rule stream

and assumptions :: 'form list

and conclusions :: 'form list

and vertices :: 'vertex fset

```
and nodeOf :: 'vertex  $\Rightarrow$  ('form, 'rule) graph-node
and edges :: ('vertex, 'form, 'var) edge' set
and vidx :: 'vertex  $\Rightarrow$  nat
and inst :: 'vertex  $\Rightarrow$  'subst +
assumes conclusions-present: set (map Conclusion conclusions)  $\subseteq$  nodeOf ' fset vertices

end
```

5 Natural Deduction

5.1 Natural_Deduction

```
theory Natural-Deduction
imports
  Abstract-Completeness.Abstract-Completeness
  Abstract-Rules
  Entailment
begin
```

Our formalization of natural deduction builds on *Abstract-Completeness.Abstract-Completeness* and refines and concretizes the structure given there as follows

- The judgements are entailments consisting of a finite set of assumptions and a conclusion, which are abstract formulas in the sense of *Incredible-Proof-Machine.Abstract-Formula*.
- The abstract rules given in *Incredible-Proof-Machine.Abstract-Rules* are used to decide the validity of a step in the derivation.

A single setep in the derivation can either be the axiom rule, the cut rule, or one of the given rules in *Incredible-Proof-Machine.Abstract-Rules*.

```
datatype 'rule NatRule = Axiom | NatRule 'rule | Cut
```

The following locale is still abstract in the set of rules (*nat-rule*), but implements the bookkeeping logic for assumptions, the *Axiom* rule and the *Cut* rule.

```
locale ND-Rules-Inst =
  Abstract-Formulas freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP
  for freshenLC :: nat  $\Rightarrow$  'var  $\Rightarrow$  'var
  and renameLCs :: ('var  $\Rightarrow$  'var)  $\Rightarrow$  'form  $\Rightarrow$  'form
  and lconsts :: 'form  $\Rightarrow$  'var set
  and closed :: 'form  $\Rightarrow$  bool
  and subst :: 'subst  $\Rightarrow$  'form  $\Rightarrow$  'form
  and subst-lconsts :: 'subst  $\Rightarrow$  'var set
  and subst-renameLCs :: ('var  $\Rightarrow$  'var)  $\Rightarrow$  ('subst  $\Rightarrow$  'subst)
  and anyP :: 'form +

  fixes nat-rule :: 'rule  $\Rightarrow$  'form  $\Rightarrow$  ('form, 'var) antecedent fset  $\Rightarrow$  bool
  and rules :: 'rule stream
begin
```

- An application of the *Axiom* rule is valid if the conclusion is among the assumptions.
- An application of a *NatRule* is more complicated. This requires some natural number a to rename local variables, and some instantiation s . It checks that
 - none of the local variables occur in the context of the judgement.
 - none of the local variables occur in the instantiation. Together, this implements the usual freshness side-conditions. Furthermore, for every antecedent of the rule, the (correctly renamed and instantiated) hypotheses need to be added to the context.
- The *Cut* rule is again easy.

```
inductive eff :: 'rule NatRule  $\Rightarrow$  'form entailment  $\Rightarrow$  'form entailment fset  $\Rightarrow$  bool where
```

```

con |∈| Γ
⇒ eff Axiom (Γ ⊢ con) {||}
|nat-rule rule c ants
⇒ (∧ ant f. ant |∈| ants ⇒ f |∈| Γ ⇒ freshenLC a ‘ (a-fresh ant) ∩ lconsts f = {})
⇒ (∧ ant. ant |∈| ants ⇒ freshenLC a ‘ (a-fresh ant) ∩ subst-lconsts s = {})
⇒ eff (NatRule rule)
  (Γ ⊢ subst s (freshen a c))
  ((λant. ((λp. subst s (freshen a p)) |‘ a-hyps ant |∪| Γ ⊢ subst s (freshen a (a-conc ant)))) |‘ ants)
|eff Cut (Γ ⊢ c’) {| (Γ ⊢ c’)|}

```

inductive-simps *eff-Cut-simps*[simp]: *eff Cut (Γ ⊢ c) S*

sublocale *RuleSystem-Defs* **where**

eff = *eff* **and** *rules* = *Cut* ## *Axiom* ## *smap NatRule rules*.

end

Now we instantiate the above locale. We duplicate each abstract rule (which can have multiple consequents) for each consequent, as the natural deduction formulation can only handle a single consequent per rule

context *Abstract-Task*

begin

inductive *natEff-Inst* **where**

c ∈ *set* (*consequent r*) ⇒ *natEff-Inst* (*r,c*) *c* (*f-antecedent r*)

definition *n-rules* **where**

n-rules = *flat* (*smap* (λ*r*. *map* (λ*c*. (*r,c*)) (*consequent r*)) *rules*)

sublocale *ND-Rules-Inst* - - - - - *natEff-Inst n-rules* ..

A task is solved if for every conclusion, there is a well-formed and finite tree that proves this conclusion, using only assumptions given in the task.

definition *solved* **where**

solved ↔ (∀ *c*. *c* |∈| *conc-forms* → (∃ Γ *t*. *fst* (*root t*) = (Γ ⊢ *c*) ∧ Γ |⊆| *ass-forms* ∧ *wf t* ∧ *tfinite t*)

end

end

6 Correctness

6.1 Incredible_Correctness

```
theory Incredible-Correctness
imports
  Abstract-Rules-To-Incredible
  Natural-Deduction
begin
```

In this theory, we prove that if we have a graph that proves a given abstract task (which is represented as the context *Tasked-Proof-Graph*), then we can prove *solved*.

```
context Tasked-Proof-Graph
begin
```

```
definition adjacentTo :: 'vertex  $\Rightarrow$  ('form, 'var) in-port  $\Rightarrow$  ('vertex  $\times$  ('form, 'var) out-port) where
adjacentTo v p = (SOME ps. (ps, (v,p))  $\in$  edges)
```

```
fun isReg where
  isReg v p = (case p of Hyp h c  $\Rightarrow$  False | Reg c  $\Rightarrow$ 
    (case nodeOf v of
      Conclusion a  $\Rightarrow$  False
    | Assumption a  $\Rightarrow$  False
    | Rule r  $\Rightarrow$  True
    | Helper  $\Rightarrow$  True
    ))
```

```
fun toNatRule where
  toNatRule v p = (case p of Hyp h c  $\Rightarrow$  Axiom | Reg c  $\Rightarrow$ 
    (case nodeOf v of
      Conclusion a  $\Rightarrow$  Axiom — a lie
    | Assumption a  $\Rightarrow$  Axiom
    | Rule r  $\Rightarrow$  NatRule (r,c)
    | Helper  $\Rightarrow$  Cut
    ))
```

```
inductive-set global-assms' :: 'var itself  $\Rightarrow$  'form set for i where
  v  $\in$  | vertices  $\Longrightarrow$  nodeOf v = Assumption p  $\Longrightarrow$  labelAtOut v (Reg p)  $\in$  global-assms' i
```

```
lemma finite-global-assms': finite (global-assms' i)
```

```
proof—
```

```
  have finite (fset vertices) by (rule finite-fset)
```

```
  moreover
```

```
  have global-assms' i  $\subseteq$  ( $\lambda$  v. case nodeOf v of Assumption p  $\Rightarrow$  labelAtOut v (Reg p)) ' fset vertices
  by (force simp add: global-assms'.simps image-iff )
```

```
  ultimately
```

```
  show ?thesis by (rule finite-surj)
```

```
qed
```

```
context includes fset.lifting
```

```
begin
```

```
  lift-definition global-assms :: 'var itself  $\Rightarrow$  'form fset is global-assms' by (rule finite-global-assms')
```

```
  lemmas global-assmsI = global-assms'.intros[Transfer.transferred]
```

```
  lemmas global-assms-simps = global-assms'.simps[Transfer.transferred]
```

```
end
```

```
fun extra-assms :: ('vertex × ('form, 'var) in-port) ⇒ 'form fset where
  extra-assms (v, p) = (λ p. labelAtOut v p) |' hyps-for (nodeOf v) p
```

```
fun hyps-along :: ('vertex, 'form, 'var) edge' list ⇒ 'form fset where
  hyps-along pth = ffUnion (extra-assms |' snd |' fset-from-list pth) |∪| global-assms TYPE('var)
```

lemma hyps-alongE[consumes 1, case-names Hyp Assumption]:

```
assumes f |∈| hyps-along pth
obtains v p h where (v,p) ∈ snd ' set pth and f = labelAtOut v h and h |∈| hyps-for (nodeOf v) p
| v pf where v |∈| vertices and nodeOf v = Assumption pf f = labelAtOut v (Reg pf)
using assms
apply (auto simp add: ffUnion.rep-eq global-assms-simps)
apply (metis image-iff snd-conv)
done
```

Here we build the natural deduction tree, by walking the graph.

```
primcorec tree :: 'vertex ⇒ ('form, 'var) in-port ⇒ ('vertex, 'form, 'var) edge' list ⇒ (('form entailment),
('rule × 'form) NatRule) dtree where
```

```
root (tree v p pth) =
  ((hyps-along ((adjacentTo v p,(v,p))#pth) ⊢ labelAtIn v p),
  (case adjacentTo v p of (v', p') ⇒ toNatRule v' p'
  ))
| cont (tree v p pth) =
  (case adjacentTo v p of (v', p') ⇒
  (if isReg v' p' then ((λ p''. tree v' p'' ((adjacentTo v p,(v,p))#pth)) |' inPorts (nodeOf v')) else {||})
  ))
```

lemma fst-root-tree[simp]: fst (root (tree v p pth)) = (hyps-along ((adjacentTo v p,(v,p))#pth) ⊢ labelAtIn v p)
by simp

lemma out-port-cases[consumes 1, case-names Assumption Hyp Rule Helper]:

```
assumes p |∈| outPorts n
obtains
  a where n = Assumption a and p = Reg a
| r h c where n = Rule r and p = Hyp h c
| r f where n = Rule r and p = Reg f
| n = Helper and p = Reg anyP
using assms by (atomize-elim, cases p; cases n) auto
```

lemma hyps-for-fimage: hyps-for (Rule r) x = (if x |∈| f-antecedent r then (λ f. Hyp f x) |' (a-hyps x) else {||})

```
apply (rule fset-eqI)
apply (rename-tac p')
apply (case-tac p')
apply (auto simp add: split: if-splits out-port.splits)
done
```

Now we prove that the thus produced tree is well-formed.

theorem wf-tree:

```
assumes valid-in-port (v,p)
assumes terminal-path v t pth
shows wf (tree v p pth)
using assms
proof (coinduction arbitrary: v p pth)
```

```

case (wf v p pth)
  let ?t = tree v p pth
  from saturated[OF wf(1)]
  obtain v' p'
  where e:((v',p'),(v,p)) ∈ edges and [simp]: adjacentTo v p = (v',p')
    by (auto simp add: adjacentTo-def, metis (no-types, lifting) eq-fst-iff tfl-some)

  let ?e = ((v',p'),(v,p))
  let ?pth' = ?e#pth
  let ?Γ = hyps-along ?pth'
  let ?l = labelAtIn v p

  from e valid-edges have v' |∈| vertices and p' |∈| outPorts (nodeOf v') by auto
  hence nodeOf v' ∈ sset nodes using valid-nodes by (meson image-eqI subsetD)

  from ⟨?e ∈ edges⟩
  have s: labelAtOut v' p' = labelAtIn v p by (rule solved)

  from ⟨p' |∈| outPorts (nodeOf v')⟩
  show ?case
  proof (cases rule: out-port-cases)
    case (Hyp r h c)

    from Hyp ⟨p' |∈| outPorts (nodeOf v')⟩
    have h |∈| a-hyps c and c |∈| f-antecedent r by auto
    hence hyps (nodeOf v') (Hyp h c) = Some c using Hyp by simp

    from well-scoped[OF ⟨- ∈ edges⟩[unfolded Hyp] this]
    have (v, p) = (v', c) ∨ v ∈ scope (v', c).
    hence (v', c) ∈ insert (v, p) (snd ' set pth)
    proof
      assume (v, p) = (v', c)
      thus ?thesis by simp
    next
      assume v ∈ scope (v', c)
      from this terminal-path-end-is-terminal[OF wf(2)] terminal-path-is-path[OF wf(2)]
      have (v', c) ∈ snd ' set pth by (rule scope-find)
      thus ?thesis by simp
    qed
  moreover

  from ⟨hyps (nodeOf v') (Hyp h c) = Some c⟩
  have Hyp h c |∈| hyps-for (nodeOf v') c by simp
  hence labelAtOut v' (Hyp h c) |∈| extra-assms (v',c) by auto
  ultimately

  have labelAtOut v' (Hyp h c) |∈| ?Γ
    by (fastforce simp add: ffUnion.rep-eq)

  hence labelAtIn v p |∈| ?Γ by (simp add: s[symmetric] Hyp)
  thus ?thesis
    using Hyp
    apply (auto intro: exI[where x = ?t] simp add: eff.simps simp del: hyps-along.simps)
    done
  next
  case (Assumption f)

```

```

from ⟨v' |∈| vertices⟩ ⟨nodeOf v' = Assumption f⟩
have labelAtOut v' (Reg f) |∈| global-assms TYPE('var)
  by (rule global-assmsI)
hence labelAtOut v' (Reg f) |∈| ?Γ by auto
hence labelAtIn v p |∈| ?Γ by (simp add: s[symmetric] Assumption)
thus ?thesis using Assumption
  by (auto intro: exI[where x = ?t] simp add: eff.simps)
next
case (Rule r f)
with ⟨nodeOf v' ∈ sset nodes⟩
have r ∈ sset rules
  by (auto simp add: nodes-def stream.set-map)

from Rule
have hyps (nodeOf v') p' = None by simp
with e ⟨terminal-path v t pth⟩
have terminal-path v' t ?pth'..

from Rule ⟨p' |∈| outPorts (nodeOf v')⟩
have f |∈| f-consequent r by simp
hence f ∈ set (consequent r) by (simp add: f-consequent-def)
with ⟨r ∈ sset rules⟩
have NatRule (r, f) ∈ sset (smap NatRule n-rules)
  by (auto simp add: stream.set-map n-rules-def no-empty-conclusions)
moreover

{
from ⟨f |∈| f-consequent r⟩
have f ∈ set (consequent r) by (simp add: f-consequent-def)
hence natEff-Inst (r, f) f (f-antecedent r)
  by (rule natEff-Inst.intros)
hence eff (NatRule (r, f)) (?Γ ⊢ subst (inst v') (freshen (vidx v') f))
  ((λant. ((λp. subst (inst v') (freshen (vidx v') p)) |' a-hyps ant |∪| ?Γ ⊢ subst (inst v') (freshen (vidx
v') (a-conc ant)))) |' f-antecedent r)
  (is eff - - ?ants)
proof (rule eff.intros)
  fix ant f
  assume ant |∈| f-antecedent r
  from ⟨v' |∈| vertices⟩ ⟨ant |∈| f-antecedent r⟩
  have valid-in-port (v', ant) by (simp add: Rule)

assume f |∈| ?Γ
thus freshenLC (vidx v') ' a-fresh ant ∩ lconsts f = {}
proof (induct rule: hyps-alongE)
  case (Hyp v'' p'' h'')

from Hyp(1) snd-set-path-verties[OF terminal-path-is-path[OF ⟨terminal-path v' t ?pth'⟩]]
have v'' |∈| vertices by (force simp add:)

from ⟨terminal-path v' t ?pth'⟩ Hyp(1)
have v'' ∉ scope (v', ant) by (rule hyps-free-path-not-in-scope)
with ⟨valid-in-port (v', ant)⟩ ⟨v'' |∈| vertices⟩
have freshenLC (vidx v') ' local-vars (nodeOf v') ant ∩ subst-lconsts (inst v'') = {}
  by (rule out-of-scope)
moreover
from hyps-free-vertices-distinct'[OF ⟨terminal-path v' t ?pth'⟩] Hyp.hyps(1)

```



```

have  $v'' \neq v'$  by (metis distinct.simps(2) fst-conv image-eqI list.set-map)
hence  $\text{vidx } v'' \neq \text{vidx } v'$  using  $\langle v' \mid \in \mid \text{vertices} \rangle \langle v'' \mid \in \mid \text{vertices} \rangle$  by (meson vidx-inj inj-onD)
hence  $\text{freshenLC } (\text{vidx } v') \text{ ' a-fresh ant } \cap \text{freshenLC } (\text{vidx } v'') \text{ ' lconsts } (\text{labelsOut } (\text{nodeOf } v'') \text{ h}'') =$ 
{}by auto
moreover
have  $\text{lconsts } f \subseteq \text{lconsts } (\text{freshen } (\text{vidx } v'') (\text{labelsOut } (\text{nodeOf } v'') \text{ h}'')) \cup \text{subst-lconsts } (\text{inst } v'')$  using
 $\langle f = - \rangle$ 
by (simp add: labelAtOut-def fv-subst)
ultimately
show ?thesis
by (fastforce simp add: lconsts-freshen)
next
case (Assumption v pf)
hence  $f = \text{subst } (\text{inst } v) (\text{freshen } (\text{vidx } v) \text{ pf})$  by (simp add: labelAtOut-def)
moreover
from Assumption have Assumption  $pf \in \text{sset nodes}$  using valid-nodes by auto
hence  $pf \in \text{set assumptions}$  unfolding nodes-def by (auto simp add: stream.set-map)
hence closed pf by (rule assumptions-closed)
ultimately
have  $\text{lconsts } f = \{\}$  by (simp add: closed-no-lconsts lconsts-freshen subst-closed freshen-closed)
thus ?thesis by simp
qed
next
fix ant
assume  $\text{ant} \mid \in \mid f\text{-antecedent } r$ 
from  $\langle v' \mid \in \mid \text{vertices} \rangle \langle \text{ant} \mid \in \mid f\text{-antecedent } r \rangle$ 
have  $\text{valid-in-port } (v', \text{ant})$  by (simp add: Rule)
moreover
note  $\langle v' \mid \in \mid \text{vertices} \rangle$ 
moreover
hence  $v' \notin \text{scope } (v', \text{ant})$  by (rule scopes-not-refl)
ultimately
have  $\text{freshenLC } (\text{vidx } v') \text{ ' local-vars } (\text{nodeOf } v') \text{ ant } \cap \text{subst-lconsts } (\text{inst } v') = \{\}$ 
by (rule out-of-scope)
thus  $\text{freshenLC } (\text{vidx } v') \text{ ' a-fresh ant } \cap \text{subst-lconsts } (\text{inst } v') = \{\}$  by simp
qed
also
have  $\text{subst } (\text{inst } v') (\text{freshen } (\text{vidx } v') \text{ f}) = \text{labelAtOut } v' \text{ p'}$  using Rule by (simp add: labelAtOut-def)
also
note  $\langle \text{labelAtOut } v' \text{ p}' = \text{labelAtIn } v \text{ p} \rangle$ 
also
have ?ants =  $((\lambda x. (\text{extra-assms } (v', x) \mid \cup \mid \text{hyps-along } ?pth' \vdash \text{labelAtIn } v' \text{ x})) \mid \uparrow \mid f\text{-antecedent } r)$ 
by (rule fimage-cong[OF refl])
(auto simp add: labelAtIn-def labelAtOut-def Rule hyps-for-fimage ffUnion.rep-eq)
finally
have eff (NatRule (r, f))
( $? \Gamma, \text{labelAtIn } v \text{ p}$ )
 $((\lambda x. \text{extra-assms } (v', x) \mid \cup \mid ? \Gamma \vdash \text{labelAtIn } v' \text{ x}) \mid \uparrow \mid f\text{-antecedent } r).$ 
}
moreover
{ fix x
assume  $x \mid \in \mid \text{cont } ?t$ 
then obtain a where  $x = \text{tree } v' \text{ a } ?pth'$  and  $a \mid \in \mid f\text{-antecedent } r$ 
by (auto simp add: Rule)
note this(1)
moreover

```

```

from ⟨ $v' \in \text{vertices}$ ⟩ ⟨ $a \in \text{f-antecedent } r$ ⟩
have  $\text{valid-in-port } (v', a)$  by ( $\text{simp add: Rule}$ )
moreover

note ⟨ $\text{terminal-path } v' t \text{ ?pth}$ ⟩
ultimately

have  $\exists v p \text{ pth. } x = \text{tree } v p \text{ pth} \wedge \text{valid-in-port } (v, p) \wedge \text{terminal-path } v t \text{ pth}$ 
by  $\text{blast}$ 
}
ultimately

show  $\text{?thesis}$  using  $\text{Rule}$ 
by ( $\text{auto intro!: exI[where } x = \text{?t}] \text{ simp add: comp-def funion-assoc}$ )
next
case  $\text{Helper}$ 
from  $\text{Helper}$ 
have  $\text{hyps } (\text{nodeOf } v') p' = \text{None}$  by  $\text{simp}$ 
with  $e \langle \text{terminal-path } v t \text{ pth} \rangle$ 
have  $\text{terminal-path } v' t \text{ ?pth'..}$ 

have  $\text{labelAtIn } v' (\text{plain-ant anyP}) = \text{labelAtIn } v p$ 
unfolding  $s[\text{symmetric}]$ 
using  $\text{Helper}$  by ( $\text{simp add: labelAtIn-def labelAtOut-def}$ )
moreover
{ fix  $x$ 
assume  $x \in \text{cont } ?t$ 

hence  $x = \text{tree } v' (\text{plain-ant anyP}) \text{ ?pth'}$ 
by ( $\text{auto simp add: Helper}$ )
note  $\text{this}(1)$ 
moreover

from ⟨ $v' \in \text{vertices}$ ⟩
have  $\text{valid-in-port } (v', \text{plain-ant anyP})$  by ( $\text{simp add: Helper}$ )
moreover

note ⟨ $\text{terminal-path } v' t \text{ ?pth}$ ⟩
ultimately

have  $\exists v p \text{ pth. } x = \text{tree } v p \text{ pth} \wedge \text{valid-in-port } (v, p) \wedge \text{terminal-path } v t \text{ pth}$ 
by  $\text{blast}$ 
}
ultimately

show  $\text{?thesis}$  using  $\text{Helper}$ 
by ( $\text{auto intro!: exI[where } x = \text{?t}] \text{ simp add: comp-def funion-assoc}$  )
qed
qed

lemma  $\text{global-in-ass: global-assms TYPE('var)} \sqsubseteq \text{ass-forms}$ 
proof
fix  $x$ 
assume  $x \in \text{global-assms TYPE('var)}$ 
then obtain  $v \text{ pf}$  where  $v \in \text{vertices}$  and  $\text{nodeOf } v = \text{Assumption pf}$  and  $x = \text{labelAtOut } v (\text{Reg pf})$ 
by ( $\text{auto simp add: global-assms-simps}$ )

