

The meta theory of the Incredible Proof Machine

Joachim Breitner Denis Lohner

October 11, 2017

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	5
2.3.1	The rose tree data type	5
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	20
5	Natural Deduction	23
5.1	Natural_Deduction	23
6	Correctness	25
6.1	Incredible_Correctness	25
7	Completeness	28
7.1	Incredible_Trees	28
7.2	Build_Incredible_Tree	34
7.3	Incredible_Completeness	35

8	Instantiations	41
8.1	Propositional_Formulas	41
8.2	Incredible_Propositional	42
8.3	Incredible_Propositional_Tasks	43
	8.3.1 Task 1.1	43
	8.3.2 Task 2.11	45
8.4	Predicate_Formulas	47
8.5	Incredible_Predicate	50
8.6	Incredible_Predicate_Tasks	52

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 <proof>
lemma mem-fset-from-list[simp]: x |∈| fset-from-list l ↔ x ∈ set l <proof>
lemma fimage-fset-from-list[simp]: f |'| fset-from-list l = fset-from-list (map f l) <proof>
lemma fset-fset-from-list[simp]: fset (fset-from-list l) = set l <proof>
lemmas fset-simps[simp] = set-simps[Transfer.transferred]
lemma size-fset-from-list[simp]: distinct l ⇒ size (fset-from-list l) = length l
  <proof>

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)
  <proof>

lemma length-list-of-fset[simp]: length (list-of-fset s) = size s
  <proof>

lemma nth-list-of-fset-mem[simp]: i < size s ⇒ list-of-fset s ! i |∈| s
  <proof>

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>
```

lemma *fmember-is-indexed-fmember*:

assumes $x \in s$
shows $\exists i. x \in_i s$

<proof>

lemma *indexed-fmember-unique*: $x \in_i s \implies y \in_j s \implies x = y \iff i = j$

<proof>

definition *indexed-members* :: $'a \text{ fset} \Rightarrow (\text{nat} \times 'a) \text{ list}$ **where**
indexed-members $s = \text{zip } [0..<\text{size } s] (\text{list-of-fset } s)$

lemma *mem-set-indexed-members*:

$(i,x) \in \text{set } (\text{indexed-members } s) \iff x \in_i s$

<proof>

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

$t \in \text{set } (\text{indexed-members } s) \iff \text{snd } t \in_{fst\ t} s$

<proof>

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

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

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

<proof>

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

<proof>

end

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

fdx $s\ x = (\text{SOME } i. x \in_i s)$

lemma *fdx-eq[simp]*: $x \in_i s \implies \text{fdx } s\ x = i$

<proof>

lemma *fdx-inj[simp]*: $x \in s \implies y \in s \implies \text{fdx } s\ x = \text{fdx } s\ y \iff x = y$

<proof>

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

<proof>

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 $'a \text{ rose-tree} = \text{RNode } (\text{root}: 'a) (\text{children}: 'a \text{ rose-tree list})$

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*: $i < \text{length} (\text{children } t) \Longrightarrow \text{children } t ! i = t' \Longrightarrow \text{it-pathsP } t' \text{ is} \Longrightarrow \text{it-pathsP } t (i\#is)$

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 = ($\lambda x. x \in \text{it-paths } t$)

<proof>

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 []) = {[]}

<proof>

lemma *it-paths-Union*: *it-paths* t \subseteq insert [] (Union ((($\lambda (i,t). (\text{op} \# i) \text{ 'it-paths } t$) ' set (List.enumerate (0::nat) (children t))))))

<proof>

lemma *finite-it-paths*[*simp*]: finite (*it-paths* t)

<proof>

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))

<proof>

lemma *it-paths-strict-prefix*:

assumes is \in *it-paths* t

assumes *strict-prefix is' is*

shows is' \in *it-paths* t

<proof>

lemma *it-paths-prefix*:

assumes is \in *it-paths* t

assumes *prefix is' is*

shows $is' \in it\text{-paths } t$
<proof>

lemma *it-paths-butlast*:
assumes $is \in it\text{-paths } t$
shows $butlast\ is \in it\text{-paths } t$
<proof>

lemma *it-path-SnocI*:
assumes $is \in it\text{-paths } t$
assumes $i < length\ (children\ (tree\text{-at } t\ is))$
shows $is\ @\ [i] \in it\text{-paths } t$
<proof>

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 *lconsts* *f* \implies *f1* *x* = *f2* *x*) \implies *renameLCs* *f1 f* = *renameLCs* *f2 f*

assumes *subst-renameLCs-cong*: (\bigwedge *x*. *x* \in *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*. *subst* *s f* = *f*) \wedge *subst-lconsts* *s* = {}

assumes *anyP-is-any*: \exists *s*. *subst* *s* *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*:

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

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma *closed-eq*:

assumes *closed f1*

assumes *closed f2*

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

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

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma *rename-noop*:

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

$\langle proof \rangle$

lemma *rename-rename-noop*:

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

$\langle proof \rangle$

lemma *rename-subst-noop*:

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

$\langle proof \rangle$

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

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

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

begin
definition *ass-forms* **where** *ass-forms* = *fset-from-list* *assumptions*
definition *conc-forms* **where** *conc-forms* = *fset-from-list* *conclusions*

lemma *mem-ass-forms[simp]*: $a \in | \text{ass-forms} \iff a \in \text{set } assumptions$
 $\langle \text{proof} \rangle$

lemma *mem-conc-forms[simp]*: $a \in | \text{conc-forms} \iff a \in \text{set } conclusions$
 $\langle \text{proof} \rangle$

lemma *subst-freshen-assumptions[simp]*:
assumes $pf \in \text{set } assumptions$
shows $\text{subst } s\ (\text{freshen } a\ pf) = pf$
 $\langle \text{proof} \rangle$

lemma *subst-freshen-conclusions[simp]*:
assumes $pf \in \text{set } conclusions$
shows $\text{subst } s\ (\text{freshen } a\ pf) = pf$
 $\langle \text{proof} \rangle$

lemma *subst-freshen-in-ass-formsI*:
assumes $pf \in \text{set } assumptions$
shows $\text{subst } s\ (\text{freshen } a\ pf) \in | \text{ass-forms}$
 $\langle \text{proof} \rangle$

```
lemma subst-freshen-in-conc-formsI:  
  assumes pf ∈ set conclusions  
  shows subst s (freshen a pf) |∈| conc-forms  
    <proof>  
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)
     $\langle$ proof $\rangle$ 

  context includes fset.lifting
  begin
  lift-definition hyps-for :: 'node  $\Rightarrow$  'inPort  $\Rightarrow$  'outPort fset is hyps-for'
     $\langle$ proof $\rangle$ 
  lemma hyps-for-simp[simp]: h  $\in$  hyps-for n p  $\longleftrightarrow$  hyps n h = Some p
     $\langle$ proof $\rangle$ 
  lemma hyps-for-simp'[simp]: h  $\in$  fset (hyps-for n p)  $\longleftrightarrow$  hyps n h = Some p
     $\langle$ proof $\rangle$ 
  lemma hyps-for-collect: fset (hyps-for n p) = {h . hyps n h = Some p}
     $\langle$ proof $\rangle$ 
  end
  lemma hyps-for-subset: hyps-for n p  $\subseteq$  outPorts n
     $\langle$ proof $\rangle$ 
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) ⟨proof⟩
lemma edge-end-tup: edge-end x = fst (snd x) ⟨proof⟩

