The meta theory of the Incredible Proof Machine

Joachim Breitner   Denis Lohner

March 17, 2025

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

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

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

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

This formalization covers the following aspects:

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

  This is the much harder direction, mostly because the freshness side condition for locally fixed constants (such as in the introduction rule for the universal quantifier) is a local check in natural deduction, but a global check in proofs graphs, and thus some elaborate renaming has to occur (*globalize* in *Incredible_Trees*).
- To explain our abstract locales, and ensure that the assumptions are consistent, we provide example instantiations for them.

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

Acknowledgements

We would like to thank Andreas Lochbihler for helpful comments.

References

[BPT14] Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel, *Abstract completeness*, Archive of Formal Proofs (2014), http://isa-afp.org/entries/Abstract_Completeness.shtml, Formal proof development.

[Bre16] Joachim Breitner, *Visual theorem proving with the Incredible Proof Machine*, ITP, 2016.

2 Auxiliary theories

2.1 Entailment

theory *Entailment*
imports *Main HOL−Library.FSet*
begin

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

abbreviation *entails* :: *'form fset* ⇒ *'form* ⇒ *'form entailment* (**infix** ‹⊢› *50*)
  where *a* ⊢ *c* ≡ (*a*, *c*)

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

end

2.2 Indexed_FSet

theory *Indexed-FSet*
imports
  HOL−Library.FSet
begin

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

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

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

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

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

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

inductive *indexed-fmember* :: *'a* ⇒ *nat* ⇒ *'a fset* ⇒ *bool* (‹- |∈|$_{-}$ -› [*50*,*50*,*50*] *50*) **where**
  i < *size s* ⟹ *list-of-fset s* ! *i* |∈|$_i$ *s*

lemma *indexed-fmember-is-fmember*: *x* |∈|$_i$ *s* ⟹ *x* |∈| *s*
proof (*induction rule*: *indexed-fmember.induct*)

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

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

lemma *indexed-fmember-unique*: $x\ |{\in}|_i\ s \Longrightarrow y\ |{\in}|_j\ s \Longrightarrow x = y \longleftrightarrow i = j$
 by (*metis distinct-list-of-fset indexed-fmember.cases length-list-of-fset nth-eq-iff-index-eq*)

definition *indexed-members* :: *′a fset* ⇒ (*nat* × *′a*) *list* **where**
 indexed-members s = *zip* [*0*..<*size s*] (*list-of-fset s*)

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

lemma *mem-set-indexed-members′*[*simp*]:
 t ∈ *set* (*indexed-members s*) ⟷ *snd t* $|{\in}|_{fst\ t}\ s$
 by (*cases t*, *simp add*: *mem-set-indexed-members*)

definition *fnth* (**infixl** ‹|!|› *100*) **where**
 s |!| *n* = *list-of-fset s* ! *n*
lemma *fnth-indexed-fmember*: $i <$ *size s* ⟹ *s* |!| *i* $|{\in}|_i\ s$
 unfolding *fnth-def* **by** (*rule indexed-fmember.intros*)
lemma *indexed-fmember-fnth*: $x\ |{\in}|_i\ s$ ⟷ (*s* |!| *i* = *x* ∧ *i* < *size s*)
 unfolding *fnth-def* **by** (*metis indexed-fmember.simps*)
end

definition *fidx* :: *′a fset* ⇒ *′a* ⇒ *nat* **where**
 fidx s x = (*SOME i*. $x\ |{\in}|_i\ s$)

lemma *fidx-eq*[*simp*]: $x\ |{\in}|_i\ s$ ⟹ *fidx s x* = *i*
 unfolding *fidx-def*
 by (*rule someI2*)(*auto simp add*: *indexed-fmember-fnth fnth-def nth-eq-iff-index-eq*)

lemma *fidx-inj*[*simp*]: x $|{\in}|$ s ⟹ y $|{\in}|$ s ⟹ *fidx s x* = *fidx s y* ⟷ *x* = *y*
 by (*auto dest*!: *fmember-is-indexed-fmember simp add*: *indexed-fmember-unique*)

lemma *inj-on-fidx*: *inj-on* (*fidx vertices*) (*fset vertices*)
 by (*rule inj-onI*) *simp*

end

2.3 Rose_Tree

theory *Rose-Tree*
imports *Main HOL−Library.Sublist*
begin

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

2.3.1 The rose tree data type

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

2.3.2 The set of paths in a rose tree

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

inductive *it-pathsP* :: *'a rose-tree* ⇒ *nat list* ⇒ *bool* **where**
  it-paths-Nil: *it-pathsP t* []
| *it-paths-Cons*: *i* < *length* (*children t*) ⟹ *children t* ! *i* = *t'* ⟹ *it-pathsP t' is* ⟹ *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* ⇒ *nat list set* **where**
  it-paths t = *Collect* (*it-pathsP t*)

lemma *it-paths-eq* [*pred-set-conv*]: *it-pathsP t* = (λ*x*. *x* ∈ *it-paths t*)
 by(*simp add*: *it-paths-def*)

lemmas *it-paths-intros* [*intro?*] = *it-pathsP.intros*[*to-set*]
lemmas *it-paths-induct* [*consumes 1*, *induct set*: *it-paths*] = *it-pathsP.induct*[*to-set*]
lemmas *it-paths-cases* [*consumes 1*, *cases set*: *it-paths*] = *it-pathsP.cases*[*to-set*]
lemmas *it-paths-ConsE* = *it-pathP-ConsE*[*to-set*]
lemmas *it-paths-RNodeE* = *it-pathP-RNodeE*[*to-set*]
lemmas *it-paths-simps* = *it-pathsP.simps*[*to-set*]

lemmas *it-paths-intros*(*1*)[*simp*]

lemma *it-paths-RNode-Nil*[*simp*]: *it-paths* (*RNode r* []) = {[]}
  by (*auto elim*: *it-paths-cases*)

lemma *it-paths-Union*: *it-paths t* ⊆ *insert* [] (*Union* (((λ (*i*,*t*). ((#) *i*) ‘ *it-paths t*) ‘ *set* (*List.enumerate* (*0*::*nat*) (*children t*)))))
  apply (*rule*)
  apply (*erule it-paths-cases*)
  apply (*auto intro*!: *bexI simp add*: *in-set-enumerate-eq*)
  done

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

2.3.3 Indexing into a rose tree

fun *tree-at* :: *'a rose-tree* ⇒ *nat list* ⇒ *'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*] ∈ *it-paths t*
  shows *is* ∈ *it-paths t* ∧ *i* < *length* (*children* (*tree-at t is*))
using *assms*
by (*induction is arbitrary*: *t*)(*auto intro*!: *it-paths-intros elim*!: *it-paths-ConsE*)

lemma *it-paths-strict-prefix*:
  assumes *is* ∈ *it-paths t*
  assumes *strict-prefix is′ is*
  shows *is′* ∈ *it-paths t*
proof−
  from *assms*(*2*)
  obtain *is′′* **where** *is* = *is′* @ *is′′* **using** *strict-prefixE′* **by** *blast*
  from *assms*(*1*)[*unfolded this*]
  show *?thesis*
    by(*induction is′ arbitrary*: *t*) (*auto elim*!: *it-paths-ConsE intro*!: *it-paths-intros*)
qed

lemma *it-paths-prefix*:
  assumes *is* ∈ *it-paths t*
  assumes *prefix is′ is*
  shows *is′* ∈ *it-paths t*
using *assms it-paths-strict-prefix strict-prefixI* **by** *fastforce*

lemma *it-paths-butlast*:
  assumes *is* ∈ *it-paths t*
  shows *butlast is* ∈ *it-paths t*
using *assms prefixeq-butlast* **by** (*rule it-paths-prefix*)

lemma *it-path-SnocI*:
  assumes *is* ∈ *it-paths t*
  assumes *i* < *length* (*children* (*tree-at t is*))
  shows *is* @ [*i*] ∈ *it-paths t*
  using *assms*
  by (*induction t arbitrary*: *is i*)
    (*auto 4 4 elim*!: *it-paths-RNodeE intro*: *it-paths-intros*)

end

3 Abstract formulas, rules and tasks

3.1 Abstract_Formula

theory *Abstract-Formula*
imports
Main
HOL−Library.FSet
HOL−Library.Stream
Indexed-FSet
begin

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

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

locale *Abstract-Formulas* =
— Variables can be renamed injectively
fixes *freshenLC* :: *nat* ⇒ *'var* ⇒ *'var*
— A variable-changing function can be mapped over a formula
fixes *renameLCs* :: (*'var* ⇒ *'var*) ⇒ (*'form* ⇒ *'form*)
— The set of variables occurring in a formula
fixes *lconsts* :: *'form* ⇒ *'var set*
— A closed formula has no variables, and substitions do not affect it.
fixes *closed* :: *'form* ⇒ *bool*
— A substitution can be applied to a formula.
fixes *subst* :: *'subst* ⇒ *'form* ⇒ *'form*
— The set of variables occurring (in the image) of a substitution.
fixes *subst-lconsts* :: *'subst* ⇒ *'var set*
— A variable-changing function can be mapped over a substitution
fixes *subst-renameLCs* :: (*'var* ⇒ *'var*) ⇒ (*'subst* ⇒ *'subst*)
— A most generic formula, can be substitutied to anything.
fixes *anyP* :: *'form*

assumes *freshenLC-eq-iff* [*simp*]: *freshenLC a v* = *freshenLC a' v'* ⟷ *a* = *a'* ∧ *v* = *v'*
assumes *lconsts-renameLCs*: *lconsts* (*renameLCs p f*) = *p* ‘ *lconsts f*
assumes *rename-closed*: *lconsts f* = {} ⟹ *renameLCs p f* = *f*
assumes *subst-closed*: *closed f* ⟹ *subst s f* = *f*
assumes *closed-no-lconsts*: *closed f* ⟹ *lconsts f* = {}
assumes *fv-subst*: *lconsts* (*subst s f*) ⊆ *lconsts f* ∪ *subst-lconsts s*
assumes *rename-rename*: *renameLCs p1* (*renameLCs p2 f*) = *renameLCs* (*p1* ∘ *p2*) *f*
assumes *rename-subst*: *renameLCs p* (*subst s f*) = *subst* (*subst-renameLCs p s*) (*renameLCs p f*)
assumes *renameLCs-cong*: (⋀ *x*. *x* ∈ *lconsts f* ⟹ *f1 x* = *f2 x*) ⟹ *renameLCs f1 f* = *renameLCs f2 f*
assumes *subst-renameLCs-cong*: (⋀ *x*. *x* ∈ *subst-lconsts s* ⟹ *f1 x* = *f2 x*) ⟹ *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*: ∃ *s*. (∀ *f*. *subst s f* = *f*) ∧ *subst-lconsts s* = {}
assumes *anyP-is-any*: ∃ *s*. *subst s anyP* = *f*
begin
definition *freshen* :: *nat* ⇒ *'form* ⇒ *'form* **where**
freshen n = *renameLCs* (*freshenLC n*)

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

empty-subst = (*SOME s.* (∀ *f. subst s f* = *f*) ∧ *subst-lconsts s* = {})

lemma *empty-subst-spec*:
(∀ *f. subst empty-subst f* = *f*) ∧ *subst-lconsts empty-subst* = {}
unfolding *empty-subst-def* **using** *empty-subst* **by** (*rule someI-ex*)
lemma *subst-empty-subst*[*simp*]: *subst empty-subst f* = *f*
by (*metis empty-subst-spec*)
lemma *subst-lconsts-empty-subst*[*simp*]: *subst-lconsts empty-subst* = {}
by (*metis empty-subst-spec*)

lemma *lconsts-freshen*: *lconsts* (*freshen a f*) = *freshenLC a* ' *lconsts f*
unfolding *freshen-def* **by** (*rule lconsts-renameLCs*)

lemma *freshen-closed*: *lconsts f* = {} ⟹ *freshen a f* = *f*
unfolding *freshen-def* **by** (*rule rename-closed*)

lemma *closed-eq*:
assumes *closed f1*
assumes *closed f2*
shows *subst s1* (*freshen a1 f1*) = *subst s2* (*freshen a2 f2*) ⟷ *f1* = *f2*
using *assms*
by (*auto simp add*: *closed-no-lconsts freshen-def lconsts-freshen subst-closed rename-closed*)

lemma *freshenLC-range-eq-iff*[*simp*]: *freshenLC a v* ∈ *range* (*freshenLC a'*) ⟷ *a* = *a'*
by *auto*

definition *rerename* :: *'var set* ⇒ *nat* ⇒ *nat* ⇒ (*'var* ⇒ *'var*) ⇒ (*'var* ⇒ *'var*) **where**
rerename V from to f x = (*if x* ∈ *freshenLC from* ' *V then freshenLC to* (*inv* (*freshenLC from*) *x*) *else f x*)

lemma *inj-freshenLC*[*simp*]: *inj* (*freshenLC i*)
by (*rule injI*) *simp*

lemma *rerename-freshen*[*simp*]: *x* ∈ *V* ⟹ *rerename V i* (*isidx is*) *f* (*freshenLC i x*) = *freshenLC* (*isidx is*) *x*
unfolding *rerename-def* **by** *simp*

lemma *range-rerename*: *range* (*rerename V from to f*) ⊆ *freshenLC to* ' *V* ∪ *range f*
by (*auto simp add*: *rerename-def split*: *if-splits*)

lemma *rerename-noop*:
x ∉ *freshenLC from* ' *V* ⟹ *rerename V from to f x* = *f x*
by (*auto simp add*: *rerename-def split*: *if-splits*)

lemma *rerename-rename-noop*:
freshenLC from ' *V* ∩ *lconsts form* = {} ⟹ *renameLCs* (*rerename V from to f*) *form* = *renameLCs f form*
by (*intro renameLCs-cong rerename-noop*) *auto*

lemma *rerename-subst-noop*:
freshenLC from ' *V* ∩ *subst-lconsts s* = {} ⟹ *subst-renameLCs* (*rerename V from to f*) *s* = *subst-renameLCs f s*
by (*intro subst-renameLCs-cong rerename-noop*) *auto*
end
end

3.2 Abstract_Rules

theory *Abstract-Rules*
imports
Abstract-Formula
begin

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

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

Each antecedent of a rule consists of

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

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

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

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

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

abbreviation *plain-ant* :: *'form* ⇒ (*'form*, *'var*) *antecedent*
where *plain-ant f* ≡ *Antecedent* {||} *f* {}

locale *Abstract-Rules* =
Abstract-Formulas freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP
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* +

fixes *antecedent* :: *'rule* ⇒ (*'form*, *'var*) *antecedent list*
and *consequent* :: *'rule* ⇒ *'form list*
and *rules* :: *'rule stream*

assumes *no-empty-conclusions*: ∀ *xs*∈*sset rules*. *consequent xs* ≠ []

assumes *no-local-consts-in-consequences*: ∀ *xs*∈*sset rules*. ⋃(*lconsts* ‘ (*set* (*consequent xs*))) = {}
assumes *no-multiple-local-consts*:
⋀ *r i i'* . *r* ∈ *sset rules* ⟹
i < *length* (*antecedent r*) ⟹
i' < *length* (*antecedent r*) ⟹

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

assumes *all-local-consts-listed*:

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

begin

definition *f-antecedent* :: *'rule* ⇒ (*'form*, *'var*) *antecedent fset*
where *f-antecedent r* = *fset-from-list* (*antecedent r*)

definition *f-consequent r* = *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* ⇒ *'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* +

fixes *assumptions* :: *'form list*
fixes *conclusions* :: *'form list*
assumes *assumptions-closed*: $\bigwedge a.\ a \in set\ assumptions \Longrightarrow closed\ a$
assumes *conclusions-closed*: $\bigwedge c.\ c \in set\ conclusions \Longrightarrow closed\ c$
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* |∈| *ass-forms* ⟷ *a* ∈ *set assumptions*
by (*auto simp add*: *ass-forms-def*)

lemma *mem-conc-forms*[*simp*]: *a* |∈| *conc-forms* ⟷ *a* ∈ *set conclusions*
by (*auto simp add*: *conc-forms-def*)