```

```

from this (1,2) valid-nodes
have Assumption pf ∈ sset nodes by (auto simp add:)
hence pf ∈ set assumptions by (auto simp add: nodes-def stream.set-map)
hence closed pf by (rule assumptions-closed)
with  $\langle x = \text{labelAtOut } v \text{ (Reg pf)} \rangle$ 
have  $x = pf$  by (auto simp add: labelAtOut-def lconsts-freshen closed-no-lconsts freshen-closed subst-closed)
thus  $x \in | \in | \text{ass-forms}$  using  $\langle pf \in \text{set assumptions} \rangle$  by (auto simp add: ass-forms-def)
qed

```

```

primcorec edge-tree :: 'vertex ⇒ ('form, 'var) in-port ⇒ ('vertex, 'form, 'var) edge' tree where
root (edge-tree v p) = (adjacentTo v p, (v,p))
| cont (edge-tree v p) =
  (case adjacentTo v p of (v', p') ⇒
    (if isReg v' p' then (( $\lambda p. \text{edge-tree } v' p$ ) |' inPorts (nodeOf v')) else {||})
  ))

```

```

lemma tfinite-map-tree: tfinite (map-tree f t)  $\longleftrightarrow$  tfinite t
proof

```

```

  assume tfinite (map-tree f t)
  thus tfinite t
  by (induction map-tree f t arbitrary: t rule: tfinite.induct)
  (fastforce intro: tfinite.intros simp add: tree.map-sel)

```

```

next

```

```

  assume tfinite t
  thus tfinite (map-tree f t)
  by (induction t rule: tfinite.induct)
  (fastforce intro: tfinite.intros simp add: tree.map-sel)

```

```

qed

```

```

lemma finite-tree-edge-tree:

```

```

  tfinite (tree v p pth)  $\longleftrightarrow$  tfinite (edge-tree v p)

```

```

proof–

```

```

  have map-tree ( $\lambda -. ()$ ) (tree v p pth) = map-tree ( $\lambda -. ()$ ) (edge-tree v p)
  by(coinduction arbitrary: v p pth)

```

```

  (fastforce simp add: tree.map-sel rel-fset-def rel-set-def split: prod.split out-port.split graph-node.split option.split)

```

```

  thus ?thesis by (metis tfinite-map-tree)

```

```

qed

```

```

coinductive forbidden-path :: 'vertex ⇒ ('vertex, 'form, 'var) edge' stream ⇒ bool where

```

```

  forbidden-path: ((v1,p1),(v2,p2)) ∈ edges ⇒ hyps (nodeOf v1) p1 = None ⇒ forbidden-path v1 pth ⇒
forbidden-path v2 (((v1,p1),(v2,p2))##pth)

```

```

lemma path-is-forbidden:

```

```

  assumes valid-in-port (v,p)

```

```

  assumes ipath (edge-tree v p) es

```

```

  shows forbidden-path v es

```

```

using assms

```

```

proof(coinduction arbitrary: v p es)

```

```

  case forbidden-path

```

```

  let ?es' = stl es

```

```

from forbidden-path(2)

```

```

obtain t' where root (edge-tree v p) = shd es and  $t' \in | \in | \text{cont}$  (edge-tree v p) and ipath t' ?es'

```

```

  by rule blast

```

```

from ⟨root (edge-tree v p) = shd es⟩
have [simp]: shd es = (adjacentTo v p, (v,p)) by simp

from saturated[OF ⟨valid-in-port (v,p)⟩]
obtain v' p'
where e:((v',p'),(v,p)) ∈ edges and [simp]: adjacentTo v p = (v',p')
by (auto simp add: adjacentTo-def, metis (no-types, lifting) eq-fst-iff tfl-some)
let ?e = ((v',p'),(v,p))

from e have p' |∈| outPorts (nodeOf v') using valid-edges by auto
thus ?case
proof(cases rule: out-port-cases)
  case Hyp
    with ⟨t' |∈| cont (edge-tree v p)⟩
    have False by auto
    thus ?thesis..
  next
    case Assumption
    with ⟨t' |∈| cont (edge-tree v p)⟩
    have False by auto
    thus ?thesis..
  next
    case (Rule r f)
    from ⟨t' |∈| cont (edge-tree v p)⟩ Rule
    obtain a where [simp]: t' = edge-tree v' a and a |∈| f-antecedent r by auto

    have es = ?e ## ?es' by (cases es rule: stream.exhaust-sel) simp
    moreover

    have ?e ∈ edges using e by simp
    moreover

    from ⟨p' = Reg f⟩ ⟨nodeOf v' = Rule r⟩
    have hyps (nodeOf v') p' = None by simp
    moreover

    from e valid-edges have v' |∈| vertices by auto
    with ⟨nodeOf v' = Rule r⟩ ⟨a |∈| f-antecedent r⟩
    have valid-in-port (v', a) by simp
    moreover

    have ipath (edge-tree v' a) ?es' using ⟨ipath t' -⟩ by simp
    ultimately

    show ?thesis by metis
  next
    case Helper
    from ⟨t' |∈| cont (edge-tree v p)⟩ Helper
    have [simp]: t' = edge-tree v' (plain-ant anyP) by simp

    have es = ?e ## ?es' by (cases es rule: stream.exhaust-sel) simp
    moreover

    have ?e ∈ edges using e by simp
    moreover

```

from $\langle p' = \text{Reg anyP} \rangle \langle \text{nodeOf } v' = \text{Helper} \rangle$
have $\text{hyps} (\text{nodeOf } v') p' = \text{None}$ **by** *simp*
moreover

from $e \text{ valid-edges}$ **have** $v' \in | \text{vertices} |$ **by** *auto*
with $\langle \text{nodeOf } v' = \text{Helper} \rangle$
have $\text{valid-in-port} (v', \text{plain-ant anyP})$ **by** *simp*
moreover

have $\text{ipath} (\text{edge-tree } v' (\text{plain-ant anyP})) ?es'$ **using** $\langle \text{ipath } t' \rightarrow \rangle$ **by** *simp*
ultimately

show $?thesis$ **by** *metis*

qed

qed

lemma *forbidden-path-prefix-is-path*:
assumes *forbidden-path* $v \text{ es}$
obtains v' **where** $\text{path } v' v (\text{rev} (\text{stake } n \text{ es}))$
using *assms*
apply (*atomize-elim*)
apply (*induction n arbitrary: v es*)
apply *simp*
apply (*simp add: path-snoc*)
apply (*subst (asm) (2) forbidden-path.simps*)
apply *auto*
done

lemma *forbidden-path-prefix-is-hyp-free*:
assumes *forbidden-path* $v \text{ es}$
shows *hyps-free* $(\text{rev} (\text{stake } n \text{ es}))$
using *assms*
apply (*induction n arbitrary: v es*)
apply (*simp add: hyps-free-def*)
apply (*subst (asm) (2) forbidden-path.simps*)
apply (*force simp add: hyps-free-def*)
done

And now we prove that the tree is finite, which requires the above notion of a *forbidden-path*, i.e. an infinite path.

theorem *finite-tree*:

assumes *valid-in-port* (v, p)

assumes *terminal-vertex* v

shows *tfinite* $(\text{tree } v p \text{ pth})$

proof(*rule ccontr*)

let $?n = \text{Suc} (\text{fcard } \text{vertices})$

assume $\neg \text{tfinite} (\text{tree } v p \text{ pth})$

hence $\neg \text{tfinite} (\text{edge-tree } v p)$ **unfolding** *finite-tree-edge-tree*.

then obtain $es :: ('vertex, 'form, 'var) \text{edge}' \text{ stream}$

where $\text{ipath} (\text{edge-tree } v p) \text{ es}$ **using** *Konig* **by** *blast*

with $\langle \text{valid-in-port} (v, p) \rangle$

have *forbidden-path* $v \text{ es}$ **by** (*rule path-is-forbidden*)

from *forbidden-path-prefix-is-path*[*OF this*] *forbidden-path-prefix-is-hyp-free*[*OF this*]

obtain v' **where** $\text{path } v' v (\text{rev} (\text{stake } ?n \text{ es}))$ **and** *hyps-free* $(\text{rev} (\text{stake } ?n \text{ es}))$

by *blast*

from *this* $\langle \text{terminal-vertex } v \rangle$

have *terminal-path* $v' v (\text{rev} (\text{stake } ?n \text{ es}))$ **by** (*rule terminal-pathI*)

hence $\text{length } (\text{rev } (\text{stake } ?n \text{ es})) \leq \text{fcard vertices}$
by (rule hyps-free-limited)
thus *False* **by** *simp*
qed

The main result of this theory.

theorem *solved*
unfolding *solved-def*
proof(*intro ballI allI conjI impI*)
fix *c*
assume $c \mid \in \mid \text{conc-forms}$
hence $c \in \text{set conclusions}$ **by** (*auto simp add: conc-forms-def*)
from *this(I) conclusions-present*
obtain *v* **where** $v \mid \in \mid \text{vertices}$ **and** $\text{nodeOf } v = \text{Conclusion } c$
by *auto*

have *valid-in-port* (*v*, (*plain-ant c*))
using $\langle v \mid \in \mid \text{vertices} \rangle \langle \text{nodeOf } - = - \rangle$ **by** *simp*

have *terminal-vertex* *v* **using** $\langle v \mid \in \mid \text{vertices} \rangle \langle \text{nodeOf } v = \text{Conclusion } c \rangle$ **by** *auto*

let $?t = \text{tree } v \text{ (plain-ant } c \text{)}$ \square

have $\text{fst } (\text{root } ?t) = (\text{global-assms } \text{TYPE}('var), c)$
using $\langle c \in \text{set conclusions} \rangle \langle \text{nodeOf } - = - \rangle$
by (*auto simp add: labelAtIn-def conclusions-closed closed-no-lconsts freshen-def rename-closed subst-closed*)
moreover

have $\text{global-assms } \text{TYPE}('var) \mid \subseteq \mid \text{ass-forms}$ **by** (*rule global-in-ass*)
moreover

from $\langle \text{terminal-vertex } v \rangle$
have *terminal-path* *v v* \square **by** (*rule terminal-path-empty*)
with $\langle \text{valid-in-port } (v, (\text{plain-ant } c)) \rangle$
have *wf* $?t$ **by** (*rule wf-tree*)
moreover

from $\langle \text{valid-in-port } (v, \text{plain-ant } c) \rangle \langle \text{terminal-vertex } v \rangle$
have *tfinite* $?t$ **by** (*rule finite-tree*)
ultimately

show $\exists \Gamma t. \text{fst } (\text{root } t) = (\Gamma \vdash c) \wedge \Gamma \mid \subseteq \mid \text{ass-forms} \wedge \text{wf } t \wedge \text{tfinite } t$ **by** *blast*
qed

end

end

7 Completeness

7.1 Incredible_Trees

```
theory Incredible-Trees
imports
  HOL-Library.Sublist
  HOL-Library.Countable
  Entailment
  Rose-Tree
  Abstract-Rules-To-Incredible
begin
```

This theory defines incredible trees, which carry roughly the same information as a (tree-shaped) incredible graph, but where the structure is still given by the data type, and not by a set of edges etc.

Tree-shape, but incredible-graph-like content (port names, explicit annotation and substitution)

```
datatype ('form,'rule,'subst,'var) itnode =
  I (iNodeOf': ('form, 'rule) graph-node)
    (iOutPort': 'form reg-out-port)
    (iAnnot': nat)
    (iSubst': 'subst)
  | H (iAnnot': nat)
    (iSubst': 'subst)
```

```
abbreviation INode n p i s ants  $\equiv$  RNode (I n p i s) ants
```

```
abbreviation HNode i s ants  $\equiv$  RNode (H i s) ants
```

```
type-synonym ('form,'rule,'subst,'var) itree = ('form,'rule,'subst,'var) itnode rose-tree
```

```
fun iNodeOf where
  iNodeOf (INode n p i s ants) = n
  | iNodeOf (HNode i s ants) = Helper
```

```
context Abstract-Formulas begin
```

```
fun iOutPort where
  iOutPort (INode n p i s ants) = p
  | iOutPort (HNode i s ants) = anyP
end
```

```
fun iAnnot where iAnnot it = iAnnot' (root it)
```

```
fun iSubst where iSubst it = iSubst' (root it)
```

```
fun iAnts where iAnts it = children it
```

```
type-synonym ('form, 'rule, 'subst) fresh-check = ('form, 'rule) graph-node  $\Rightarrow$  nat  $\Rightarrow$  'subst  $\Rightarrow$  'form entailment  $\Rightarrow$  bool
```

```
context Abstract-Task
begin
```

The well-formedness of the tree. The first argument can be varied, depending on whether we are interested in the local freshness side-conditions or not.

```
inductive iwf :: ('form, 'rule, 'subst) fresh-check  $\Rightarrow$  ('form,'rule,'subst,'var) itree  $\Rightarrow$  'form entailment  $\Rightarrow$  bool
```

```

for  $fc$ 
where
   $iwf$ :  $\llbracket$ 
     $n \in sset\ nodes$ ;
     $Reg\ p\ |\in|\ outPorts\ n$ ;
     $list-all2\ (\lambda\ ip\ t.\ iwfc\ fc\ t\ ((\lambda\ h.\ subst\ s\ (freshen\ i\ (labelsOut\ n\ h)))\ |\uparrow\ hyps-for\ n\ ip\ |\cup|\ \Gamma \vdash\ subst\ s\ (freshen$ 
 $i\ (labelsIn\ n\ ip))))$ 
     $(inPorts'\ n)\ ants$ ;
     $fc\ n\ i\ s\ (\Gamma \vdash\ c)$ ;
     $c = subst\ s\ (freshen\ i\ p)$ 
   $\rrbracket \implies iwfc\ fc\ (INode\ n\ p\ i\ s\ ants)\ (\Gamma \vdash\ c)$ 
|  $iwfH$ :  $\llbracket$ 
   $c\ |\notin|\ ass-forms$ ;
   $c\ |\in|\ \Gamma$ ;
   $c = subst\ s\ (freshen\ i\ anyP)$ 
   $\rrbracket \implies iwfc\ fc\ (HNode\ i\ s\ [])\ (\Gamma \vdash\ c)$ 

```

lemma $iwf-subst-freshen-outPort$:

```

 $iwfc\ lc\ ts\ ent \implies$ 
 $snd\ ent = subst\ (iSubst\ ts)\ (freshen\ (iAnnot\ ts)\ (iOutPort\ ts))$ 
by  $(auto\ elim:\ iwfc.cases)$ 

```

definition $all-local-vars$:: $(\text{'form}, \text{'rule})\ graph-node \implies \text{'var set}$ **where**
 $all-local-vars\ n = \bigcup (local-vars\ n\ \text{'fset}\ (inPorts\ n))$

lemma $all-local-vars-Helper[simp]$:

```

 $all-local-vars\ Helper = \{\}$ 
unfolding  $all-local-vars-def$  by  $simp$ 

```

lemma $all-local-vars-Assumption[simp]$:

```

 $all-local-vars\ (Assumption\ c) = \{\}$ 
unfolding  $all-local-vars-def$  by  $simp$ 

```

Local freshness side-conditions, corresponding what we have in the theory *Natural-Deduction*.

inductive $local-fresh-check$:: $(\text{'form}, \text{'rule}, \text{'subst})\ fresh-check$ **where**

```

 $\llbracket \wedge\ f.\ f\ |\in|\ \Gamma \implies freshenLC\ i\ \text{'}\ (all-local-vars\ n) \cap\ lconsts\ f = \{\}$ ;
 $freshenLC\ i\ \text{'}\ (all-local-vars\ n) \cap\ subst-lconsts\ s = \{\}$ 
   $\rrbracket \implies local-fresh-check\ n\ i\ s\ (\Gamma \vdash\ c)$ 

```

abbreviation $local-iwf \equiv iwfc\ local-fresh-check$

No freshness side-conditions. Used with the tree that comes out of *globalize*, where we establish the (global) freshness conditions separately.

inductive $no-fresh-check$:: $(\text{'form}, \text{'rule}, \text{'subst})\ fresh-check$ **where**

```

 $no-fresh-check\ n\ i\ s\ (\Gamma \vdash\ c)$ 

```

abbreviation $plain-iwf \equiv iwfc\ no-fresh-check$

fun $isHNode$ **where**

```

 $isHNode\ (HNode\ -\ -\ -) = True$ 
 $isHNode\ - = False$ 

```

lemma $iwf-edge-match$:

```

assumes  $iwfc\ fc\ t\ ent$ 
assumes  $is@[i] \in it-paths\ t$ 
shows  $subst\ (iSubst\ (tree-at\ t\ (is@[i])))\ (freshen\ (iAnnot\ (tree-at\ t\ (is@[i])))\ (iOutPort\ (tree-at\ t\ (is@[i])))$ 

```



```

= subst (iSubst (tree-at t is)) (freshen (iAnnot (tree-at t is)) (a-conc (inPorts' (iNodeOf (tree-at t is)) !
i)))
using assms
apply (induction arbitrary: is i)
apply (auto elim!: it-paths-SnocE)[1]
apply (rename-tac is i)
apply (case-tac is)
apply (auto dest!: list-all2-nthD2)[1]
using iwf-subst-freshen-outPort
apply (solves <(auto)[1]>)
apply (auto elim!: it-paths-ConsE dest!: list-all2-nthD2)[1]
using it-path-SnocI
apply (solves blast)
apply (solves auto)
done

```

lemma *iwf-length-inPorts*:
assumes *iwf fc t ent*
assumes *is ∈ it-paths t*
shows $\text{length } (iAnts \text{ (tree-at t is)}) \leq \text{length } (inPorts' (iNodeOf (tree-at t is)))$
using *assms*
by (induction arbitrary: *is rule: iwf.induct*)
(auto elim!: *it-paths-RNodeE dest: list-all2-lengthD list-all2-nthD2*)

lemma *iwf-local-not-in-subst*:
assumes *local-iwf t ent*
assumes *is ∈ it-paths t*
assumes *var ∈ all-local-vars (iNodeOf (tree-at t is))*
shows $\text{freshenLC } (iAnnot \text{ (tree-at t is)}) \text{ var} \notin \text{subst-lconsts } (iSubst \text{ (tree-at t is)})$
using *assms*
by (induction arbitrary: *is rule: iwf.induct*)
(auto 4 4 elim!: *it-paths-RNodeE local-fresh-check.cases dest: list-all2-lengthD list-all2-nthD2*)

lemma *iwf-length-inPorts-not-HNode*:
assumes *iwf fc t ent*
assumes *is ∈ it-paths t*
assumes $\neg (isHNode \text{ (tree-at t is)})$
shows $\text{length } (iAnts \text{ (tree-at t is)}) = \text{length } (inPorts' (iNodeOf (tree-at t is)))$
using *assms*
by (induction arbitrary: *is rule: iwf.induct*)
(auto 4 4 elim!: *it-paths-RNodeE dest: list-all2-lengthD list-all2-nthD2*)

lemma *iNodeOf-outPorts*:
iwf fc t ent $\implies is \in it-paths t \implies outPorts (iNodeOf (tree-at t is)) = \{\}\implies False$
by (induction arbitrary: *is rule: iwf.induct*)
(auto 4 4 elim!: *it-paths-RNodeE dest: list-all2-lengthD list-all2-nthD2*)

lemma *iNodeOf-tree-at*:
iwf fc t ent $\implies is \in it-paths t \implies iNodeOf (tree-at t is) \in sset \text{ nodes}$
by (induction arbitrary: *is rule: iwf.induct*)
(auto 4 4 elim!: *it-paths-RNodeE dest: list-all2-lengthD list-all2-nthD2*)

lemma *iwf-outPort*:
assumes *iwf fc t ent*
assumes *is ∈ it-paths t*
shows $\text{Reg } (iOutPort \text{ (tree-at t is)}) \mid \in \mid outPorts (iNodeOf (tree-at t is))$
using *assms*

by (induction arbitrary: is rule: iwf.induct)
(auto 4 4 elim!: it-paths-RNodeE dest: list-all2-lengthD list-all2-nthD2)

inductive-set *hyps-along* for *t* is where

prefix (*is'*@[*i*]) *is* \implies
i < length (*inPorts'* (*iNodeOf* (*tree-at t is'*))) \implies
hyps (*iNodeOf* (*tree-at t is'*)) *h* = *Some* (*inPorts'* (*iNodeOf* (*tree-at t is'*)) ! *i*) \implies
subst (*iSubst* (*tree-at t is'*)) (*freshen* (*iAnnot* (*tree-at t is'*)) (*labelsOut* (*iNodeOf* (*tree-at t is'*)) *h*)) \in *hyps-along*
t is

lemma *hyps-along-Nil[simp]*: *hyps-along t []* = {}
by (auto simp add: *hyps-along.simps*)

lemma *prefix-app-Cons-elim*:

assumes *prefix* (*xs*@[*y*]) (*z#zs*)
obtains *xs* = [] and *y* = *z*
| *xs'* where *xs* = *z#xs'* and *prefix* (*xs'*@[*y*]) *zs*
using *assms* by (cases *xs*) auto

lemma *hyps-along-Cons*:

assumes *iwf fc t ent*
assumes *i#is* \in *it-paths t*
shows *hyps-along t (i#is)* =
($\lambda h. \text{subst } (iSubst\ t) (\text{freshen } (iAnnot\ t) (\text{labelsOut } (iNodeOf\ t)\ h))$) ' *fset* (*hyps-for* (*iNodeOf t*) (*inPorts'*
(*iNodeOf t*) ! *i*))
 \cup *hyps-along* (*iAnts t* ! *i*) *is* (is ?*S1* = ?*S2* \cup ?*S3*)

proof–

from *assms*
have *i* < length (*iAnts t*) and *is* \in *it-paths* (*iAnts t* ! *i*)
by (auto elim: *it-paths-ConsE*)
let ?*t'* = *iAnts t* ! *i*

show ?*thesis*

proof (*rule*; *rule*)

fix *x*

assume *x* \in *hyps-along t (i # is)*

then obtain *is'* *i'* *h* where

prefix (*is'*@[*i'*]) (*i#is*)

and *i'* < length (*inPorts'* (*iNodeOf* (*tree-at t is'*)))

and *hyps* (*iNodeOf* (*tree-at t is'*)) *h* = *Some* (*inPorts'* (*iNodeOf* (*tree-at t is'*)) ! *i'*)

and [simp]: *x* = *subst* (*iSubst* (*tree-at t is'*)) (*freshen* (*iAnnot* (*tree-at t is'*)) (*labelsOut* (*iNodeOf* (*tree-at*
t is')) *h*))

by (auto elim!: *hyps-along.cases*)

from *this*(1)

show *x* \in ?*S2* \cup ?*S3*

proof(*cases rule: prefix-app-Cons-elim*)

assume *is'* = [] and *i'* = *i*

with $\langle \text{hyps } (iNodeOf\ (tree-at\ t\ is'))\ h = \text{Some } - \rangle$

have *x* \in ?*S2* by auto

thus ?*thesis*..