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
  ⟨proof⟩

lemma path-appendI: path v v' pth1 ⇒ path v' v'' pth2 ⇒ path v v'' (pth1@pth2)
  ⟨proof⟩

lemma path-split: path v v' (pth1@[e]@pth2) ⟷ path v (edge-end e) (pth1@[e]) ∧ path (edge-end e) v'
pth2
  ⟨proof⟩

lemma path-split2: path v v' (pth1@(e#pth2)) ⟷ path v (edge-begin e) pth1 ∧ path (edge-begin e) v'
(e#pth2)
  ⟨proof⟩

lemma path-snoc: path v v' (pth1@[e]) ⟷ e ∈ edges ∧ path v (edge-begin e) pth1 ∧ edge-end e = v'
  ⟨proof⟩

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
  ⟨proof⟩

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
  ⟨proof⟩

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  $ps \notin snd\ 'set\ pth1$ 
 $\langle proof \rangle$ 
end

```

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

```

locale Port-Graph = Pre-Port-Graph +
assumes valid-nodes:  $nodeOf\ 'fset\ vertices \subseteq sset\ nodes$ 
assumes valid-edges:  $\forall (ps1, ps2) \in edges.\ valid-out-port\ ps1 \wedge valid-in-port\ ps2$ 
begin
lemma snd-set-path-verties:  $path\ v\ v'\ pth \implies fst\ 'snd\ 'set\ pth \subseteq fset\ vertices$ 
 $\langle proof \rangle$ 

lemma fst-set-path-verties:  $path\ v\ v'\ pth \implies fst\ 'fst\ 'set\ pth \subseteq fset\ vertices$ 
 $\langle proof \rangle$ 
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:  $\bigwedge v.\ v \in |vertices| \implies (\exists pth\ v'.\ path\ v\ v'\ pth \wedge terminal-vertex\ v')$ 
begin
lemma scopes-not-refl:
assumes  $v \in |vertices|$ 
shows  $v \notin scope\ (v, p)$ 
 $\langle proof \rangle$ 

```

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

```

lemma scopes-nest:
fixes  $ps1\ ps2$ 
shows  $scope\ ps1 \subseteq scope\ ps2 \vee scope\ ps2 \subseteq scope\ ps1 \vee scope\ ps1 \cap scope\ ps2 = \{\}$ 
 $\langle proof \rangle$ 
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:  $((v_1, p_1), (v_2, p_2)) \in edges \implies hyps\ (nodeOf\ v_1)\ p_1 = Some\ p' \implies (v_2, p_2) = (v_1, p') \vee v_2 \in scope\ (v_1, p')$ 

```

```

context Scoped-Graph
begin

```

```

definition hyps-free where
 $hyps-free\ pth = (\forall v_1\ p_1\ v_2\ p_2.\ ((v_1, p_1), (v_2, p_2)) \in set\ pth \longrightarrow hyps\ (nodeOf\ v_1)\ p_1 = None)$ 

```

```

lemma hyps-free-Nil[simp]:  $hyps-free\ [] \langle proof \rangle$ 

```

```

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

```


lemma *path-vertices-shift*:
assumes *path v v' pth*
shows $\text{map fst (map fst pth)}@[v'] = v \# \text{map fst (map snd pth)}$
 $\langle \text{proof} \rangle$

inductive *terminal-path* **where**
terminal-path-empty: $\text{terminal-vertex } v \implies \text{terminal-path } v \ [] \ |$
terminal-path-cons: $((v_1, p_1), (v_2, p_2)) \in \text{edges} \implies \text{terminal-path } v_2 \ v' \ pth \implies \text{hyps (nodeOf } v_1) \ p_1 = \text{None}$
 $\implies \text{terminal-path } v_1 \ v' \ (((v_1, p_1), (v_2, p_2)) \# pth)$

lemma *terminal-path-is-path*:
assumes *terminal-path v v' pth*
shows *path v v' pth*
 $\langle \text{proof} \rangle$

lemma *terminal-path-is-hyps-free*:
assumes *terminal-path v v' pth*
shows *hyps-free pth*
 $\langle \text{proof} \rangle$

lemma *terminal-path-end-is-terminal*:
assumes *terminal-path v v' pth*
shows *terminal-vertex v'*
 $\langle \text{proof} \rangle$

lemma *terminal-pathI*:
assumes *path v v' pth*
assumes *hyps-free pth*
assumes *terminal-vertex v'*
shows *terminal-path v v' pth*
 $\langle \text{proof} \rangle$
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*: $\text{path } v \ v \ pth \implies \text{hyps-free } pth \implies pth = []$
begin

lemma *hyps-free-vertices-distinct*:
assumes *terminal-path v v' pth*
shows $\text{distinct (map fst (map fst pth)}@[v'])$
 $\langle \text{proof} \rangle$

lemma *hyps-free-vertices-distinct'*:
assumes *terminal-path v v' pth*
shows $\text{distinct (v \# map fst (map snd pth))}$
 $\langle \text{proof} \rangle$

lemma *hyps-free-limited*:
assumes *terminal-path v v' pth*
shows $\text{length } pth \leq \text{fcard vertices}$
 $\langle \text{proof} \rangle$

lemma *hyps-free-path-not-in-scope*:
assumes *terminal-path v t pth*
assumes $(v', p') \in \text{snd ' set } pth$

shows $v' \notin \text{scope}(v, p)$
 $\langle \text{proof} \rangle$

end

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

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

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 ⇒ '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 +
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') = {}
  ⟨proof⟩
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 *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 ⟨proof⟩

```

```

lemma Assumption-in-nodes[simp]:

```

```

  Assumption a ∈ sset nodes ⟷ a ∈ set assumptions ⟨proof⟩

```

```

lemma Conclusion-in-nodes[simp]:

```

```

  Conclusion c ∈ sset nodes ⟷ c ∈ set conclusions ⟨proof⟩

```

```

lemma Rule-in-nodes[simp]:

```

```

  Rule r ∈ sset nodes ⟷ r ∈ sset rules ⟨proof⟩

```

```

fun inPorts' :: ('form, 'rule) graph-node ⇒ ('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 ⇒ ('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*:

$inPorts\ n = fset-from-list\ (inPorts'\ n)$
 $\langle proof \rangle$

definition *outPortsRule* **where**

$outPortsRule\ r = ffUnion\ ((\lambda\ a.\ (\lambda\ h.\ Hyp\ h\ a)\ |'\ a-hyps\ a)\ |'\ f-antecedent\ r)\ |\cup|\ Reg\ |'\ f-consequent\ r$

lemma *Reg-in-outPortsRule[simp]*: $Reg\ c\ |\in|\ outPortsRule\ r \iff c\ |\in|\ f-consequent\ r$
 $\langle proof \rangle$

lemma *Hyp-in-outPortsRule[simp]*: $Hyp\ h\ c\ |\in|\ outPortsRule\ r \iff c\ |\in|\ f-antecedent\ r \wedge h\ |\in|\ a-hyps\ c$
 $\langle proof \rangle$

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**

$hyps\ (Rule\ r)\ (Hyp\ h\ a) = (if\ a\ |\in|\ f-antecedent\ r \wedge h\ |\in|\ a-hyps\ a\ then\ Some\ a\ else\ None)$
 $|hyps\ -\ - = None$

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

sublocale *Labeled-Signature nodes inPorts outPorts hyps labelsIn labelsOut*
 $\langle proof \rangle$

lemma *hyps-for-conclusion[simp]*: $hyps-for\ (Conclusion\ n)\ p = \{|\}$
 $\langle proof \rangle$

lemma *hyps-for-Helper[simp]*: $hyps-for\ Helper\ p = \{|\}$
 $\langle proof \rangle$

lemma *hyps-for-Rule[simp]*: $ip\ |\in|\ f-antecedent\ r \implies hyps-for\ (Rule\ r)\ ip = (\lambda\ h.\ Hyp\ h\ ip)\ |'\ a-hyps\ ip$
 $\langle proof \rangle$

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 $freshenLC :: nat \Rightarrow 'var \Rightarrow 'var$
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 antecedent :: 'rule  $\Rightarrow$  ('form, 'var) antecedent list
and consequent :: 'rule  $\Rightarrow$  '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 vid $\alpha$  :: '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* 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 *Abstract-Formula*.
- The abstract rules given in *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 *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* <proof>

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* <proof>

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*.