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

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

lemma *subst-freshen-in-ass-formsI*:
assumes *pf* ∈ *set assumptions*

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

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

end

4 Incredible Proof Graphs

4.1 Incredible_Signatures

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

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

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

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

locale Port-Graph-Signature-Scoped =
  Port-Graph-Signature +
  fixes hyps :: 'node ⇒ 'outPort ⇀ 'inPort
  assumes hyps-correct: hyps n p1 = Some p2 ⟹ p1 |∈| outPorts n ∧ p2 |∈| inPorts n
begin
  inductive-set hyps-for' :: 'node ⇒ 'inPort ⇒ 'outPort set for n p
    where hyps n h = Some p ⟹ h ∈ hyps-for' n p

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

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

locale Labeled-Signature =
  Port-Graph-Signature-Scoped +
  fixes labelsIn :: 'node ⇒ 'inPort ⇒ 'form
  fixes labelsOut :: 'node ⇒ 'outPort ⇒ '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* ⇒ *'inPort fset* **and** *outPorts* :: *'node* ⇒ *'outPort fset*
and *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* +

fixes *local-vars* :: *'node* ⇒ *'inPort* ⇒ *'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* ⇒ *'inPort fset*
and *outPorts* :: *'node* ⇒ *'outPort fset* +
fixes *vertices* :: *'v fset*
fixes *nodeOf* :: *'v* ⇒ *'node*
begin
fun *valid-out-port* **where** *valid-out-port* (*v*,*p*) ⟷ *v* |∈| *vertices* ∧ *p* |∈| *outPorts* (*nodeOf v*)
fun *valid-in-port* **where** *valid-in-port* (*v*,*p*) ⟷ *v* |∈| *vertices* ∧ *p* |∈| *inPorts* (*nodeOf v*)

fun *terminal-node* **where**
terminal-node n ⟷ *outPorts n* = {||}
fun *terminal-vertex* **where**
terminal-vertex v ⟷ *v* |∈| *vertices* ∧ *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* × *'outPort*) × (*'v* × *'inPort*))

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  from ‹path - - -› and ‹pth = pth1@[e]@pth2›
  have path v (edge-end e) (pth1@[e]) and path (edge-end e) v' pth2 by (metis path-split)+
  show thesis
  proof(rule that)
    show pth = (pth1@[e])@pth2 using ‹pth= -› by simp
     show path v (fst ps) (pth1@[e]) using ‹path v (edge-end e) (pth1@[e])›  ‹snd e = ps› by (simp add:
edge-end-tup)
    show path (fst ps) v' pth2 using ‹path (edge-end e) v' pth2›  ‹snd e = ps› by (simp add: edge-end-tup)
    show ps ∉ snd ‘ set pth1 by fact
    show snd e = ps by fact
  qed
qed
end
```

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

```
locale Port-Graph = Pre-Port-Graph +
  assumes valid-nodes: nodeOf ‘ fset vertices  ⊆ sset nodes
  assumes valid-edges: ∀ (ps1,ps2) ∈ edges. valid-out-port ps1 ∧ valid-in-port ps2
begin
  lemma snd-set-path-verties: path v v' pth ⟹ fst ‘ snd ‘ set pth ⊆ fset vertices
    apply (induction rule: path.induct)
    apply auto
    apply (metis valid-in-port.elims(2) edge-end.simps case-prodD valid-edges)
    done

  lemma fst-set-path-verties: path v v' pth ⟹ fst ‘ fst ‘ set pth ⊆ fset vertices
    apply (induction rule: path.induct)
```

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

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

```
locale Pruned-Port-Graph = Port-Graph +
  assumes pruned: ⋀v.  v |∈| vertices ⟹ (∃pth v'. path v v' pth ∧ terminal-vertex v')
begin
  lemma scopes-not-refl:
    assumes v |∈| vertices
    shows v ∉ scope (v,p)
  proof(rule notI)
    assume v ∈ scope (v,p)

    from pruned[OF assms]
     obtain pth t where terminal-vertex t and path v t pth and least: ∀ pth'. path v t pth' ⟶ length pth ≤ length pth'
      by atomize-elim (auto simp del: terminal-vertex.simps elim: ex-has-least-nat)

    from scope-split[OF ‹v ∈ scope (v,p)› ‹path v t pth› ‹terminal-vertex t›]
    obtain pth1 e pth2 where pth = (pth1 @ [e]) @ pth2 path v t pth2 by (metis fst-conv)

    from this(2) least
    have length pth ≤ length pth2 by auto
    with ‹pth = -›
    show False by auto
  qed
```

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

```
  lemma scopes-nest:
    fixes ps1 ps2
    shows scope ps1 ⊆ scope ps2 ∨ scope ps2 ⊆ scope ps1 ∨ scope ps1 ∩ scope ps2 = {}
  proof(cases ps1 = ps2)
    assume ps1 ≠ ps2
    {
    fix v
    assume v ∈ scope ps1 ∩ scope ps2
    hence v |∈| vertices using scope.simps by auto
    then obtain pth t where path v t pth and terminal-vertex t using pruned by blast

    from ‹path v t pth› and ‹terminal-vertex t› and ‹v ∈ scope ps1 ∩ scope ps2›
    obtain pth1a e1 pth1b  where pth = (pth1a@[e1])@pth1b and path v (fst ps1) (pth1a@[e1]) and snd e1 = ps1 and ps1 ∉ snd ` set pth1a
      by (auto elim: scope-split)

    from ‹path v t pth› and ‹terminal-vertex t› and ‹v ∈ scope ps1 ∩ scope ps2›
    obtain pth2a e2 pth2b  where pth = (pth2a@[e2])@pth2b and path v (fst ps2) (pth2a@[e2]) and snd e2 = ps2 and ps2 ∉ snd ` set pth2a
      by (auto elim: scope-split)

    from ‹pth = (pth1a@[e1])@pth1b› ‹pth = (pth2a@[e2])@pth2b›
    have set pth1a ⊆ set pth2a ∨ set pth2a ⊆ set pth1a by (auto simp add: append-eq-append-conv2)
    hence scope ps1 ⊆ scope ps2 ∨ scope ps2 ⊆ scope ps1
    proof
      assume set pth1a ⊆ set pth2a with ‹ps2 ∉ -›
```

```
      have ps2 ∉ snd ‘ set (pth1a@[e1]) using ‹ps1 ≠ ps2› ‹snd e1 = ps1› by auto

      have scope ps1 ⊆ scope ps2
      proof
        fix v′
        assume v′ ∈ scope ps1
        hence v′ |∈| vertices using scope.simps by auto
        thus v′ ∈ scope ps2
        proof(rule scope.intros)
          fix pth′ t′
          assume path v′ t′ pth′ and terminal-vertex t′
          with ‹v′ ∈ scope ps1›
          obtain pth3a e3 pth3b where pth′ = (pth3a@[e3])@pth3b and path (fst ps1) t′ pth3b
            by (auto elim: scope-split)

           have path v t′ ((pth1a@[e1]) @ pth3b) using ‹path v (fst ps1) (pth1a@[e1])› and ‹path (fst ps1) t′
pth3b›
            by (rule path-appendI)
          with ‹terminal-vertex t′› ‹v ∈ scope ps1 ∩ scope ps2›
          have ps2 ∈ snd ‘ set ((pth1a@[e1]) @ pth3b) by (meson IntD2 scope.cases)
          hence ps2 ∈ snd ‘ set pth3b using ‹ps2 ∉ snd ‘ set (pth1a@[e1])› by auto
          thus ps2 ∈ snd ‘ set pth′ using ‹pth′=-› by auto
        qed
      qed
      thus ?thesis by simp
    next
      assume set pth2a ⊆ set pth1a with ‹ps1 ∉ -›
      have ps1 ∉ snd ‘ set (pth2a@[e2]) using ‹ps1 ≠ ps2› ‹snd e2 = ps2› by auto

      have scope ps2 ⊆ scope ps1
      proof
        fix v′
        assume v′ ∈ scope ps2
        hence v′ |∈| vertices using scope.simps by auto
        thus v′ ∈ scope ps1
        proof(rule scope.intros)
          fix pth′ t′
          assume path v′ t′ pth′ and terminal-vertex t′
          with ‹v′ ∈ scope ps2›
          obtain pth3a e3 pth3b where pth′ = (pth3a@[e3])@pth3b and path (fst ps2) t′ pth3b
            by (auto elim: scope-split)

           have path v t′ ((pth2a@[e2]) @ pth3b) using ‹path v (fst ps2) (pth2a@[e2])› and ‹path (fst ps2) t′
pth3b›
            by (rule path-appendI)
          with ‹terminal-vertex t′› ‹v ∈ scope ps1 ∩ scope ps2›
          have ps1 ∈ snd ‘ set ((pth2a@[e2]) @ pth3b) by (meson IntD1 scope.cases)
          hence ps1 ∈ snd ‘ set pth3b using ‹ps1 ∉ snd ‘ set (pth2a@[e2])› by auto
          thus ps1 ∈ snd ‘ set pth′ using ‹pth′=-› by auto
        qed
      qed
      thus ?thesis by simp
    qed
    }
    thus ?thesis by blast
  qed simp
end
```

A well-scoped graph is one where a port marked to be a local hypothesis is only connected to the corresponding input port, either directly or via a path. It must not be, however, that there is a 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*: $((v_1,p_1),(v_2,p_2)) \in edges \Longrightarrow hyps\ (nodeOf\ v_1)\ p_1 = Some\ p' \Longrightarrow (v_2,p_2) = (v_1,p') \vee v_2 \in scope\ (v_1,p')$

context *Scoped-Graph*
begin

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

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

lemma *hyps-free-Cons*[*simp*]: $hyps\text{-}free\ (e\#pth) \longleftrightarrow hyps\text{-}free\ pth \wedge hyps\ (nodeOf\ (fst\ (fst\ e)))\ (snd\ (fst\ e)) = None$
  by (*auto simp add*: *hyps-free-def*) (*metis prod.collapse*)

lemma *path-vertices-shift*:
  assumes $path\ v\ v'\ pth$
  shows $map\ fst\ (map\ fst\ pth)@[v'] = v\#map\ fst\ (map\ snd\ pth)$
using *assms* **by** *induction auto*

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

lemma *terminal-path-is-path*:
  assumes $terminal\text{-}path\ v\ v'\ pth$
  shows $path\ v\ v'\ pth$
using *assms* **by** *induction* (*auto simp add*: *path-cons-simp*)

lemma *terminal-path-is-hyps-free*:
  assumes $terminal\text{-}path\ v\ v'\ pth$
  shows *hyps-free pth*
using *assms* **by** *induction* (*auto simp add*: *hyps-free-def*)

lemma *terminal-path-end-is-terminal*:
  assumes $terminal\text{-}path\ v\ v'\ pth$
  shows $terminal\text{-}vertex\ v'$
using *assms* **by** *induction*

lemma *terminal-pathI*:
  assumes $path\ v\ v'\ pth$
  assumes *hyps-free pth*
  assumes $terminal\text{-}vertex\ v'$
  shows $terminal\text{-}path\ v\ v'\ pth$
using *assms*
by *induction* (*auto intro*: *terminal-path.intros*)
end

An acyclic graph is one where there are no non-trivial cyclic paths (disregarding edges that are local hypotheses – these are naturally and benignly cyclic).

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

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

lemma hyps-free-limited:
  assumes terminal-path v v' pth
  shows length pth ≤ fcard vertices
```

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

lemma hyps-free-path-not-in-scope:
  assumes terminal-path v t pth
  assumes (v',p') ∈ snd ‘ set pth
  shows   v' ∉ scope (v, p)
proof
  assume v' ∈ scope (v,p)

  from ‹(v',p') ∈ snd ‘ set pth›
  obtain pth1 pth2 e  where pth = pth1@[e]@pth2 and snd e = (v',p') by (rule snd-set-split)
  from terminal-path-is-path[OF assms(1), unfolded ‹pth = - ›] ‹snd e = -›
  have path v v' (pth1@[e]) and path v' t pth2 unfolding path-split by (auto simp add: edge-end-tup)

  from ‹v' ∈ scope (v,p)› terminal-path-end-is-terminal[OF assms(1)] ‹path v' t pth2›
  have (v,p) ∈ snd ‘ set pth2 by (rule scope-find)
  then obtain pth2a e' pth2b  where pth2 = pth2a@[e']@pth2b and snd e' = (v,p)  by (rule snd-set-split)
  from ‹path v' t pth2›[unfolded ‹pth2 = - ›] ‹snd e' = -›
  have path v' v (pth2a@[e']) and path v t pth2b unfolding path-split by (auto simp add: edge-end-tup)

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

end
```

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

```
locale Saturated-Graph = Port-Graph +
  assumes saturated: valid-in-port (v,p) ⟹ ∃ e ∈ edges . 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* ⇒ *'inPort* ⇒ *'form* **and** *labelsOut* :: *'node* ⇒ *'outPort* ⇒ *'form*
and *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* +
fixes *vidx* :: *'vertex* ⇒ *nat*
and *inst* :: *'vertex* ⇒ *'subst*
assumes *vidx-inj*: *inj-on vidx* (*fset vertices*)
begin
definition *labelAtIn* :: *'vertex* ⇒ *'inPort* ⇒ *'form* **where**
labelAtIn v p = *subst* (*inst v*) (*freshen* (*vidx v*) (*labelsIn* (*nodeOf v*) *p*))
definition *labelAtOut* :: *'vertex* ⇒ *'outPort* ⇒ *'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 labeld with the same formula.

locale *Solution* =
Instantiation - - - - - edges **for** *edges* :: ((*'vertex* × *'outPort*) × *'vertex* × *'inPort*) *set* +
assumes *solved*: ((v_1,p_1),(v_2,p_2)) ∈ *edges* ⟹ *labelAtOut* v_1 p_1 = *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'*) = {}
using *well-scoped-inst* **by** *auto*
end

The following locale assembles all these conditions.