next

fix *is''*

assume [simp]: *is'* = *i # is''* and *prefix* (*is''*@[*i'*]) *is*

have *tree-at t is'* = *tree-at ?t' is''* by simp

note $\langle \text{prefix } (is''\ @\ [i'])\ is \rangle$

$\langle i' < \text{length } (inPorts'\ (iNodeOf\ (tree-at\ t\ is')) \rangle$

```

    ⟨hyps (iNodeOf (tree-at t is^)) h = Some (inPorts' (iNodeOf (tree-at t is')) ! i)⟩
  from this[unfolded ⟨tree-at t is' = tree-at ?t' is''⟩]
  have subst (iSubst (tree-at (iAnts t ! i) is'')) (freshen (iAnnot (tree-at (iAnts t ! i) is'')) (labelsOut
(iNodeOf (tree-at (iAnts t ! i) is'')) h))
    ∈ hyps-along (iAnts t ! i) is by (rule hyps-along.intros)
  hence x ∈ ?S3 by simp
  thus ?thesis..
qed
next
fix x
assume x ∈ ?S2 ∪ ?S3
thus x ∈ ?S1
proof
  have prefix ([]@[i]) (i#is) by simp
  moreover
  from ⟨iwf - t -⟩
  have length (iAnts t) ≤ length (inPorts' (iNodeOf (tree-at t [])))
    by cases (auto dest: list-all2-lengthD)
  with ⟨i < -⟩
  have i < length (inPorts' (iNodeOf (tree-at t []))) by simp
  moreover
  assume x ∈ ?S2
  then obtain h where h |∈| hyps-for (iNodeOf t) (inPorts' (iNodeOf t) ! i)
    and [simp]: x = subst (iSubst t) (freshen (iAnnot t) (labelsOut (iNodeOf t) h)) by auto
  from this(1)
  have hyps (iNodeOf (tree-at t [])) h = Some (inPorts' (iNodeOf (tree-at t [])) ! i) by simp
  ultimately
  have subst (iSubst (tree-at t [])) (freshen (iAnnot (tree-at t [])) (labelsOut (iNodeOf (tree-at t [])) h)) ∈
hyps-along t (i # is)
    by (rule hyps-along.intros)
  thus x ∈ hyps-along t (i # is) by simp
next
assume x ∈ ?S3
thus x ∈ ?S1
  apply (auto simp add: hyps-along.simps)
  apply (rule-tac x = i#is' in exI)
  apply auto
done
qed
qed
qed

```

lemma *iwf-hyps-exist*:

assumes *iwf lc it ent*

assumes *is ∈ it-paths it*

assumes *tree-at it is = (HNode i s ants[^])*

assumes *fst ent |⊆| ass-forms*

shows *subst s (freshen i anyP) ∈ hyps-along it is*

proof—

from *assms(1,2,3)*

have *subst s (freshen i anyP) ∈ hyps-along it is*

∨ *subst s (freshen i anyP) |∈| fst ent*

∧ *subst s (freshen i anyP) |∉| ass-forms*

proof(induction arbitrary: *is* rule: *iwf.induct*)

case (*iwf n p s' a' Γ ants c is*)

have *iwf lc (INode n p a' s' ants) (Γ ⊢ c)*

```

using iwf(1,2,3,4,5)
by (auto intro!: iwf.intros elim!: list-all2-mono)

show ?case
proof(cases is)
  case Nil
  with  $\langle \text{tree-at } (INode\ n\ p\ a'\ s'\ ants)\ is = HNode\ i\ s\ ants' \rangle$ 
  show ?thesis by auto
next
  case (Cons i' is')
  with  $\langle is \in \text{it-paths } (INode\ n\ p\ a'\ s'\ ants) \rangle$ 
  have  $i' < \text{length } ants$  and  $is' \in \text{it-paths } (ants\ !\ i')$ 
  by (auto elim: it-paths-ConsE)

let  $\Gamma' = (\lambda h. \text{subst } s' (\text{freshen } a' (\text{labelsOut } n\ h))) \mid \mid \langle \text{hyps-for } n\ (\text{inPorts}'\ n\ !\ i') \rangle$ 

from  $\langle \text{tree-at } (INode\ n\ p\ a'\ s'\ ants)\ is = HNode\ i\ s\ ants' \rangle$ 
have  $\text{tree-at } (ants\ !\ i')\ is' = HNode\ i\ s\ ants'$  using Cons by simp

from iwf.IH  $\langle i' < \text{length } ants \rangle \langle is' \in \text{it-paths } (ants\ !\ i') \rangle$  this
have  $\text{subst } s (\text{freshen } i\ anyP) \in \text{hyps-along } (ants\ !\ i')\ is'$ 
   $\vee \text{subst } s (\text{freshen } i\ anyP) \mid \in \mid \Gamma' \mid \cup \mid \Gamma \wedge \text{subst } s (\text{freshen } i\ anyP) \mid \notin \mid \text{ass-forms}$ 
  by (auto dest: list-all2-nthD2)
moreover
from  $\langle is \in \text{it-paths } (INode\ n\ p\ a'\ s'\ ants) \rangle$ 
have  $\text{hyps-along } (INode\ n\ p\ a'\ s'\ ants)\ is = \text{fset } \Gamma' \cup \text{hyps-along } (ants\ !\ i')\ is'$ 
  using  $\langle is = - \rangle$ 
  by (simp add: hyps-along-Cons[OF iwf.lc (INode n p a' s' ants) (Γ ⊢ c)])
ultimately
show ?thesis by auto
qed
next
  case (iwfH c Γ s' i' is)
  hence [simp]:  $is = []\ i' = i\ s' = s$  by simp-all
  from  $\langle c = \text{subst } s' (\text{freshen } i'\ anyP) \rangle \langle c \mid \in \mid \Gamma \rangle \langle c \mid \notin \mid \text{ass-forms} \rangle$ 
  show ?case by simp
qed
with assms(4)
show ?thesis by blast
qed

```

definition *hyp-port-for'* :: ('form, 'rule, 'subst, 'var) itree \Rightarrow nat list \Rightarrow 'form \Rightarrow nat list \times nat \times ('form, 'var) out-port **where**

```

hyp-port-for' t is f = (SOME x.
  (case x of (is', i, h)  $\Rightarrow$ 
    prefix (is' @ [i]) is  $\wedge$ 
     $i < \text{length } (\text{inPorts}' (iNodeOf (\text{tree-at } t\ is')))) \wedge$ 
     $\text{hyps } (iNodeOf (\text{tree-at } t\ is'))\ h = \text{Some } (\text{inPorts}' (iNodeOf (\text{tree-at } t\ is'))\ !\ i) \wedge$ 
     $f = \text{subst } (iSubst (\text{tree-at } t\ is')) (\text{freshen } (iAnnot (\text{tree-at } t\ is')) (\text{labelsOut } (iNodeOf (\text{tree-at } t\ is'))\ h))$ 
  ))

```

lemma *hyp-port-for-spec'*:

assumes $f \in \text{hyps-along } t\ is$

shows (case *hyp-port-for'* t is f of (is', i, h) \Rightarrow

prefix (is' @ [i]) is \wedge

$i < \text{length } (\text{inPorts}' (iNodeOf (\text{tree-at } t\ is')))) \wedge$

$\text{hyps } (iNodeOf (\text{tree-at } t\ is'))\ h = \text{Some } (\text{inPorts}' (iNodeOf (\text{tree-at } t\ is'))\ !\ i) \wedge$

$f = \text{subst } (i\text{Subst } (\text{tree-at } t \text{ is }')) (\text{freshen } (i\text{Annot } (\text{tree-at } t \text{ is }')) (\text{labelsOut } (i\text{NodeOf } (\text{tree-at } t \text{ is }')) h)))$
using *assms* **unfolding** *hyp-_{along}.simps hyp-port-for'-def* **by** $-(\text{rule } \text{someI-ex, blast})$

definition *hyp-port-path-for* :: ('form, 'rule, 'subst, 'var) itree \Rightarrow nat list \Rightarrow 'form \Rightarrow nat list
where *hyp-port-path-for* *t is f* = fst (*hyp-port-for'* *t is f*)

definition *hyp-port-i-for* :: ('form, 'rule, 'subst, 'var) itree \Rightarrow nat list \Rightarrow 'form \Rightarrow nat
where *hyp-port-i-for* *t is f* = fst (snd (*hyp-port-for'* *t is f*))

definition *hyp-port-h-for* :: ('form, 'rule, 'subst, 'var) itree \Rightarrow nat list \Rightarrow 'form \Rightarrow ('form, 'var) out-port
where *hyp-port-h-for* *t is f* = snd (snd (*hyp-port-for'* *t is f*))

lemma *hyp-port-prefix*:

assumes $f \in \text{hyp-_{along} } t \text{ is}$

shows *prefix* (*hyp-port-path-for* *t is f*@[*hyp-port-i-for* *t is f*]) *is*

using *hyp-port-for-spec'*[*OF assms*] **unfolding** *hyp-port-path-for-def hyp-port-i-for-def* **by** *auto*

lemma *hyp-port-strict-prefix*:

assumes $f \in \text{hyp-_{along} } t \text{ is}$

shows *strict-prefix* (*hyp-port-path-for* *t is f*) *is*

using *hyp-port-prefix*[*OF assms*] **by** (*simp add: strict-prefixI' prefix-order.dual-order.strict-trans1*)

lemma *hyp-port-it-paths*:

assumes $is \in \text{it-paths } t$

assumes $f \in \text{hyp-_{along} } t \text{ is}$

shows *hyp-port-path-for* *t is f* \in *it-paths* *t*

using *assms* **by** (*rule it-paths-strict-prefix*[*OF - hyp-port-strict-prefix*])

lemma *hyp-port-hyps*:

assumes $f \in \text{hyp-_{along} } t \text{ is}$

shows *hyps* (*iNodeOf* (*tree-at* *t* (*hyp-port-path-for* *t is f*))) (*hyp-port-h-for* *t is f*) = *Some* (*inPorts'* (*iNodeOf* (*tree-at* *t* (*hyp-port-path-for* *t is f*))) ! *hyp-port-i-for* *t is f*)

using *hyp-port-for-spec'*[*OF assms*] **unfolding** *hyp-port-path-for-def hyp-port-i-for-def hyp-port-h-for-def* **by** *auto*

lemma *hyp-port-outPort*:

assumes $f \in \text{hyp-_{along} } t \text{ is}$

shows (*hyp-port-h-for* *t is f*) \in *outPorts* (*iNodeOf* (*tree-at* *t* (*hyp-port-path-for* *t is f*)))

using *hyps-correct*[*OF hyp-port-hyps*[*OF assms*]]..

lemma *hyp-port-eq*:

assumes $f \in \text{hyp-_{along} } t \text{ is}$

shows $f = \text{subst } (i\text{Subst } (\text{tree-at } t \text{ (hyp-port-path-for } t \text{ is } f))) (\text{freshen } (i\text{Annot } (\text{tree-at } t \text{ (hyp-port-path-for } t \text{ is } f))) (\text{labelsOut } (i\text{NodeOf } (\text{tree-at } t \text{ (hyp-port-path-for } t \text{ is } f))) (\text{hyp-port-h-for } t \text{ is } f)))$

using *hyp-port-for-spec'*[*OF assms*] **unfolding** *hyp-port-path-for-def hyp-port-i-for-def hyp-port-h-for-def* **by** *auto*

definition *isidx* :: nat list \Rightarrow nat **where** *isidx* *xs* = *to-nat* (*Some* *xs*)

definition *v-away* :: nat **where** *v-away* = *to-nat* (*None* :: nat list option)

lemma *isidx-inj*[*simp*]: *isidx* *xs* = *isidx* *ys* \iff *xs* = *ys*

unfolding *isidx-def* **by** *simp*

lemma *isidx-v-away*[*simp*]: *isidx* *xs* \neq *v-away*

unfolding *isidx-def v-away-def* **by** *simp*

definition *mapWithIndex* **where** *mapWithIndex* *f* *xs* = *map* ($\lambda (i,t) . f i t$) (*List.enumerate* 0 *xs*)

lemma *mapWithIndex-cong* [*fundef-cong*]:

$xs = ys \implies (\bigwedge x i. x \in \text{set } ys \implies f i x = g i x) \implies \text{mapWithIndex } f \text{ } xs = \text{mapWithIndex } g \text{ } ys$
unfolding *mapWithIndex-def* **by** (*auto simp add: in-set-enumerate-eq*)

lemma *mapWithIndex-Nil[simp]*: $\text{mapWithIndex } f \ [] = []$
unfolding *mapWithIndex-def* **by** *simp*

lemma *length-mapWithIndex[simp]*: $\text{length } (\text{mapWithIndex } f \text{ } xs) = \text{length } xs$
unfolding *mapWithIndex-def* **by** *simp*

lemma *nth-mapWithIndex[simp]*: $i < \text{length } xs \implies \text{mapWithIndex } f \text{ } xs ! i = f i (xs ! i)$
unfolding *mapWithIndex-def* **by** (*auto simp add: nth-enumerate-eq*)

lemma *list-all2-mapWithIndex2E*:

assumes *list-all2* P *as* *bs*

assumes $\bigwedge i a b. i < \text{length } bs \implies P a b \implies Q a (f i b)$

shows *list-all2* Q *as* (*mapWithIndex* f *bs*)

using *assms(1)*

by (*auto simp add: list-all2-conv-all-nth mapWithIndex-def nth-enumerate-eq intro: assms(2) split: prod.split*)

The *globalize* function, which renames all local constants so that they cannot clash with local constants occurring anywhere else in the tree.

fun *globalize-node* :: $\text{nat list} \Rightarrow ('var \Rightarrow 'var) \Rightarrow ('form, 'rule, 'subst, 'var) \text{ itnode} \Rightarrow ('form, 'rule, 'subst, 'var) \text{ itnode}$ **where**

globalize-node is f $(I n p i s) = I n p (isidx is) (subst-renameLCs f s)$

| *globalize-node is f* $(H i s) = H (isidx is) (subst-renameLCs f s)$

fun *globalize* :: $\text{nat list} \Rightarrow ('var \Rightarrow 'var) \Rightarrow ('form, 'rule, 'subst, 'var) \text{ itree} \Rightarrow ('form, 'rule, 'subst, 'var) \text{ itree}$ **where**

globalize is f $(RNode r ants) = RNode$

(*globalize-node is f* r)

(*mapWithIndex* $(\lambda i' t.$

globalize (is@[i'])

(*rerename (a-fresh (inPorts' (iNodeOf (RNode r ants)) ! i')*)

(*iAnnot (RNode r ants) (isidx is) f*)

t

) *ants*)

lemma *iAnnot'-globalize-node[simp]*: $iAnnot' (\text{globalize-node is f } n) = isidx is$
by (*cases n*) *auto*

lemma *iAnnot-globalize*:

assumes $is' \in \text{it-paths } (\text{globalize is f } t)$

shows $iAnnot (\text{tree-at } (\text{globalize is f } t) is') = isidx (is@is')$

using *assms*

by (*induction t arbitrary: f is is'*) (*auto elim!: it-paths-RNodeE*)

lemma *all-local-consts-listed'*:

assumes $n \in \text{sset nodes}$

assumes $p \in | \text{inPorts } n$

shows $\text{lconsts } (a\text{-conc } p) \cup (\bigcup (\text{lconsts } 'fset (a\text{-hyps } p))) \subseteq a\text{-fresh } p$

using *assms*

by (*auto simp add: nodes-def stream.set-map lconsts-anyP closed-no-lconsts conclusions-closed f-antecedent-def dest!: all-local-consts-listed*)

lemma *no-local-consts-in-consequences'*:

$n \in \text{sset nodes} \implies \text{Reg } p \in | \text{outPorts } n \implies \text{lconsts } p = \{\}$

using *no-local-consts-in-consequences*

by (auto simp add: nodes-def lconsts-anyP closed-no-lconsts assumptions-closed stream.set-map f-consequent-def)

lemma iwf-globalize:

assumes local-iwf t ($\Gamma \vdash c$)

shows plain-iwf (globalize is f t) (renameLCs f | Γ \vdash renameLCs f c)

using assms

proof (induction t $\Gamma \vdash c$ arbitrary: is f Γ c rule: iwf.induct)

case (iwf n p s i Γ ants c is f)

note $\langle n \in \text{sset nodes} \rangle$

moreover

note $\langle \text{Reg } p \mid \in \mid \text{outPorts } n \rangle$

moreover

{ **fix** i'

let $?V = a\text{-fresh } (inPorts' n ! i')$

let $?f' = \text{rename } ?V i (isidx is) f$

let $?t = \text{globalize } (is @ [i']) ?f' (ants ! i')$

let $?ip = inPorts' n ! i'$

let $?T' = (\lambda h. \text{subst } (\text{subst-renameLCs } f s) (\text{freshen } (isidx is) (\text{labelsOut } n h))) \mid \Gamma \text{ hyps-for } n ?ip$

let $?c' = \text{subst } (\text{subst-renameLCs } f s) (\text{freshen } (isidx is) (\text{labelsIn } n ?ip))$

assume $i' < \text{length } (inPorts' n)$

hence $(inPorts' n ! i') \mid \in \mid inPorts n$ by (simp add: inPorts-fset-of)

from $\langle i' < \text{length } (inPorts' n) \rangle$

have subset-V: $?V \subseteq \text{all-local-vars } n$

unfolding all-local-vars-def

by (auto simp add: inPorts-fset-of set-conv-nth)

from $\langle \text{local-fresh-check } n i s (\Gamma \vdash c) \rangle$

have freshenLC $i' \text{ ' all-local-vars } n \cap \text{subst-lconsts } s = \{\}$

by (rule local-fresh-check.cases) simp

hence freshenLC $i' \text{ ' } ?V \cap \text{subst-lconsts } s = \{\}$

using subset-V **by** auto

hence rename-subst: $\text{subst-renameLCs } ?f' s = \text{subst-renameLCs } f s$

by (rule rename-subst-noop)

from all-local-consts-listed'[OF $\langle n \in \text{sset nodes} \rangle \langle (inPorts' n ! i') \mid \in \mid inPorts n \rangle$]

have subset-conc: $\text{lconsts } (a\text{-conc } (inPorts' n ! i')) \subseteq ?V$

and subset-hyp': $\bigwedge hyp . hyp \mid \in \mid a\text{-hyps } (inPorts' n ! i') \implies \text{lconsts } hyp \subseteq ?V$

by auto

from List.list-all2-nthD[OF $\langle \text{list-all2 } - - \rightarrow \langle i' < \text{length } (inPorts' n) \rangle, \text{simplified} \rangle$]

have plain-iwf ?t

(renameLCs ?f' | Γ \vdash (($\lambda h. \text{subst } s (\text{freshen } i (\text{labelsOut } n h))) \mid \Gamma \text{ hyps-for } n ?ip \mid \cup \mid \Gamma$) \vdash renameLCs ?f' (subst s (freshen i (a-conc ?ip))))

by simp

also have renameLCs ?f' | Γ \vdash (($\lambda h. \text{subst } s (\text{freshen } i (\text{labelsOut } n h))) \mid \Gamma \text{ hyps-for } n ?ip \mid \cup \mid \Gamma$)

= ($\lambda x. \text{subst } (\text{subst-renameLCs } ?f' s) (\text{renameLCs } ?f' (\text{freshen } i (\text{labelsOut } n x))) \mid \Gamma \text{ hyps-for } n ?ip \mid \cup \mid \Gamma$)

renameLCs ?f' | Γ

by (simp add: fimage-fimage fimage-funion comp-def rename-subst)

also have renameLCs ?f' | Γ = renameLCs f | Γ

proof(rule fimage-cong[OF refl])

fix x

assume $x \mid \in \mid \Gamma$

with $\langle \text{local-fresh-check } n i s (\Gamma \vdash c) \rangle$

have freshenLC $i' \text{ ' all-local-vars } n \cap \text{lconsts } x = \{\}$

```

    by (elim local-fresh-check.cases) simp
  hence freshenLC i ' ?V ∩ lconsts x = {}
    using subset-V by auto
  thus renameLCs ?f' x = renameLCs f x
    by (rule rerename-rename-noop)
qed
also have (λx. subst (subst-renameLCs ?f' s) (renameLCs ?f' (freshen i (labelsOut n x)))) |' hyps-for n
?ip = ?Γ'
proof(rule fimage-cong[OF refl])
  fix hyp
  assume hyp |∈| hyps-for n (inPorts' n ! i')
  hence labelsOut n hyp |∈| a-hyps (inPorts' n ! i')
    apply (cases hyp)
    apply (solves simp)
    apply (cases n)
    apply (auto split: if-splits)
  done
from subset-hyp'[OF this]
have subset-hyp: lconsts (labelsOut n hyp) ⊆ ?V.

show subst (subst-renameLCs ?f' s) (renameLCs ?f' (freshen i (labelsOut n hyp))) =
  subst (subst-renameLCs f s) (freshen (isidx is) (labelsOut n hyp))
  apply (simp add: freshen-def rename-rename rerename-subst)
  apply (rule arg-cong[OF renameLCs-cong])
  apply (auto dest: subsetD[OF subset-hyp])
  done
qed
also have renameLCs ?f' (subst s (freshen i (a-conc ?ip))) = subst (subst-renameLCs ?f' s) (renameLCs
?f' (freshen i (a-conc ?ip))) by (simp add: rename-subst)
also have ... = ?c'
  apply (simp add: freshen-def rename-rename rerename-subst)
  apply (rule arg-cong[OF renameLCs-cong])
  apply (auto dest: subsetD[OF subset-conc])
  done
finally
  have plain-ivf ?t (?Γ' |∪| renameLCs f |' Γ ⊢ ?c').
}
with list-all2-lengthD[OF ‹list-all2 - - -›]
have list-all2
  (λip t. plain-ivf t ((λh. subst (subst-renameLCs f s)
    (freshen (isidx is) (labelsOut n h))) |' hyps-for n ip |∪| renameLCs f |' Γ ⊢ subst (subst-renameLCs f s)
    (freshen (isidx is) (labelsIn n ip))))
  (inPorts' n)
  (mapWithIndex (λ i' t. globalize (is@[i']) (rename (a-fresh (inPorts' n ! i')) i (isidx is) f) t) ants)
  by (auto simp add: list-all2-conv-all-nth)
moreover
have no-fresh-check n (isidx is) (subst-renameLCs f s) (renameLCs f |' Γ ⊢ renameLCs f c)..
moreover
from ‹n ∈ sset nodes› ‹Reg p |∈| outPorts n›
have lconsts p = {} by (rule no-local-consts-in-consequences')
with ‹c = subst s (freshen i p)›
have renameLCs f c = subst (subst-renameLCs f s) (freshen (isidx is) p)
  by (simp add: rename-subst rename-closed freshen-closed)
ultimately
show ?case
  unfolding globalize.simps globalize-node.simps iNodeOf.simps iAnnot.simps itnode.sel rose-tree.sel Let-def