```

lemma ffUnion-fempty[simp]: ffUnion fempty = fempty
  <proof>

```

```

lemma ffUnion-finsert[simp]: ffUnion (finsert x S) = x | $\cup$ | ffUnion S
  <proof>

```

```

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>

```

```

context includes fset.lifting
begin

```

```

  lift-definition global-assms :: 'var itself  $\Rightarrow$  'form fset is global-assms' <proof>

```

```

  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)
⟨proof⟩

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) ⟨proof⟩

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
⟨proof⟩

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

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)
⟨proof⟩

lemma *global-in-ass*: *global-assms* TYPE('var) |⊆| *ass-forms*
⟨proof⟩

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 ((λ p. edge-tree v' p) |' inPorts (nodeOf v')) else {||})
))
```

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

lemma *finite-tree-edge-tree*:
 $tfinite (tree v p pth) \longleftrightarrow tfinite (edge-tree v p)$
 ⟨proof⟩

coinductive *forbidden-path* :: 'vertex \Rightarrow ('vertex, 'form, 'var) edge' stream \Rightarrow bool **where**
 $forbidden-path: ((v_1, p_1), (v_2, p_2)) \in edges \Longrightarrow hyps (nodeOf v_1) p_1 = None \Longrightarrow forbidden-path v_1 pth \Longrightarrow$
 $forbidden-path v_2 (((v_1, p_1), (v_2, p_2)) \#\# pth)$

lemma *path-is-forbidden*:
assumes *valid-in-port* (v, p)
assumes *ipath* (edge-tree v p) es
shows *forbidden-path* v es
 ⟨proof⟩

lemma *forbidden-path-prefix-is-path*:
assumes *forbidden-path* v es
obtains v' **where** *path* v' v (rev (stake n es))
 ⟨proof⟩

lemma *forbidden-path-prefix-is-hyp-free*:
assumes *forbidden-path* v es
shows *hyps-free* (rev (stake n es))
 ⟨proof⟩

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* (tree v p pth)
 ⟨proof⟩

The main result of this theory.

theorem *solved*
 ⟨proof⟩

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

lemma *iwf-subst-freshen-outPort*:

$\text{iwf } lc \ ts \ \text{ent} \implies$
 $\text{snd } \text{ent} = \text{subst } (i\text{Subst } ts) \ (\text{freshen } (i\text{Annot } ts) \ (i\text{OutPort } ts))$
 $\langle \text{proof} \rangle$

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

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

$\text{all-local-vars } \text{Helper} = \{\}$
 $\langle \text{proof} \rangle$

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

$\text{all-local-vars } (\text{Assumption } c) = \{\}$
 $\langle \text{proof} \rangle$

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

inductive *local-fresh-check* :: ('form, 'rule, 'subst) fresh-check **where**

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

abbreviation *local-iwf* \equiv *iwf 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* :: ('form, 'rule, 'subst) fresh-check **where**

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

abbreviation *plain-iwf* \equiv *iwf no-fresh-check*

fun *isHNode* **where**

$\text{isHNode } (\text{HNode } - \ - \ -) = \text{True}$
 $\text{isHNode } - = \text{False}$

lemma *iwf-edge-match*:

assumes *iwf fc t ent*
assumes $\text{is@[i]} \in \text{it-paths } t$
shows $\text{subst } (i\text{Subst } (\text{tree-at } t \ (\text{is@[i]}))) \ (\text{freshen } (i\text{Annot } (\text{tree-at } t \ (\text{is@[i]}))) \ (i\text{OutPort } (\text{tree-at } t \ (\text{is@[i]}))))$
 $= \text{subst } (i\text{Subst } (\text{tree-at } t \ is)) \ (\text{freshen } (i\text{Annot } (\text{tree-at } t \ is)) \ (\text{a-conc } (\text{inPorts}' \ (i\text{NodeOf } (\text{tree-at } t \ is)) \ !))$

i)))
 ⟨proof⟩

lemma *iwf-length-inPorts*:
 assumes *iwf fc t ent*
 assumes $is \in \text{it-paths } t$
 shows $\text{length } (i\text{Ants } (\text{tree-at } t \text{ is})) \leq \text{length } (\text{inPorts}' (i\text{NodeOf } (\text{tree-at } t \text{ is})))$
 ⟨proof⟩

lemma *iwf-local-not-in-subst*:
 assumes *local-iwf t ent*
 assumes $is \in \text{it-paths } t$
 assumes $var \in \text{all-local-vars } (i\text{NodeOf } (\text{tree-at } t \text{ is}))$
 shows $\text{freshenLC } (i\text{Annot } (\text{tree-at } t \text{ is})) \text{ var} \notin \text{subst-lconsts } (i\text{Subst } (\text{tree-at } t \text{ is}))$
 ⟨proof⟩

lemma *iwf-length-inPorts-not-HNode*:
 assumes *iwf fc t ent*
 assumes $is \in \text{it-paths } t$
 assumes $\neg (i\text{HNode } (\text{tree-at } t \text{ is}))$
 shows $\text{length } (i\text{Ants } (\text{tree-at } t \text{ is})) = \text{length } (\text{inPorts}' (i\text{NodeOf } (\text{tree-at } t \text{ is})))$
 ⟨proof⟩

lemma *iNodeOf-outPorts*:
 $i\text{wfc } t \text{ ent} \implies is \in \text{it-paths } t \implies \text{outPorts } (i\text{NodeOf } (\text{tree-at } t \text{ is})) = \{\}\implies \text{False}$
 ⟨proof⟩

lemma *iNodeOf-tree-at*:
 $i\text{wfc } t \text{ ent} \implies is \in \text{it-paths } t \implies i\text{NodeOf } (\text{tree-at } t \text{ is}) \in \text{sset nodes}$
 ⟨proof⟩

lemma *iwf-outPort*:
 assumes *iwf fc t ent*
 assumes $is \in \text{it-paths } t$
 shows $\text{Reg } (i\text{OutPort } (\text{tree-at } t \text{ is})) \mid \in \mid \text{outPorts } (i\text{NodeOf } (\text{tree-at } t \text{ is}))$
 ⟨proof⟩

inductive-set *hyps-along* for *t is* where
 prefix $(is'@[i]) \text{ is} \implies$
 $i < \text{length } (\text{inPorts}' (i\text{NodeOf } (\text{tree-at } t \text{ is}')))) \implies$
 $\text{hyps } (i\text{NodeOf } (\text{tree-at } t \text{ is}')) \text{ h} = \text{Some } (i\text{NodeOf } (\text{tree-at } t \text{ is}')) ! i \implies$
 $\text{subst } (i\text{Subst } (\text{tree-at } t \text{ is}')) (\text{freshen } (i\text{Annot } (\text{tree-at } t \text{ is}')) (\text{labelsOut } (i\text{NodeOf } (\text{tree-at } t \text{ is}')) \text{ h})) \in \text{hyps-along}$
t is

lemma *hyps-along-Nil[simp]*: $\text{hyps-along } t \ [] = \{\}$
 ⟨proof⟩

lemma *prefix-app-Cons-elim*:
 assumes prefix $(xs@[y]) (z\#zs)$
 obtains $xs = []$ and $y = z$
 | xs' where $xs = z\#xs'$ and prefix $(xs'@[y]) zs$
 ⟨proof⟩

lemma *hyps-along-Cons*:
 assumes *iwf fc t ent*
 assumes $i\#is \in \text{it-paths } t$
 shows $\text{hyps-along } t (i\#is) =$

$(\lambda h. \text{subst } (i\text{Subst } t) (\text{freshen } (i\text{Annot } t) (\text{labelsOut } (i\text{NodeOf } t) h))) \text{ 'fset } (\text{hyps-for } (i\text{NodeOf } t) (\text{inPorts}'$
 $(i\text{NodeOf } t) ! i))$
 $\cup \text{hyps-along } (i\text{Ants } t ! i) \text{ is } (\text{is } ?S1 = ?S2 \cup ?S3)$
 $\langle \text{proof} \rangle$

lemma *iwf-hyps-exist*:
assumes *iwf lc it ent*
assumes *is* \in *it-paths it*
assumes *tree-at it is* = $(HNode\ i\ s\ ants')$
assumes *fst ent* \sqsubseteq *ass-forms*
shows *subst s* $(\text{freshen } i\ anyP) \in \text{hyps-along } it\ is$
 $\langle \text{proof} \rangle$

definition *hyp-port-for'* :: $(\text{'form}, \text{'rule}, \text{'subst}, \text{'var})\ itree \Rightarrow \text{nat list} \Rightarrow \text{'form} \Rightarrow \text{nat list} \times \text{nat} \times (\text{'form},$
 $\text{'var})\ \text{out-port}$ **where**
hyp-port-for' t is f = $(SOME\ x.$
 $(\text{case } x\ \text{of } (is', i, h) \Rightarrow$
 $\text{prefix } (is' @ [i])\ is \wedge$
 $i < \text{length } (\text{inPorts}' (i\text{NodeOf } (\text{tree-at } t\ is')))) \wedge$
 $\text{hyps } (i\text{NodeOf } (\text{tree-at } t\ is'))\ h = \text{Some } (\text{inPorts}' (i\text{NodeOf } (\text{tree-at } t\ is')) ! i) \wedge$
 $f = \text{subst } (i\text{Subst } (\text{tree-at } t\ is')) (\text{freshen } (i\text{Annot } (\text{tree-at } t\ is')) (\text{labelsOut } (i\text{NodeOf } (\text{tree-at } t\ is'))\ h))$
 $\rangle)$

lemma *hyp-port-for-spec'*:
assumes *f* \in *hyps-along t is*
shows $(\text{case } \text{hyp-port-for}'\ t\ is\ f\ \text{of } (is', i, h) \Rightarrow$
 $\text{prefix } (is' @ [i])\ is \wedge$
 $i < \text{length } (\text{inPorts}' (i\text{NodeOf } (\text{tree-at } t\ is')))) \wedge$