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

end

4.3 Abstract_Rules_To_Incredible

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

Incredible-Deduction
Abstract-Rules
begin

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

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

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

type-synonym (*'form*, *'var*) *in-port* = (*'form*, *'var*) *antecedent*
type-synonym *'form reg-out-port* = *'form*
type-synonym *'form hyp* = *'form*
datatype (*'form*, *'var*) *out-port* = *Reg 'form reg-out-port* | *Hyp 'form hyp* (*'form*, *'var*) *in-port*
type-synonym (*'v*, *'form*, *'var*) *edge'* = ((*'v* × (*'form*, *'var*) *out-port*) × (*'v* × (*'form*, *'var*) *in-port*))

context *Abstract-Task*
begin
definition *nodes* :: (*'form*, *'rule*) *graph-node stream* **where**
nodes = *Helper* ## *shift* (*map Assumption assumptions*) (*shift* (*map Conclusion conclusions*) (*smap Rule rules*))

lemma *Helper-in-nodes*[*simp*]:
Helper ∈ *sset nodes* **by** (*simp add*: *nodes-def*)
lemma *Assumption-in-nodes*[*simp*]:
Assumption a ∈ *sset nodes* ⟷ *a* ∈ *set assumptions* **by** (*auto simp add*: *nodes-def stream.set-map*)
lemma *Conclusion-in-nodes*[*simp*]:
Conclusion c ∈ *sset nodes* ⟷ *c* ∈ *set conclusions* **by** (*auto simp add*: *nodes-def stream.set-map*)
lemma *Rule-in-nodes*[*simp*]:
Rule r ∈ *sset nodes* ⟷ *r* ∈ *sset rules* **by** (*auto simp add*: *nodes-def stream.set-map*)

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

definition *outPortsRule* **where**
outPortsRule r = *ffUnion* ((λ *a*. (λ *h*. *Hyp h a*) |'| *a-hyps a*) |'| *f-antecedent r*) |∪| *Reg* |'| *f-consequent r*

lemma *Reg-in-outPortsRule*[*simp*]: *Reg c* |∈| *outPortsRule r* ⟷ *c* |∈| *f-consequent r*
by (*auto simp add*: *outPortsRule-def ffUnion.rep-eq*)
lemma *Hyp-in-outPortsRule*[*simp*]: *Hyp h c* |∈| *outPortsRule r* ⟷ *c* |∈| *f-antecedent r* ∧ *h* |∈| *a-hyps c*
by (*auto simp add*: *outPortsRule-def ffUnion.rep-eq*)

fun *outPorts* **where**
outPorts (*Rule r*) = *outPortsRule r*

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

fun labelsIn where
  labelsIn - p = a-conc p

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

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

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


sublocale Labeled-Signature nodes inPorts outPorts hyps labelsIn labelsOut
proof(standard,goal-cases)
  case (1 n p1 p2)
  thus ?case by(induction n p1 rule: hyps.induct) (auto  split: if-splits)
qed

lemma hyps-for-conclusion[simp]: hyps-for (Conclusion n) p = {||}
  using hyps-for-subset by auto
lemma hyps-for-Helper[simp]: hyps-for Helper p = {||}
  using hyps-for-subset by auto
lemma hyps-for-Rule[simp]: ip |∈| f-antecedent r ⟹ hyps-for (Rule r) ip = (λ h. Hyp h ip) |`| a-hyps ip
  by (auto elim!: hyps.elims split: if-splits)
end
```

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

```
locale Tasked-Proof-Graph =
  Abstract-Task freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP  antecedent
consequent rules assumptions conclusions  +
  Scoped-Proof-Graph freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP  inPorts
outPorts nodeOf hyps nodes vertices labelsIn labelsOut vidx inst edges local-vars
  for freshenLC :: nat ⇒ 'var ⇒ 'var
   and renameLCs :: ('var ⇒ 'var) ⇒ 'form ⇒ 'form
   and lconsts :: 'form ⇒ 'var set
   and closed :: 'form ⇒ bool
   and subst :: 'subst ⇒ 'form ⇒ 'form
   and subst-lconsts :: 'subst ⇒ 'var set
   and subst-renameLCs :: ('var ⇒ 'var) ⇒ ('subst ⇒ 'subst)
   and anyP :: 'form

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

   and assumptions :: 'form list
   and conclusions :: 'form list

   and vertices :: 'vertex fset
```

and *nodeOf* :: *'vertex* ⇒ (*'form*, *'rule*) *graph-node*
and *edges* :: (*'vertex*, *'form*, *'var*) *edge'* *set*
and *vidx* :: *'vertex* ⇒ *nat*
and *inst* :: *'vertex* ⇒ *'subst* +
assumes *conclusions-present*: *set* (*map Conclusion conclusions*) ⊆ *nodeOf* ‘ *fset vertices*

end

5 Natural Deduction

5.1 Natural_Deduction

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

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

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

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

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

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

locale *ND-Rules-Inst =*
Abstract-Formulas freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP
for *freshenLC* :: *nat ⇒ '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 +*

fixes *nat-rule* :: *'rule ⇒ 'form ⇒ ('form, 'var) antecedent fset ⇒ 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 ⇒ 'form entailment ⇒ 'form entailment fset ⇒ 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*))
((λ*ant*. ((λ*p*. *subst s* (*freshen a p*)) |‘| *a-hyps ant* |∪| Γ ⊢ *subst s* (*freshen a* (*a-conc ant*)))) |‘| *ants*)
|*eff Cut* (Γ ⊢ *c*′) {| (Γ ⊢ *c*′)|}

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

sublocale *RuleSystem-Defs* **where**
eff = *eff* **and** *rules* = *Cut* ## *Axiom* ## *smap NatRule rules*.
end

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

context *Abstract-Task*
begin
inductive *natEff-Inst* **where**
c ∈ *set* (*consequent r*) ⟹ *natEff-Inst* (*r*,*c*) *c* (*f-antecedent r*)

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

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

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

definition *solved* **where**
solved ⟷ (∀ *c*. *c* |∈| *conc-forms* ⟶ (∃ Γ *t*. *fst* (*root t*) = (Γ ⊢ *c*) ∧ Γ |⊆| *ass-forms* ∧ *wf t* ∧ *tfinite t*))
end

end

6 Correctness

6.1 Incredible_Correctness

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

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

context *Tasked-Proof-Graph*
begin

definition *adjacentTo* :: *'vertex* ⇒ (*'form*, *'var*) *in-port* ⇒ (*'vertex* × (*'form*, *'var*) *out-port*) **where**
 adjacentTo v p = (*SOME ps*. (*ps*, (*v*,*p*)) ∈ *edges*)

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

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

inductive-set *global-assms'* :: *'var itself* ⇒ *'form set* **for** *i* **where**
  v |∈| *vertices* ⟹ *nodeOf v* = *Assumption p* ⟹ *labelAtOut v* (*Reg p*) ∈ *global-assms' i*

lemma *finite-global-assms'*: *finite* (*global-assms' i*)
proof−
  have *finite* (*fset vertices*) **by** (*rule finite-fset*)
  moreover
  have *global-assms' i* ⊆ (λ *v*. *case nodeOf v of Assumption p* ⇒ *labelAtOut v* (*Reg p*)) ‘ *fset vertices*
    by (*force simp add*: *global-assms'.simps image-iff*)
  ultimately
  show *?thesis* **by** (*rule finite-surj*)
qed

context includes *fset.lifting*
begin
  lift-definition *global-assms* :: *'var itself* ⇒ *'form fset* **is** *global-assms'* **by** (*rule finite-global-assms'*)
  lemmas *global-assmsI* = *global-assms'.intros*[*Transfer.transferred*]
  lemmas *global-assms-simps* = *global-assms'.simps*[*Transfer.transferred*]
end

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

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

lemma *hyps-alongE*[*consumes 1*, *case-names Hyp Assumption*]:
assumes *f* |∈| *hyps-along pth*
obtains *v p h* **where** (*v*,*p*) ∈ *snd* ' *set pth* **and** *f* = *labelAtOut v h* **and** *h* |∈| *hyps-for* (*nodeOf v*) *p*
| *v pf* **where** *v* |∈| *vertices* **and** *nodeOf v* = *Assumption pf f* = *labelAtOut v* (*Reg pf*)
using *assms*
apply (*auto simp add*: *ffUnion.rep-eq global-assms-simps*)
apply (*metis image-iff snd-conv*)
done

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

primcorec *tree* :: '*vertex* ⇒ ('*form*, '*var*) *in-port* ⇒ ('*vertex*, '*form*, '*var*) *edge*' *list* ⇒ (('*form entailment*), ('*rule* × '*form*) *NatRule*) *dtree* **where**
root (*tree v p pth*) =
((*hyps-along* ((*adjacentTo v p*,(*v*,*p*))#*pth*) ⊢ *labelAtIn v p*),
(*case adjacentTo v p of* (*v*', *p*') ⇒ *toNatRule v*' *p*'
))
| *cont* (*tree v p pth*) =
(*case adjacentTo v p of* (*v*', *p*') ⇒
(*if isReg v*' *p*' *then* ((λ *p*''. *tree v*' *p*'' ((*adjacentTo v p*,(*v*,*p*))#*pth*)) |'| *inPorts* (*nodeOf v*')) *else* {||}
))

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

lemma *out-port-cases*[*consumes 1*, *case-names Assumption Hyp Rule Helper*]:
assumes *p* |∈| *outPorts n*
obtains
a **where** *n* = *Assumption a* **and** *p* = *Reg a*
| *r h c* **where** *n* = *Rule r* **and** *p* = *Hyp h c*
| *r f* **where** *n* = *Rule r* **and** *p* = *Reg f*
| *n* = *Helper* **and** *p* = *Reg anyP*
using *assms* **by** (*atomize-elim*, *cases p*; *cases n*) *auto*

lemma *hyps-for-fimage*: *hyps-for* (*Rule r*) *x* = (*if x* |∈| *f-antecedent r then* (λ *f*. *Hyp f x*) |'| (*a-hyps x*) *else* {||})
apply (*rule fset-eqI*)
apply (*rename-tac p*')
apply (*case-tac p*')
apply (*auto simp add*: *split*: *if-splits out-port.splits*)
done

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

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

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

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

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

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

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

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

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


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

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

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

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

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

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

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

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

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

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

have *v''* ≠ *v'* **by** (*metis distinct.simps*(*2*) *fst-conv image-eqI list.set-map*)
hence *vidx v''* ≠ *vidx v'* **using** ‹*v'* |∈| *vertices*› ‹*v''* |∈| *vertices*› **by** (*meson vidx-inj inj-onD*)
hence *freshenLC* (*vidx v'*) ' *a-fresh ant* ∩ *freshenLC* (*vidx v''*) ' *lconsts* (*labelsOut* (*nodeOf v''*) *h''*) = {}**by** *auto*
moreover
have *lconsts f* ⊆ *lconsts* (*freshen* (*vidx v''*) (*labelsOut* (*nodeOf v''*) *h''*)) ∪ *subst-lconsts* (*inst v''*) **using** ‹*f* = -›
by (*simp add*: *labelAtOut-def fv-subst*)
ultimately
show *?thesis*
by (*fastforce simp add*: *lconsts-freshen*)
next
case (*Assumption v pf*)
hence *f* = *subst* (*inst v*) (*freshen* (*vidx v*) *pf*) **by** (*simp add*: *labelAtOut-def*)
moreover
from *Assumption* **have** *Assumption pf* ∈ *sset nodes* **using** *valid-nodes* **by** *auto*
hence *pf* ∈ *set assumptions* **unfolding** *nodes-def* **by** (*auto simp add*: *stream.set-map*)
hence *closed pf* **by** (*rule assumptions-closed*)
ultimately
have *lconsts f* = {} **by** (*simp add*: *closed-no-lconsts lconsts-freshen subst-closed freshen-closed*)
thus *?thesis* **by** *simp*
qed
next
fix *ant*
assume *ant* |∈| *f-antecedent r*
from ‹*v'* |∈| *vertices*› ‹*ant* |∈| *f-antecedent r*›
have *valid-in-port* (*v'*,*ant*) **by** (*simp add*: *Rule*)
moreover
note ‹*v'* |∈| *vertices*›
moreover
hence *v'* ∉ *scope* (*v'*, *ant*) **by** (*rule scopes-not-refl*)
ultimately
have *freshenLC* (*vidx v'*) ' *local-vars* (*nodeOf v'*) *ant* ∩ *subst-lconsts* (*inst v'*) = {}
by (*rule out-of-scope*)
thus *freshenLC* (*vidx v'*) ' *a-fresh ant* ∩ *subst-lconsts* (*inst v'*) = {} **by** *simp*
qed
also
have *subst* (*inst v'*) (*freshen* (*vidx v'*) *f*) = *labelAtOut v' p'* **using** *Rule* **by** (*simp add*: *labelAtOut-def*)
also
note ‹*labelAtOut v' p'* = *labelAtIn v p*›
also
have *?ants* = ((λ*x*. (*extra-assms* (*v'*,*x*) |∪| *hyps-along ?pth'* ⊢ *labelAtIn v' x*)) |'| *f-antecedent r*)
by (*rule fimage-cong*[*OF refl*])
(*auto simp add*: *labelAtIn-def labelAtOut-def Rule hyps-for-fimage ffUnion.rep-eq*)
finally
have *eff* (*NatRule* (*r*, *f*))
(*?*Γ, *labelAtIn v p*)
((λ*x*. *extra-assms* (*v'*,*x*) |∪| *?*Γ ⊢ *labelAtIn v' x*) |'| *f-antecedent r*).
}
moreover

{ **fix** *x*
assume *x* |∈| *cont ?t*
then obtain *a* **where** *x* = *tree v' a ?pth'* **and** *a* |∈| *f-antecedent r*
by (*auto simp add*: *Rule*)
note *this*(*1*)
moreover

from ‹v' |∈| *vertices*› ‹*a* |∈| *f-antecedent r*›
have *valid-in-port* (v',*a*) **by** (*simp add*: *Rule*)
moreover

note ‹*terminal-path* v' *t ?pth*$'$›
ultimately

have ∃ *v p pth*. *x* = *tree v p pth* ∧ *valid-in-port* (*v*,*p*) ∧ *terminal-path v t pth*
by *blast*
}
ultimately

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

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

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

from ‹v' |∈| *vertices*›
have *valid-in-port* (v',*plain-ant anyP*) **by** (*simp add*: *Helper*)
moreover

note ‹*terminal-path* v' *t ?pth*$'$›
ultimately

have ∃ *v p pth*. *x* = *tree v p pth* ∧ *valid-in-port* (*v*,*p*) ∧ *terminal-path v t pth*
by *blast*
}
ultimately

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

lemma *global-in-ass*: *global-assms TYPE*($'$*var*) |⊆| *ass-forms*
proof
fix *x*
assume *x* |∈| *global-assms TYPE*($'$*var*)
then obtain *v pf* **where** *v* |∈| *vertices* **and** *nodeOf v* = *Assumption pf* **and** *x* = *labelAtOut v* (*Reg pf*)
by (*auto simp add*: *global-assms-simps*)

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

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

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


lemma finite-tree-edge-tree:
  tfinite (tree v p pth) ⟷ tfinite (edge-tree v p)
proof−
  have map-tree (λ -. ())  (tree v p pth) = map-tree (λ -. ()) (edge-tree v p)
   by(coinduction arbitrary: v p pth)
      (fastforce simp add: tree.map-sel rel-fset-def rel-set-def split: prod.split out-port.split graph-node.split option.split)
  thus ?thesis by (metis tfinite-map-tree)
qed

coinductive forbidden-path :: 'vertex ⇒ ('vertex, 'form, 'var) edge' stream ⇒ bool   where
    forbidden-path: ((v₁,p₁),(v₂,p₂)) ∈ edges ⟹ hyps (nodeOf v₁) p₁ = None ⟹ forbidden-path v₁ pth ⟹
forbidden-path v₂ (((v₁,p₁),(v₂,p₂))##pth)

lemma path-is-forbidden:
  assumes valid-in-port (v,p)
  assumes ipath (edge-tree v p) es
  shows forbidden-path v es
using assms
proof(coinduction arbitrary: v p es)
  case forbidden-path

  let ?es' = stl es

  from forbidden-path(2)
  obtain t' where root (edge-tree v p) = shd es and t' |∈| cont (edge-tree v p) and ipath t' ?es'
    by rule blast
```

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

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

from *e* **have** *p′* |∈| *outPorts* (*nodeOf v′*) **using** *valid-edges* **by** *auto*
thus *?case*
proof(*cases rule*: *out-port-cases*)
  case *Hyp*
  with ‹*t′* |∈| *cont* (*edge-tree v p*)›
  have *False* **by** *auto*
  thus *?thesis*..
next
  case *Assumption*
  with ‹*t′* |∈| *cont* (*edge-tree v p*)›
  have *False* **by** *auto*
  thus *?thesis*..
next
  case (*Rule r f*)
  from ‹*t′* |∈| *cont* (*edge-tree v p*)› *Rule*
  obtain *a* **where** [*simp*]: *t′* = *edge-tree v′ a* **and** *a* |∈| *f-antecedent r* **by** *auto*