```


by (*rule iwf.intros(1)*)
next
case (*iwfH c Γ s i is f*)
from $\langle c \mid \notin \mid \text{ass-forms} \rangle$
have *renameLCs f c $\mid \notin \mid \text{ass-forms}$*
using *assumptions-closed closed-no-lconsts lconsts-renameLCs rename-closed* **by** *fastforce*
moreover
from $\langle c \mid \in \mid \Gamma \rangle$
have *renameLCs f c $\mid \in \mid \text{renameLCs f} \mid \uparrow \Gamma$* **by** *auto*
moreover
from $\langle c = \text{subst } s \text{ (freshen } i \text{ anyP)} \rangle$
have *renameLCs f c = subst (subst-renameLCs f s) (freshen (isidx is) anyP)*
by (*metis freshen-closed lconsts-anyP rename-closed rename-subst*)
ultimately
show *plain-iwf (globalize is f (HNode i s [])) (renameLCs f $\mid \uparrow \Gamma \vdash \text{renameLCs f c}$)*
unfolding *globalize.simps globalize-node.simps mapWithIndex-Nil Let-def*
by (*rule iwf.intros(2)*)
qed

definition *fresh-at where*

fresh-at t xs =
(case rev xs of [] \Rightarrow {}
| (i#is') \Rightarrow freshenLC (iAnnot (tree-at t (rev is')) ' (a-fresh (inPorts' (iNodeOf (tree-at t (rev is')) ! i))))

lemma *fresh-at-Nil[simp]:*

fresh-at t [] = {}
unfolding *fresh-at-def* **by** *simp*

lemma *fresh-at-snoc[simp]:*

fresh-at t (is@[i]) = freshenLC (iAnnot (tree-at t is)) ' (a-fresh (inPorts' (iNodeOf (tree-at t is)) ! i))
unfolding *fresh-at-def* **by** *simp*

lemma *fresh-at-def':*

fresh-at t is =
(if is = [] then {}
else freshenLC (iAnnot (tree-at t (butlast is))) ' (a-fresh (inPorts' (iNodeOf (tree-at t (butlast is)) ! last is))))
unfolding *fresh-at-def* **by** (*auto split: list.split*)

lemma *fresh-at-Cons[simp]:*

fresh-at t (i#is) = (if is = [] then freshenLC (iAnnot t) ' (a-fresh (inPorts' (iNodeOf t) ! i)) else (let t' = iAnts t ! i in fresh-at t' is))
unfolding *fresh-at-def'*
by (*auto simp add: Let-def*)

definition *fresh-at-path where*

fresh-at-path t is = \bigcup (fresh-at t ' set (prefixes is))

lemma *fresh-at-path-Nil[simp]:*

fresh-at-path t [] = {}
unfolding *fresh-at-path-def* **by** *simp*

lemma *fresh-at-path-Cons[simp]:*

fresh-at-path t (i#is) = fresh-at t [i] \cup fresh-at-path (iAnts t ! i) is
unfolding *fresh-at-path-def*
by (*fastforce split: if-splits*)

```

lemma globalize-local-consts:
  assumes  $is' \in it\_paths$  (globalize is f t)
  shows subst-lconsts ( $iSubst$  (tree-at (globalize is f t)  $is'$ ))  $\subseteq$ 
    fresh-at-path (globalize is f t)  $is' \cup range\ f$ 
  using assms
  apply (induction is f t arbitrary: is' rule:globalize.induct)
  apply (rename-tac is f r ants is')
  apply (case-tac r)
  apply (auto simp add: subst-lconsts-subst-rewriteLCs elim!: it-paths-RNodeE)
  apply (solves <force dest!: subsetD[OF range-rewrite]>)
  apply (solves <force dest!: subsetD[OF range-rewrite]>)
  done

lemma iwf-globalize':
  assumes local-iwf t ent
  assumes  $\bigwedge x. x \in |fst\ ent \implies closed\ x$ 
  assumes closed (snd ent)
  shows plain-iwf (globalize is (freshenLC v-away)  $t$ ) ent
using assms
proof(induction ent rule: prod.induct)
  case (Pair  $\Gamma\ c$ )
    have plain-iwf (globalize is (freshenLC v-away)  $t$ ) (renameLCs (freshenLC v-away)  $|'\ \Gamma \vdash renameLCs$ 
      (freshenLC v-away)  $c$ )
      by (rule iwf-globalize[OF Pair(1)])
    also
    from Pair(3) have closed c by simp
    hence renameLCs (freshenLC v-away)  $c = c$  by (simp add: closed-no-lconsts rename-closed)
    also
    from Pair(2)
    have renameLCs (freshenLC v-away)  $|'\ \Gamma = \Gamma$ 
      by (auto simp add: closed-no-lconsts rename-closed image-iff)
    finally show ?case.
qed
end

end

```

7.2 Build_Incredible_Tree

```

theory Build-Incredible-Tree
imports Incredible-Trees Natural-Deduction
begin

```

This theory constructs an incredible tree (with freshness checked only locally) from a natural deduction tree.

```

lemma image-eq-to-f:
  assumes  $f1 \text{ ' } S1 = f2 \text{ ' } S2$ 
  obtains  $f$  where  $\bigwedge x. x \in S2 \implies f\ x \in S1 \wedge f1\ (f\ x) = f2\ x$ 
proof (atomize-elim)
  from assms
  have  $\forall x. x \in S2 \longrightarrow (\exists y. y \in S1 \wedge f1\ y = f2\ x)$  by (metis image-iff)
  thus  $\exists f. \forall x. x \in S2 \longrightarrow f\ x \in S1 \wedge f1\ (f\ x) = f2\ x$  by metis
qed

```

```

context includes fset.lifting

```

```

begin
lemma fimage-eq-to-f:
  assumes f1 | $\cdot$ | S1 = f2 | $\cdot$ | S2
  obtains f where  $\bigwedge x. x \in S2 \implies f x \in S1 \wedge f1 (f x) = f2 x$ 
using assms apply transfer using image-eq-to-f by metis
end

context Abstract-Task
begin

lemma build-local-iwf:
  fixes t :: ('form entailment  $\times$  ('rule  $\times$  'form) NatRule) tree
  assumes tfinite t
  assumes wf t
  shows  $\exists it. local-iwf\ it\ (fst\ (root\ t))$ 
using assms
proof(induction)
  case (tfinite t)
  from  $\langle wf\ t \rangle$ 
  have snd (root t)  $\in R$  using wf.simps by blast

  from  $\langle wf\ t \rangle$ 
  have eff (snd (root t)) (fst (root t)) ((fst  $\circ$  root) | $\cdot$ | cont t) using wf.simps by blast

  from  $\langle wf\ t \rangle$ 
  have  $\bigwedge t'. t' \in cont\ t \implies wf\ t'$  using wf.simps by blast
  hence IH:  $\bigwedge \Gamma' t'. t' \in cont\ t \implies (\exists it'. local-iwf\ it'\ (fst\ (root\ t')))$  using tfinite(2) by blast
  then obtain its where its:  $\bigwedge t'. t' \in cont\ t \implies local-iwf\ (its\ t')\ (fst\ (root\ t'))$  by metis

  from  $\langle eff\ -\ - \rangle$ 
  show ?case
proof(cases rule: eff.cases[case-names Axiom NatRule Cut])
  case (Axiom c  $\Gamma$ )
  show ?thesis
  proof (cases c | $\in$ | ass-forms)
    case True
    then have c  $\in set\ assumptions$  by (auto simp add: ass-forms-def)

    let ?it = INode (Assumption c) c undefined undefined [] :: ('form, 'rule, 'subst, 'var) itree

    from  $\langle c \in set\ assumptions \rangle$ 
    have local-iwf ?it ( $\Gamma \vdash c$ )
      by (auto intro: iwf local-fresh-check.intros)

    thus ?thesis unfolding Axiom..
  next
  case False
  obtain s where subst s anyP = c by atomize-elim (rule anyP-is-any)
  hence [simp]: subst s (freshen undefined anyP) = c by (simp add: lconsts-anyP freshen-closed)

  let ?it = HNode undefined s [] :: ('form, 'rule, 'subst, 'var) itree

  from  $\langle c \in \Gamma \rangle$  False
  have local-iwf ?it ( $\Gamma \vdash c$ ) by (auto intro: iwfH)
  thus ?thesis unfolding Axiom..
qed
next

```

```

case (NatRule rule c ants  $\Gamma$  i s)
  from  $\langle \text{natEff-Inst rule c ants} \rangle$ 
  have snd rule = c and [simp]: ants = f-antecedent (fst rule) and c  $\in$  set (consequent (fst rule))
    by (auto simp add: natEff-Inst.simps)

  from  $\langle (\text{fst} \circ \text{root}) \mid \! \! \! \uparrow \mid \text{cont } t = (\lambda \text{ant. } (\lambda p. \text{subst } s (\text{freshen } i \ p))) \mid \! \! \! \uparrow \mid \text{a-hyps ant} \mid \cup \mid \Gamma \vdash \text{subst } s (\text{freshen } i \ (a\text{-conc ant}))) \mid \! \! \! \uparrow \mid \text{ants} \rangle$ 
  obtain to-t where  $\bigwedge \text{ant. ant} \mid \in \mid \text{ants} \implies \text{to-t ant} \mid \in \mid \text{cont } t \wedge (\text{fst} \circ \text{root}) (\text{to-t ant}) = ((\lambda p. \text{subst } s (\text{freshen } i \ p))) \mid \! \! \! \uparrow \mid \text{a-hyps ant} \mid \cup \mid \Gamma \vdash \text{subst } s (\text{freshen } i \ (a\text{-conc ant})))$ 
    by (rule fimage-eq-to-f) (rule that)
  hence to-t-in-cont:  $\bigwedge \text{ant. ant} \mid \in \mid \text{ants} \implies \text{to-t ant} \mid \in \mid \text{cont } t$ 
  and to-t-root:  $\bigwedge \text{ant. ant} \mid \in \mid \text{ants} \implies \text{fst} (\text{root} (\text{to-t ant})) = ((\lambda p. \text{subst } s (\text{freshen } i \ p))) \mid \! \! \! \uparrow \mid \text{a-hyps ant} \mid \cup \mid \Gamma \vdash \text{subst } s (\text{freshen } i \ (a\text{-conc ant})))$ 
    by auto

  let ?ants' = map ( $\lambda \text{ant. its} (\text{to-t ant}))$  (antecedent (fst rule))
  let ?it = INode (Rule (fst rule)) c i s ?ants' :: ('form, 'rule, 'subst, 'var) itree

  from  $\langle \text{snd} (\text{root } t) \in R \rangle$ 
  have fst rule  $\in$  sset rules
    unfolding NatRule
    by (auto simp add: stream.set-map n-rules-def no-empty-conclusions )
  moreover
  from  $\langle c \in \text{set} (\text{consequent} (\text{fst rule})) \rangle$ 
  have c  $\mid \in \mid$  f-consequent (fst rule) by (simp add: f-consequent-def)
  moreover
  { fix ant
    assume ant  $\in$  set (antecedent (fst rule))
    hence ant  $\mid \in \mid$  ants by (simp add: f-antecedent-def)
    from its[OF to-t-in-cont[OF this]]
    have local-iwf (its (to-t ant)) (fst (root (to-t ant))).
    also have fst (root (to-t ant)) =
       $((\lambda p. \text{subst } s (\text{freshen } i \ p))) \mid \! \! \! \uparrow \mid \text{a-hyps ant} \mid \cup \mid \Gamma \vdash \text{subst } s (\text{freshen } i \ (a\text{-conc ant})))$ 
      by (rule to-t-root[OF  $\langle \text{ant} \mid \in \mid \text{ants} \rangle$ ])
    also have ... =
       $((\lambda h. \text{subst } s (\text{freshen } i \ (\text{labelsOut} (\text{Rule} (\text{fst rule})) \ h))) \mid \! \! \! \uparrow \mid \text{hyps-for} (\text{Rule} (\text{fst rule})) \ \text{ant} \mid \cup \mid \Gamma \vdash \text{subst } s (\text{freshen } i \ (a\text{-conc ant})))$ 
      using  $\langle \text{ant} \mid \in \mid \text{ants} \rangle$ 
      by auto
    finally
    have local-iwf (its (to-t ant))
       $((\lambda h. \text{subst } s (\text{freshen } i \ (\text{labelsOut} (\text{Rule} (\text{fst rule})) \ h))) \mid \! \! \! \uparrow \mid \text{hyps-for} (\text{Rule} (\text{fst rule})) \ \text{ant} \mid \cup \mid \Gamma \vdash \text{subst } s (\text{freshen } i \ (a\text{-conc ant})))$ .
  }
  moreover
  from NatRule(5,6)
  have local-fresh-check (Rule (fst rule)) i s ( $\Gamma \vdash \text{subst } s (\text{freshen } i \ c)$ )
    by (fastforce intro!: local-fresh-check.intros simp add: all-local-vars-def)
  ultimately
  have local-iwf ?it ( $\Gamma \vdash \text{subst } s (\text{freshen } i \ c)$ )
    by (intro iwf ) (auto simp add: list-all2-map2 list-all2-same)
  thus ?thesis unfolding NatRule..
}
next
case (Cut  $\Gamma$  con)
  obtain s where subst s anyP = con by atomize-elim (rule anyP-is-any)
  hence [simp]: subst s (freshen undefined anyP) = con by (simp add: lconsts-anyP freshen-closed)

```

```

from ⟨fst ∘ root⟩ |q cont t = {Γ ⊢ con}⟩
obtain t' where t' |∈| cont t and [simp]: fst (root t') = (Γ ⊢ con)
  by (cases cont t) auto

from ⟨t' |∈| cont t⟩ obtain it' where local-iwf it' (Γ ⊢ con) using IH by force

let ?it = INode Helper anyP undefined s [it'] :: ('form, 'rule, 'subst, 'var) itree

from ⟨local-iwf it' (Γ ⊢ con)⟩
have local-iwf ?it (Γ ⊢ con) by (auto intro!: iwf local-fresh-check.intros)
thus ?thesis unfolding Cut..
qed
qed

definition to-it :: ('form entailment × ('rule × 'form) NatRule) tree ⇒ ('form, 'rule, 'subst, 'var) itree where
  to-it t = (SOME it. local-iwf it (fst (root t)))

lemma iwf-to-it:
  assumes tfinite t and wf t
  shows local-iwf (to-it t) (fst (root t))
unfolding to-it-def using build-local-iwf[OF assms] by (rule someI2-ex)
end
end

```

7.3 Incredible_Completeness

```

theory Incredible-Completeness
imports Natural-Deduction Incredible-Deduction Build-Incredible-Tree
begin

```

This theory takes the tree produced in *Incredible-Proof-Machine.Build-Incredible-Tree*, globalizes it using *globalize*, and then builds the incredible proof graph out of it.

```

type-synonym 'form vertex = ('form × nat list)
type-synonym ('form, 'var) edge'' = ('form vertex, 'form, 'var) edge'

```

```

locale Solved-Task =
  Abstract-Task freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP antecedent
  consequent rules assumptions conclusions
  for freshenLC :: nat ⇒ 'var ⇒ 'var
  and renameLCs :: ('var ⇒ 'var) ⇒ 'form ⇒ 'form
  and lconsts :: 'form ⇒ 'var set
  and closed :: 'form ⇒ bool
  and subst :: 'subst ⇒ 'form ⇒ 'form
  and subst-lconsts :: 'subst ⇒ 'var set
  and subst-renameLCs :: ('var ⇒ 'var) ⇒ ('subst ⇒ 'subst)
  and anyP :: 'form
  and antecedent :: 'rule ⇒ ('form, 'var) antecedent list
  and consequent :: 'rule ⇒ 'form list
  and rules :: 'rule stream
  and assumptions :: 'form list
  and conclusions :: 'form list +
  assumes solved: solved
begin

```

Let us get our hand on concrete trees.

```

definition ts :: 'form ⇒ (('form entailment) × ('rule × 'form) NatRule) tree where

```

$ts\ c = (SOME\ t.\ snd\ (fst\ (root\ t)) = c \wedge fst\ (fst\ (root\ t)) \mid\subseteq\ ass\text{-forms} \wedge wf\ t \wedge tfinite\ t)$

lemma

assumes $c \mid\subseteq\ conc\text{-forms}$
shows $ts\text{-conc}: snd\ (fst\ (root\ (ts\ c))) = c$
and $ts\text{-context}: fst\ (fst\ (root\ (ts\ c))) \mid\subseteq\ ass\text{-forms}$
and $ts\text{-wf}: wf\ (ts\ c)$
and $ts\text{-finite}[simp]: tfinite\ (ts\ c)$
unfolding $atomize\text{-conj}\ conj\text{-assoc}\ ts\text{-def}$
apply $(rule\ someI\text{-ex})$
using $solved\ assms$
by $(force\ simp\ add:\ solved\text{-def})$

abbreviation it' **where**

$it'\ c \equiv globalize\ [fix\ conc\text{-forms}\ c,\ 0]\ (freshenLC\ v\text{-away})\ (to\text{-it}\ (ts\ c))$

lemma $iwf\text{-it}$:

assumes $c \in set\ conclusions$
shows $plain\text{-iwf}\ (it'\ c)\ (fst\ (root\ (ts\ c)))$
using $assms$
apply $(auto\ simp\ add:\ ts\text{-conc}\ conclusions\text{-closed}\ intro!\ iwf\text{-globalize}'\ iwf\text{-to}\text{-it}\ ts\text{-finite}\ ts\text{-wf})$
by $(meson\ assumptions\text{-closed}\ fset\text{-mp}\ mem\text{-ass}\text{-forms}\ mem\text{-conc}\text{-forms}\ ts\text{-context})$

definition $vertices :: 'form\ vertex\ fset$ **where**

$vertices = Abs\text{-fset}\ (Union\ (set\ (map\ (\lambda\ c.\ insert\ (c,\ [])\ ((\lambda\ p.\ (c,\ 0\ \#)\ p))\ ' (it\text{-paths}\ (it'\ c))))\ conclusions))$

lemma $mem\text{-vertices}: v \mid\subseteq\ vertices \longleftrightarrow (fst\ v \in set\ conclusions \wedge (snd\ v = [] \vee snd\ v \in ((\#)\ 0)\ ' it\text{-paths}\ (it'\ (fst\ v))))$

unfolding $vertices\text{-def}\ ffUnion.\text{rep}\text{-eq}$
by $(cases\ v)(auto\ simp\ add:\ Abs\text{-fset}\text{-inverse}\ Bex\text{-def})$

lemma $prefixeq\text{-vertices}: (c, is) \mid\subseteq\ vertices \implies prefix\ is'\ is \implies (c, is') \mid\subseteq\ vertices$

by $(cases\ is')(auto\ simp\ add:\ mem\text{-vertices}\ intro!\ imageI\ elim:\ it\text{-paths}\text{-prefix})$

lemma $none\text{-vertices}[simp]: (c, []) \mid\subseteq\ vertices \longleftrightarrow c \in set\ conclusions$

by $(simp\ add:\ mem\text{-vertices})$

lemma $some\text{-vertices}[simp]: (c, i\#\is) \mid\subseteq\ vertices \longleftrightarrow c \in set\ conclusions \wedge i = 0 \wedge is \in it\text{-paths}\ (it'\ c)$

by $(auto\ simp\ add:\ mem\text{-vertices})$

lemma $vertices\text{-cases}[consumes\ 1,\ case\text{-names}\ None\ Some]$:

assumes $v \mid\subseteq\ vertices$

obtains c **where** $c \in set\ conclusions$ **and** $v = (c, [])$

| c **is** **where** $c \in set\ conclusions$ **and** $is \in it\text{-paths}\ (it'\ c)$ **and** $v = (c, 0\#\is)$

using $assms$ **by** $(cases\ v;\ rename\text{-tac}\ is;\ case\text{-tac}\ is;\ auto)$

lemma $vertices\text{-induct}[consumes\ 1,\ case\text{-names}\ None\ Some]$:

assumes $v \mid\subseteq\ vertices$

assumes $\bigwedge c.\ c \in set\ conclusions \implies P\ (c,\ [])$

assumes $\bigwedge c\ is.\ c \in set\ conclusions \implies is \in it\text{-paths}\ (it'\ c) \implies P\ (c,\ 0\#\is)$

shows $P\ v$

using $assms$ **by** $(cases\ v;\ rename\text{-tac}\ is;\ case\text{-tac}\ is;\ auto)$

fun $nodeOf :: 'form\ vertex \implies ('form,\ 'rule)\ graph\text{-node}$ **where**