 $\text{hyps } (i\text{NodeOf } (\text{tree-at } t\ is'))\ h = \text{Some } (\text{inPorts}' (i\text{NodeOf } (\text{tree-at } t\ is')) ! i) \wedge$
 $f = \text{subst } (i\text{Subst } (\text{tree-at } t\ is')) (\text{freshen } (i\text{Annot } (\text{tree-at } t\ is')) (\text{labelsOut } (i\text{NodeOf } (\text{tree-at } t\ is'))\ h))$
 $\rangle)$
 $\langle \text{proof} \rangle$

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

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

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

lemma *hyp-port-prefix*:
assumes *f* \in *hyps-along t is*
shows $\text{prefix } (\text{hyp-port-path-for } t\ is\ f @ [\text{hyp-port-i-for } t\ is\ f])\ is$
 $\langle \text{proof} \rangle$

lemma *hyp-port-strict-prefix*:
assumes *f* \in *hyps-along t is*
shows *strict-prefix* $(\text{hyp-port-path-for } t\ is\ f)\ is$
 $\langle \text{proof} \rangle$

lemma *hyp-port-it-paths*:
assumes *is* \in *it-paths t*
assumes *f* \in *hyps-along t is*
shows *hyp-port-path-for t is f* \in *it-paths t*
 $\langle \text{proof} \rangle$

lemma *hyp-port-hyps*:

assumes $f \in \text{hyps-along } t \text{ is}$
shows $\text{hyps } (iNodeOf (tree-at t (hyp-port-path-for t is f))) (hyp-port-h-for t is f) = \text{Some } (inPorts' (iNodeOf (tree-at t (hyp-port-path-for t is f)))) ! \text{hyp-port-i-for } t \text{ is } f$
 $\langle \text{proof} \rangle$

lemma *hyp-port-outPort*:
assumes $f \in \text{hyps-along } t \text{ is}$
shows $(hyp-port-h-for t is f) \in | \text{outPorts } (iNodeOf (tree-at t (hyp-port-path-for t is f)))$
 $\langle \text{proof} \rangle$

lemma *hyp-port-eq*:
assumes $f \in \text{hyps-along } t \text{ is}$
shows $f = \text{subst } (iSubst (tree-at t (hyp-port-path-for t is f))) (\text{freshen } (iAnnot (tree-at t (hyp-port-path-for t is f)))) (\text{labelsOut } (iNodeOf (tree-at t (hyp-port-path-for t is f))) (hyp-port-h-for t is f))$
 $\langle \text{proof} \rangle$

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

definition *v-away* :: nat **where** $v-away = \text{to-nat } (\text{None} :: \text{nat list option})$

lemma *isidx-inj[simp]*: $\text{isidx } xs = \text{isidx } ys \iff xs = ys$
 $\langle \text{proof} \rangle$

lemma *isidx-v-away[simp]*: $\text{isidx } xs \neq v-away$
 $\langle \text{proof} \rangle$

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

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

$xs = ys \implies (\bigwedge x \ i. x \in \text{set } ys \implies f \ i \ x = g \ i \ x) \implies \text{mapWithIndex } f \ xs = \text{mapWithIndex } g \ ys$
 $\langle \text{proof} \rangle$

lemma *mapWithIndex-Nil[simp]*: $\text{mapWithIndex } f \ [] = []$
 $\langle \text{proof} \rangle$

lemma *length-mapWithIndex[simp]*: $\text{length } (\text{mapWithIndex } f \ xs) = \text{length } xs$
 $\langle \text{proof} \rangle$

lemma *nth-mapWithIndex[simp]*: $i < \text{length } xs \implies \text{mapWithIndex } f \ xs \ ! \ i = f \ i \ (xs \ ! \ i)$
 $\langle \text{proof} \rangle$

lemma *list-all2-mapWithIndex2E*:

assumes $\text{list-all2 } P \ as \ bs$

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

shows $\text{list-all2 } Q \ as \ (\text{mapWithIndex } f \ bs)$

$\langle \text{proof} \rangle$

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**

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

| $\text{globalize-node } is \ f \ (H \ i \ s) = H \ (\text{isidx } is) \ (\text{subst-renameLCs } f \ s)$

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

$\text{globalize } is \ f \ (RNode \ r \ ants) = RNode$

$(\text{globalize-node } is \ f \ r)$

$(\text{mapWithIndex } (\lambda \ i' \ t.))$

$globalize (is@[i'])$
 $(rename (a-fresh (inPorts' (iNodeOf (RNode r ants)) ! i'))$
 $(iAnnot (RNode r ants)) (isidx is) f)$
 t
 $) ants)$

lemma *iAnnot'-globalize-node[simp]*: $iAnnot' (globalize-node is f n) = isidx is$
 $\langle proof \rangle$

lemma *iAnnot-globalize*:
assumes $is' \in it-paths (globalize is f t)$
shows $iAnnot (tree-at (globalize is f t) is') = isidx (is@is')$
 $\langle proof \rangle$

lemma *all-local-consts-listed'*:
assumes $n \in sset nodes$
assumes $p \in inPorts n$
shows $lconsts (a-conc p) \cup (\bigcup (lconsts 'fset (a-hyps p))) \subseteq a-fresh p$
 $\langle proof \rangle$

lemma *no-local-consts-in-consequences'*:
 $n \in sset nodes \implies Reg p \in outPorts n \implies lconsts p = \{\}$
 $\langle proof \rangle$

lemma *iwf-globalize*:
assumes $local-iwf t (\Gamma \vdash c)$
shows $plain-iwf (globalize is f t) (renameLCs f |' \Gamma \vdash renameLCs f c)$
 $\langle proof \rangle$

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]*:
 $fresh-at t [] = \{\}$
 $\langle proof \rangle$

lemma *fresh-at-snoc[simp]*:
 $fresh-at t (is@[i]) = freshenLC (iAnnot (tree-at t is)) ' (a-fresh (inPorts' (iNodeOf (tree-at t is)) ! i))$
 $\langle proof \rangle$

lemma *fresh-at-def'*:
 $fresh-at t is =$
 $(if is = [] then \{\}$
 $else freshenLC (iAnnot (tree-at t (butlast is))) ' (a-fresh (inPorts' (iNodeOf (tree-at t (butlast is)) ! last$
 $is)))$
 $\langle proof \rangle$

lemma *fresh-at-Cons[simp]*:
 $fresh-at t (i\#is) = (if is = [] then freshenLC (iAnnot t) ' (a-fresh (inPorts' (iNodeOf t) ! i)) else (let t' =$
 $iAnts t ! i in fresh-at t' is))$
 $\langle proof \rangle$

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 $t \ [] = \{\}$
<proof>

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

fresh-at-path $t \ (i \# is) = \text{fresh-at } t \ [i] \cup \text{fresh-at-path } (i \text{Ants } t \ ! \ i) \ is$
<proof>

lemma *globalize-local-consts*:

assumes $is' \in \text{it-paths } (\text{globalize } is \ f \ t)$
shows $\text{subst-lconsts } (i \text{Subst } (\text{tree-at } (\text{globalize } is \ f \ t) \ is')) \subseteq$
 $\text{fresh-at-path } (\text{globalize } is \ f \ t) \ is' \cup \text{range } f$
<proof>

lemma *iwf-globalize'*:

assumes *local-iwf* $t \ ent$
assumes $\bigwedge x. x \ | \in \ | \ \text{fst } ent \implies \text{closed } x$
assumes *closed* $(\text{snd } ent)$
shows *plain-iwf* $(\text{globalize } is \ (\text{freshenLC } v \text{-away}) \ t) \ ent$
<proof>
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 $f1 \ ' \ S1 = f2 \ ' \ S2$
obtains $f \ \text{where } \bigwedge x. x \in S2 \implies f \ x \in S1 \wedge f1 \ (f \ x) = f2 \ x$
<proof>

context *includes* *fset.lifting*

begin

lemma *fimage-eq-to-f*:

assumes $f1 \ | \ ' \ S1 = f2 \ | \ ' \ S2$
obtains $f \ \text{where } \bigwedge x. x \ | \ \in \ | \ S2 \implies f \ x \ | \ \in \ | \ S1 \wedge f1 \ (f \ x) = f2 \ x$
<proof>
end

context *Abstract-Task*

begin

lemma *build-local-iwf*:

fixes $t :: ('form \ \text{entailment} \times ('rule \times 'form) \ \text{NatRule}) \ \text{tree}$
assumes *tfinite* t
assumes *wf* t
shows $\exists \ \text{it. local-iwf } \text{it} \ (\text{fst } (\text{root } t))$
<proof>

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))*
⟨*proof*⟩
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 *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 ∧ fst (fst (root t)) |⊆| *ass-forms* ∧ wf t ∧ *tfinite t*)

lemma
assumes *c |∈| conc-forms*
shows *ts-conc: snd (fst (root (ts c))) = c*
and *ts-context: fst (fst (root (ts c))) |⊆| ass-forms*
and *ts-wf: wf (ts c)*
and *ts-finite[simp]: tfinite (ts c)*
⟨*proof*⟩

abbreviation *it'* **where**
it' c ≡ *globalize [fix conc-forms c, 0] (freshenLC v-away) (to-it (ts c))*

lemma *iwf-it*:

assumes $c \in \text{set conclusions}$
shows $\text{plain-iwf } (it' c) (\text{fst } (\text{root } (ts c)))$
 $\langle \text{proof} \rangle$

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

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

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

lemma *prefixeq-vertices*: $(c, is) \in \text{vertices} \Longrightarrow \text{prefix } is' is \Longrightarrow (c, is') \in \text{vertices}$
 $\langle \text{proof} \rangle$