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

  have *?e* ∈ *edges* **using** *e* **by** *simp*
  moreover

  from ‹*p′* = *Reg f*› ‹*nodeOf v′* = *Rule r*›
  have *hyps* (*nodeOf v′*) *p′* = *None* **by** *simp*
  moreover

  from *e valid-edges* **have** *v′* |∈| *vertices* **by** *auto*
  with ‹*nodeOf v′* = *Rule r*› ‹*a* |∈| *f-antecedent r*›
  have *valid-in-port* (*v′*, *a*) **by** *simp*
  moreover

  have *ipath* (*edge-tree v′ a*) *?es′* **using** ‹*ipath t′* -› **by** *simp*
  ultimately

  show *?thesis* **by** *metis*
next
  case *Helper*
  from ‹*t′* |∈| *cont* (*edge-tree v p*)› *Helper*
  have [*simp*]: *t′* = *edge-tree v′* (*plain-ant anyP*) **by** *simp*

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

  have *?e* ∈ *edges* **using** *e* **by** *simp*
  moreover

from ‹$p' = Reg\ anyP$› ‹$nodeOf\ v' = Helper$›
have *hyps* (*nodeOf* v') $p' = None$ **by** *simp*
moreover

from *e valid-edges* **have** v' |∈| *vertices* **by** *auto*
with ‹$nodeOf\ v' = Helper$›
have *valid-in-port* (v', *plain-ant anyP*) **by** *simp*
moreover

have *ipath* (*edge-tree* v' (*plain-ant anyP*)) *?es'* **using** ‹*ipath t' -*› **by** *simp*
ultimately

show *?thesis* **by** *metis*
qed
qed

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

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

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

theorem *finite-tree*:
assumes *valid-in-port* (*v*,*p*)
assumes *terminal-vertex v*
shows *tfinite* (*tree v p pth*)
proof(*rule ccontr*)
let *?n* = *Suc* (*fcard vertices*)
assume ¬ *tfinite* (*tree v p pth*)
hence ¬ *tfinite* (*edge-tree v p*) **unfolding** *finite-tree-edge-tree*.
then obtain *es* :: (*'vertex*, *'form*, *'var*) *edge'* *stream*
where *ipath* (*edge-tree v p*) *es* **using** *Konig* **by** *blast*
with ‹*valid-in-port* (*v*,*p*)›
have *forbidden-path v es* **by** (*rule path-is-forbidden*)
from *forbidden-path-prefix-is-path*[*OF this*] *forbidden-path-prefix-is-hyp-free*[*OF this*]
obtain v' **where** *path* v' *v* (*rev* (*stake ?n es*)) **and** *hyps-free* (*rev* (*stake ?n es*))
by *blast*
from *this* ‹*terminal-vertex v*›
have *terminal-path* v' *v* (*rev* (*stake ?n es*)) **by** (*rule terminal-pathI*)

hence *length* (*rev* (*stake ?n es*)) ≤ *fcard vertices*
  by (*rule hyps-free-limited*)
thus *False* **by** *simp*
qed

The main result of this theory.

theorem *solved*
unfolding *solved-def*
proof(*intro ballI allI conjI impI*)
  fix *c*
  assume *c* |∈| *conc-forms*
  hence *c* ∈ *set conclusions* **by** (*auto simp add*: *conc-forms-def*)
  from *this*(*1*) *conclusions-present*
  obtain *v* **where** *v* |∈| *vertices* **and** *nodeOf v* = *Conclusion c*
    by *auto*

  have *valid-in-port* (*v*, (*plain-ant c*))
    using ‹*v* |∈| *vertices*› ‹*nodeOf* - = - › **by** *simp*

  have *terminal-vertex v* **using** ‹*v* |∈| *vertices*› ‹*nodeOf v* = *Conclusion c*› **by** *auto*

  let *?t* = *tree v* (*plain-ant c*) []

  have *fst* (*root ?t*) = (*global-assms TYPE*(′*var*), *c*)
    using ‹*c* ∈ *set conclusions*› ‹*nodeOf* - = -›
   by (*auto simp add*: *labelAtIn-def conclusions-closed closed-no-lconsts freshen-def rename-closed subst-closed*)
  moreover

  have *global-assms TYPE*(′*var*) |⊆| *ass-forms* **by** (*rule global-in-ass*)
  moreover

  from ‹*terminal-vertex v*›
  have *terminal-path v v* [] **by** (*rule terminal-path-empty*)
  with ‹*valid-in-port* (*v*, (*plain-ant c*))›
  have *wf ?t* **by** (*rule wf-tree*)
  moreover

  from ‹*valid-in-port* (*v*, *plain-ant c*)› ‹*terminal-vertex v*›
  have *tfinite ?t* **by** (*rule finite-tree*)
  ultimately

  show ∃Γ *t*. *fst* (*root t*) = (Γ ⊢ *c*) ∧ Γ |⊆| *ass-forms* ∧ *wf t* ∧ *tfinite t* **by** *blast*
qed

end

end

7 Completeness

7.1 Incredible_Trees

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

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

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

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

abbreviation *INode n p i s ants* ≡ *RNode* (*I n p i s*) *ants*
abbreviation *HNode i s ants* ≡ *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* ⇒ *nat* ⇒ *'subst* ⇒ *'form entailment* ⇒ *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* ⇒ (*'form*,*'rule*,*'subst*,*'var*) *itree* ⇒ *'form entailment* ⇒ *bool*

for *fc*
where
iwf: ⟦
n ∈ *sset nodes*;
Reg p |∈| *outPorts n*;
list-all2 (λ *ip t*. *iwf fc t* ((λ *h* . *subst s* (*freshen i* (*labelsOut n h*))) |'| *hyps-for n ip* |∪| Γ ⊢ *subst s* (*freshen i* (*labelsIn n ip*))))
(*inPorts′ n*) *ants*;
fc n i s (Γ ⊢ *c*);
c = *subst s* (*freshen i p*)
⟧ ⟹ *iwf fc* (*INode n p i s ants*) (Γ ⊢ *c*)
| *iwfH*: ⟦
c |∉| *ass-forms*;
c |∈| Γ;
c = *subst s* (*freshen i anyP*)
⟧ ⟹ *iwf fc* (*HNode i s* []) (Γ ⊢ *c*)

lemma *iwf-subst-freshen-outPort*:
iwf lc ts ent ⟹
snd ent = *subst* (*iSubst ts*) (*freshen* (*iAnnot ts*) (*iOutPort ts*))
by (*auto elim*: *iwf.cases*)

definition *all-local-vars* :: (*′form*, *′rule*) *graph-node* ⇒ *′var set* **where**
all-local-vars n = ⋃(*local-vars n* ' *fset* (*inPorts n*))

lemma *all-local-vars-Helper*[*simp*]:
all-local-vars Helper = {}
unfolding *all-local-vars-def* **by** *simp*

lemma *all-local-vars-Assumption*[*simp*]:
all-local-vars (*Assumption c*) = {}
unfolding *all-local-vars-def* **by** *simp*

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

inductive *local-fresh-check* :: (*′form*, *′rule*, *′subst*) *fresh-check* **where**
⟦⋀ *f*. *f* |∈| Γ ⟹ *freshenLC i* ' (*all-local-vars n*) ∩ *lconsts f* = {};
freshenLC i ' (*all-local-vars n*) ∩ *subst-lconsts s* = {}
⟧ ⟹ *local-fresh-check n i s* (Γ ⊢ *c*)

abbreviation *local-iwf* ≡ *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**
no-fresh-check n i s (Γ ⊢ *c*)

abbreviation *plain-iwf* ≡ *iwf no-fresh-check*

fun *isHNode* **where**
isHNode (*HNode* - - -) = *True*
|*isHNode* - = *False*

lemma *iwf-edge-match*:
assumes *iwf fc t ent*
assumes *is*@[*i*] ∈ *it-paths t*
shows *subst* (*iSubst* (*tree-at t* (*is*@[*i*]))) (*freshen* (*iAnnot* (*tree-at t* (*is*@[*i*]))) (*iOutPort* (*tree-at t* (*is*@[*i*]))))

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

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

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

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

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

lemma *iNodeOf-tree-at*:
iwf fc t ent ⟹ *is* ∈ *it-paths t* ⟹ *iNodeOf* (*tree-at t is*) ∈ *sset nodes*
by (*induction arbitrary*: *is rule*: *iwf.induct*)
(*auto 4 4 elim*!: *it-paths-RNodeE dest*: *list-all2-lengthD list-all2-nthD2*)

lemma *iwf-outPort*:
assumes *iwf fc t ent*
assumes *is* ∈ *it-paths t*
shows *Reg* (*iOutPort* (*tree-at t is*)) |∈| *outPorts* (*iNodeOf* (*tree-at t is*))
using *assms*

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

inductive-set *hyps-along* **for** *t is* **where**
prefix (*is'*@[*i*]) *is* ⟹
i < *length* (*inPorts'* (*iNodeOf* (*tree-at t is'*))) ⟹
hyps (*iNodeOf* (*tree-at t is'*)) *h* = *Some* (*inPorts'* (*iNodeOf* (*tree-at t is'*)) ! *i*) ⟹
subst (*iSubst* (*tree-at t is'*)) (*freshen* (*iAnnot* (*tree-at t is'*)) (*labelsOut* (*iNodeOf* (*tree-at t is'*)) *h*)) ∈ *hyps-along t is*

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

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

lemma *hyps-along-Cons*:
assumes *iwf fc t ent*
assumes *i*#*is* ∈ *it-paths t*
shows *hyps-along t* (*i*#*is*) =
(λ*h*. *subst* (*iSubst t*) (*freshen* (*iAnnot t*) (*labelsOut* (*iNodeOf t*) *h*))) ' *fset* (*hyps-for* (*iNodeOf t*) (*inPorts'* (*iNodeOf t*) ! *i*))
∪ *hyps-along* (*iAnts t* ! *i*) *is* (**is** *?S1* = *?S2* ∪ *?S3*)
proof−
from *assms*
have *i* < *length* (*iAnts t*) **and** *is* ∈ *it-paths* (*iAnts t* ! *i*)
by (*auto elim*: *it-paths-ConsE*)
let *?t'* = *iAnts t* ! *i*

show *?thesis*
proof (*rule*; *rule*)
fix *x*
assume *x* ∈ *hyps-along t* (*i* # *is*)
then obtain *is' i' h* **where**
prefix (*is'*@[*i'*]) (*i*#*is*)
and *i'* < *length* (*inPorts'* (*iNodeOf* (*tree-at t is'*)))
and *hyps* (*iNodeOf* (*tree-at t is'*)) *h* = *Some* (*inPorts'* (*iNodeOf* (*tree-at t is'*)) ! *i'*)
and [*simp*]: *x* = *subst* (*iSubst* (*tree-at t is'*)) (*freshen* (*iAnnot* (*tree-at t is'*)) (*labelsOut* (*iNodeOf* (*tree-at t is'*)) *h*))
by (*auto elim*!: *hyps-along.cases*)
from *this*(*1*)
show *x* ∈ *?S2* ∪ *?S3*
proof(*cases rule*: *prefix-app-Cons-elim*)
assume *is'* = [] **and** *i'* = *i*
with ‹*hyps* (*iNodeOf* (*tree-at t is'*)) *h* = *Some* -›
have *x* ∈ *?S2* **by** *auto*
thus *?thesis*..
next
fix *is''*
assume [*simp*]: *is'* = *i* # *is''* **and** *prefix* (*is''* @ [*i'*]) *is*
have *tree-at t is'* = *tree-at ?t' is''* **by** *simp*

note ‹*prefix* (*is''* @ [*i'*]) *is*›
‹*i'* < *length* (*inPorts'* (*iNodeOf* (*tree-at t is'*)))›

‹hyps (iNodeOf (tree-at t is′)) h = Some (inPorts′ (iNodeOf (tree-at t is′)) ! i′)›
from *this*[*unfolded* ‹tree-at t is′ = tree-at ?t′ is′′›]
have *subst* (*iSubst* (*tree-at* (*iAnts* *t* ! *i*) *is′′*)) (*freshen* (*iAnnot* (*tree-at* (*iAnts* *t* ! *i*) *is′′*)) (*labelsOut* (*iNodeOf* (*tree-at* (*iAnts* *t* ! *i*) *is′′*)) *h*))
∈ *hyps-along* (*iAnts* *t* ! *i*) *is* **by** (*rule hyps-along.intros*)
hence *x* ∈ *?S3* **by** *simp*
thus *?thesis*..
qed
next
fix *x*
assume *x* ∈ *?S2* ∪ *?S3*
thus *x* ∈ *?S1*
proof
have *prefix* ([]@[*i*]) (*i*#*is*) **by** *simp*
moreover
from ‹iwf - t -›
have *length* (*iAnts* *t*) ≤ *length* (*inPorts′* (*iNodeOf* (*tree-at* *t* [])))
by *cases* (*auto dest*: *list-all2-lengthD*)
with ‹i < -›
have *i* < *length* (*inPorts′* (*iNodeOf* (*tree-at* *t* []))) **by** *simp*
moreover
assume *x* ∈ *?S2*
then obtain *h* **where** *h* |∈| *hyps-for* (*iNodeOf* *t*) (*inPorts′* (*iNodeOf* *t*) ! *i*)
and [*simp*]: *x* = *subst* (*iSubst* *t*) (*freshen* (*iAnnot* *t*) (*labelsOut* (*iNodeOf* *t*) *h*)) **by** *auto*
from *this*(*1*)
have *hyps* (*iNodeOf* (*tree-at* *t* [])) *h* = *Some* (*inPorts′* (*iNodeOf* (*tree-at* *t* [])) ! *i*) **by** *simp*
ultimately
have *subst* (*iSubst* (*tree-at* *t* [])) (*freshen* (*iAnnot* (*tree-at* *t* [])) (*labelsOut* (*iNodeOf* (*tree-at* *t* [])) *h*)) ∈ *hyps-along* *t* (*i* # *is*)
by (*rule hyps-along.intros*)
thus *x* ∈ *hyps-along* *t* (*i* # *is*) **by** *simp*
next
assume *x* ∈ *?S3*
thus *x* ∈ *?S1*
apply (*auto simp add*: *hyps-along.simps*)
apply (*rule-tac* *x* = *i*#*is′* **in** *exI*)
apply *auto*
done
qed
qed
qed

lemma *iwf-hyps-exist*:
assumes *iwf lc it ent*
assumes *is* ∈ *it-paths it*
assumes *tree-at it is* = (*HNode i s ants′*)
assumes *fst ent* |⊆| *ass-forms*
shows *subst s* (*freshen i anyP*) ∈ *hyps-along it is*
proof−
from *assms*(*1*,*2*,*3*)
have *subst s* (*freshen i anyP*) ∈ *hyps-along it is*
∨ *subst s* (*freshen i anyP*) |∈| *fst ent*
∧ *subst s* (*freshen i anyP*) |∉| *ass-forms*
proof(*induction arbitrary*: *is rule*: *iwf.induct*)
case (*iwf n p s′ a′* Γ *ants c is*)