$nodeOf\ (pf,\ []) = Conclusion\ pf$

| $nodeOf\ (pf,\ i\#\is) = iNodeOf\ (tree\text{-at}\ (it'\ pf)\ is)$

fun *inst where*

inst (*c*, \square) = *empty-subst*

| *inst* (*c*, *i#is*) = *iSubst* (*tree-at* (*it' c*) *is*)

lemma *terminal-is-nil[simp]*: $v \in | \text{vertices} \implies \text{outPorts} (\text{nodeOf } v) = \{\square\} \longleftrightarrow \text{snd } v = \square$

by (*induction v rule: nodeOf.induct*)

(*auto elim: iNodeOf-outPorts[rotated] iwf-it*)

sublocale *Vertex-Graph nodes inPorts outPorts vertices nodeOf.*

definition *edge-from* :: '*form* \Rightarrow *nat list* \Rightarrow ('*form* *vertex* \times ('*form*, '*var*) *out-port*) **where**

edge-from c is = ((*c*, 0 # *is*), *Reg* (*iOutPort* (*tree-at* (*it' c*) *is*)))

lemma *fst-edge-from[simp]*: *fst* (*edge-from c is*) = (*c*, 0 # *is*)

by (*simp add: edge-from-def*)

fun *in-port-at* :: ('*form* \times *nat list*) \Rightarrow *nat* \Rightarrow ('*form*, '*var*) *in-port* **where**

in-port-at (*c*, \square) - = *plain-ant c*

| *in-port-at* (*c*, -#*is*) *i* = *inPorts'* (*iNodeOf* (*tree-at* (*it' c*) *is*)) ! *i*

definition *edge-to* :: '*form* \Rightarrow *nat list* \Rightarrow ('*form* *vertex* \times ('*form*, '*var*) *in-port*) **where**

edge-to c is =

(*case rev is of* $\square \Rightarrow$ ((*c*, \square), *in-port-at* (*c*, \square) 0)

| *i#is* \Rightarrow ((*c*, 0 # (*rev is*)), *in-port-at* (*c*, (0#*rev is*) *i*))

lemma *edge-to-Nil[simp]*: *edge-to c* \square = ((*c*, \square), *plain-ant c*)

by (*simp add: edge-to-def*)

lemma *edge-to-Snoc[simp]*: *edge-to c* (*is@[i]*) = ((*c*, 0 # *is*), *in-port-at* ((*c*, 0 # *is*) *i*)

by (*simp add: edge-to-def*)

definition *edge-at* :: '*form* \Rightarrow *nat list* \Rightarrow ('*form*, '*var*) *edge''* **where**

edge-at c is = (*edge-from c is*, *edge-to c is*)

lemma *fst-edge-at[simp]*: *fst* (*edge-at c is*) = *edge-from c is* **by** (*simp add: edge-at-def*)

lemma *snd-edge-at[simp]*: *snd* (*edge-at c is*) = *edge-to c is* **by** (*simp add: edge-at-def*)

lemma *hyps-exist'*:

assumes *c* \in *set conclusions*

assumes *is* \in *it-paths* (*it' c*)

assumes *tree-at* (*it' c*) *is* = (*HNode i s ants*)

shows *subst s* (*freshen i anyP*) \in *hyps-along* (*it' c*) *is*

proof—

from *assms*(1)

have *plain-iwf* (*it' c*) (*fst* (*root* (*ts c*))) **by** (*rule iwf-it*)

moreover

note *assms*(2,3)

moreover

have *fst* (*fst* (*root* (*ts c*))) \subseteq | *ass-forms*

by (*simp add: assms*(1) *ts-context*)

ultimately

show *?thesis* **by** (*rule iwf-hyps-exist*)

qed

definition *hyp-edge-to* :: '*form* \Rightarrow *nat list* \Rightarrow ('*form* *vertex* \times ('*form*, '*var*) *in-port*) **where**

hyp-edge-to c *is* = $((c, 0 \# is), \text{plain-ant any}P)$

definition *hyp-edge-from* :: 'form \Rightarrow nat list \Rightarrow nat \Rightarrow 'subst \Rightarrow ('form vertex \times ('form,'var) out-port)
where

hyp-edge-from c *is* n s =
 $((c, 0 \# \text{hyp-port-path-for } (it' c) \text{ is } (\text{subst } s (\text{freshen } n \text{ any}P))),$
 $\text{hyp-port-h-for } (it' c) \text{ is } (\text{subst } s (\text{freshen } n \text{ any}P)))$

definition *hyp-edge-at* :: 'form \Rightarrow nat list \Rightarrow nat \Rightarrow 'subst \Rightarrow ('form, 'var) edge'' **where**
hyp-edge-at c *is* n s = $(\text{hyp-edge-from } c \text{ is } n \text{ s}, \text{hyp-edge-to } c \text{ is})$

lemma *fst-hyp-edge-at[simp]*:

fst (*hyp-edge-at* c *is* n s) = *hyp-edge-from* c *is* n s **by** (*simp add: hyp-edge-at-def*)

lemma *snd-hyp-edge-at[simp]*:

snd (*hyp-edge-at* c *is* n s) = *hyp-edge-to* c *is* **by** (*simp add: hyp-edge-at-def*)

inductive-set *edges* **where**

regular-edge: $c \in \text{set conclusions} \Longrightarrow is \in \text{it-paths } (it' c) \Longrightarrow \text{edge-at } c \text{ is} \in \text{edges}$
 $| \text{hyp-edge}$: $c \in \text{set conclusions} \Longrightarrow is \in \text{it-paths } (it' c) \Longrightarrow \text{tree-at } (it' c) \text{ is} = \text{HNode } n \text{ s ants} \Longrightarrow \text{hyp-edge-at } c \text{ is } n \text{ s} \in \text{edges}$

sublocale *Pre-Port-Graph* *nodes inPorts outPorts vertices nodeOf edges*.

lemma *edge-from-valid-out-port*:

assumes $p \in \text{it-paths } (it' c)$
assumes $c \in \text{set conclusions}$
shows *valid-out-port* (*edge-from* c p)

using *assms*

by (*auto simp add: edge-from-def intro: iwf-outPort iwf-it*)

lemma *edge-to-valid-in-port*:

assumes $p \in \text{it-paths } (it' c)$
assumes $c \in \text{set conclusions}$
shows *valid-in-port* (*edge-to* c p)
using *assms*
apply (*auto simp add: edge-to-def inPorts-fset-of split: list.split elim!: it-paths-SnocE*)
apply (*rule nth-mem*)
apply (*drule (1) iwf-length-inPorts[OF iwf-it]*)
apply *auto*
done

lemma *hyp-edge-from-valid-out-port*:

assumes $is \in \text{it-paths } (it' c)$
assumes $c \in \text{set conclusions}$
assumes $\text{tree-at } (it' c) \text{ is} = \text{HNode } n \text{ s ants}$
shows *valid-out-port* (*hyp-edge-from* c *is* n s)

using *assms*

by(*auto simp add: hyp-edge-from-def intro: hyp-port-outPort it-paths-strict-prefix hyp-port-strict-prefix hyps-exist'*)

lemma *hyp-edge-to-valid-in-port*:

assumes $is \in \text{it-paths } (it' c)$
assumes $c \in \text{set conclusions}$
assumes $\text{tree-at } (it' c) \text{ is} = \text{HNode } n \text{ s ants}$
shows *valid-in-port* (*hyp-edge-to* c *is*)

using *assms* **by** (*auto simp add: hyp-edge-to-def*)

inductive *scope'* :: 'form vertex \Rightarrow ('form,'var) in-port \Rightarrow 'form \times nat list \Rightarrow bool **where**
c \in set conclusions \Longrightarrow
is' \in ((#) 0) ' it-paths (it' c) \Longrightarrow
prefix (is@[i]) *is'* \Longrightarrow
ip = in-port-at (c,*is*) *i* \Longrightarrow
scope' (c, *is*) *ip* (c, *is'*)

inductive-simps *scope-simp*: *scope'* v i v'

inductive-cases *scope-cases*: *scope'* v i v'

lemma *scope-valid*:

scope' v i v' \Longrightarrow v' \in vertices

by (auto elim: *scope-cases*)

lemma *scope-valid-inport*:

v' \in vertices \Longrightarrow *scope'* v ip v' \longleftrightarrow (\exists i. fst v = fst v' \wedge prefix (snd v@[i]) (snd v') \wedge ip = in-port-at v i)

by (cases v; cases v') (auto simp add: *scope'.simps mem-vertices*)

definition *terminal-path-from* :: 'form \Rightarrow nat list \Rightarrow ('form, 'var) edge'' list **where**

terminal-path-from c is = map (edge-at c) (rev (prefixes is))

lemma *terminal-path-from-Nil*[*simp*]:

terminal-path-from c [] = [edge-at c []]

by (*simp* add: *terminal-path-from-def*)

lemma *terminal-path-from-Snoc*[*simp*]:

terminal-path-from c (is @ [i]) = edge-at c (is@[i]) # *terminal-path-from* c is

by (*simp* add: *terminal-path-from-def*)

lemma *path-terminal-path-from*:

c \in set conclusions \Longrightarrow

is \in it-paths (it' c) \Longrightarrow

path (c, 0 # is) (c, []) (*terminal-path-from* c is)

by (induction is rule: rev-induct)

(auto simp add: *path-cons-simp intro!*: regular-edge elim: it-paths-SnocE)

lemma *edge-step*:

assumes (((a, b), ba), ((aa, bb), bc)) \in edges

obtains

i **where** a = aa **and** b = bb@[i] **and** bc = in-port-at (aa,bb) *i* **and** hyps (nodeOf (a, b)) ba = None
| *i* **where** a = aa **and** prefix (b@[i]) bb **and** hyps (nodeOf (a, b)) ba = Some (in-port-at (a,b) *i*)

using *assms*

proof(cases rule: edges.cases[consumes 1, case-names Reg Hyp])

case (Reg c is)

then obtain *i* **where** a = aa **and** b = bb@[i] **and** bc = in-port-at (aa,bb) *i* **and** hyps (nodeOf (a, b)) ba = None

by (auto elim!: edges.cases *simp* add: edge-at-def edge-from-def edge-to-def split: list.split list.split-asm)

thus thesis **by** (rule that)

next

case (Hyp c is n s)

let ?i = hyp-port-i-for (it' c) is (subst s (freshen n anyP))

from Hyp **have** a = aa **and** prefix (b@[?i]) bb **and**

hyps (nodeOf (a, b)) ba = Some (in-port-at (a,b) ?i)

by (auto simp add: edge-at-def edge-from-def edge-to-def hyp-edge-at-def hyp-edge-to-def hyp-edge-from-def

intro: hyp-port-prefix hyps-exist' hyp-port-hyps)

thus thesis **by** (rule that)

qed

lemma *path-has-prefixes*:

assumes *path* $v\ v'\ pth$

assumes $snd\ v' = []$

assumes *prefix* $(is' @ [i])\ (snd\ v)$

shows $((fst\ v,\ is'), (in-port-at\ (fst\ v,\ is')\ i)) \in snd\ 'set\ pth$

using *assms*

by (*induction rule: path.induct*)(*auto elim!: edge-step dest: prefix-snocD*)

lemma *in-scope*: $valid-in-port\ (v',\ p') \implies v \in scope\ (v',\ p') \iff scope'\ v'\ p'\ v$

proof

assume $v \in scope\ (v',\ p')$

hence $v \in |vertices|$ **and** $\bigwedge pth\ t.\ path\ v\ t\ pth \implies terminal-vertex\ t \implies (v',\ p') \in snd\ 'set\ pth$

by (*auto simp add: scope.simps*)

from *this*

show $scope'\ v'\ p'\ v$

proof (*induction rule: vertices-induct*)

case $(None\ c)$

from $None(2)[of\ (c,\ [])\ [],\ simplified,\ OF\ None(1)]$

have *False*.

thus $scope'\ v'\ p'\ (c,\ [])..$

next

case $(Some\ c\ is)$

from $\langle c \in set\ conclusions \rangle \langle is \in it-paths\ (it'\ c) \rangle$

have $path\ (c,\ 0\ \#is)\ (c,\ [])$ (*terminal-path-from c is*)

by (*rule path-terminal-path-from*)

moreover

from $\langle c \in set\ conclusions \rangle$

have *terminal-vertex* $(c,\ [])$ **by** *simp*

ultimately

have $(v',\ p') \in snd\ 'set\ (terminal-path-from\ c\ is)$

by (*rule Some(3)*)

hence $(v',\ p') \in set\ (map\ (edge-to\ c)\ (prefixes\ is))$

unfolding *terminal-path-from-def* **by** *auto*

then obtain is' **where** *prefix is' is* **and** $(v',\ p') = edge-to\ c\ is'$

by *auto*

show $scope'\ v'\ p'\ (c,\ 0\ \#is)$

proof(*cases is' rule: rev-cases*)

case *Nil*

with $\langle (v',\ p') = edge-to\ c\ is' \rangle$

have $v' = (c,\ [])$ **and** $p' = plain-ant\ c$

by (*auto simp add: edge-to-def*)

with $\langle c \in set\ conclusions \rangle \langle is \in it-paths\ (it'\ c) \rangle$

show *?thesis* **by** (*auto intro!: scope'.intros*)

next

case $(snoc\ is''\ i)$

with $\langle (v',\ p') = edge-to\ c\ is' \rangle$

have $v' = (c,\ 0\ \#is'')$ **and** $p' = in-port-at\ v'\ i$

by (*auto simp add: edge-to-def*)

with $\langle c \in set\ conclusions \rangle \langle is \in it-paths\ (it'\ c) \rangle \langle prefix\ is'\ is \rangle [unfolded\ snoc]$

show *?thesis*

by (*auto intro!: scope'.intros*)

qed

qed

next

```

assume valid-in-port ( $v'$ ,  $p'$ )
assume scope'  $v'$   $p'$   $v$ 
then obtain  $c$   $is'$   $i$  is where
   $v' = (c, is')$  and  $v = (c, is)$  and  $c \in \text{set conclusions}$  and
   $p' = \text{in-port-at } v' \ i$  and
   $is \in (\#) \ 0 \ \text{'it-paths } (it' \ c)$  and prefix ( $is' \ @ \ [i]$ ) is
  by (auto simp add: scope'.simps)

from  $\langle \text{scope}' \ v' \ p' \ v \rangle$ 
have  $(c, is) \in \text{vertices unfolding } \langle v = - \rangle$  by (rule scope-valid)
hence  $(c, is) \in \text{scope } ((c, is'), p')$ 
proof(rule scope.intros)
  fix  $pth \ t$ 
  assume path  $(c, is) \ t \ pth$ 

  assume terminal-vertex  $t$ 
  hence  $\text{snd } t = []$  by auto

  from path-has-prefixes[OF  $\langle \text{path } (c, is) \ t \ pth \rangle \langle \text{snd } t = [] \rangle$ , simplified, OF  $\langle \text{prefix } (is' \ @ \ [i]) \ is \rangle$ ]
  show  $((c, is'), p') \in \text{snd 'set pth unfolding } \langle p' = - \rangle \langle v' = - \rangle$ .
qed
thus  $v \in \text{scope } (v', p')$  using  $\langle v = - \rangle \langle v' = - \rangle$  by simp
qed

```

sublocale *Port-Graph nodes inPorts outPorts vertices nodeOf edges*
proof

```

show nodeOf 'fset vertices  $\subseteq$  sset nodes
  apply (auto simp add: mem-vertices)
  apply (auto simp add: stream.set-map dest: iNodeOf-tree-at[OF iwf-it])
  done
next

```

```

have  $\forall e \in \text{edges. valid-out-port } (fst \ e) \wedge \text{valid-in-port } (snd \ e)$ 
  by (auto elim!: edges.cases simp add: edge-at-def
    dest: edge-from-valid-out-port edge-to-valid-in-port
    dest: hyp-edge-from-valid-out-port hyp-edge-to-valid-in-port)

```

```

thus  $\forall (ps1, ps2) \in \text{edges. valid-out-port } ps1 \wedge \text{valid-in-port } ps2$  by auto
qed

```

sublocale *Scoped-Graph nodes inPorts outPorts vertices nodeOf edges hysps..*

lemma *hyps-free-path-length:*

```

assumes path  $v \ v' \ pth$ 
assumes hyps-free  $pth$ 
shows  $\text{length } pth + \text{length } (snd \ v') = \text{length } (snd \ v)$ 
using assms by induction (auto elim!: edge-step)

```

fun *vidx* :: 'form vertex \Rightarrow nat **where**

```

  vidx  $(c, []) = \text{isidx } [fidx \ \text{conc-forms } c]$ 
   $|\text{vidx } (c, -\#is) = \text{iAnnot } (\text{tree-at } (it' \ c) \ is)$ 

```

lemma *my-vidx-inj: inj-on vidx (fset vertices)*

```

by (rule inj-onI)
  (auto simp add: mem-vertices iAnnot-globalize simp del: iAnnot.simps)

```

lemma *vidx-not-v-away*[simp]: $v \in \text{vertices} \implies \text{vidx } v \neq v\text{-away}$
by (*cases v rule:vidx.cases*) (*auto simp add: iAnnot-globalize simp del: iAnnot.simps*)

sublocale *Instantiation inPorts outPorts nodeOf hyps nodes edges vertices labelsIn labelsOut freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP vidx inst*

proof

show *inj-on vidx* (*fset vertices*) **by** (*rule my-vidx-inj*)

qed

sublocale *Well-Scoped-Graph nodes inPorts outPorts vertices nodeOf edges hyps*

proof

fix $v_1 p_1 v_2 p_2 p'$

assume *assms*: $((v_1, p_1), (v_2, p_2)) \in \text{edges hyps (nodeOf } v_1) p_1 = \text{Some } p'$

from *assms*(1) *hyps-correct*[OF *assms*(2)]

have *valid-out-port* (v_1, p_1) **and** *valid-in-port* (v_2, p_2) **and** *valid-in-port* (v_1, p') **and** $v_2 \in \text{vertices}$
using *valid-edges* **by** *auto*

from *assms*

have $\exists i. \text{fst } v_1 = \text{fst } v_2 \wedge \text{prefix (snd } v_1 @ [i]) (\text{snd } v_2) \wedge p' = \text{in-port-at } v_1 i$

by (*cases v1; cases v2; auto elim!: edge-step*)

hence *scope'* $v_1 p' v_2$

unfolding *scope-valid-inport*[OF $\langle v_2 \in \text{vertices} \rangle$].

hence $v_2 \in \text{scope } (v_1, p')$

unfolding *in-scope*[OF $\langle \text{valid-in-port } (v_1, p') \rangle$].

thus $(v_2, p_2) = (v_1, p') \vee v_2 \in \text{scope } (v_1, p') ..$

qed

sublocale *Acyclic-Graph nodes inPorts outPorts vertices nodeOf edges hyps*

proof

fix $v \text{ pth}$

assume *path* $v v \text{ pth}$ **and** *hyps-free* pth

from *hyps-free-path-length*[OF *this*]

show $\text{pth} = []$ **by** *simp*

qed

sublocale *Saturated-Graph nodes inPorts outPorts vertices nodeOf edges*

proof

fix $v p$

assume *valid-in-port* (v, p)

thus $\exists e \in \text{edges. snd } e = (v, p)$

proof(*induction v*)

fix $c \text{ cis}$

assume *valid-in-port* $((c, \text{cis}), p)$

hence $c \in \text{set conclusions}$ **by** (*auto simp add: mem-vertices*)

show $\exists e \in \text{edges. snd } e = ((c, \text{cis}), p)$

proof(*cases cis*)

case *Nil*

with $\langle \text{valid-in-port } ((c, \text{cis}), p) \rangle$

have [simp]: $p = \text{plain-ant } c$ **by** *simp*

have $[] \in \text{it-paths (it' } c)$ **by** *simp*

with $\langle c \in \text{set conclusions} \rangle$

have *edge-at* $c [] \in \text{edges}$ **by** (*rule regular-edge*)

moreover

have *snd* (*edge-at* $c []$) = $((c, []), \text{plain-ant } c)$

by (*simp add: edge-to-def*)

ultimately
show *?thesis* **by** (*auto simp add: Nil simp del: snd-edge-at*)
next
case (*Cons c' is*)
with $\langle \text{valid-in-port } ((c, cis), p) \rangle$
have [*simp*]: $c' = 0$ **and** $is \in \text{it-paths } (it' c)$
and $p \in \text{inPorts } (iNodeOf (tree-at (it' c) is))$ **by** *auto*

from *this(3)* **obtain** *i* **where**
 $i < \text{length } (\text{inPorts}' (iNodeOf (tree-at (it' c) is)))$ **and**
 $p = \text{inPorts}' (iNodeOf (tree-at (it' c) is)) ! i$
by (*auto simp add: inPorts-fset-of in-set-conv-nth*)

show *?thesis*
proof (*cases tree-at (it' c) is*)
case [*simp*]: (*RNode r ants*)
show *?thesis*
proof(*cases r*)
case *I*
hence $\neg \text{isHNode } (tree-at (it' c) is)$ **by** *simp*
from $\text{iwf-length-inPorts-not-HNode}[OF \text{iwf-it}[OF \langle c \in \text{set conclusions} \rangle] \langle is \in \text{it-paths } (it' c) \rangle \text{this}]$
 $\langle i < \text{length } (\text{inPorts}' (iNodeOf (tree-at (it' c) is))) \rangle$
have $i < \text{length } (\text{children } (tree-at (it' c) is))$ **by** *simp*
with $\langle is \in \text{it-paths } (it' c) \rangle$
have $is@[i] \in \text{it-paths } (it' c)$ **by** (*rule it-path-SnocI*)
from $\langle c \in \text{set conclusions} \rangle \text{this}$
have $\text{edge-at } c (is@[i]) \in \text{edges}$ **by** (*rule regular-edge*)
moreover
have $\text{snd } (\text{edge-at } c (is@[i])) = ((c, 0 \# is), \text{inPorts}' (iNodeOf (tree-at (it' c) is)) ! i)$
by (*simp add: edge-to-def*)
ultimately
show *?thesis* **by** (*auto simp add: Cons $\langle p = \rightarrow \text{simp del: snd-edge-at} \rangle$*)
next
case (*H n s*)
hence $\text{tree-at } (it' c) is = \text{HNode } n s \text{ ants}$ **by** *simp*
from $\langle c \in \text{set conclusions} \rangle \langle is \in \text{it-paths } (it' c) \rangle \text{this}$
have $\text{hyp-edge-at } c is n s \in \text{edges..}$
moreover
from $H \langle p \in \text{inPorts } (iNodeOf (tree-at (it' c) is)) \rangle$
have [*simp*]: $p = \text{plain-ant anyP}$ **by** *simp*

have $\text{snd } (\text{hyp-edge-at } c is n s) = ((c, 0 \# is), p)$
by (*simp add: hyp-edge-to-def*)
ultimately
show *?thesis* **by** (*auto simp add: Cons simp del: snd-hyp-edge-at*)
qed
qed
qed
qed
qed

sublocale *Pruned-Port-Graph nodes inPorts outPorts vertices nodeOf edges*
proof
fix *v*
assume $v \in \text{vertices}$
thus $\exists \text{pth } v'. \text{path } v v' \text{pth} \wedge \text{terminal-vertex } v'$
proof(*induct rule: vertices-induct*)

```

case (None c)
hence terminal-vertex (c,[]) by simp
with path.intros(1)
show ?case by blast
next
case (Some c is)
hence path (c, 0 # is) (c, []) (terminal-path-from c is)
  by (rule path-terminal-path-from)
moreover
have terminal-vertex (c,[]) using Some(1) by simp
ultimately
show ?case by blast
qed
qed

```

sublocale Well-Shaped-Graph nodes inPorts outPorts vertices nodeOf edges hyps..

sublocale sol:Solution inPorts outPorts nodeOf hyps nodes vertices labelsIn labelsOut freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP vidx inst edges

proof

```

fix v1 p1 v2 p2
assume ((v1, p1), (v2, p2)) ∈ edges
thus labelAtOut v1 p1 = labelAtIn v2 p2
proof(cases rule:edges.cases)
  case (regular-edge c is)

```

```

from ⟨((v1, p1), v2, p2) = edge-at c is⟩
have (v1,p1) = edge-from c is using fst-edge-at by (metis fst-conv)
hence [simp]: v1 = (c, 0 # is) by (simp add: edge-from-def)

```

show ?thesis

proof(cases is rule:rev-cases)

```

  case Nil
  let ?t' = it' c
  have labelAtOut v1 p1 = subst (iSubst ?t') (freshen (vidx v1) (iOutPort ?t'))
    using regular-edge Nil by (simp add: labelAtOut-def edge-at-def edge-from-def)
  also have vidx v1 = iAnnot ?t' by (simp add: Nil)
  also have subst (iSubst ?t') (freshen (iAnnot ?t') (iOutPort ?t')) = snd (fst (root (ts c)))
    unfolding iwf-subst-freshen-outPort[OF iwf-it[OF ⟨c ∈ set conclusions⟩]]..
  also have ... = c using ⟨c ∈ set conclusions⟩ by (simp add: ts-conc)
  also have ... = labelAtIn v2 p2
    using ⟨c ∈ set conclusions⟩ regular-edge Nil
    by (simp add: labelAtIn-def edge-at-def freshen-closed conclusions-closed closed-no-lconsts)
  finally show ?thesis.