lemma *none-vertices[simp]*: $(c, []) \in \text{vertices} \longleftrightarrow c \in \text{set conclusions}$
 $\langle \text{proof} \rangle$

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

lemma *vertices-cases[consumes 1, case-names None Some]*:

assumes $v \in \text{vertices}$
obtains $c \in \text{set conclusions}$ **and** $v = (c, [])$
| $c \text{ is}$ **where** $c \in \text{set conclusions}$ **and** $is \in \text{it-paths } (it' c)$ **and** $v = (c, 0 \# is)$
 $\langle \text{proof} \rangle$

lemma *vertices-induct[consumes 1, case-names None Some]*:

assumes $v \in \text{vertices}$
assumes $\bigwedge c. c \in \text{set conclusions} \Longrightarrow P (c, [])$
assumes $\bigwedge c \text{ is}. c \in \text{set conclusions} \Longrightarrow is \in \text{it-paths } (it' c) \Longrightarrow P (c, 0 \# is)$
shows $P v$
 $\langle \text{proof} \rangle$

fun *nodeOf* :: 'form vertex \Rightarrow ('form, 'rule) graph-node **where**

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

fun *inst* **where**

$\text{inst } (c, []) = \text{empty-subst}$
| $\text{inst } (c, i \# is) = \text{iSubst } (\text{tree-at } (it' c) is)$

lemma *terminal-is-nil[simp]*: $v \in \text{vertices} \Longrightarrow \text{outPorts } (\text{nodeOf } v) = \{[]\} \longleftrightarrow \text{snd } v = []$
 $\langle \text{proof} \rangle$

sublocale *Vertex-Graph* nodes inPorts outPorts vertices nodeOf $\langle \text{proof} \rangle$

definition *edge-from* :: 'form \Rightarrow nat list \Rightarrow ('form vertex \times ('form, 'var) out-port) **where**
 $\text{edge-from } c \text{ is} = ((c, 0 \# is), \text{Reg } (\text{iOutPort } (\text{tree-at } (it' c) is)))$

lemma *fst-edge-from[simp]*: $\text{fst } (\text{edge-from } c \text{ is}) = (c, 0 \# is)$
 $\langle \text{proof} \rangle$

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

$\text{in-port-at } (c, []) - = \text{plain-ant } c$
| $\text{in-port-at } (c, - \# is) i = \text{inPorts}' (\text{iNodeOf } (\text{tree-at } (it' c) is)) ! i$

definition *edge-to* :: 'form \Rightarrow nat list \Rightarrow ('form vertex \times ('form,'var) in-port) **where**
edge-to *c* *is* =
 (case rev *is* of [] \Rightarrow ((*c*, []), in-port-at (*c*, []) 0)
 | *i*#*is* \Rightarrow ((*c*, 0 # (rev *is*)), in-port-at (*c*, (0#rev *is*)) *i*))

lemma *edge-to-Nil[simp]*: *edge-to* *c* [] = ((*c*, []), plain-ant *c*)
 <proof>

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

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* <proof>

lemma *snd-edge-at[simp]*: snd (*edge-at* *c* *is*) = *edge-to* *c* *is* <proof>

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>

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*), plain-ant anyP)

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 # hyp-port-path-for (it' *c*) *is* (subst *s* (freshen *n* anyP))),
 hyp-port-h-for (it' *c*) *is* (subst *s* (freshen *n* anyP)))

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

lemma *fst-hyp-edge-at[simp]*:
 fst (hyp-edge-at *c* *is* *n* *s*) = hyp-edge-from *c* *is* *n* *s* <proof>

lemma *snd-hyp-edge-at[simp]*:
 snd (hyp-edge-at *c* *is* *n* *s*) = hyp-edge-to *c* *is* <proof>

inductive-set *edges* **where**

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

sublocale Pre-Port-Graph nodes inPorts outPorts vertices nodeOf edges <proof>

lemma *edge-from-valid-out-port*:
 assumes *p* \in it-paths (it' *c*)
 assumes *c* \in set conclusions
 shows valid-out-port (edge-from *c* *p*)
 <proof>

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* $c p$)
 $\langle \text{proof} \rangle$

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

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

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

inductive-simps *scope-simp*: $\text{scope}' v i v'$
inductive-cases *scope-cases*: $\text{scope}' v i v'$

lemma *scope-valid*:
 $\text{scope}' v i v' \Longrightarrow v' \in \text{vertices}$
 $\langle \text{proof} \rangle$

lemma *scope-valid-inport*:
 $v' \in \text{vertices} \Longrightarrow \text{scope}' v ip v' \longleftrightarrow (\exists i. \text{fst } v = \text{fst } v' \wedge \text{prefix } (\text{snd } v@[i]) (\text{snd } v') \wedge ip = \text{in-port-at } v$
 $i)$
 $\langle \text{proof} \rangle$

definition *terminal-path-from* :: *'form* \Rightarrow *nat list* \Rightarrow (*'form, 'var*) *edge'' list* **where**
 $\text{terminal-path-from } c is = \text{map } (\text{edge-at } c) (\text{rev } (\text{prefixes } is))$

lemma *terminal-path-from-Nil[simp]*:
 $\text{terminal-path-from } c [] = [\text{edge-at } c []]$
 $\langle \text{proof} \rangle$

lemma *terminal-path-from-Snoc[simp]*:
 $\text{terminal-path-from } c (is @ [i]) = \text{edge-at } c (is@[i]) \# \text{terminal-path-from } c is$
 $\langle \text{proof} \rangle$

lemma *path-terminal-path-from*:
 $c \in \text{set conclusions} \Longrightarrow$
 $is \in \text{it-paths } (it' c) \Longrightarrow$
 $\text{path } (c, 0 \# is) (c, []) (\text{terminal-path-from } c is)$
 $\langle \text{proof} \rangle$

lemma *edge-step*:

assumes $((a, b), ba), ((aa, bb), bc) \in \text{edges}$

obtains

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

$\langle \text{proof} \rangle$

lemma *path-has-prefixes*:

assumes $\text{path } v \ v' \ pth$

assumes $\text{snd } v' = []$

assumes $\text{prefix } (is' @ [i]) \ (\text{snd } v)$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

sublocale *Scoped-Graph nodes inPorts outPorts vertices nodeOf edges hyps* $\langle \text{proof} \rangle$

lemma *hyps-free-path-length*:

assumes $\text{path } v \ v' \ pth$

assumes $\text{hyps-free } pth$

shows $\text{length } pth + \text{length } (\text{snd } v') = \text{length } (\text{snd } v)$

$\langle \text{proof} \rangle$

fun *vidx* :: $'form \ \text{vertex} \implies \text{nat}$ **where**

$\text{vidx } (c, []) = \text{isidx } [\text{fidx } \text{conc-forms } c]$

$|\text{vidx } (c, -\#is) = \text{iAnnot } (\text{tree-at } (it' \ c) \ is)$

lemma *my-vidx-inj*: $\text{inj-on } \text{vidx } (\text{fset } \text{vertices})$

$\langle \text{proof} \rangle$

lemma *vidx-not-v-away[simp]*: $v \in |\text{vertices} \implies \text{vidx } v \neq v\text{-away}$

$\langle \text{proof} \rangle$

sublocale *Instantiation inPorts outPorts nodeOf hyps nodes edges vertices labelsIn labelsOut freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP vidx inst*

$\langle \text{proof} \rangle$

sublocale *Well-Scoped-Graph nodes inPorts outPorts vertices nodeOf edges hyps*

$\langle \text{proof} \rangle$

sublocale *Acyclic-Graph nodes inPorts outPorts vertices nodeOf edges hyps*

$\langle \text{proof} \rangle$

sublocale *Saturated-Graph nodes inPorts outPorts vertices nodeOf edges*

$\langle \text{proof} \rangle$

sublocale *Pruned-Port-Graph nodes inPorts outPorts vertices nodeOf edges*

$\langle \text{proof} \rangle$

sublocale *Well-Shaped-Graph nodes inPorts outPorts vertices nodeOf edges hyps* $\langle \text{proof} \rangle$

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⟩

lemma *node-disjoint-fresh-vars:*

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

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

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

shows $a\text{-fresh } (\text{inPorts}' n \ ! \ i) \cap a\text{-fresh } (\text{inPorts}' n \ ! \ i') = \{\}$ $\vee i = i'$

⟨proof⟩

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⟩

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*⟨proof⟩

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⟩

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
begin

class infinite =
  assumes infinite-UNIV: infinite (UNIV::'a set)

instance nat :: infinite
  <proof>
instance prod :: (infinite, type) infinite
  <proof>
instance list :: (type) infinite
  <proof>

lemma countable-infinite-ex-bij:  $\exists f::('a::\{countable,infinite\} \Rightarrow 'b::\{countable,infinite\}). \text{bij } f$ 
  <proof>