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

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

    show ?case
    proof(cases is)
      case Nil
      with ‹tree-at (INode n p a' s' ants) is = HNode i s ants'›
      show ?thesis by auto
    next
      case (Cons i' is')
      with ‹is ∈ it-paths (INode n p a' s' ants)›
      have i' < length ants and is' ∈ it-paths (ants ! i')
        by (auto elim: it-paths-ConsE)

      let ?Γ' = (λh. subst s' (freshen a' (labelsOut n h))) |`| hyps-for n (inPorts' n ! i')

      from ‹tree-at (INode n p a' s' ants) is = HNode i s ants'›
      have tree-at (ants ! i') is' = HNode i s ants' using Cons by simp

      from  iwf.IH ‹i' < length ants›  ‹is' ∈ it-paths (ants ! i')› this
      have  subst s (freshen i anyP) ∈ hyps-along (ants ! i') is'
        ∨ subst s (freshen i anyP) |∈| ?Γ' |∪| Γ ∧ subst s (freshen i anyP) |∉| ass-forms
        by (auto dest: list-all2-nthD2)
      moreover
      from  ‹is ∈ it-paths (INode n p a' s' ants)›
      have hyps-along (INode n p a' s' ants) is = fset ?Γ' ∪ hyps-along (ants ! i') is'
        using ‹is = -›
        by (simp add: hyps-along-Cons[OF ‹iwf lc (INode n p a' s' ants) (Γ ⊢ c)›])
      ultimately
      show ?thesis by auto
    qed
  next
    case (iwfH c  Γ s' i' is)
    hence [simp]: is = [] i' = i s' = s by simp-all
    from ‹c = subst s' (freshen i' anyP)› ‹c |∈| Γ› ‹c |∉| ass-forms›
    show ?case by simp
  qed
  with assms(4)
  show ?thesis by blast
qed

definition hyp-port-for' :: ('form, 'rule, 'subst, 'var) itree ⇒ nat list ⇒ 'form ⇒ nat list × nat × ('form, 'var)
out-port where
  hyp-port-for' t is f = (SOME x.
   (case x of (is', i, h) ⇒
      prefix (is' @ [i]) is ∧
      i < length (inPorts' (iNodeOf (tree-at t is'))) ∧
      hyps (iNodeOf (tree-at t is')) h =  Some (inPorts' (iNodeOf (tree-at t is')) ! i) ∧
      f = subst (iSubst  (tree-at t is')) (freshen (iAnnot (tree-at t is')) (labelsOut (iNodeOf (tree-at t is')) h))
   ))