```

next

```

  case (snoc is' i)
  let ?t1 = tree-at (it' c) (is'@[i])
  let ?t2 = tree-at (it' c) is'
  have labelAtOut v1 p1 = subst (iSubst ?t1) (freshen (vidx v1) (iOutPort ?t1))
    using regular-edge snoc by (simp add: labelAtOut-def edge-at-def edge-from-def)
  also have vidx v1 = iAnnot ?t1 using snoc regular-edge(3) by simp
  also have subst (iSubst ?t1) (freshen (iAnnot ?t1) (iOutPort ?t1))
    = subst (iSubst ?t2) (freshen (iAnnot ?t2) (a-conc (inPorts' (iNodeOf ?t2) ! i)))
    by (rule iwf-edge-match[OF iwf-it[OF ⟨c ∈ set conclusions⟩] ⟨is ∈ it-paths (it' c)⟩[unfolded snoc]])
  also have iAnnot ?t2 = vidx (c, 0 # is') by simp
  also have subst (iSubst ?t2) (freshen (vidx (c, 0 # is')) (a-conc (inPorts' (iNodeOf ?t2) ! i))) = labelAtIn

```

v₂ p₂

```

    using regular-edge snoc by (simp add: labelAtIn-def edge-at-def)
  finally show ?thesis.
qed
next
  case (hyp-edge c is n s ants)
  let ?f = subst s (freshen n anyP)
  let ?h = hyp-port-h-for (it' c) is ?f
  let ?his = hyp-port-path-for (it' c) is ?f
  let ?t1 = tree-at (it' c) ?his
  let ?t2 = tree-at (it' c) is

  from ⟨c ∈ set conclusions⟩ ⟨is ∈ it-paths (it' c)⟩ ⟨tree-at (it' c) is = HNode n s ants⟩
  have ?f ∈ hyps-along (it' c) is
    by (rule hyps-exist')

  from ⟨((v1, p1), v2, p2) = hyp-edge-at c is n s⟩
  have (v1, p1) = hyp-edge-from c is n s using fst-hyp-edge-at by (metis fst-conv)
  hence [simp]: v1 = (c, 0 # ?his) by (simp add: hyp-edge-from-def)

  have labelAtOut v1 p1 = subst (iSubst ?t1) (freshen (vidx v1) (labelsOut (iNodeOf ?t1) ?h))
    using hyp-edge by (simp add: hyp-edge-at-def hyp-edge-from-def labelAtOut-def)
  also have vidx v1 = iAnnot ?t1 by simp
  also have subst (iSubst ?t1) (freshen (iAnnot ?t1) (labelsOut (iNodeOf ?t1) ?h)) = ?f using ⟨?f ∈
hyps-along (it' c) is⟩ by (rule local.hyp-port-eq[symmetric])
  also have ... = subst (iSubst ?t2) (freshen (iAnnot ?t2) anyP) using hyp-edge by simp
  also have subst (iSubst ?t2) (freshen (iAnnot ?t2) anyP) = labelAtIn v2 p2
    using hyp-edge by (simp add: labelAtIn-def hyp-edge-at-def hyp-edge-to-def)
  finally show ?thesis.
qed
qed

lemma node-disjoint-fresh-vars:
  assumes n ∈ sset nodes
  assumes i < length (inPorts' n)
  assumes i' < length (inPorts' n)
  shows a-fresh (inPorts' n ! i) ∩ a-fresh (inPorts' n ! i') = {} ∨ i = i'
  using assms no-multiple-local-consts
  by (fastforce simp add: nodes-def stream.set-map)

sublocale Well-Scoped-Instantiation freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs
anyP inPorts outPorts nodeOf hyps nodes vertices labelsIn labelsOut vidx inst edges local-vars
proof
  fix v p var v'
  assume valid-in-port (v, p)
  hence v |∈| vertices by simp

  obtain c is where v = (c, is) by (cases v, auto)

  from ⟨valid-in-port (v, p)⟩ ⟨v = -⟩
  have (c, is) |∈| vertices and p |∈| inPorts (nodeOf (c, is)) by simp-all
  hence c ∈ set conclusions by (simp add: mem-vertices)

  from ⟨p |∈| -⟩ obtain i where
    i < length (inPorts' (nodeOf (c, is))) and
    p = inPorts' (nodeOf (c, is)) ! i by (auto simp add: inPorts-fset-of in-set-conv-nth)

```

hence $p = \text{in-port-at } (c, is) \text{ } i \text{ by } (\text{cases } is) \text{ } \text{auto}$

assume $v' \in | \text{vertices}$

then obtain $c' \text{ } is' \text{ where } v' = (c', is') \text{ by } (\text{cases } v', \text{ auto})$

assume $var \in \text{local-vars } (\text{nodeOf } v) \text{ } p$

hence $var \in a\text{-fresh } p \text{ by simp}$

assume $\text{freshenLC } (vidx \ v) \text{ } var \in \text{subst-lconsts } (\text{inst } v')$

then obtain $is'' \text{ where } is' = 0 \# is'' \text{ and } is'' \in \text{it-paths } (it' \ c')$

using $\langle v' \in | \text{vertices} \rangle$

by $(\text{cases } is') \text{ } (\text{auto simp add: } \langle v' = - \rangle)$

note $\langle \text{freshenLC } (vidx \ v) \text{ } var \in \text{subst-lconsts } (\text{inst } v') \rangle$

also

have $\text{subst-lconsts } (\text{inst } v') = \text{subst-lconsts } (i\text{Subst } (\text{tree-at } (it' \ c') \ is''))$

by $(\text{simp add: } \langle v' = - \rangle \langle is' = - \rangle)$

also

from $\langle is'' \in \text{it-paths } (it' \ c') \rangle$

have $\dots \subseteq \text{fresh-at-path } (it' \ c') \ is'' \cup \text{range } (\text{freshenLC } v\text{-away})$

by $(\text{rule } \text{globalize-local-consts})$

finally

have $\text{freshenLC } (vidx \ v) \text{ } var \in \text{fresh-at-path } (it' \ c') \ is''$

using $\langle v \in | \text{vertices} \rangle \text{ by auto}$

then obtain $is''' \text{ where prefix } is''' \ is'' \text{ and } \text{freshenLC } (vidx \ v) \text{ } var \in \text{fresh-at } (it' \ c') \ is'''$

unfolding fresh-at-path-def by auto

then obtain $i' \ is'''' \text{ where prefix } (is''''@[i']) \ is''$

and $\text{freshenLC } (vidx \ v) \text{ } var \in \text{fresh-at } (it' \ c') \ (is''''@[i'])$

using $\text{append-butlast-last-id}[\text{where } xs = is''', \text{ symmetric}]$

apply $(\text{cases } is''' = [])$

apply $(\text{auto simp del: } \text{fresh-at-snoc } \text{append-butlast-last-id})$

apply metis

done

from $\langle is'' \in \text{it-paths } (it' \ c') \rangle \langle \text{prefix } (is''''@[i']) \ is'' \rangle$

have $(is''''@[i']) \in \text{it-paths } (it' \ c') \text{ by } (\text{rule } \text{it-paths-prefix})$

hence $is'''' \in \text{it-paths } (it' \ c') \text{ using } \text{append-prefixD } \text{it-paths-prefix} \text{ by blast}$

from $\text{this } \langle \text{freshenLC } (vidx \ v) \text{ } var \in \text{fresh-at } (it' \ c') \ (is''''@[i']) \rangle$

have $c = c' \wedge is = 0 \# is'''' \wedge var \in a\text{-fresh } (\text{inPorts}' (i\text{NodeOf } (\text{tree-at } (it' \ c') \ is'''')) ! i')$

unfolding $\text{fresh-at-def}'$ using $\langle v \in | \text{vertices} \rangle \langle v' \in | \text{vertices} \rangle$

apply $(\text{cases } is)$

apply $(\text{auto split: } \text{if-splits } \text{simp add: } i\text{Annot-globalize } \text{it-paths-butlast } \langle v = - \rangle \langle v' = - \rangle \langle is' = - \rangle \text{ simp del: } i\text{Annot.simps})$

done

hence $c' = c \text{ and } is = 0 \# is'''' \text{ and } var \in a\text{-fresh } (\text{inPorts}' (i\text{NodeOf } (\text{tree-at } (it' \ c') \ is'''')) ! i') \text{ by } \text{simp-all}$

from $\langle (is''''@[i']) \in \text{it-paths } (it' \ c') \rangle$

have $i' < \text{length } (\text{inPorts}' (\text{nodeOf } (c, is)))$

using $\text{iwf-length-inPorts}[OF \ \text{iwf-it}[OF \ \langle c \in \text{set conclusions} \rangle]]$

by $(\text{auto elim!: } \text{it-paths-SnocE } \text{simp add: } \langle is = - \rangle \langle c' = - \rangle \text{ order.strict-trans2})$

have $\text{nodeOf } (c, is) \in \text{sset nodes}$

unfolding $\langle is = - \rangle \langle c' = - \rangle \text{ nodeOf.simps}$

by $(\text{rule } i\text{NodeOf-tree-at}[OF \ \text{iwf-it}[OF \ \langle c \in \text{set conclusions} \rangle]] \ \langle is'''' \in \text{it-paths } (it' \ c') \rangle [\text{unfolded } \langle c' = - \rangle])$

from $\langle \text{var} \in a\text{-fresh } (\text{inPorts}' (\text{iNodeOf } (\text{tree-at } (\text{it}' c') \text{ is}''')) ! i') \rangle$
 $\langle \text{var} \in a\text{-fresh } p \rangle \langle p = \text{inPorts}' (\text{nodeOf } (c, \text{is})) ! i \rangle$
 $\text{node-disjoint-fresh-vars}[OF$
 $\langle \text{nodeOf } (c, \text{is}) \in \text{sset nodes} \rangle$
 $\langle i < \text{length } (\text{inPorts}' (\text{nodeOf } (c, \text{is}))) \rangle \langle i' < \text{length } (\text{inPorts}' (\text{nodeOf } (c, \text{is}))) \rangle]$
have $i' = i$ **by** $(\text{auto simp add: } \langle \text{is} \Rightarrow \rangle \langle c' = c \rangle)$

from $\langle \text{prefix } (\text{is}''''@[i']) \text{ is}'' \rangle$
have $\text{prefix } (\text{is } @[i']) \text{ is}'$ **by** $(\text{simp add: } \langle \text{is}' \Rightarrow \rangle \langle \text{is} \Rightarrow \rangle)$

from $\langle c \in \text{set conclusions} \rangle \langle \text{is}'' \in \text{it-paths } (\text{it}' c') \rangle \langle \text{prefix } (\text{is } @[i']) \text{ is}' \rangle$
 $\langle p = \text{in-port-at } (c, \text{is}) i \rangle$
have $\text{scope}' v p v'$
unfolding $\langle v \Rightarrow \rangle \langle v' \Rightarrow \rangle \langle c' = - \rangle \langle \text{is}' = - \rangle \langle i' \Rightarrow \rangle$ **by** $(\text{auto intro: } \text{scope}'.\text{intros})$
thus $v' \in \text{scope } (v, p)$ **using** $\langle \text{valid-in-port } (v, p) \rangle$ **by** $(\text{simp add: } \text{in-scope})$

qed

sublocale *Scoped-Proof-Graph freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP inPorts outPorts nodeOf hyps nodes vertices labelsIn labelsOut vidx inst edges local-vars..*

sublocale *tpg:Tasked-Proof-Graph freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP antecedent consequent rules assumptions conclusions vertices nodeOf edges vidx inst*

proof

show $\text{set } (\text{map } \text{Conclusion } \text{conclusions}) \subseteq \text{nodeOf } ' \text{fset vertices}$

proof–

{

fix c

assume $c \in \text{set conclusions}$

hence $(c, []) \in \text{vertices}$ **by** *simp*

hence $\text{nodeOf } (c, []) \in \text{nodeOf } ' \text{fset vertices}$

by $(\text{rule } \text{imageI})$

hence $\text{Conclusion } c \in \text{nodeOf } ' \text{fset vertices}$ **by** *simp*

} **thus** *?thesis* **by** *auto*

qed

qed

end

end

8 Instantiations

To ensure that our locale assumption are fulfillable, we instantiate them with small examples.

8.1 Propositional_Formulas

```

theory Propositional-Formulas
imports
  Abstract-Formula
  HOL-Library.Countable
  HOL-Library.Infinite-Set
  HOL-Library.Infinite-Typeclass
begin

lemma countable-infinite-ex-bij:  $\exists f::('a::\{countable,infinite\}\Rightarrow'b::\{countable,infinite\})$ . bij f
proof –
  have infinite (range (to-nat::a  $\Rightarrow$  nat))
    using finite-imageD infinite-UNIV by blast
  moreover have infinite (range (to-nat::b  $\Rightarrow$  nat))
    using finite-imageD infinite-UNIV by blast
  ultimately have  $\exists f$ . bij-betw f (range (to-nat::a  $\Rightarrow$  nat)) (range (to-nat::b  $\Rightarrow$  nat))
    by (meson bij-betw-inv bij-betw-trans bij-enumerate)
  then obtain f where f-def: bij-betw f (range (to-nat::a  $\Rightarrow$  nat)) (range (to-nat::b  $\Rightarrow$  nat)) ..
  then have f-range-trans: f ‘ (range (to-nat::a  $\Rightarrow$  nat)) = range (to-nat::b  $\Rightarrow$  nat)
    unfolding bij-betw-def by simp
  have surj ((from-nat::nat  $\Rightarrow$  'b)  $\circ$  f  $\circ$  (to-nat::a  $\Rightarrow$  nat))
proof (rule surjI)
  fix a
  obtain b where [simp]: to-nat (a::'b) = b by blast
  hence b  $\in$  range (to-nat::b  $\Rightarrow$  nat) by blast
  with f-range-trans have b  $\in$  f ‘ (range (to-nat::a  $\Rightarrow$  nat)) by simp
  from imageE [OF this] obtain c where [simp]: f c = b and c  $\in$  range (to-nat::a  $\Rightarrow$  nat)
    by auto
  with f-def have [simp]: inv-into (range (to-nat::a  $\Rightarrow$  nat)) f b = c
    by (meson bij-betw-def inv-into-f-f)
  then obtain d where cd: from-nat c = (d::'a) by blast
  with  $\langle c \in \text{range to-nat} \rangle$  have [simp]: to-nat d = c by auto
  from  $\langle \text{to-nat } a = b \rangle$  have [simp]: from-nat b = a
    using from-nat-to-nat by blast
  show (from-nat  $\circ$  f  $\circ$  to-nat) (((from-nat::nat  $\Rightarrow$  'a)  $\circ$  inv-into (range (to-nat::a  $\Rightarrow$  nat)) f  $\circ$  (to-nat::b
 $\Rightarrow$  nat)) a) = a
    by (clarsimp simp: cd)
  qed
  moreover have inj ((from-nat::nat  $\Rightarrow$  'b)  $\circ$  f  $\circ$  (to-nat::a  $\Rightarrow$  nat))
    apply (rule injI)
    apply auto
  apply (metis bij-betw-inv-into-left f-def f-inv-into-f f-range-trans from-nat-def image-eqI rangeI to-nat-split)
  done
  ultimately show ?thesis by (blast intro: bijI)
qed

```

Propositional formulas are either a variable from an infinite but countable set, or a function given by a name and the arguments.

```

datatype ('var,'cname) pform =
  Var 'var::\{countable,infinite\}

```

| *Fun* (*name*:*'cname*) (*params*: (*'var*,*'cname*) *pform list*)

Substitution on and closedness of propositional formulas is straight forward.

```
fun subst :: ('var::{countable,infinite} => ('var,'cname) pform) => ('var,'cname) pform => ('var,'cname) pform
where subst s (Var v) = s v
| subst s (Fun n ps) = Fun n (map (subst s) ps)
```

```
fun closed :: ('var::{countable,infinite},'cname) pform => bool
where closed (Var v) <=> False
| closed (Fun n ps) <=> list-all closed ps
```

Now we can interpret *Abstract-Formulas*. As there are no locally fixed constants in propositional formulas, most of the locale parameters are dummy values

interpretation *propositional*: *Abstract-Formulas*

— No need to freshen locally fixed constants

curry (*SOME* *f*. *bij* *f*):: *nat* => *'var* => *'var*

— also no renaming needed as there are no locally fixed constants

λ -. *id* λ -. {}

— closedness and substitution as defined above

closed :: ('var::{countable,infinite},'cname) pform => bool *subst*

— no substitution and renaming of locally fixed constants

λ -. {} λ -. *id*

— most generic formula

Var *undefined*

proof

fix *a* *v* *a'* *v'*

from *countable-infinite-ex-bij* **obtain** *f* **where** *bij* (*f*::*nat* × *'var* => *'var*) **by** *blast*

```
then show (curry (SOME f. bij (f::nat × 'var => 'var)) (a::nat) (v::'var) = curry (SOME f. bij f) (a'::nat)
(v'::'var)) =
  (a = a' ∧ v = v')
```

```
apply (rule someI2 [where Q= $\lambda$ f. curry f a v = curry f a' v' <=> a = a' ∧ v = v'])
```

```
by auto (metis bij-pointE prod.inject)+
```

next

fix *f* *s*

assume *closed* (*f*::('var, 'cname) pform)

then show *subst* *s* *f* = *f*

proof (*induction* *s* *f* *rule*: *subst.induct*)

case (*2* *s* *n* *ps*)

thus ?*case* **by** (*induction* *ps*) *auto*

qed *auto*

next

have *subst* *Var* *f* = *f* **for** *f* :: ('var,'cname) pform

by (*induction* *f*) (*auto* *intro*: *map-idI*)

then show \exists *s*. (\forall *f*. *subst* *s* (*f*::('var,'cname) pform) = *f*) ∧ {} = {}

by (*rule-tac* *x*=*Var* **in** *exI*; *clarsimp*)

qed *auto*

declare *propositional.subst-lconsts-empty-subst* [*simp* *del*]

end

8.2 Incredible_Propositional

theory *Incredible-Propositional* **imports**

Abstract-Rules-To-Incredible

Propositional-Formulas

begin

Our concrete interpretation with propositional logic will cover conjunction and implication as well as constant symbols. The type for variables will be *string*.

datatype *prop-funs* = *and* | *imp* | *Const string*

The rules are introduction and elimination of conjunction and implication.

datatype *prop-rule* = *andI* | *andE* | *impI* | *impE*

definition *prop-rules* :: *prop-rule stream*
where *prop-rules* = *cycle [andI, andE, impI, impE]*

lemma *iR-prop-rules [simp]*: *sset prop-rules* = {*andI, andE, impI, impE*}
unfolding *prop-rules-def* **by** *simp*

Just some short notation.

abbreviation *X* :: (*string, 'a*) *pform*
where *X* ≡ *Var "X"*

abbreviation *Y* :: (*string, 'a*) *pform*
where *Y* ≡ *Var "Y"*

Finally the right- and left-hand sides of the rules.