```

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\}  $\Rightarrow$  ('var,'cname) pform)  $\Rightarrow$  ('var,'cname) pform  $\Rightarrow$  ('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  $\Rightarrow$  bool
  where closed (Var v)  $\longleftrightarrow$  False
  | closed (Fun n ps)  $\longleftrightarrow$  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  $\Rightarrow$  'var  $\Rightarrow$  'var
  — also no renaming needed as there are no locally fixed constants
   $\lambda$ -. id  $\lambda$ -. {}
  — closedness and substitution as defined above
  closed :: ('var::\{countable,infinite\},'cname) pform  $\Rightarrow$  bool subst
  — no substitution and renaming of locally fixed constants
   $\lambda$ -. {}  $\lambda$ -. id
  — most generic formula
  Var undefined

```

⟨proof⟩

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*}

⟨proof⟩

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⟩

```

```
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
```

```
  ⇒ entail = (Γ ⊢ subst s (freshen a c))
```

```
  ⇒ hyps = ((λant. ((λp. subst s (freshen a p)) |' a-hyps ant |∪| Γ ⊢ subst s (freshen a (a-conc ant)))) |' 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) entail hyps
```

```
  ⟨proof⟩
```

```
end
```

```
context Abstract-Task begin
```

```
lemma natEff-InstI:
```

```
  rule = (r,c)
```

```
  ⇒ c ∈ set (consequent r)
```

```
  ⇒ antec = f-antecedent r
```

```
  ⇒ natEff-Inst rule c antec
```

```
  ⟨proof⟩
```

```
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 ≡ Fun (Const "A") []
```