lemma hyp-port-for-spec':
  assumes f ∈ hyps-along t is
  shows (case hyp-port-for' t is f of (is', i, h) ⇒
      prefix (is' @ [i]) is ∧
      i < length (inPorts' (iNodeOf (tree-at t is'))) ∧
      hyps (iNodeOf (tree-at t is')) h =  Some (inPorts' (iNodeOf (tree-at t is')) ! i) ∧
```

f = *subst* (*iSubst* (*tree-at t is'*)) (*freshen* (*iAnnot* (*tree-at t is'*)) (*labelsOut* (*iNodeOf* (*tree-at t is'*)) *h*)))
using *assms* **unfolding** *hyps-along.simps hyp-port-for'-def* **by** −(*rule someI-ex*, *blast*)

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

lemma *hyp-port-prefix*:
assumes *f* ∈ *hyps-along t is*
shows *prefix* (*hyp-port-path-for t is f*@[*hyp-port-i-for t is f*]) *is*
using *hyp-port-for-spec'*[*OF assms*] **unfolding** *hyp-port-path-for-def hyp-port-i-for-def* **by** *auto*

lemma *hyp-port-strict-prefix*:
assumes *f* ∈ *hyps-along t is*
shows *strict-prefix* (*hyp-port-path-for t is f*) *is*
using *hyp-port-prefix*[*OF assms*] **by** (*simp add*: *strict-prefixI' prefix-order.dual-order.strict-trans1*)

lemma *hyp-port-it-paths*:
assumes *is* ∈ *it-paths t*
assumes *f* ∈ *hyps-along t is*
shows *hyp-port-path-for t is f* ∈ *it-paths t*
using *assms* **by** (*rule it-paths-strict-prefix*[*OF - hyp-port-strict-prefix*])

lemma *hyp-port-hyps*:
assumes *f* ∈ *hyps-along t is*
shows *hyps* (*iNodeOf* (*tree-at t* (*hyp-port-path-for t is f*))) (*hyp-port-h-for t is f*) = *Some* (*inPorts'* (*iNodeOf* (*tree-at t* (*hyp-port-path-for t is f*))) ! *hyp-port-i-for t is f*)
using *hyp-port-for-spec'*[*OF assms*] **unfolding** *hyp-port-path-for-def hyp-port-i-for-def hyp-port-h-for-def* **by** *auto*

lemma *hyp-port-outPort*:
assumes *f* ∈ *hyps-along t is*
shows (*hyp-port-h-for t is f*) |∈| *outPorts* (*iNodeOf* (*tree-at t* (*hyp-port-path-for t is f*)))
using *hyps-correct*[*OF hyp-port-hyps*[*OF assms*]]**..**

lemma *hyp-port-eq*:
assumes *f* ∈ *hyps-along t is*
shows *f* = *subst* (*iSubst* (*tree-at t* (*hyp-port-path-for t is f*))) (*freshen* (*iAnnot* (*tree-at t* (*hyp-port-path-for t is f*))) (*labelsOut* (*iNodeOf* (*tree-at t* (*hyp-port-path-for t is f*))) (*hyp-port-h-for t is f*)))
using *hyp-port-for-spec'*[*OF assms*] **unfolding** *hyp-port-path-for-def hyp-port-i-for-def hyp-port-h-for-def* **by** *auto*

definition *isidx* :: *nat list* ⇒ *nat* **where** *isidx xs* = *to-nat* (*Some xs*)
definition *v-away* :: *nat* **where** *v-away* = *to-nat* (*None* :: *nat list option*)
lemma *isidx-inj*[*simp*]: *isidx xs* = *isidx ys* ⟷ *xs* = *ys*
unfolding *isidx-def* **by** *simp*
lemma *isidx-v-away*[*simp*]: *isidx xs* ≠ *v-away*
unfolding *isidx-def v-away-def* **by** *simp*

definition *mapWithIndex* **where** *mapWithIndex f xs* = *map* (λ (*i*,*t*) . *f i t*) (*List.enumerate 0 xs*)
lemma *mapWithIndex-cong* [*fundef-cong*]:

xs = *ys* ⟹ (⋀*x i*. *x* ∈ *set ys* ⟹ *f i x* = *g i x*) ⟹ *mapWithIndex f xs* = *mapWithIndex g ys*
unfolding *mapWithIndex-def* **by** (*auto simp add*: *in-set-enumerate-eq*)

lemma *mapWithIndex-Nil*[*simp*]: *mapWithIndex f* [] = []
unfolding *mapWithIndex-def* **by** *simp*

lemma *length-mapWithIndex*[*simp*]: *length* (*mapWithIndex f xs*) = *length xs*
unfolding *mapWithIndex-def* **by** *simp*

lemma *nth-mapWithIndex*[*simp*]: *i* < *length xs* ⟹ *mapWithIndex f xs* ! *i* = *f i* (*xs* ! *i*)
unfolding *mapWithIndex-def* **by** (*auto simp add*: *nth-enumerate-eq*)

lemma *list-all2-mapWithIndex2E*:
assumes *list-all2 P as bs*
assumes ⋀ *i a b* . *i* < *length bs* ⟹ *P a b* ⟹ *Q a* (*f i b*)
shows *list-all2 Q as* (*mapWithIndex f bs*)
using *assms*(*1*)
by (*auto simp add*: *list-all2-conv-all-nth mapWithIndex-def nth-enumerate-eq intro*: *assms*(*2*) *split*: *prod.split*)

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

fun *globalize-node* :: *nat list* ⇒ ('*var* ⇒ '*var*) ⇒ ('*form*,'*rule*,'*subst*,'*var*) *itnode* ⇒ ('*form*,'*rule*,'*subst*,'*var*) *itnode* **where**
globalize-node is f (*I n p i s*) = *I n p* (*isidx is*) (*subst-renameLCs f s*)
| *globalize-node is f* (*H i s*) = *H* (*isidx is*) (*subst-renameLCs f s*)

fun *globalize* :: *nat list* ⇒ ('*var* ⇒ '*var*) ⇒ ('*form*,'*rule*,'*subst*,'*var*) *itree* ⇒ ('*form*,'*rule*,'*subst*,'*var*) *itree* **where**
globalize is f (*RNode r ants*) = *RNode*
(*globalize-node is f r*)
(*mapWithIndex* (λ *i*' *t*.
globalize (*is*@[*i*'])
(*rerename* (*a-fresh* (*inPorts*' (*iNodeOf* (*RNode r ants*)) ! *i*'))
(*iAnnot* (*RNode r ants*)) (*isidx is*) *f*)
t
) *ants*)

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

lemma *iAnnot-globalize*:
assumes *is*' ∈ *it-paths* (*globalize is f t*)
shows *iAnnot* (*tree-at* (*globalize is f t*) *is*') = *isidx* (*is*@*is*')
using *assms*
by (*induction t arbitrary*: *f is is*') (*auto elim*!: *it-paths-RNodeE*)

lemma *all-local-consts-listed*':
assumes *n* ∈ *sset nodes*
assumes *p* |∈| *inPorts n*
shows *lconsts* (*a-conc p*) ∪ (⋃(*lconsts* ‘ *fset* (*a-hyps p*))) ⊆ *a-fresh p*
using *assms*
by (*auto simp add*: *nodes-def stream.set-map lconsts-anyP closed-no-lconsts conclusions-closed f-antecedent-def dest*!: *all-local-consts-listed*)

lemma *no-local-consts-in-consequences*':
n ∈ *sset nodes* ⟹ *Reg p* |∈| *outPorts n* ⟹ *lconsts p* = {}
using *no-local-consts-in-consequences*

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

lemma *iwf-globalize*:
assumes *local-iwf t* (Γ ⊢ *c*)
shows *plain-iwf* (*globalize is f t*) (*renameLCs f* |'| Γ ⊢ *renameLCs f c*)
using *assms*
proof (*induction t* Γ ⊢ *c arbitrary*: *is f* Γ *c rule*: *iwf.induct*)
case (*iwf n p s i* Γ *ants c is f*)

note ‹*n* ∈ *sset nodes*›
moreover
note ‹*Reg p* |∈| *outPorts n*›
moreover
{ **fix** *i'*
let *?V* = *a-fresh* (*inPorts' n* ! *i'*)
let *?f'* = *rerename ?V i* (*isidx is*) *f*
let *?t* = *globalize* (*is* @ [*i'*]) *?f'* (*ants* ! *i'*)
let *?ip* = *inPorts' n* ! *i'*
let *?Γ'* = (λ*h*. *subst* (*subst-renameLCs f s*) (*freshen* (*isidx is*) (*labelsOut n h*))) |'| *hyps-for n ?ip*
let *?c'* = *subst* (*subst-renameLCs f s*) (*freshen* (*isidx is*) (*labelsIn n ?ip*))

assume *i'* < *length* (*inPorts' n*)
hence (*inPorts' n* ! *i'*) |∈| *inPorts n* **by** (*simp add*: *inPorts-fset-of*)

from ‹*i'* < *length* (*inPorts' n*)›
have *subset-V*: *?V* ⊆ *all-local-vars n*
unfolding *all-local-vars-def*
by (*auto simp add*: *inPorts-fset-of set-conv-nth*)

from ‹*local-fresh-check n i s* (Γ ⊢ *c*)›
have *freshenLC i* ' *all-local-vars n* ∩ *subst-lconsts s* = {}
by (*rule local-fresh-check.cases*) *simp*
hence *freshenLC i* ' *?V* ∩ *subst-lconsts s* = {}
using *subset-V* **by** *auto*
hence *rerename-subst*: *subst-renameLCs ?f' s* = *subst-renameLCs f s*
by (*rule rerename-subst-noop*)

from *all-local-consts-listed'*[*OF* ‹ *n* ∈ *sset nodes*› ‹(*inPorts' n* ! *i'*) |∈| *inPorts n*›]
have *subset-conc*: *lconsts* (*a-conc* (*inPorts' n* ! *i'*)) ⊆ *?V*
and *subset-hyp'*: ⋀ *hyp* . *hyp* |∈| *a-hyps* (*inPorts' n* ! *i'*) ⟹ *lconsts hyp* ⊆ *?V*
by *auto*

from *List.list-all2-nthD*[*OF* ‹*list-all2 - - -*› ‹*i'* < *length* (*inPorts' n*)›,*simplified*]
have *plain-iwf ?t*
(*renameLCs ?f'* |'| ((λ*h*. *subst s* (*freshen i* (*labelsOut n h*))) |'| *hyps-for n ?ip* |∪| Γ) ⊢
renameLCs ?f' (*subst s* (*freshen i* (*a-conc ?ip*))))
by *simp*
also have *renameLCs ?f'* |'| ((λ*h*. *subst s* (*freshen i* (*labelsOut n h*))) |'| *hyps-for n ?ip* |∪| Γ)
= (λ*x*. *subst* (*subst-renameLCs ?f' s*) (*renameLCs ?f'* (*freshen i* (*labelsOut n x*)))) |'| *hyps-for n ?ip* |∪| *renameLCs ?f'* |'| Γ
by (*simp add*: *fimage-fimage fimage-funion comp-def rename-subst*)
also have *renameLCs ?f'* |'| Γ = *renameLCs f* |'| Γ
proof(*rule fimage-cong*[*OF refl*])
fix *x*
assume *x* |∈| Γ
with ‹*local-fresh-check n i s* (Γ ⊢ *c*)›
have *freshenLC i* ' *all-local-vars n* ∩ *lconsts x* = {}

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

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

by (*rule iwf.intros*(*1*))
next
case (*iwfH c* Γ *s i is f*)
from ‹*c* |∉| *ass-forms*›
have *renameLCs f c* |∉| *ass-forms*
using *assumptions-closed closed-no-lconsts lconsts-renameLCs rename-closed* **by** *fastforce*
moreover
from ‹*c* |∈| Γ›
have *renameLCs f c* |∈| *renameLCs f* |'| Γ **by** *auto*
moreover
from ‹*c* = *subst s* (*freshen i anyP*)›
have *renameLCs f c* = *subst* (*subst-renameLCs f s*) (*freshen* (*isidx is*) *anyP*)
by (*metis freshen-closed lconsts-anyP rename-closed rename-subst*)
ultimately
show *plain-iwf* (*globalize is f* (*HNode i s* [])) (*renameLCs f* |'| Γ ⊢ *renameLCs f c*)
unfolding *globalize.simps globalize-node.simps mapWithIndex-Nil Let-def*
by (*rule iwf.intros*(*2*))
qed

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

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

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

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

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

definition *fresh-at-path* **where**
fresh-at-path t is = ⋃(*fresh-at t* ' *set* (*prefixes is*))

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

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

lemma *globalize-local-consts*:
  assumes *is′* ∈ *it-paths* (*globalize is f t*)
  shows *subst-lconsts* (*iSubst* (*tree-at* (*globalize is f t*) *is′*)) ⊆
    fresh-at-path (*globalize is f t*) *is′* ∪ *range f*
  using *assms*
  apply (*induction is f t arbitrary*: *is′ rule*:*globalize.induct*)
  apply (*rename-tac is f r ants is′*)
  apply (*case-tac r*)
   apply (*auto simp add*: *subst-lconsts-subst-renameLCs elim*!: *it-paths-RNodeE*)
   apply (*solves* ‹*force dest*!: *subsetD*[*OF range-rerename*]›)
  apply (*solves* ‹*force dest*!: *subsetD*[*OF range-rerename*]›)
  done

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

end

7.2 Build_Incredible_Tree

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

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

lemma *image-eq-to-f*:
  assumes *f1* ' *S1* = *f2* ' *S2*
  obtains *f* **where** ⋀ *x*. *x* ∈ *S2* ⟹ *f x* ∈ *S1* ∧ *f1* (*f x*) = *f2 x*
proof (*atomize-elim*)
  from *assms*
  have ∀ *x*. *x* ∈ *S2* ⟶ (∃ *y*. *y* ∈ *S1* ∧ *f1 y* = *f2 x*) **by** (*metis image-iff*)
  thus ∃ *f*. ∀ *x*. *x* ∈ *S2* ⟶ *f x* ∈ *S1* ∧ *f1* (*f x*) = *f2 x* **by** *metis*
qed

context includes *fset.lifting*

begin
lemma *fimage-eq-to-f*:
  assumes *f1* |'| *S1* = *f2* |'| *S2*
  obtains *f* **where** ⋀ *x*. *x* |∈| *S2* ⟹ *f x* |∈| *S1* ∧ *f1* (*f x*) = *f2 x*
using *assms* **apply** *transfer* **using** *image-eq-to-f* **by** *metis*
end

context *Abstract-Task*
begin

lemma *build-local-iwf*:
  fixes *t* :: ('*form entailment* × ('*rule* × '*form*) *NatRule*) *tree*
  assumes *tfinite t*
  assumes *wf t*
  shows ∃ *it*. *local-iwf it* (*fst* (*root t*))
using *assms*
proof(*induction*)
  case (*tfinite t*)
  from ‹*wf t*›
  have *snd* (*root t*) ∈ *R* **using** *wf.simps* **by** *blast*

  from ‹*wf t*›
  have *eff* (*snd* (*root t*)) (*fst* (*root t*)) ((*fst* ∘ *root*) |'| *cont t*) **using** *wf.simps* **by** *blast*

  from ‹*wf t*›
  have ⋀ *t*′. *t*′ |∈| *cont t* ⟹ *wf t*′ **using** *wf.simps* **by** *blast*
  hence *IH*: ⋀ Γ′ *t*′. *t*′ |∈| *cont t* ⟹ (∃ *it*′. *local-iwf it*′ (*fst* (*root t*′))) **using** *tfinite*(*2*) **by** *blast*
  then obtain *its* **where** *its*: ⋀ *t*′. *t*′ |∈| *cont t* ⟹ *local-iwf* (*its t*′) (*fst* (*root t*′)) **by** *metis*

  from ‹*eff* - - -›
  show *?case*
  proof(*cases rule*: *eff.cases*[*case-names Axiom NatRule Cut*])
  case (*Axiom c* Γ)
    show *?thesis*
    proof (*cases c* |∈| *ass-forms*)
      case *True*
      then have *c* ∈ *set assumptions* **by** (*auto simp add*: *ass-forms-def*)

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

      from ‹*c* ∈ *set assumptions*›
      have *local-iwf ?it* (Γ ⊢ *c*)
        by (*auto intro*!: *iwf local-fresh-check.intros*)

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

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

      from ‹*c* |∈| Γ› *False*
      have *local-iwf ?it* (Γ ⊢ *c*) **by** (*auto intro*: *iwfH*)
      thus *?thesis* **unfolding** *Axiom*..
    qed
  next

case (*NatRule rule c ants* Γ *i s*)
from ‹*natEff-Inst rule c ants*›
have *snd rule* = *c* **and** [*simp*]: *ants* = *f-antecedent* (*fst rule*) **and** *c* ∈ *set* (*consequent* (*fst rule*))
by (*auto simp add*: *natEff-Inst.simps*)

from ‹(*fst* ∘ *root*) |'| *cont t* = (λ*ant*. (λ*p*. *subst s* (*freshen i p*)) |'| *a-hyps ant* |∪| Γ ⊢ *subst s* (*freshen i* (*a-conc ant*))) |'| *ants*›
obtain *to-t* **where** ⋀ *ant*. *ant* |∈| *ants* ⟹ *to-t ant* |∈| *cont t* ∧ (*fst* ∘ *root*) (*to-t ant*) = ((λ*p*. *subst s* (*freshen i p*)) |'| *a-hyps ant* |∪| Γ ⊢ *subst s* (*freshen i* (*a-conc ant*)))
by (*rule fimage-eq-to-f*) (*rule that*)
hence *to-t-in-cont*: ⋀ *ant*. *ant* |∈| *ants* ⟹ *to-t ant* |∈| *cont t*
and *to-t-root*: ⋀ *ant*. *ant* |∈| *ants* ⟹ *fst* (*root* (*to-t ant*)) = ((λ*p*. *subst s* (*freshen i p*)) |'| *a-hyps ant* |∪| Γ ⊢ *subst s* (*freshen i* (*a-conc ant*)))
by *auto*

let *?ants'* = *map* (λ *ant*. *its* (*to-t ant*)) (*antecedent* (*fst rule*))
let *?it* = *INode* (*Rule* (*fst rule*)) *c i s ?ants'* :: ('*form*, '*rule*, '*subst*, '*var*) *itree*

from ‹*snd* (*root t*) ∈ *R*›
have *fst rule* ∈ *sset rules*
unfolding *NatRule*
by (*auto simp add*: *stream.set-map n-rules-def no-empty-conclusions*)
moreover
from ‹*c* ∈ *set* (*consequent* (*fst rule*))›
have *c* |∈| *f-consequent* (*fst rule*) **by** (*simp add*: *f-consequent-def*)
moreover
{ fix *ant*
assume *ant* ∈ *set* (*antecedent* (*fst rule*))
hence *ant* |∈| *ants* **by** (*simp add*: *f-antecedent-def*)
from *its*[*OF to-t-in-cont*[*OF this*]]
have *local-iwf* (*its* (*to-t ant*)) (*fst* (*root* (*to-t ant*)))**.**
also have *fst* (*root* (*to-t ant*)) =
((λ*p*. *subst s* (*freshen i p*)) |'| *a-hyps ant* |∪| Γ ⊢ *subst s* (*freshen i* (*a-conc ant*)))
by (*rule to-t-root*[*OF* ‹*ant* |∈| *ants*›])
also have . . . =
((λ*h*. *subst s* (*freshen i* (*labelsOut* (*Rule* (*fst rule*)) *h*))) |'| *hyps-for* (*Rule* (*fst rule*)) *ant* |∪| Γ ⊢ *subst s* (*freshen i* (*a-conc ant*)))
using ‹*ant* |∈| *ants*›
by *auto*
finally
have *local-iwf* (*its* (*to-t ant*))
((λ*h*. *subst s* (*freshen i* (*labelsOut* (*Rule* (*fst rule*)) *h*))) |'| *hyps-for* (*Rule* (*fst rule*)) *ant* |∪| Γ ⊢ *subst s* (*freshen i* (*a-conc ant*)))**.**
}
moreover
from *NatRule*(*5*,*6*)
have *local-fresh-check* (*Rule* (*fst rule*)) *i s* (Γ ⊢ *subst s* (*freshen i c*))
by (*fastforce intro*!: *local-fresh-check.intros simp add*: *all-local-vars-def*)
ultimately
have *local-iwf ?it* ((Γ ⊢ *subst s* (*freshen i c*)))
by (*intro iwf*) (*auto simp add*: *list-all2-map2 list-all2-same*)
thus *?thesis* **unfolding** *NatRule***..**
next
case (*Cut* Γ *con*)
obtain *s* **where** *subst s anyP* = *con* **by** *atomize-elim* (*rule anyP-is-any*)
hence [*simp*]: *subst s* (*freshen undefined anyP*) = *con* **by** (*simp add*: *lconsts-anyP freshen-closed*)

from ‹(*fst* ∘ *root*) |‘| *cont t* = {|Γ ⊢ *con*|}›
obtain *t'* **where** *t'* |∈| *cont t* **and** [*simp*]: *fst* (*root t'*) = (Γ ⊢ *con*)
by (*cases cont t*) *auto*

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

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

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

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

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

7.3 Incredible_Completeness

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

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

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

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

Let us get our hand on concrete trees.

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

ts c = (*SOME t*. *snd* (*fst* (*root t*)) = *c* ∧ *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*)
  unfolding *atomize-conj conj-assoc ts-def*
  apply (*rule someI-ex*)
  using *solved assms*
  by (*force simp add*: *solved-def*)

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

lemma *iwf-it*:
  assumes *c* ∈ *set conclusions*
  shows *plain-iwf* (*it′ c*) (*fst* (*root* (*ts c*)))
  using *assms*
  apply (*auto simp add*: *ts-conc conclusions-closed intro*!: *iwf-globalize′ iwf-to-it ts-finite ts-wf*)
  by (*meson assumptions-closed fset-mp mem-ass-forms mem-conc-forms ts-context*)

definition *vertices* :: *′form vertex fset* **where**
  vertices = *Abs-fset* (*Union* (*set* (*map* (λ *c*. *insert* (*c*, []) ((λ *p*. (*c*, *0* # *p*)) ‘ (*it-paths* (*it′ c*)))) *conclusions*)))

lemma *mem-vertices*: *v* |∈| *vertices* ⟷ (*fst v* ∈ *set conclusions* ∧ (*snd v* = [] ∨ *snd v* ∈ ((#) *0*) ‘ *it-paths* (*it′* (*fst v*))))
  unfolding *vertices-def ffUnion.rep-eq*
  by (*cases v*)(*auto simp add*: *Abs-fset-inverse Bex-def*)

lemma *prefixeq-vertices*: (*c*,*is*) |∈| *vertices* ⟹ *prefix is′ is* ⟹ (*c*, *is′*) |∈| *vertices*
  by (*cases is′*) (*auto simp add*: *mem-vertices intro*!: *imageI elim*: *it-paths-prefix*)

lemma *none-vertices*[*simp*]: (*c*, []) |∈| *vertices* ⟷ *c* ∈ *set conclusions*
  by (*simp add*: *mem-vertices*)

lemma *some-vertices*[*simp*]: (*c*, *i*#*is*) |∈| *vertices* ⟷ *c* ∈ *set conclusions* ∧ *i* = *0* ∧ *is* ∈ *it-paths* (*it′ c*)
  by (*auto simp add*: *mem-vertices*)

lemma *vertices-cases*[*consumes 1*, *case-names None Some*]:
  assumes *v* |∈| *vertices*
  obtains *c* **where** *c* ∈ *set conclusions* **and** *v* = (*c*, [])
    | *c is* **where** *c* ∈ *set conclusions* **and** *is* ∈ *it-paths* (*it′ c*) **and** *v* = (*c*, *0*#*is*)
using *assms* **by** (*cases v*; *rename-tac is*; *case-tac is*; *auto*)

lemma *vertices-induct*[*consumes 1*, *case-names None Some*]:
  assumes *v* |∈| *vertices*
  assumes ⋀ *c*. *c* ∈ *set conclusions* ⟹ *P* (*c*, [])
  assumes ⋀ *c is* . *c* ∈ *set conclusions* ⟹ *is* ∈ *it-paths* (*it′ c*) ⟹ *P* (*c*, *0*#*is*)
  shows *P v*
using *assms* **by** (*cases v*; *rename-tac is*; *case-tac is*; *auto*)

fun *nodeOf* :: *′form vertex* ⇒ (*′form*, *′rule*) *graph-node* **where**
  nodeOf (*pf*, []) = *Conclusion pf*
| *nodeOf* (*pf*, *i*#*is*) = *iNodeOf* (*tree-at* (*it′ pf*) *is*)

fun *inst* **where**
inst (*c*,[]) = *empty-subst*
|*inst* (*c*, *i*#*is*) = *iSubst* (*tree-at* (*it′ c*) *is*)

lemma *terminal-is-nil*[*simp*]: *v* |∈| *vertices* ⟹ *outPorts* (*nodeOf v*) = {||} ⟷ *snd v* = []
by (*induction v rule*: *nodeOf.induct*)
(*auto elim*: *iNodeOf-outPorts*[*rotated*] *iwf-it*)

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

definition *edge-from* :: *′form* ⇒ *nat list* => (*′form vertex* × (*′form*,*′var*) *out-port*) **where**
edge-from c is = ((*c*, *0* # *is*), *Reg* (*iOutPort* (*tree-at* (*it′ c*) *is*)))

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

fun *in-port-at* :: (*′form* × *nat list*) ⇒ *nat* ⇒ (*′form*,*′var*) *in-port* **where**
in-port-at (*c*, []) - = *plain-ant c*
| *in-port-at* (*c*, -#*is*) *i* = *inPorts′* (*iNodeOf* (*tree-at* (*it′ c*) *is*)) ! *i*

definition *edge-to* :: *′form* ⇒ *nat list* => (*′form vertex* × (*′form*,*′var*) *in-port*) **where**
edge-to c is =
(*case rev is of* [] ⇒ ((*c*, []), *in-port-at* (*c*, []) *0*)
| *i*#*is* ⇒ ((*c*, *0* # (*rev is*)), *in-port-at* (*c*, (*0*#*rev is*)) *i*))

lemma *edge-to-Nil*[*simp*]: *edge-to c* [] = ((*c*, []), *plain-ant c*)
by (*simp add*: *edge-to-def*)

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

definition *edge-at* :: *′form* ⇒ *nat list* => (*′form*, *′var*) *edge′′* **where**
edge-at c is = (*edge-from c is*, *edge-to c is*)

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

lemma *hyps-exist′*:
assumes *c* ∈ *set conclusions*
assumes *is* ∈ *it-paths* (*it′ c*)
assumes *tree-at* (*it′ c*) *is* = (*HNode i s ants*)
shows *subst s* (*freshen i anyP*) ∈ *hyps-along* (*it′ c*) *is*
proof−
from *assms*(*1*)
have *plain-iwf* (*it′ c*) (*fst* (*root* (*ts c*))) **by** (*rule iwf-it*)
moreover
note *assms*(*2*,*3*)
moreover
have *fst* (*fst* (*root* (*ts c*))) |⊆| *ass-forms*
by (*simp add*: *assms*(*1*) *ts-context*)
ultimately
show *?thesis* **by** (*rule iwf-hyps-exist*)
qed

definition *hyp-edge-to* :: *′form* ⇒ *nat list* => (*′form vertex* × (*′form*,*′var*) *in-port*) **where**

hyp-edge-to c is = ((*c*, *0* # *is*), *plain-ant anyP*)

definition *hyp-edge-from* :: *'form* ⇒ *nat list* => *nat* ⇒ *'subst* ⇒ (*'form vertex* × (*'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* ⇒ *nat list* => *nat* ⇒ *'subst* ⇒ (*'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* **by** (*simp add*:*hyp-edge-at-def*)
lemma *snd-hyp-edge-at*[*simp*]:
snd (*hyp-edge-at c is n s*) = *hyp-edge-to c is* **by** (*simp add*:*hyp-edge-at-def*)

inductive-set *edges* **where**
regular-edge: *c* ∈ *set conclusions* ⟹ *is* ∈ *it-paths* (*it' c*) ⟹ *edge-at c is* ∈ *edges*
| *hyp-edge*: *c* ∈ *set conclusions* ⟹ *is* ∈ *it-paths* (*it' c*) ⟹ *tree-at* (*it' c*) *is* = *HNode n s ants* ⟹ *hyp-edge-at c is n s* ∈ *edges*

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

lemma *edge-from-valid-out-port*:
assumes *p* ∈ *it-paths* (*it' c*)
assumes *c* ∈ *set conclusions*
shows *valid-out-port* (*edge-from c p*)
using *assms*
by (*auto simp add*: *edge-from-def intro*: *iwf-outPort iwf-it*)

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

lemma *hyp-edge-from-valid-out-port*:
assumes *is* ∈ *it-paths* (*it' c*)
assumes *c* ∈ *set conclusions*
assumes *tree-at* (*it' c*) *is* = *HNode n s ants*
shows *valid-out-port* (*hyp-edge-from c is n s*)
using *assms*
by(*auto simp add*: *hyp-edge-from-def intro*: *hyp-port-outPort it-paths-strict-prefix hyp-port-strict-prefix hyps-exist'*)

lemma *hyp-edge-to-valid-in-port*:
assumes *is* ∈ *it-paths* (*it' c*)
assumes *c* ∈ *set conclusions*
assumes *tree-at* (*it' c*) *is* = *HNode n s ants*
shows *valid-in-port* (*hyp-edge-to c is*)
using *assms* **by** (*auto simp add*: *hyp-edge-to-def*)

inductive *scope'* :: *'form vertex* ⇒ (*'form*,*'var*) *in-port* ⇒ *'form* × *nat list* ⇒ *bool* **where**
  c ∈ *set conclusions* ⟹
   is' ∈ ((#) *0*) ‘ *it-paths* (*it'* *c*) ⟹
   prefix (*is*@[*i*]) *is'* ⟹
   ip = *in-port-at* (*c*,*is*) *i* ⟹
   scope' (*c*, *is*) *ip* (*c*, *is'*)

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

lemma *scope-valid*:
  scope' *v i v'* ⟹ *v'* |∈| *vertices*
by (*auto elim*: *scope-cases*)

lemma *scope-valid-inport*:
  v' |∈| *vertices* ⟹ *scope'* *v ip* *v'* ⟷ (∃ *i*. *fst v* = *fst v'* ∧ *prefix* (*snd v*@[*i*]) (*snd v'*) ∧ *ip* = *in-port-at v i*)
by (*cases v*; *cases v'*) (*auto simp add*: *scope'.simps mem-vertices*)

definition *terminal-path-from* :: *'form* ⇒ *nat list* => (*'form*, *'var*) *edge''* *list* **where**
   terminal-path-from c is = *map* (*edge-at c*) (*rev* (*prefixes is*))

lemma *terminal-path-from-Nil*[*simp*]:
  terminal-path-from c [] = [*edge-at c* []]
  by (*simp add*: *terminal-path-from-def*)

lemma *terminal-path-from-Snoc*[*simp*]:
  terminal-path-from c (*is* @ [*i*]) = *edge-at* *c* (*is*@[*i*]) # *terminal-path-from c is*
  by (*simp add*: *terminal-path-from-def*)

lemma *path-terminal-path-from*:
  c ∈ *set conclusions* ⟹
  is ∈ *it-paths* (*it'* *c*) ⟹
  path (*c*, *0* # *is*) (*c*, []) (*terminal-path-from c is*)
by (*induction is rule*: *rev-induct*)
   (*auto simp add*: *path-cons-simp intro*!: *regular-edge elim*: *it-paths-SnocE*)

lemma *edge-step*:
  assumes (((*a*, *b*), *ba*), ((*aa*, *bb*), *bc*)) ∈ *edges*
  obtains
    i **where** *a* = *aa* **and** *b* = *bb*@[*i*] **and** *bc* = *in-port-at* (*aa*,*bb*) *i* **and** *hyps* (*nodeOf* (*a*, *b*)) *ba* = *None*
  | *i* **where** *a* = *aa* **and** *prefix* (*b*@[*i*]) *bb* **and** *hyps* (*nodeOf* (*a*, *b*)) *ba* = *Some* (*in-port-at* (*a*,*b*) *i*)
using *assms*
proof(*cases rule*: *edges.cases*[*consumes 1*, *case-names Reg Hyp*])
  case (*Reg c is*)
  then obtain *i* **where** *a* = *aa* **and** *b* = *bb*@[*i*] **and** *bc* = *in-port-at* (*aa*,*bb*) *i* **and** *hyps* (*nodeOf* (*a*, *b*)) *ba* = *None*
    by (*auto elim*!: *edges.cases simp add*: *edge-at-def edge-from-def edge-to-def split*: *list.split list.split-asm*)
  thus *thesis* **by** (*rule that*)
next
  case (*Hyp c is n s*)
  let *?i* = *hyp-port-i-for* (*it'* *c*) *is* (*subst s* (*freshen n anyP*))
  from *Hyp* **have** *a* = *aa* **and** *prefix* (*b*@[*?i*]) *bb* **and**
    hyps (*nodeOf* (*a*, *b*)) *ba* = *Some* (*in-port-at* (*a*,*b*) *?i*)
  by (*auto simp add*: *edge-at-def edge-from-def edge-to-def hyp-edge-at-def hyp-edge-to-def hyp-edge-from-def*
      intro: *hyp-port-prefix hyps-exist' hyp-port-hyps*)
  thus *thesis* **by** (*rule that*)

qed

lemma *path-has-prefixes*:
  assumes *path v v′ pth*
  assumes *snd v′* = []
  assumes *prefix* (*is′* @ [*i*]) (*snd v*)
  shows ((*fst v*, *is′*), (*in-port-at* (*fst v*, *is′*) *i*)) ∈ *snd* ‘ *set pth*
  using *assms*
  by (*induction rule*: *path.induct*)(*auto elim*!: *edge-step dest*: *prefix-snocD*)

lemma *in-scope*: *valid-in-port* (*v′*, *p′*) ⟹ *v* ∈ *scope* (*v′*, *p′*) ⟷ *scope′ v′ p′ v*
proof
  assume *v* ∈ *scope* (*v′*, *p′*)
  hence *v* |∈| *vertices* **and** ⋀ *pth t*. *path v t pth* ⟹ *terminal-vertex t* ⟹ (*v′*, *p′*) ∈ *snd* ‘ *set pth*
    by (*auto simp add*: *scope.simps*)
  from *this*
  show *scope′ v′ p′ v*
  proof (*induction rule*: *vertices-induct*)
    case (*None c*)
    from *None*(*2*)[*of* (*c*, []) [], *simplified*, *OF None*(*1*)]
    have *False*.
    thus *scope′ v′ p′* (*c*, [])..
  next
    case (*Some c is*)

    from ‹*c* ∈ *set conclusions*› ‹*is* ∈ *it-paths* (*it′ c*)›
    have *path* (*c*, *0*#*is*) (*c*, []) (*terminal-path-from c is*)
      by (*rule path-terminal-path-from*)
    moreover
    from ‹*c* ∈ *set conclusions*›
    have *terminal-vertex* (*c*, []) **by** *simp*
    ultimately
    have (*v′*, *p′*) ∈ *snd* ‘ *set* (*terminal-path-from c is*)
      by (*rule Some*(*3*))
    hence (*v′*,*p′*) ∈ *set* (*map* (*edge-to c*) (*prefixes is*))
      unfolding *terminal-path-from-def* **by** *auto*
    then obtain *is′* **where** *prefix is′ is* **and** (*v′*,*p′*) = *edge-to c is′*
      by *auto*
    show *scope′ v′ p′* (*c*, *0*#*is*)
    proof(*cases is′ rule*: *rev-cases*)
      case *Nil*
      with ‹(*v′*,*p′*) = *edge-to c is′*›
      have *v′* = (*c*, []) **and** *p′* = *plain-ant c*
        by (*auto simp add*: *edge-to-def*)
      with ‹*c* ∈ *set conclusions*› ‹*is* ∈ *it-paths* (*it′ c*)›
      show *?thesis* **by** (*auto intro*!: *scope′.intros*)
    next
      case (*snoc is″ i*)
      with ‹(*v′*,*p′*) = *edge-to c is′*›
      have *v′* = (*c*, *0* # *is″*) **and** *p′* = *in-port-at v′ i*
        by (*auto simp add*: *edge-to-def*)
      with ‹*c* ∈ *set conclusions*› ‹*is* ∈ *it-paths* (*it′ c*)› ‹*prefix is′ is*›[*unfolded snoc*]
      show *?thesis*
        by (*auto intro*!: *scope′.intros*)
    qed
  qed
next

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

  from ‹scope' v' p' v›
  have (c, is) |∈| vertices unfolding ‹v = -› by (rule scope-valid)
  hence (c, is) ∈ scope ((c, is'), p')
  proof(rule scope.intros)
    fix pth t
    assume path (c,is) t pth

    assume terminal-vertex t
    hence snd t = [] by auto

    from path-has-prefixes[OF ‹path (c,is) t pth› ‹snd t = []›, simplified, OF ‹prefix (is' @ [i]) is›]
    show ((c, is'), p') ∈ snd ` set pth unfolding ‹p' = - › ‹v' = - ›.
  qed
  thus v ∈ scope (v', p') using ‹v =-› ‹v' = -› by simp
qed

sublocale Port-Graph nodes inPorts outPorts vertices nodeOf edges
proof
  show nodeOf ` fset vertices ⊆ sset nodes
    apply (auto simp add: mem-vertices)
    apply (auto simp add: stream.set-map dest: iNodeOf-tree-at[OF iwf-it])
    done
  next

  have ∀ e ∈ edges. valid-out-port (fst e) ∧ valid-in-port (snd e)
    by (auto elim!: edges.cases simp add: edge-at-def
        dest: edge-from-valid-out-port edge-to-valid-in-port
        dest: hyp-edge-from-valid-out-port hyp-edge-to-valid-in-port)

  thus ∀(ps1, ps2)∈edges. valid-out-port ps1 ∧ valid-in-port ps2 by auto
qed


sublocale Scoped-Graph nodes inPorts outPorts vertices nodeOf edges hyps..

lemma hyps-free-path-length:
  assumes path v v' pth
  assumes hyps-free pth
  shows length pth + length (snd v') = length (snd v)
using assms by induction (auto elim!: edge-step )

fun vidx :: 'form vertex ⇒ nat where
   vidx (c, [])   = isidx [fidx conc-forms c]
  |vidx (c, -#is) = iAnnot (tree-at (it' c) is)

lemma my-vidx-inj: inj-on vidx (fset vertices)
  by (rule inj-onI)
     (auto simp add:  mem-vertices iAnnot-globalize simp del: iAnnot.simps)
```