fun *consequent* :: *prop-rule* ⇒ (*string, prop-funs*) *pform list*
where *consequent andI* = [*Fun and [X, Y]*]
| *consequent andE* = [*X, Y*]
| *consequent impI* = [*Fun imp [X, Y]*]
| *consequent impE* = [*Y*]

fun *antecedent* :: *prop-rule* ⇒ ((*string, prop-funs*) *pform, string*) *antecedent list*
where *antecedent andI* = [*plain-ant X, plain-ant Y*]
| *antecedent andE* = [*plain-ant (Fun and [X, Y])*]
| *antecedent impI* = [*Antecedent {|X|} Y {|}*]
| *antecedent impE* = [*plain-ant (Fun imp [X, Y]), plain-ant X*]

interpretation *propositional: Abstract-Rules*
curry (SOME f. bij f):: *nat* ⇒ *string* ⇒ *string*

λ-. id

λ-. {}

closed :: (*string, prop-funs*) *pform* ⇒ *bool*

subst

λ-. {}

λ-. id

Var undefined

antecedent

consequent

prop-rules

proof

show $\forall xs \in \text{sset } \text{prop-rules}. \text{consequent } xs \neq []$

unfolding *prop-rules-def*

using *consequent.elims* **by** *blast*

next

show $\forall xs \in \text{sset } \text{prop-rules}. \bigcup ((\lambda-. \{\}) \text{ ` set (consequent } xs)) = \{\}$
by *clarsimp*

```

next
  fix i' r i ia
  assume r ∈ sset prop-rules
  and ia < length (antecedent r)
  and i' < length (antecedent r)
  then show a-fresh (antecedent r ! ia) ∩ a-fresh (antecedent r ! i') = {} ∨ ia = i'
  by (cases i'; auto)
next
  fix p
  show {} ∪ ∪((λ-. {}) ' fset (a-hyps p)) ⊆ a-fresh p by clarsimp
qed

end

```

8.3 Incredible_Propositional_Tasks

theory *Incredible-Propositional-Tasks*

imports

Incredible-Completeness

Incredible-Propositional

begin

context *ND-Rules-Inst* **begin**

lemma *eff-NatRuleI*:

nat-rule rule c ants

\implies *entail* = $(\Gamma \vdash \text{subst } s (\text{freshen } a \ c))$

\implies *hyps* = $((\lambda \text{ant. } ((\lambda p. \text{subst } s (\text{freshen } a \ p)) \mid^{\cdot} \text{a-hyps } \text{ant} \mid \cup \mid \Gamma \vdash \text{subst } s (\text{freshen } a \ (\text{a-conc } \text{ant})))) \mid^{\cdot}$

ants)

$\implies (\bigwedge \text{ant } f. \text{ant} \mid \in \mid \text{ants} \implies f \mid \in \mid \Gamma \implies \text{freshenLC } a \ ' (\text{a-fresh } \text{ant}) \cap \text{lconsts } f = \{\})$

$\implies (\bigwedge \text{ant. } \text{ant} \mid \in \mid \text{ants} \implies \text{freshenLC } a \ ' (\text{a-fresh } \text{ant}) \cap \text{subst-lconsts } s = \{\})$

\implies *eff* (*NatRule rule*) *entail hyps*

by (*drule eff.intros(2)*) *simp-all*

end

context *Abstract-Task* **begin**

lemma *natEff-InstI*:

rule = (r,c)

$\implies c \in \text{set } (\text{consequent } r)$

$\implies \text{antec} = \text{f-antecedent } r$

$\implies \text{natEff-Inst rule } c \text{ antec}$

by (*metis natEff-Inst.intros*)

end

context **begin**

8.3.1 Task 1.1

This is the very first task of the Incredible Proof Machine: $A \longrightarrow A$

abbreviation *A* :: (*string,prop-funs*) *pform*

where *A* \equiv *Fun* (*Const "A"*) \square

First the task is defined as an *Abstract-Task*.

interpretation *task1-1*: *Abstract-Task*

curry (*SOME f. bij f*):: *nat* \Rightarrow *string* \Rightarrow *string*

$\lambda-. \text{id}$

```

λ-. {}
closed :: (string, prop-funs) pform ⇒ bool
subst
λ-. {}
λ-. id
Var undefined
antecedent
consequent
prop-rules
[A]
[A]
by unfold-locales simp

```

Then we show, that this task has a proof within our formalization of natural deduction by giving a concrete proof tree.

```

lemma task1-1.solved
  unfolding task1-1.solved-def
  apply clarsimp
  apply (rule-tac x={|A|} in exI)
  apply clarsimp
  apply (rule-tac x=Node ({|A|} ⊢ A, Axiom) {||} in exI)
  apply clarsimp
  apply (rule conjI)
  apply (rule task1-1.wf)
  apply (solves clarsimp)
  apply clarsimp
  apply (rule task1-1.eff.intros(1))
  apply (solves simp)
  apply (solves clarsimp)
by (auto intro: tfinite.intros)

```

print-locale *Vertex-Graph*

interpretation *task1-1: Vertex-Graph* *task1-1.nodes task1-1.inPorts task1-1.outPorts* $\{|0::nat,1|\}$
undefined(0 := Assumption A, 1 := Conclusion A)

.

print-locale *Pre-Port-Graph*

interpretation *task1-1: Pre-Port-Graph* *task1-1.nodes task1-1.inPorts task1-1.outPorts* $\{|0::nat,1|\}$
undefined(0 := Assumption A, 1 := Conclusion A)
 $\{((0,Reg A),(1,plain-ant A))\}$

.

print-locale *Instantiation*

interpretation *task1-1: Instantiation*
task1-1.inPorts
task1-1.outPorts
undefined(0 := Assumption A, 1 := Conclusion A)
task1-1.hyps
task1-1.nodes
 $\{((0,Reg A),(1,plain-ant A))\}$
 $\{|0::nat,1|\}$
task1-1.labelsIn
task1-1.labelsOut
curry (SOME f. bij f):: nat ⇒ string ⇒ string
λ-. id
λ-. {}

```

closed :: (string, prop-funs) pform => bool
subst
λ-. {}
λ-. id
Var undefined
id
undefined
by unfold-locales simp

declare One-nat-def [simp del]

lemma path-one-edge[simp]:
  task1-1.path v1 v2 pth <=>
    (v1 = 0 ∧ v2 = 1 ∧ pth = [((0,Reg A),(1,plain-ant A))] ∨
     pth = [] ∧ v1 = v2)
  apply (cases pth)
  apply (auto simp add: task1-1.path-cons-simp')
  apply (rename-tac list, case-tac list, auto simp add: task1-1.path-cons-simp')+
  done

```

Finally we can also show that there is a proof graph for this task.

```

interpretation Tasked-Proof-Graph
  curry (SOME f. bij f):: nat => string => string
  λ-. id
  λ-. {}
  closed :: (string, prop-funs) pform => bool
  subst
  λ-. {}
  λ-. id
  Var undefined
  antecedent
  consequent
  prop-rules
  [A]
  [A]
  {|0::nat,1|}
  undefined(0 := Assumption A, 1 := Conclusion A)
  {((0,Reg A),(1,plain-ant A))}
  id
  undefined
apply unfold-locales
  apply (solves simp)
  apply (solves clarsimp)
  apply (solves clarsimp)
  apply (solves clarsimp)
  apply (solves clarsimp)
  apply (solves fastforce)
  apply (solves fastforce)
  apply (solves <clarsimp simp add: task1-1.labelAtOut-def task1-1.labelAtIn-def>)
  apply (solves clarsimp)
apply (solves clarsimp)
done

```

8.3.2 Task 2.11

This is a slightly more interesting task as it involves both our connectives: $P \wedge Q \longrightarrow R \implies P \longrightarrow Q \longrightarrow R$

abbreviation $B :: (\text{string}, \text{prop-funs}) \text{ pform}$
where $B \equiv \text{Fun } (\text{Const } "B") []$
abbreviation $C :: (\text{string}, \text{prop-funs}) \text{ pform}$
where $C \equiv \text{Fun } (\text{Const } "C") []$

interpretation *task2-11: Abstract-Task*
 $\text{curry } (\text{SOME } f. \text{bij } f) :: \text{nat} \Rightarrow \text{string} \Rightarrow \text{string}$
 $\lambda-. \text{id}$
 $\lambda-. \{\}$
 $\text{closed} :: (\text{string}, \text{prop-funs}) \text{ pform} \Rightarrow \text{bool}$
 subst
 $\lambda-. \{\}$
 $\lambda-. \text{id}$
 Var undefined
 antecedent
 consequent
 prop-rules
 $[\text{Fun imp } [\text{Fun and } [A, B], C]]$
 $[\text{Fun imp } [A, \text{Fun imp } [B, C]]]$
by *unfold-locales simp-all*

abbreviation $n\text{-andI} \equiv \text{task2-11.n-rules} !! 0$
abbreviation $n\text{-andE1} \equiv \text{task2-11.n-rules} !! 1$
abbreviation $n\text{-andE2} \equiv \text{task2-11.n-rules} !! 2$
abbreviation $n\text{-impI} \equiv \text{task2-11.n-rules} !! 3$
abbreviation $n\text{-impE} \equiv \text{task2-11.n-rules} !! 4$

lemma $n\text{-andI}$ [*simp*]: $n\text{-andI} = (\text{andI}, \text{Fun and } [X, Y])$
unfolding *task2-11.n-rules-def* **by** (*simp add: prop-rules-def*)
lemma $n\text{-andE1}$ [*simp*]: $n\text{-andE1} = (\text{andE}, X)$
unfolding *task2-11.n-rules-def One-nat-def* **by** (*simp add: prop-rules-def*)
lemma $n\text{-andE2}$ [*simp*]: $n\text{-andE2} = (\text{andE}, Y)$
unfolding *task2-11.n-rules-def numeral-2-eq-2* **by** (*simp add: prop-rules-def*)
lemma $n\text{-impI}$ [*simp*]: $n\text{-impI} = (\text{impI}, \text{Fun imp } [X, Y])$
unfolding *task2-11.n-rules-def numeral-3-eq-3* **by** (*simp add: prop-rules-def*)
lemma $n\text{-impE}$ [*simp*]: $n\text{-impE} = (\text{impE}, Y)$
proof –
have $n\text{-impE} = \text{task2-11.n-rules} !! \text{Suc } 3$ **by** *simp*
also have $\dots = (\text{impE}, Y)$
unfolding *task2-11.n-rules-def numeral-3-eq-3* **by** (*simp add: prop-rules-def*)
finally show *?thesis* .
qed

lemma *subst-Var-eq-id* [*simp*]: $\text{subst Var} = \text{id}$
by (*rule ext*) (*induct-tac x; auto simp: map-idI*)

lemma *xy-update*: $f = \text{undefined}("X" := x, "Y" := y) \Longrightarrow x = f "X" \wedge y = f "Y"$ **by** *force*
lemma *y-update*: $f = \text{undefined}("Y" := y) \Longrightarrow y = f "Y"$ **by** *force*

declare *snth.simps(1)* [*simp del*]

By interpreting *Solved-Task* we show that there is a proof tree for the task. We get the existence of the proof graph for free by using the completeness theorem.

interpretation *task2-11: Solved-Task*
 $\text{curry } (\text{SOME } f. \text{bij } f) :: \text{nat} \Rightarrow \text{string} \Rightarrow \text{string}$
 $\lambda-. \text{id}$


```

λ-. {}
closed :: (string, prop-funs) pform ⇒ bool
subst
λ-. {}
λ-. id
Var undefined
antecedent
consequent
prop-rules
[Fun imp [Fun and [A,B],C]]
[Fun imp [A, Fun imp [B,C]]]
apply unfold-locales
  unfolding task2-11.solved-def
apply clarsimp
apply (rule-tac x={|Fun imp [Fun and [A,B],C]|} in exI)
apply clarsimp
— The actual proof tree for this task.
apply (rule-tac x=Node ({|Fun imp [Fun and [A, B], C]|} ⊢ Fun imp [A, Fun imp [B, C]], NatRule n-impI)
  {|Node ({|Fun imp [Fun and [A, B], C], A|} ⊢ Fun imp [B,C], NatRule n-impI)
    {|Node ({|Fun imp [Fun and [A, B], C], A, B|} ⊢ C, NatRule n-impE)
      {|Node ({|Fun imp [Fun and [A, B], C], A, B|} ⊢ Fun imp [Fun and [A,B], C], Axiom) {|}|,
        Node ({|Fun imp [Fun and [A, B], C], A, B|} ⊢ Fun and [A,B], NatRule n-andI)
          {|Node ({|Fun imp [Fun and [A, B], C], A, B|} ⊢ A, Axiom) {|}|,
            Node ({|Fun imp [Fun and [A, B], C], A, B|} ⊢ B, Axiom) {|}|
          }|}
    }|}
  }|}
  }|}
  }|} in exI)
apply clarsimp
apply (rule conjI)

apply (rule task1-1.wf)
  apply (solves ⟨clarsimp; metis n-impI snth-smap snth-sset⟩)
apply clarsimp
apply (rule task1-1.eff-NatRuleI [unfolded propositional.freshen-def, simplified]) apply simp-all[4]
  apply (rule task2-11.natEff-InstI)
    apply (solves simp)
    apply (solves simp)
    apply (solves simp)
  apply (intro conjI; simp; rule xy-update)
  apply (solves simp)
apply (solves ⟨fastforce simp: propositional.f-antecedent-def⟩)

apply (rule task1-1.wf)
  apply (solves ⟨clarsimp; metis n-impI snth-smap snth-sset⟩)
apply clarsimp
apply (rule task1-1.eff-NatRuleI [unfolded propositional.freshen-def, simplified]) apply simp-all[4]
  apply (rule task2-11.natEff-InstI)
    apply (solves simp)
    apply (solves simp)
    apply (solves simp)
  apply (intro conjI; simp; rule xy-update)
  apply (solves simp)
apply (solves ⟨fastforce simp: propositional.f-antecedent-def⟩)

apply (rule task1-1.wf)
  apply (solves ⟨clarsimp; metis n-impE snth-smap snth-sset⟩)

```

```

apply clarsimp
apply (rule task1-1.eff-NatRuleI [unfolded propositional.freshen-def, simplified, where s=undefined("Y":=C,"X":=Fun
and [A,B])]) apply simp-all[4]
  apply (rule task2-11.natEff-InstI)
    apply (solves simp)
    apply (solves simp)
    apply (solves simp)
  apply (solves <intro conjI; simp>)
apply (solves <simp add: propositional.f-antecedent-def>)
apply (erule disjE)

```

```

apply (auto intro: task1-1.wf intro!: task1-1.eff.intros(1))[1]

```

```

apply (rule task1-1.wf)
  apply (solves <clarsimp; metis n-andI snth-smap snth-sset>)
apply clarsimp
apply (rule task1-1.eff-NatRuleI [unfolded propositional.freshen-def, simplified]) apply simp-all[4]
  apply (rule task2-11.natEff-InstI)
    apply (solves simp)
    apply (solves simp)
    apply (solves simp)
  apply (intro conjI; simp; rule xy-update)
  apply (solves simp)
apply (solves <simp add: propositional.f-antecedent-def>)
apply clarsimp

```

```

apply (erule disjE)
apply (solves <rule task1-1.wf; auto intro: task1-1.eff.intros(1)>)
apply (solves <rule task1-1.wf; auto intro: task1-1.eff.intros(1)>)

```

```

by (rule tfinite.intros; auto)+

```

interpretation *Tasked-Proof-Graph*

```

curry (SOME f. bij f):: nat  $\Rightarrow$  string  $\Rightarrow$  string

```

```

λ-. id

```

```

λ-. {}

```

```

closed :: (string, prop-funs) pform  $\Rightarrow$  bool

```

```

subst

```

```

λ-. {}

```

```

λ-. id

```

```

Var undefined

```

```

antecedent

```

```

consequent

```

```

prop-rules

```

```

[Fun imp [Fun and [A,B],C]]

```

```

[Fun imp [A, Fun imp [B,C]]]

```

```

task2-11.vertices

```

```

task2-11.nodeOf

```

```

task2-11.edges

```

```

task2-11.vidx

```

```

task2-11.inst

```

```

by unfold-locales

```

```

end

```

```

end

```

8.4 Predicate_Formulas

theory *Predicate-Formulas*

imports

HOL-Library.Countable
HOL-Library.Infinite-Set
HOL-Eisbach.Eisbach
Abstract-Formula

begin

This theory contains an example instantiation of *Abstract-Formulas* with an formula type with local constants. It is a rather ad-hoc type that may not be very useful to work with, though.

type-synonym *var* = *nat*

type-synonym *lconst* = *nat*

We support higher order variables, in order to express $\forall x. ?P x$. But we stay first order, i.e. the parameters of such a variables will only be instantiated with ground terms.

datatype *form* =

Var (*var:var*) (*params: form list*)
| *LC* (*var:lconst*)
| *Op* (*name:string*) (*params: form list*)
| *Quant* (*name:string*) (*var:nat*) (*body: form*)

type-synonym *schema* = *var list* \times *form*

type-synonym *subst* = (*nat* \times *schema*) *list*

fun *fv* :: *form* \Rightarrow *var set* **where**

fv (*Var v xs*) = *insert v (Union (fv ' set xs))*
| *fv* (*LC v*) = *{}*
| *fv* (*Op n xs*) = *Union (fv ' set xs)*
| *fv* (*Quant n v f*) = *fv f - {v}*

definition *fresh-for* :: *var set* \Rightarrow *var* **where**

fresh-for V = (*SOME n. n* \notin *V*)

lemma *fresh-for-fresh*: *finite V* \Longrightarrow *fresh-for V* \notin *V*

unfolding *fresh-for-def*

apply (*rule someI2-ex*)

using *infinite-nat-iff-unbounded-le*

apply *auto*

done

Free variables

fun *fv-schema* :: *schema* \Rightarrow *var set* **where**

fv-schema (ps,f) = *fv f - set ps*

definition *fv-subst* :: *subst* \Rightarrow *var set* **where**

fv-subst s = \bigcup (*fv-schema ' ran (map-of s)*)

definition *fv-subst1* **where**

fv-subst1 s = \bigcup (*fv ' snd ' set s*)

lemma *fv-subst-Nil[simp]*: *fv-subst1 []* = *{}*

unfolding *fv-subst1-def* **by** *auto*

Local constants, separate from free variables.

```
fun lc :: form  $\Rightarrow$  lconst set where
  lc (Var v xs) = Union (lc ' set xs)
| lc (LC c) = {c}
| lc (Op n xs) = Union (lc ' set xs)
| lc (Quant n v f) = lc f
```

```
fun lc-schema :: schema  $\Rightarrow$  lconst set where
  lc-schema (ps,f) = lc f
```

```
definition lc-subst1 where
  lc-subst1 s =  $\bigcup$  (lc ' snd ' set s)
```

```
fun lc-subst :: subst  $\Rightarrow$  lconst set where
  lc-subst s =  $\bigcup$  (lc-schema ' snd ' set s)
```

```
fun map-lc :: (lconst  $\Rightarrow$  lconst)  $\Rightarrow$  form  $\Rightarrow$  form where
  map-lc f (Var v xs) = Var v (map (map-lc f) xs)
| map-lc f (LC n) = LC (f n)
| map-lc f (Op n xs) = Op n (map (map-lc f) xs)
| map-lc f (Quant n v f') = Quant n v (map-lc f f')
```

```
lemma fv-map-lc[simp]: fv (map-lc p f) = fv f
by (induction f) auto
```

```
lemma lc-map-lc[simp]: lc (map-lc p f) = p ' lc f
by (induction f) auto
```

```
lemma map-lc-map-lc[simp]: map-lc p1 (map-lc p2 f) = map-lc (p1  $\circ$  p2) f
by (induction f) auto
```

```
fun map-lc-subst1 :: (lconst  $\Rightarrow$  lconst)  $\Rightarrow$  (var  $\times$  form) list  $\Rightarrow$  (var  $\times$  form) list where
  map-lc-subst1 f s = map (apsnd (map-lc f)) s
```

```
fun map-lc-subst :: (lconst  $\Rightarrow$  lconst)  $\Rightarrow$  subst  $\Rightarrow$  subst where
  map-lc-subst f s = map (apsnd (apsnd (map-lc f))) s
```

```
lemma map-lc-noop[simp]: lc f = {}  $\implies$  map-lc p f = f
by (induction f) (auto simp add: map-idI)
```

```
lemma map-lc-cong[cong]: ( $\bigwedge x. x \in \text{lc } f \implies f1\ x = f2\ x$ )  $\implies$  map-lc f1 f = map-lc f2 f
by (induction f) auto
```

```
lemma [simp]: fv-subst1 (map (apsnd (map-lc p)) s) = fv-subst1 s
unfolding fv-subst1-def
by auto
```

```
lemma map-lc-subst-cong[cong]:
  assumes ( $\bigwedge x. x \in \text{lc-subst } s \implies f1\ x = f2\ x$ )
  shows map-lc-subst f1 s = map-lc-subst f2 s
by (force intro!: map-lc-cong assms)
```

In order to make the termination checker happy, we define substitution in two stages: One that substitutes only ground terms for variables, and the real one that can substitute schematic terms (or lambda expression, if you want).

```
fun subst1 :: (var  $\times$  form) list  $\Rightarrow$  form  $\Rightarrow$  form where
```

```

  subst1 s (Var v []) = (case map-of s v of Some f => f | None => Var v [])
| subst1 s (Var v xs) = Var v xs
| subst1 s (LC n) = LC n
| subst1 s (Op n xs) = Op n (map (subst1 s) xs)
| subst1 s (Quant n v f) =
  (if v ∈ fv-subst1 s then
   (let v' = fresh-for (fv-subst1 s)
    in Quant n v' (subst1 ((v, Var v' [])#s) f))
  else Quant n v (subst1 s f))