First the task is defined as an *Abstract-Task*.

```
interpretation task1-1: Abstract-Task
```

```
  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]
  ⟨proof⟩

```

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
  ⟨proof⟩

```

```

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)
  ⟨proof⟩

```

```

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))}
  ⟨proof⟩

```

```

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
  ⟨proof⟩

```

```

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
  ⟨proof⟩

```

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
⟨proof⟩

```

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 :: (string,prop-funs) pform
where B ≡ Fun (Const "B") []
abbreviation C :: (string,prop-funs) pform
where C ≡ Fun (Const "C") []

```

interpretation *task2-11: Abstract-Task*

```

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
[Fun imp [Fun and [A,B],C]]
[Fun imp [A,Fun imp [B,C]]]
⟨proof⟩

```

```

abbreviation n-andI ≡ task2-11.n-rules !! 0
abbreviation n-andE1 ≡ task2-11.n-rules !! 1
abbreviation n-andE2 ≡ task2-11.n-rules !! 2
abbreviation n-impI ≡ task2-11.n-rules !! 3
abbreviation n-impE ≡ task2-11.n-rules !! 4

```

```

lemma n-andI [simp]: n-andI = (andI, Fun and [X,Y])
⟨proof⟩

```

```

lemma n-andE1 [simp]: n-andE1 = (andE, X)
⟨proof⟩

```

```

lemma n-andE2 [simp]: n-andE2 = (andE, Y)
  ⟨proof⟩
lemma n-impI [simp]: n-impI = (impI, Fun imp [X, Y])
  ⟨proof⟩
lemma n-impE [simp]: n-impE = (impE, Y)
  ⟨proof⟩

```

```

lemma subst-Var-eq-id [simp]: subst Var = id
  ⟨proof⟩

```

```

lemma xy-update: f = undefined("X" := x, "Y" := y)  $\implies$  x = f "X"  $\wedge$  y = f "Y" ⟨proof⟩
lemma y-update: f = undefined("Y" := y)  $\implies$  y = f "Y" ⟨proof⟩

```

```

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
  curry (SOME f. bij f):: nat  $\Rightarrow$  string  $\Rightarrow$  string
   $\lambda$ -. id
   $\lambda$ -. {}
  closed :: (string, prop-funs) pform  $\Rightarrow$  bool
  subst
   $\lambda$ -. {}
   $\lambda$ -. id
  Var undefined
  antecedent
  consequent
  prop-rules
  [Fun imp [Fun and [A, B], C]]
  [Fun imp [A, Fun imp [B, C]]]
  ⟨proof⟩

```

```

interpretation Tasked-Proof-Graph
  curry (SOME f. bij f):: nat  $\Rightarrow$  string  $\Rightarrow$  string
   $\lambda$ -. id
   $\lambda$ -. {}
  closed :: (string, prop-funs) pform  $\Rightarrow$  bool
  subst
   $\lambda$ -. {}
   $\lambda$ -. 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
  ⟨proof⟩

```

```

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:var*) (*body: form*)

type-synonym *schema* = *var list* \times *form*

type-synonym *subst* = (*var* \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*

<proof>

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 []* = *{}*

<proof>

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
  <proof>
```

```
lemma lc-map-lc[simp]: lc (map-lc p f) = p ' lc f
  <proof>
```

```
lemma map-lc-map-lc[simp]: map-lc p1 (map-lc p2 f) = map-lc (p1  $\circ$  p2) f
  <proof>
```

```
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
  <proof>
```

```
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
  <proof>
```

```
lemma [simp]: fv-subst1 (map (apsnd (map-lc p)) s) = fv-subst1 s
  <proof>
```