lemma *vidx-not-v-away*[*simp*]: v |∈| *vertices* ⟹ *vidx* $v \neq$ *v-away*
  by (*cases v rule*:*vidx.cases*) (*auto simp add*: *iAnnot-globalize simp del*: *iAnnot.simps*)

sublocale *Instantiation inPorts outPorts nodeOf hyps nodes edges vertices labelsIn labelsOut freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP vidx inst*
proof
  show *inj-on vidx* (*fset vertices*) **by** (*rule my-vidx-inj*)
qed

sublocale *Well-Scoped-Graph nodes inPorts outPorts vertices nodeOf edges hyps*
proof
  fix v_1 p_1 v_2 p_2 p'
  assume *assms*: ((v_1, p_1), (v_2, p_2)) ∈ *edges hyps* (*nodeOf* v_1) p_1 = *Some* p'
  from *assms*(*1*) *hyps-correct*[*OF assms*(*2*)]
  have *valid-out-port* (v_1, p_1) **and** *valid-in-port* (v_2, p_2) **and** *valid-in-port* (v_1, p') **and** v_2 |∈| *vertices*
    using *valid-edges* **by** *auto*

  from *assms*
  have ∃ *i*. *fst* v_1 = *fst* v_2 ∧ *prefix* (*snd* v_1@[*i*]) (*snd* v_2) ∧ p' = *in-port-at* v_1 *i*
    by (*cases* v_1; *cases* v_2; *auto elim*!: *edge-step*)
  hence *scope*′ v_1 p' v_2
    unfolding *scope-valid-inport*[*OF* ‹v_2 |∈| *vertices*›].
  hence v_2 ∈ *scope* (v_1, p')
    unfolding *in-scope*[*OF* ‹*valid-in-port* (v_1, p')›].
  thus (v_2, p_2) = (v_1, p') ∨ v_2 ∈ *scope* (v_1, p') **..**
qed

sublocale *Acyclic-Graph nodes inPorts outPorts vertices nodeOf edges hyps*
proof
  fix *v pth*
  assume *path v v pth* **and** *hyps-free pth*
  from *hyps-free-path-length*[*OF this*]
  show *pth* = [] **by** *simp*
qed

sublocale *Saturated-Graph nodes inPorts outPorts vertices nodeOf edges*
proof
  fix *v p*
  assume *valid-in-port* (*v*, *p*)
  thus ∃ *e*∈*edges*. *snd e* = (*v*, *p*)
  proof(*induction v*)
    fix *c cis*
    assume *valid-in-port* ((*c*, *cis*), *p*)
    hence *c* ∈ *set conclusions* **by** (*auto simp add*: *mem-vertices*)

    show ∃ *e*∈*edges*. *snd e* = ((*c*, *cis*), *p*)
    proof(*cases cis*)
      case *Nil*
      with ‹*valid-in-port* ((*c*, *cis*), *p*)›
      have [*simp*]: *p* = *plain-ant c* **by** *simp*

      have [] ∈ *it-paths* (*it*′ *c*) **by** *simp*
      with ‹*c* ∈ *set conclusions*›
      have *edge-at c* [] ∈ *edges* **by** (*rule regular-edge*)
      moreover
      have *snd* (*edge-at c* []) = ((*c*, []), *plain-ant c*)
        by (*simp add*: *edge-to-def*)

```
      ultimately
      show ?thesis by (auto simp add: Nil simp del: snd-edge-at)
    next
      case (Cons c' is)
      with ‹valid-in-port ((c, cis), p)›
      have [simp]: c' = 0 and is ∈ it-paths (it' c)
        and p |∈| inPorts (iNodeOf (tree-at (it' c) is)) by auto

      from this(3) obtain i where
        i < length (inPorts' (iNodeOf (tree-at (it' c) is))) and
        p = inPorts' (iNodeOf (tree-at (it' c) is)) ! i
          by (auto simp add: inPorts-fset-of in-set-conv-nth)

      show ?thesis
      proof (cases tree-at (it' c) is)
        case [simp]: (RNode r ants)
        show ?thesis
        proof(cases r)
          case I
          hence ¬ isHNode (tree-at (it' c) is) by simp
          from iwf-length-inPorts-not-HNode[OF iwf-it[OF ‹c ∈ set conclusions›] ‹is ∈ it-paths (it' c)› this]
               ‹i < length (inPorts' (iNodeOf (tree-at (it' c) is)))›
          have i < length (children (tree-at (it' c) is)) by simp
          with ‹is ∈ it-paths (it' c)›
          have is@[i] ∈ it-paths (it' c) by (rule it-path-SnocI)
          from ‹c ∈ set conclusions› this
          have edge-at c (is@[i]) ∈ edges by (rule regular-edge)
          moreover
          have snd (edge-at c (is@[i])) = ((c, 0 # is), inPorts' (iNodeOf (tree-at (it' c) is)) ! i)
            by (simp add: edge-to-def)
          ultimately
          show ?thesis by (auto simp add: Cons ‹p = -› simp del: snd-edge-at)
        next
          case (H n s)
          hence tree-at (it' c) is = HNode n s ants by simp
          from ‹c ∈ set conclusions› ‹is ∈ it-paths (it' c)› this
          have hyp-edge-at c is n s ∈ edges..
          moreover
          from H ‹p |∈| inPorts (iNodeOf (tree-at (it' c) is))›
          have [simp]: p = plain-ant anyP by simp

          have snd (hyp-edge-at c is n s) = ((c, 0 # is), p)
            by (simp add: hyp-edge-to-def)
          ultimately
          show ?thesis by (auto simp add: Cons simp del: snd-hyp-edge-at)
        qed
      qed
     qed
   qed
qed

sublocale Pruned-Port-Graph nodes inPorts outPorts vertices nodeOf edges
proof
  fix v
  assume v |∈| vertices
  thus ∃pth v'. path v v' pth ∧ terminal-vertex v'
  proof(induct rule: vertices-induct)
```

```
    case (None c)
    hence terminal-vertex (c,[]) by simp
    with path.intros(1)
    show ?case by blast
  next
    case (Some c is)
    hence path (c, 0 # is) (c, []) (terminal-path-from c is)
      by (rule path-terminal-path-from)
    moreover
    have terminal-vertex (c,[]) using Some(1) by simp
    ultimately
    show ?case by blast
  qed
qed

sublocale Well-Shaped-Graph  nodes inPorts outPorts vertices nodeOf edges hyps..

sublocale sol:Solution inPorts outPorts nodeOf hyps nodes vertices  labelsIn labelsOut freshenLC renameLCs
lconsts closed subst subst-lconsts subst-renameLCs anyP vidx inst edges
proof
  fix v1 p1 v2 p2
  assume ((v1, p1), (v2, p2)) ∈ edges
  thus labelAtOut v1 p1 = labelAtIn v2 p2
  proof(cases rule:edges.cases)
    case (regular-edge c is)

    from ‹((v1, p1), v2, p2) = edge-at c is›
    have (v1,p1) = edge-from c is using fst-edge-at by (metis fst-conv)
    hence [simp]: v1 = (c, 0 # is) by (simp add: edge-from-def)

    show ?thesis
    proof(cases is rule:rev-cases)
      case Nil
      let ?t' = it' c
      have labelAtOut v1 p1 = subst (iSubst ?t') (freshen (vidx v1) (iOutPort ?t'))
        using regular-edge Nil by (simp add: labelAtOut-def edge-at-def edge-from-def)
      also have vidx v1 = iAnnot ?t' by (simp add:  Nil)
      also have subst (iSubst ?t') (freshen (iAnnot ?t') (iOutPort ?t')) = snd (fst (root (ts c)))
        unfolding iwf-subst-freshen-outPort[OF iwf-it[OF ‹c ∈ set conclusions›]]..
      also have ... = c using ‹c ∈ set conclusions› by (simp add: ts-conc)
      also have ... = labelAtIn v2 p2
        using  ‹c ∈ set conclusions›  regular-edge Nil
        by (simp add: labelAtIn-def edge-at-def freshen-closed conclusions-closed closed-no-lconsts)
      finally show ?thesis.
    next
      case (snoc is' i)
      let ?t1 = tree-at (it' c) (is'@[i])
      let ?t2 = tree-at (it' c) is'
      have labelAtOut v1 p1 = subst (iSubst ?t1) (freshen (vidx v1) (iOutPort ?t1))
        using regular-edge snoc by (simp add: labelAtOut-def edge-at-def edge-from-def)
      also have vidx v1 = iAnnot ?t1 using snoc regular-edge(3) by simp
      also have subst (iSubst ?t1) (freshen (iAnnot ?t1) (iOutPort ?t1))
          = subst (iSubst ?t2) (freshen (iAnnot ?t2) (a-conc (inPorts' (iNodeOf ?t2) ! i)))
        by (rule iwf-edge-match[OF iwf-it[OF ‹c ∈ set conclusions›] ‹is ∈ it-paths (it' c)›[unfolded snoc]])
      also have iAnnot ?t2 = vidx (c, 0 # is') by simp
     also have subst (iSubst ?t2) (freshen (vidx (c, 0 # is')) (a-conc (inPorts' (iNodeOf ?t2) ! i))) = labelAtIn
v2 p2
```