```

lemma *subst1-Nil[simp]*: $subst1 [] f = f$
by (*induction* []::(var × form) list *f* *rule*:subst1.induct)
 (*auto simp add*: map-idI split: option.splits)

lemma *lc-subst1*: $lc (subst1 s f) \subseteq lc f \cup \bigcup (lc \text{ ‘ } snd \text{ ‘ } set s)$
by (*induction* s f *rule*: subst1.induct)
 (*auto split*: option.split *dest*: map-of-SomeD *simp add*: Let-def)

lemma *apsnd-def'*: $apsnd f = (\lambda(k, v). (k, f v))$
by *auto*

lemma *map-of-map-apsnd*:
 $map-of (map (apsnd f) xs) = map-option f \circ map-of xs$
unfolding *apsnd-def'* **by** (*rule* map-of-map)

lemma *map-lc-subst1[simp]*: $map-lc p (subst1 s f) = subst1 (map-lc-subst1 p s) (map-lc p f)$
apply (*induction* s f *rule*: subst1.induct)
apply (*auto split*: option.splits *simp add*: map-of-map-apsnd Let-def)
apply (subst subst1.simps, *auto split*: option.splits)[1]
apply (subst subst1.simps, *auto split*: option.splits)[1]
apply (subst subst1.simps, *auto split*: option.splits)[1]
apply (subst subst1.simps, *auto split*: option.splits, *simp only*: Let-def map-lc.simps)[1]
apply (subst subst1.simps, *auto split*: option.splits)
done

fun *subst'* :: subst => form => form **where**
subst' s (Var v xs) =
 (case map-of s v of None => (Var v (map (subst' s) xs))
 | Some (ps,rhs) =>
 if length ps = length xs
 then subst1 (zip ps (map (subst' s) xs)) rhs
 else (Var v (map (subst' s) xs)))
| *subst'* s (LC n) = LC n
| *subst'* s (Op n xs) = Op n (map (subst' s) xs)
| *subst'* s (Quant n v f) =
 (if v ∈ fv-subst s then
 (let v' = fresh-for (fv-subst s)
 in Quant n v' (subst' ((v,([], Var v' []))#s) f))
 else Quant n v (subst' s f))

lemma *subst'-Nil[simp]*: $subst' [] f = f$
by (*induction* f) (*auto simp add*: map-idI fv-subst-def)

lemma *lc-subst'*: $lc (subst' s f) \subseteq lc f \cup lc-subst s$
apply (*induction* s f *rule*: subst'.induct)

```

apply (auto split: option.splits dest: map-of-SomeD dest!: subsetD[OF lc-subst1] simp add: fv-subst-def)
apply (fastforce dest!: set-zip-rightD)+
done

```

```

lemma ran-map-option-comp[simp]:
  ran (map-option f ◦ m) = f ‘ ran m
unfolding comp-def by (rule ran-map-option)

```

```

lemma fv-schema-apsnd-map-lc[simp]:
  fv-schema (apsnd (map-lc p) a) = fv-schema a
by (cases a) auto

```

```

lemma fv-subst-map-apsnd-map-lc[simp]:
  fv-subst (map (apsnd (apsnd (map-lc p))) s) = fv-subst s
unfolding fv-subst-def
by (auto simp add: map-of-map-apsnd)

```

```

lemma map-apsnd-zip[simp]: map (apsnd f) (zip a b) = zip a (map f b)
by (simp add: apsnd-def' zip-map2)

```

```

lemma map-lc-subst'[simp]: map-lc p (subst' s f) = subst' (map-lc-subst p s) (map-lc p f)
apply (induction s f rule: subst'.induct)
  apply (auto split: option.splits dest: map-of-SomeD simp add: map-of-map-apsnd Let-def)
  apply (solves <(subst subst'.simps, auto split: option.splits)[1]>)
  apply (solves <(subst subst'.simps, auto split: option.splits cong: map-cong)[1]>)
  apply (solves <(subst subst'.simps, auto split: option.splits)[1]>)
  apply (solves <(subst subst'.simps, auto split: option.splits)[1]>)
  apply (solves <(subst subst'.simps, auto split: option.splits, simp only: Let-def map-lc.simps)[1]>)
  apply (solves <(subst subst'.simps, auto split: option.splits)[1]>)
done

```

Since `subst'` happily renames quantified variables, we have a simple wrapper that ensures that the substitution is minimal, and is empty if f is closed. This is a hack to support lemma `subst-noop`.

```

fun subst :: subst ⇒ form ⇒ form where
  subst s f = subst' (filter (λ (v,s). v ∈ fv f) s) f

```

```

lemma subst-Nil[simp]: subst [] f = f
by auto

```

```

lemma subst-noop[simp]: fv f = {} ⇒ subst s f = f
by simp

```

```

lemma lc-subst: lc (subst s f) ⊆ lc f ∪ lc-subst s
by (auto dest: subsetD[OF lc-subst'])

```

```

lemma lc-subst-map-lc-subst[simp]: lc-subst (map-lc-subst p s) = p ‘ lc-subst s
by force

```

```

lemma map-lc-subst[simp]: map-lc p (subst s f) = subst (map-lc-subst p s) (map-lc p f)
unfolding subst.simps
by (auto simp add: filter-map intro!: arg-cong[OF filter-cong] )

```

```

fun closed :: form ⇒ bool where
  closed f ⇔ fv f = {} ∧ lc f = {}

```

interpretation predicate: *Abstract-Formulas*

```

curry to-nat :: nat => var => var
map-lc
lc
closed
subst
lc-subst
map-lc-subst
Var 0 []
apply unfold-locales
apply (solves fastforce)
apply (solves fastforce)
apply (solves fastforce)
apply (solves fastforce)
apply (solves fastforce)
apply (solves <rule lc-subst>)
apply (solves fastforce)
apply (solves fastforce)
apply (solves fastforce)
apply (solves <metis map-lc-subst-cong>)
apply (solves <rule lc-subst-map-lc-subst>)
apply (solves simp)
apply (solves <rule exI[where x = [], simp]>)
apply (solves <rename-tac f, rule-tac x = [(0, ([,f]))] in exI, simp>)
done

```

```
declare predicate.subst-lconsts-empty-subst [simp del]
```

```
end
```

8.5 Incredible_Predicate

```
theory Incredible-Predicate imports
```

```
  Abstract-Rules-To-Incredible
```

```
  Predicate-Formulas
```

```
begin
```

Our example interpretation with predicate logic will cover implication and the universal quantifier.

The rules are introduction and elimination of implication and universal quantifiers.

```
datatype prop-rule = allI | allE | impI | impE
```

```
definition prop-rules :: prop-rule stream
  where prop-rules = cycle [allI, allE, impI, impE]
```

```
lemma iR-prop-rules [simp]: sset prop-rules = {allI, allE, impI, impE}
  unfolding prop-rules-def by simp
```

Just some short notation.

```
abbreviation X :: form
  where X ≡ Var 10 []
abbreviation Y :: form
  where Y ≡ Var 11 []
abbreviation x :: form
  where x ≡ Var 9 []
abbreviation t :: form
```

```

where  $t \equiv \text{Var } 13 \ []$ 
abbreviation  $P :: \text{form} \Rightarrow \text{form}$ 
where  $P f \equiv \text{Var } 12 [f]$ 
abbreviation  $Q :: \text{form} \Rightarrow \text{form}$ 
where  $Q f \equiv \text{Op } "Q" [f]$ 
abbreviation  $\text{imp} :: \text{form} \Rightarrow \text{form} \Rightarrow \text{form}$ 
where  $\text{imp } f1 f2 \equiv \text{Op } "imp" [f1, f2]$ 
abbreviation  $\text{ForallX} :: \text{form} \Rightarrow \text{form}$ 
where  $\text{ForallX } f \equiv \text{Quant } "all" 9 f$ 

```

Finally the right- and left-hand sides of the rules.

```

fun consequent :: prop-rule  $\Rightarrow$  form list
where consequent allI = [ForallX (P x)]
| consequent allE = [P t]
| consequent impI = [imp X Y]
| consequent impE = [Y]

```

```

abbreviation allI-input where allI-input  $\equiv \text{Antecedent } \{\{\}\} (P (LC 0)) \{0\}$ 
abbreviation impI-input where impI-input  $\equiv \text{Antecedent } \{|X|\} Y \{\}$ 

```

```

fun antecedent :: prop-rule  $\Rightarrow$  (form, lconst) antecedent list
where antecedent allI = [allI-input]
| antecedent allE = [plain-ant (ForallX (P x))]
| antecedent impI = [impI-input]
| antecedent impE = [plain-ant (imp X Y), plain-ant X]

```

interpretation *predicate: Abstract-Rules*

```

curry to-nat :: nat  $\Rightarrow$  var  $\Rightarrow$  var
map-lc
lc
closed
subst
lc-subst
map-lc-subst
Var 0 []
antecedent
consequent
prop-rules

```

proof

```

show  $\forall xs \in \text{sset } \text{prop-rules}. \text{consequent } xs \neq []$ 
unfolding prop-rules-def
using consequent.elims by blast

```

next

```

show  $\forall xs \in \text{sset } \text{prop-rules}. \bigcup (lc \text{ ' set } (\text{consequent } xs)) = \{\}$ 
by auto

```

next

```

fix  $i' r i ia$ 
assume  $r \in \text{sset } \text{prop-rules}$ 
and  $ia < \text{length } (\text{antecedent } r)$ 
and  $i' < \text{length } (\text{antecedent } r)$ 
then show  $a\text{-fresh } (\text{antecedent } r ! ia) \cap a\text{-fresh } (\text{antecedent } r ! i') = \{\} \vee ia = i'$ 
by (cases i'; auto)

```

next

```

fix  $r p$ 
assume  $r \in \text{sset } \text{prop-rules}$ 
and  $p \in \text{set } (\text{antecedent } r)$ 
thus  $lc (a\text{-conc } p) \cup \bigcup (lc \text{ ' fset } (a\text{-hyps } p)) \subseteq a\text{-fresh } p$  by auto

```


qed

end

8.6 Incredible_Predicate_Tasks

theory *Incredible-Predicate-Tasks*

imports

Incredible-Completeness

Incredible-Predicate

HOL-Eisbach.Eisbach

begin

declare *One-nat-def* [*simp del*]

context *ND-Rules-Inst* **begin**

lemma *eff-NatRuleI*:

nat-rule rule c ants

$\implies \text{entail} = (\Gamma \vdash \text{subst } s \text{ (freshen } a \text{ } c))$

$\implies \text{hyps} = ((\lambda \text{ant. } ((\lambda p. \text{subst } s \text{ (freshen } a \text{ } p)) \mid^{\dagger} a\text{-hyps } \text{ant} \mid \cup \mid \Gamma \vdash \text{subst } s \text{ (freshen } a \text{ (} a\text{-conc } \text{ant})))) \mid^{\dagger}$
ants)

$\implies (\bigwedge \text{ant } f. \text{ant} \mid \in \mid \text{ants} \implies f \mid \in \mid \Gamma \implies \text{freshenLC } a \text{ ' (} a\text{-fresh } \text{ant} \text{) } \cap \text{lconsts } f = \{\})$

$\implies (\bigwedge \text{ant. } \text{ant} \mid \in \mid \text{ants} \implies \text{freshenLC } a \text{ ' (} a\text{-fresh } \text{ant} \text{) } \cap \text{subst-lconsts } s = \{\})$

$\implies \text{eff (NatRule rule) entail hyps}$

by (*drule eff.intros(2)*) *simp-all*

end

context *Abstract-Task* **begin**

lemma *natEff-InstI*:

rule = (r,c)

$\implies c \in \text{set (consequent } r)$

$\implies \text{antec} = f\text{-antecedent } r$

$\implies \text{natEff-Inst rule } c \text{ antec}$

by (*metis natEff-Inst.intros*)

end

context **begin**

A typical task with local constants:: $\forall x. Q x \longrightarrow Q x$

First the task is defined as an *Abstract-Task*.

interpretation *task: Abstract-Task*

curry to-nat :: nat \Rightarrow var \Rightarrow var

map-lc

lc

closed

subst

lc-subst

map-lc-subst

Var 0 []

antecedent

consequent

prop-rules

[]

[ForallX (imp (Q x) (Q x))]

by *unfold-locales auto*

Then we show, that this task has a proof within our formalization of natural deduction by giving a concrete proof tree.

abbreviation $lx :: nat$ **where** $lx \equiv to\text{-}nat (1::nat, 0::nat)$

abbreviation $base\text{-}tree :: ((form\ fset \times form) \times (prop\text{-}rule \times form)\ NatRule)\ tree$ **where**
 $base\text{-}tree \equiv Node (\{|Q (LC\ lx)|\} \vdash Q (LC\ lx), Axiom)\ \{\|\}$

abbreviation $imp\text{-}tree :: ((form\ fset \times form) \times (prop\text{-}rule \times form)\ NatRule)\ tree$ **where**
 $imp\text{-}tree \equiv Node (\{\|\} \vdash imp (Q (LC\ lx)) (Q (LC\ lx)), NatRule (impI, imp\ X\ Y))\ \{|base\text{-}tree|\}$

abbreviation $solution\text{-}tree :: ((form\ fset \times form) \times (prop\text{-}rule \times form)\ NatRule)\ tree$ **where**
 $solution\text{-}tree \equiv Node (\{\|\} \vdash ForallX (imp (Q\ x) (Q\ x)), NatRule (allI, ForallX (P\ x)))\ \{|imp\text{-}tree|\}$

abbreviation $s1$ **where** $s1 \equiv [(12, ([9], imp (Q\ x) (Q\ x)))]$

abbreviation $s2$ **where** $s2 \equiv [(10, ([], Q (LC\ lx))), (11, ([], Q (LC\ lx)))]$

lemma $fv\text{-}subst\text{-}s1[simp]: fv\text{-}subst\ s1 = \{\}$
by ($simp\ add: fv\text{-}subst\text{-}def$)

lemma $subst1\text{-}simps[simp]:$
 $subst\ s1 (P (LC\ n)) = imp (Q (LC\ n)) (Q (LC\ n))$
 $subst\ s1 (ForallX (P\ x)) = ForallX (imp (Q\ x) (Q\ x))$
by $simp\text{-}all$

lemma $subst2\text{-}simps[simp]:$
 $subst\ s2\ X = Q (LC\ lx)$
 $subst\ s2\ Y = Q (LC\ lx)$
 $subst\ s2 (imp\ X\ Y) = imp (subst\ s2\ X) (subst\ s2\ Y)$
by $simp\text{-}all$

lemma $substI1: ForallX (imp (Q\ x) (Q\ x)) = subst\ s1 (predicate.freshen\ 1 (ForallX (P\ x)))$
by ($auto\ simp\ add: predicate.freshen\text{-}def\ Let\text{-}def$)

lemma $substI2: imp (Q (LC\ lx)) (Q (LC\ lx)) = subst\ s2 (predicate.freshen\ 2 (imp\ X\ Y))$
by ($auto\ simp\ add: predicate.freshen\text{-}def\ Let\text{-}def$)

declare $subst.simps[simp\ del]$

lemma $task.solved$
unfolding $task.solved\text{-}def$
apply $clarsimp$
apply ($rule\text{-}tac\ x=\{\|\}$ **in** exI)
apply $clarsimp$
apply ($rule\text{-}tac\ x=solution\text{-}tree$ **in** exI)
apply $clarsimp$
apply ($rule\ conjI$)

apply ($rule\ task.wf$)
apply ($solves\ \langle auto\ simp\ add: stream.set\text{-}map\ task.n\text{-}rules\text{-}def \rangle [1]$)
apply $clarsimp$
apply ($rule\ task.eff\text{-}NatRuleI$)
apply ($solves\ \langle rule\ task.natEff\text{-}Inst.intros; simp \rangle$)
apply $clarsimp$
apply ($rule\ conjI$)
apply ($solves\ \langle simp \rangle$)
apply ($solves\ \langle rule\ substI1 \rangle$)
apply ($simp\ add: predicate.f\text{-}antecedent\text{-}def\ predicate.freshen\text{-}def$)

```

apply (subst antecedent.sel(2))
apply (solves ⟨simp⟩)
apply (solves ⟨simp⟩)
apply (solves ⟨simp⟩)
apply simp

apply (rule task.wf)
apply (solves ⟨(auto simp add: stream.set-map task.n-rules-def)[1]⟩)
apply clarsimp
apply (rule task.eff-NatRuleI)
  apply (solves ⟨rule task.natEff-Inst.intros; simp⟩)
  apply clarsimp
  apply (rule conjI)
  apply (solves ⟨simp⟩)
  apply (solves ⟨rule substI2⟩)
  apply (solves ⟨simp add: predicate.f-antecedent-def predicate.freshen-def⟩)
  apply (solves ⟨simp⟩)
apply (solves ⟨simp add: predicate.f-antecedent-def⟩)
apply simp

apply (solves ⟨(auto intro: task.wf intro!: task.eff.intros(1))[1]⟩)
apply (solves ⟨(rule tfinite.intros, simp)+⟩)
done

```

abbreviation vertices **where** vertices $\equiv \{|0::nat, 1, 2 \}$

fun nodeOf **where**
 nodeOf n = [Conclusion (ForallX (imp (Q x) (Q x))),
 Rule allI,
 Rule impI] ! n

fun inst **where**
 inst n = [|, s1, s2] ! n

interpretation task: Vertex-Graph task.nodes task.inPorts task.outPorts vertices nodeOf.

abbreviation e1 :: (nat, form, nat) edge'
where e1 $\equiv ((1, \text{Reg } (\text{ForallX } (P \ x))), (0, \text{plain-ant } (\text{ForallX } (\text{imp } (Q \ x) (Q \ x))))$

abbreviation e2 :: (nat, form, nat) edge'
where e2 $\equiv ((2, \text{Reg } (\text{imp } X \ Y)), (1, \text{allI-input}))$

abbreviation e3 :: (nat, form, nat) edge'
where e3 $\equiv ((2, \text{Hyp } X (\text{impI-input})), (2, \text{impI-input}))$

abbreviation task-edges :: (nat, form, nat) edge' set **where** task-edges $\equiv \{e1, e2, e3\}$

interpretation task: Scoped-Graph task.nodes task.inPorts task.outPorts vertices nodeOf task-edges task.hyps
by standard (auto simp add: predicate.f-consequent-def predicate.f-antecedent-def)

interpretation task: Instantiation

task.inPorts
 task.outPorts
 nodeOf
 task.hyps
 task.nodes
 task-edges
 vertices
 task.labelsIn

```

task.labelsOut
curry to-nat :: nat ⇒ var ⇒ var
map-lc
lc
closed
subst
lc-subst
map-lc-subst
Var 0 []
id
inst
by unfold-locales simp

```

Finally we can also show that there is a proof graph for this task.

interpretation *Well-Scoped-Graph*

```

task.nodes
task.inPorts
task.outPorts
vertices
nodeOf
task-edges
task.hyps
by standard (auto split: if-splits)

```

lemma *no-path-01*[simp]: $task.path\ 0\ v\ pth \longleftrightarrow (pth = [] \wedge v = 0)$

by (cases pth) (auto simp add: task.path-cons-simp)

lemma *no-path-12*[simp]: $\neg task.path\ 1\ 2\ pth$

by (cases pth) (auto simp add: task.path-cons-simp)

interpretation *Acyclic-Graph*

```

task.nodes
task.inPorts
task.outPorts
vertices
nodeOf
task-edges
task.hyps

```

proof

fix $v\ pth$

assume $task.path\ v\ v\ pth$ **and** $task.hyps-free\ pth$

thus $pth = []$

by (cases pth) (auto simp add: task.path-cons-simp predicate.f-antecedent-def)

qed

interpretation *Saturated-Graph*

```

task.nodes
task.inPorts
task.outPorts
vertices
nodeOf
task-edges

```

by standard

(auto simp add: predicate.f-consequent-def predicate.f-antecedent-def)

interpretation *Pruned-Port-Graph*

```

task.nodes
task.inPorts

```

```

task.outPorts
vertices
nodeOf
task-edges
proof
fix v
assume v |∈| vertices
hence ∃ pth. task.path v 0 pth
  apply auto
  apply (rule exI[where x = [e1]], auto simp add: task.path-cons-simp)
  apply (rule exI[where x = [e2,e1]], auto simp add: task.path-cons-simp)
  done
moreover
have task.terminal-vertex 0 by auto
ultimately
show ∃ pth v'. task.path v v' pth ∧ task.terminal-vertex v' by blast
qed

```

interpretation *Well-Shaped-Graph*

```

task.nodes
task.inPorts
task.outPorts
vertices
nodeOf task-edges
task.hyps
..

```

interpretation *Solution*

```

task.inPorts
task.outPorts
nodeOf
task.hyps
task.nodes
vertices
task.labelsIn
task.labelsOut
curry to-nat :: nat ⇒ var ⇒ var
map-lc
lc
closed
subst
lc-subst
map-lc-subst
Var 0 []
id
inst
task-edges
by standard
(auto simp add: task.labelAtOut-def task.labelAtIn-def predicate.freshen-def, subst antecedent.sel, simp)

```

interpretation *Proof-Graph*

```

task.nodes
task.inPorts
task.outPorts
vertices
nodeOf
task-edges

```

```

task.hyps
task.labelsIn
task.labelsOut
curry to-nat :: nat ⇒ var ⇒ var
map-lc
lc
closed
subst
lc-subst
map-lc-subst
Var 0 []
id
inst
..

```

```

lemma path-20:
  assumes task.path 2 0 pth
  shows (1, allI-input) ∈ snd ‘ set pth
proof–
  { fix v
    assume task.path v 0 pth
    hence v = 0 ∨ v = 1 ∨ (1, allI-input) ∈ snd ‘ set pth
    by (induction v 0::nat pth rule: task.path.induct) auto
  }
from this[OF assms]
show ?thesis by auto
qed

```

```

lemma scope-21: 2 ∈ task.scope (1, allI-input)
by (auto intro!: task.scope.intros elim: path-20 simp add: task.outPortsRule-def predicate.f-antecedent-def predicate.f-consequent-def)

```

interpretation *Scoped-Proof-Graph*

```

curry to-nat :: nat ⇒ var ⇒ var
map-lc
lc
closed
subst
lc-subst
map-lc-subst
Var 0 []
task.inPorts
task.outPorts
nodeOf
task.hyps
task.nodes
vertices
task.labelsIn
task.labelsOut
id
inst
task-edges
task.local-vars

```

by standard (auto simp add: predicate.f-antecedent-def scope-21)

interpretation *Tasked-Proof-Graph*

```

curry to-nat :: nat ⇒ var ⇒ var

```

```
map-lc
lc
closed
subst
lc-subst
map-lc-subst
Var 0 []
antecedent
consequent
prop-rules
[]
[ForallX (imp (Q x) (Q x))]
vertices
nodeOf
task-edges
id
inst
by unfold-locales auto

end

end
```