```
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
  <proof>
```

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  $\Rightarrow$  f | None  $\Rightarrow$  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]*: $subst1 [] f = f$
 ⟨proof⟩

lemma *lc-subst1*: $lc (subst1 s f) \subseteq lc f \cup \bigcup (lc \text{ ‘ } snd \text{ ‘ } set s)$
 ⟨proof⟩

lemma *apsnd-def'*: $apsnd f = (\lambda(k, v). (k, f v))$
 ⟨proof⟩

lemma *map-of-map-apsnd*:
 $map\text{-of} (map (apsnd f) xs) = map\text{-option} f \circ map\text{-of} xs$
 ⟨proof⟩

lemma *map-lc-subst1[simp]*: $map\text{-lc} p (subst1 s f) = subst1 (map\text{-lc}\text{-subst1} p s) (map\text{-lc} p f)$
 ⟨proof⟩

fun *subst'* :: $subst \Rightarrow form \Rightarrow form$ **where**
 $subst' s (Var v xs) =$
 (case $map\text{-of} s v$ of $None \Rightarrow (Var v (map (subst' s) xs))$
 | $Some (ps, rhs) \Rightarrow$
 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 $v \in fv\text{-subst} s$ then
 (let $v' = fresh\text{-for} (fv\text{-subst} s)$
 in $Quant n v' (subst' ((v, ([], Var v' []))\#s) f)$)
 else $Quant n v (subst' s f)$)

lemma *subst'-Nil[simp]*: $subst' [] f = f$
 ⟨proof⟩

lemma *lc-subst'*: $lc (subst' s f) \subseteq lc f \cup lc\text{-subst} s$
 ⟨proof⟩

lemma *ran-map-option-comp[simp]*:
 $ran (map\text{-option} f \circ m) = f \text{ ‘ } ran m$
 ⟨proof⟩

lemma *fv-schema-apsnd-map-lc[simp]*:
 $fv\text{-schema} (apsnd (map\text{-lc} p) a) = fv\text{-schema} a$
 ⟨proof⟩

lemma *fv-subst-map-apsnd-map-lc[simp]*:
 $fv\text{-subst} (map (apsnd (apsnd (map\text{-lc} p))) s) = fv\text{-subst} s$

<proof>

lemma *map-apsnd-zip[simp]*: $\text{map } (\text{apsnd } f) (\text{zip } a \ b) = \text{zip } a \ (\text{map } f \ b)$
<proof>

lemma *map-lc-subst'[simp]*: $\text{map-lc } p \ (\text{subst}' \ s \ f) = \text{subst}' \ (\text{map-lc-subst } p \ s) \ (\text{map-lc } p \ f)$
<proof>

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* \Rightarrow *form* \Rightarrow *form* **where**
subst *s* *f* = *subst'* (*filter* ($\lambda (v,s). v \in \text{fv } f$) *s*) *f*

lemma *subst-Nil[simp]*: $\text{subst } [] \ f = f$
<proof>

lemma *subst-noop[simp]*: $\text{fv } f = \{\} \implies \text{subst } s \ f = f$
<proof>

lemma *lc-subst*: $\text{lc } (\text{subst } s \ f) \subseteq \text{lc } f \cup \text{lc-subst } s$
<proof>

lemma *lc-subst-map-lc-subst[simp]*: $\text{lc-subst } (\text{map-lc-subst } p \ s) = p \ \text{'lc-subst } s$
<proof>

lemma *map-lc-subst[simp]*: $\text{map-lc } p \ (\text{subst } s \ f) = \text{subst } (\text{map-lc-subst } p \ s) \ (\text{map-lc } p \ f)$
<proof>

fun *closed* :: *form* \Rightarrow *bool* **where**
closed *f* $\longleftrightarrow \text{fv } f = \{\} \wedge \text{lc } f = \{\}$

interpretation *predicate*: *Abstract-Formulas*

curry to-nat :: *nat* \Rightarrow *var* \Rightarrow *var*

map-lc

lc

closed

subst

lc-subst

map-lc-subst

Var 0 []

<proof>

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*}
 ⟨*proof*⟩

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* ≡ *Var* 13 []
abbreviation *P* :: *form* ⇒ *form*
 where *P* *f* ≡ *Var* 12 [*f*]
abbreviation *Q* :: *form* ⇒ *form*
 where *Q* *f* ≡ *Op* "Q" [*f*]
abbreviation *imp* :: *form* ⇒ *form* ⇒ *form*
 where *imp* *f1* *f2* ≡ *Op* "imp" [*f1*, *f2*]
abbreviation *ForallX* :: *form* ⇒ *form*
 where *ForallX* *f* ≡ *Quant* "all" 9 *f*

Finally the right- and left-hand sides of the rules.

fun *consequent* :: *prop-rule* ⇒ *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* ≡ *Antecedent* {} (*P* (*LC* 0)) {0}
abbreviation *impI-input* **where** *impI-input* ≡ *Antecedent* {*X*} *Y* {}

fun *antecedent* :: *prop-rule* ⇒ (*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* ⇒ *var* ⇒ *var*
map-lc
lc
closed
subst
lc-subst
map-lc-subst
Var 0 []
antecedent
consequent
prop-rules
⟨*proof*⟩

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

$\implies \text{entail} = (\Gamma \vdash \text{subst } s \text{ (freshen } a \text{ } c))$

$\implies \text{hyps} = ((\lambda \text{ant. } ((\lambda p. \text{subst } s \text{ (freshen } a \text{ } p)) \text{ |' | } a\text{-hyps } \text{ant} \text{ |} \cup \text{ | } \Gamma \vdash \text{subst } s \text{ (freshen } a \text{ (} a\text{-conc } \text{ant} \text{)))) \text{ |' | } \text{ants})$

$\implies (\bigwedge \text{ant } f. \text{ant} \text{ |} \in \text{ | } \text{ants} \implies f \text{ |} \in \text{ | } \Gamma \implies \text{freshenLC } a \text{ ' (} a\text{-fresh } \text{ant}) \cap \text{lconsts } f = \{\})$

$\implies (\bigwedge \text{ant. } \text{ant} \text{ |} \in \text{ | } \text{ants} \implies \text{freshenLC } a \text{ ' (} a\text{-fresh } \text{ant}) \cap \text{subst-lconsts } s = \{\})$

$\implies \text{eff (NatRule rule) entail hyps}$

<proof>

end

context *Abstract-Task* **begin**

lemma *natEff-InstI*:

rule = (r,c)

$\implies c \in \text{set (consequent } r)$

$\implies \text{antec} = \text{f-antecedent } r$

$\implies \text{natEff-Inst rule } c \text{ antec}$

<proof>

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))]

<proof>

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 $fv\text{-}subst\text{-}s1[simp]$: $fv\text{-}subst\ s1 = \{\}$
 $\langle proof \rangle$

lemma $subst1\text{-}simps[simp]$:
 $subst\ s1\ (P\ (LC\ n)) = imp\ (Q\ (LC\ n))\ (Q\ (LC\ n))$
 $subst\ s1\ (ForallX\ (P\ x)) = ForallX\ (imp\ (Q\ x)\ (Q\ x))$
 $\langle proof \rangle$

lemma $subst2\text{-}simps[simp]$:
 $subst\ s2\ X = Q\ (LC\ lx)$
 $subst\ s2\ Y = Q\ (LC\ lx)$
 $subst\ s2\ (imp\ X\ Y) = imp\ (subst\ s2\ X)\ (subst\ s2\ Y)$
 $\langle proof \rangle$

lemma $substI1$: $ForallX\ (imp\ (Q\ x)\ (Q\ x)) = subst\ s1\ (predicate.freshen\ 1\ (ForallX\ (P\ x)))$
 $\langle proof \rangle$

lemma $substI2$: $imp\ (Q\ (LC\ lx))\ (Q\ (LC\ lx)) = subst\ s2\ (predicate.freshen\ 2\ (imp\ X\ Y))$
 $\langle proof \rangle$

declare $subst.simps[simp\ del]$

lemma $task.solved$
 $\langle proof \rangle$

abbreviation $vertices$ **where** $vertices \equiv \{[0::nat,1,2]\}$

fun $nodeOf$ **where**
 $nodeOf\ n = [Conclusion\ (ForallX\ (imp\ (Q\ x)\ (Q\ x))),$
 $\quad Rule\ allI,$
 $\quad Rule\ impI] ! n$

fun $inst$ **where**
 $inst\ n = [[],s1,s2] ! n$

interpretation $task$: $Vertex\text{-}Graph\ task.nodes\ task.inPorts\ task.outPorts\ vertices\ nodeOf\ \langle proof \rangle$

abbreviation $e1 :: (nat,\ form,\ nat)\ edge'$
where $e1 \equiv ((1,Reg\ (ForallX\ (P\ x))),\ (0,plain\text{-}ant\ (ForallX\ (imp\ (Q\ x)\ (Q\ x))))$

abbreviation $e2 :: (nat,\ form,\ nat)\ edge'$
where $e2 \equiv ((2,Reg\ (imp\ X\ Y)),\ (1,allI\text{-}input))$

abbreviation $e3 :: (\text{nat}, \text{form}, \text{nat}) \text{edge}'$

where $e3 \equiv ((2, \text{Hyp } X \text{ (impI-input)}), (2, \text{impI-input}))$

abbreviation $\text{task-edges} :: (\text{nat}, \text{form}, \text{nat}) \text{edge}' \text{ set}$ **where** $\text{task-edges} \equiv \{e1, e2, e3\}$

interpretation $\text{task}: \text{Scoped-Graph } \text{task.nodes } \text{task.inPorts } \text{task.outPorts } \text{vertices } \text{nodeOf } \text{task-edges } \text{task.hyps}$
 $\langle \text{proof} \rangle$

interpretation $\text{task}: \text{Instantiation}$

task.inPorts

task.outPorts

nodeOf

task.hyps

task.nodes

task-edges

vertices

task.labelsIn

task.labelsOut

$\text{curry to-nat} :: \text{nat} \Rightarrow \text{var} \Rightarrow \text{var}$

map-lc

lc

closed

subst

lc-subst

map-lc-subst

$\text{Var } 0 \square$

id

inst

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma $\text{no-path-01}[\text{simp}]: \text{task.path } 0 \ v \ \text{pth} \longleftrightarrow (\text{pth} = \square \wedge v = 0)$

$\langle \text{proof} \rangle$

lemma $\text{no-path-12}[\text{simp}]: \neg \text{task.path } 1 \ 2 \ \text{pth}$

$\langle \text{proof} \rangle$

interpretation Acyclic-Graph

task.nodes

task.inPorts

task.outPorts

vertices

nodeOf

task-edges

task.hyps

$\langle \text{proof} \rangle$

interpretation Saturated-Graph

task.nodes
task.inPorts
task.outPorts
vertices
nodeOf
task-edges
<proof>

interpretation *Pruned-Port-Graph*

task.nodes
task.inPorts
task.outPorts
vertices
nodeOf
task-edges
<proof>

interpretation *Well-Shaped-Graph*

task.nodes
task.inPorts
task.outPorts
vertices
nodeOf task-edges
task.hyps
<proof>

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
<proof>

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
 ⟨*proof*⟩

lemma *path-20*:
assumes *task.path 2 0 pth*
shows $(1, \text{allI-input}) \in \text{snd } \text{'set } pth$
 ⟨*proof*⟩

lemma *scope-21*: $2 \in \text{task.scope } (1, \text{allI-input})$
 ⟨*proof*⟩

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
 ⟨*proof*⟩

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  
⟨proof⟩  
  
end  
  
end
```