using *regular-edge snoc* **by** (*simp add*: *labelAtIn-def edge-at-def*)
finally show *?thesis*.
qed
next
case (*hyp-edge c is n s ants*)
let *?f* = *subst s* (*freshen n anyP*)
let *?h* = *hyp-port-h-for* (*it′ c*) *is ?f*
let *?his* = *hyp-port-path-for* (*it′ c*) *is ?f*
let *?t1* = *tree-at* (*it′ c*) *?his*
let *?t2* = *tree-at* (*it′ c*) *is*

from ‹*c* ∈ *set conclusions*› ‹*is* ∈ *it-paths* (*it′ c*)› ‹*tree-at* (*it′ c*) *is* = *HNode n s ants*›
have *?f* ∈ *hyps-along* (*it′ c*) *is*
by (*rule hyps-exist′*)

from ‹$((v_1, p_1), v_2, p_2)$ = *hyp-edge-at c is n s*›
have (v_1,p_1) = *hyp-edge-from c is n s* **using** *fst-hyp-edge-at* **by** (*metis fst-conv*)
hence [*simp*]: v_1 = (*c*, *0* # *?his*) **by** (*simp add*: *hyp-edge-from-def*)

have *labelAtOut* v_1 p_1 = *subst* (*iSubst ?t1*) (*freshen* (*vidx* v_1) (*labelsOut* (*iNodeOf ?t1*) *?h*))
using *hyp-edge* **by** (*simp add*: *hyp-edge-at-def hyp-edge-from-def labelAtOut-def*)
also have *vidx* v_1 = *iAnnot ?t1* **by** *simp*
also have *subst* (*iSubst ?t1*) (*freshen* (*iAnnot ?t1*) (*labelsOut* (*iNodeOf ?t1*) *?h*)) = *?f* **using** ‹*?f* ∈ *hyps-along* (*it′ c*) *is*› **by** (*rule local.hyp-port-eq*[*symmetric*])
also have . . . = *subst* (*iSubst ?t2*) (*freshen* (*iAnnot ?t2*) *anyP*) **using** *hyp-edge* **by** *simp*
also have *subst* (*iSubst ?t2*) (*freshen* (*iAnnot ?t2*) *anyP*) = *labelAtIn* v_2 p_2
using *hyp-edge* **by** (*simp add*: *labelAtIn-def hyp-edge-at-def hyp-edge-to-def*)
finally show *?thesis*.
qed
qed

lemma *node-disjoint-fresh-vars*:
assumes *n* ∈ *sset nodes*
assumes *i* < *length* (*inPorts′ n*)
assumes *i′* < *length* (*inPorts′ n*)
shows *a-fresh* (*inPorts′ n* ! *i*) ∩ *a-fresh* (*inPorts′ n* ! *i′*) = {} ∨ *i* = *i′*
using *assms no-multiple-local-consts*
by (*fastforce simp add*: *nodes-def stream.set-map*)

sublocale *Well-Scoped-Instantiation freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP inPorts outPorts nodeOf hyps nodes vertices labelsIn labelsOut vidx inst edges local-vars*
proof
fix *v p var v′*
assume *valid-in-port* (*v*, *p*)
hence *v* |∈| *vertices* **by** *simp*

obtain *c is* **where** *v* = (*c*,*is*) **by** (*cases v*, *auto*)

from ‹*valid-in-port* (*v*, *p*)› ‹*v*= -›
have (*c*,*is*) |∈| *vertices* **and** *p* |∈| *inPorts* (*nodeOf* (*c*, *is*)) **by** *simp-all*
hence *c* ∈ *set conclusions* **by** (*simp add*: *mem-vertices*)

from ‹*p* |∈| -› **obtain** *i* **where**
i < *length* (*inPorts′* (*nodeOf* (*c*, *is*))) **and**
p = *inPorts′* (*nodeOf* (*c*, *is*)) ! *i* **by** (*auto simp add*: *inPorts-fset-of in-set-conv-nth*)

hence *p* = *in-port-at* (*c*, *is*) *i* **by** (*cases is*) *auto*

assume *v'* |∈| *vertices*
then obtain *c' is'* **where** *v'* = (*c'*,*is'*) **by** (*cases v'*, *auto*)

assume *var* ∈ *local-vars* (*nodeOf v*) *p*
hence *var* ∈ *a-fresh p* **by** *simp*

assume *freshenLC* (*vidx v*) *var* ∈ *subst-lconsts* (*inst v'*)
then obtain *is''* **where** *is'* = *0#is''* **and** *is''* ∈ *it-paths* (*it' c'*)
using ‹*v'* |∈| *vertices*›
by (*cases is'*) (*auto simp add*: ‹*v'*=-›)

note ‹*freshenLC* (*vidx v*) *var* ∈ *subst-lconsts* (*inst v'*)›
also
have *subst-lconsts* (*inst v'*) = *subst-lconsts* (*iSubst* (*tree-at* (*it' c'*) *is''*))
by (*simp add*: ‹*v'*=-› ‹*is'*=-›)
also
from ‹*is''* ∈ *it-paths* (*it' c'*)›
have . . . ⊆ *fresh-at-path* (*it' c'*) *is''* ∪ *range* (*freshenLC v-away*)
by (*rule globalize-local-consts*)
finally
have *freshenLC* (*vidx v*) *var* ∈ *fresh-at-path* (*it' c'*) *is''*
using ‹*v* |∈| *vertices*› **by** *auto*
then obtain *is'''* **where** *prefix is''' is''* **and** *freshenLC* (*vidx v*) *var* ∈ *fresh-at* (*it' c'*) *is'''*
unfolding *fresh-at-path-def* **by** *auto*
then obtain *i' is''''* **where** *prefix* (*is''''*@[*i'*]) *is''*
and *freshenLC* (*vidx v*) *var* ∈ *fresh-at* (*it' c'*) (*is''''*@[*i'*])
using *append-butlast-last-id*[**where** *xs* = *is'''*, *symmetric*]
apply (*cases is'''* = [])
apply (*auto simp del*: *fresh-at-snoc append-butlast-last-id*)
apply *metis*
done

from ‹*is''* ∈ *it-paths* (*it' c'*)› ‹*prefix* (*is''''*@[*i'*]) *is''*›
have (*is''''*@[*i'*]) ∈ *it-paths* (*it' c'*) **by** (*rule it-paths-prefix*)
hence *is''''* ∈ *it-paths* (*it' c'*) **using** *append-prefixD it-paths-prefix* **by** *blast*

from *this* ‹*freshenLC* (*vidx v*) *var* ∈ *fresh-at* (*it' c'*) (*is''''*@[*i'*])›
have *c* = *c'* ∧ *is* = *0* # *is''''* ∧ *var* ∈ *a-fresh* (*inPorts'* (*iNodeOf* (*tree-at* (*it' c'*) *is''''*)) ! *i'*)
unfolding *fresh-at-def'* **using** ‹*v* |∈| *vertices*› ‹*v'* |∈| *vertices*›
apply (*cases is*)
apply (*auto split*: *if-splits simp add*: *iAnnot-globalize it-paths-butlast* ‹*v*=-› ‹*v'*=-› ‹*is'*=-› *simp del*: *iAnnot.simps*)
done
hence *c'* = *c* **and** *is* = *0* # *is''''* **and** *var* ∈ *a-fresh* (*inPorts'* (*iNodeOf* (*tree-at* (*it' c'*) *is''''*)) ! *i'*) **by** *simp-all*

from ‹(*is''''*@[*i'*]) ∈ *it-paths* (*it' c'*)›
have *i'* < *length* (*inPorts'* (*nodeOf* (*c*, *is*)))
using *iwf-length-inPorts*[*OF iwf-it*[*OF* ‹*c* ∈ *set conclusions*›]]
by (*auto elim*!: *it-paths-SnocE simp add*: ‹*is*=-› ‹*c'* = -› *order.strict-trans2*)

have *nodeOf* (*c*, *is*) ∈ *sset nodes*
unfolding ‹*is* = -› ‹*c'* = -› *nodeOf.simps*
by (*rule iNodeOf-tree-at*[*OF iwf-it*[*OF* ‹*c* ∈ *set conclusions*›] ‹*is''''* ∈ *it-paths* (*it' c'*)›[*unfolded* ‹*c'* = -›]])

```
  from ‹var ∈ a-fresh (inPorts' (iNodeOf (tree-at (it' c') is''')) ! i')›
      ‹var ∈ a-fresh p› ‹p = inPorts' (nodeOf (c, is)) ! i›
      node-disjoint-fresh-vars[OF
         ‹nodeOf (c, is) ∈ sset nodes›
         ‹i < length (inPorts' (nodeOf (c, is)))› ‹i' < length (inPorts' (nodeOf (c, is)))›]
  have i' = i by (auto simp add: ‹is=-› ‹c'=c›)

  from ‹prefix (is'''@[i']) is''›
  have prefix (is @ [i']) is' by (simp add: ‹is'=-› ‹is=-›)


  from ‹c ∈ set conclusions› ‹is'' ∈ it-paths (it' c')› ‹prefix (is @ [i']) is'›
     ‹p = in-port-at (c, is) i›
  have scope' v p v'
  unfolding ‹v=-› ‹v'=-› ‹c' = -› ‹is' = -› ‹i'=-› by (auto intro: scope'.intros)
  thus v' ∈ scope (v, p) using ‹valid-in-port (v, p)› by (simp add: in-scope)
qed

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


sublocale tpg:Tasked-Proof-Graph freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs
anyP antecedent consequent rules assumptions conclusions
  vertices nodeOf edges vidx inst
proof
  show set (map Conclusion conclusions) ⊆ nodeOf ` fset vertices
  proof−
  {
    fix c
    assume c ∈ set conclusions
    hence (c, []) |∈| vertices by simp
    hence nodeOf (c, []) ∈ nodeOf ` fset vertices
      by (rule imageI)
    hence Conclusion c ∈ nodeOf ` fset vertices by simp
  } thus ?thesis by auto
  qed
qed

end

end
```

8 Instantiations

To ensure that our locale assumption are fulfillable, we instantiate them with small examples.

8.1 Propositional_Formulas

theory *Propositional-Formulas*
imports
  Abstract-Formula
  HOL−Library.Countable
  HOL−Library.Infinite-Set
  HOL−Library.Infinite-Typeclass
begin

lemma *countable-infinite-ex-bij*: ∃*f*::(*'a*::{*countable*,*infinite*}⇒*'b*::{*countable*,*infinite*}). *bij f*
proof −
  have *infinite* (*range* (*to-nat*::*'a* ⇒ *nat*))
    using *finite-imageD infinite-UNIV* **by** *blast*
  moreover have *infinite* (*range* (*to-nat*::*'b* ⇒ *nat*))
    using *finite-imageD infinite-UNIV* **by** *blast*
  ultimately have ∃*f*. *bij-betw f* (*range* (*to-nat*::*'a* ⇒ *nat*)) (*range* (*to-nat*::*'b* ⇒ *nat*))
    by (*meson bij-betw-inv bij-betw-trans bij-enumerate*)
  then obtain *f* **where** *f-def*: *bij-betw f* (*range* (*to-nat*::*'a* ⇒ *nat*)) (*range* (*to-nat*::*'b* ⇒ *nat*)) **..**
  then have *f-range-trans*: *f* ‘ (*range* (*to-nat*::*'a* ⇒ *nat*)) = *range* (*to-nat*::*'b* ⇒ *nat*)
    unfolding *bij-betw-def* **by** *simp*
  have *surj* ((*from-nat*::*nat* ⇒ *'b*) ∘ *f* ∘ (*to-nat*::*'a* ⇒ *nat*))
  proof (*rule surjI*)
    fix *a*
    obtain *b* **where** [*simp*]: *to-nat* (*a*::*'b*) = *b* **by** *blast*
    hence *b* ∈ *range* (*to-nat*::*'b* ⇒ *nat*) **by** *blast*
    with *f-range-trans* **have** *b* ∈ *f* ‘ (*range* (*to-nat*::*'a* ⇒ *nat*)) **by** *simp*
    from *imageE* [*OF this*] **obtain** *c* **where** [*simp*]:*f c* = *b* **and** *c* ∈ *range* (*to-nat*::*'a* ⇒ *nat*)
      by *auto*
    with *f-def* **have** [*simp*]: *inv-into* (*range* (*to-nat*::*'a* ⇒ *nat*)) *f b* = *c*
      by (*meson bij-betw-def inv-into-f-f*)
    then obtain *d* **where** *cd*: *from-nat c* = (*d*::*'a*) **by** *blast*
    with ‹*c* ∈ *range to-nat*› **have** [*simp*]:*to-nat d* = *c* **by** *auto*
    from ‹*to-nat a* = *b*› **have** [*simp*]: *from-nat b* = *a*
      using *from-nat-to-nat* **by** *blast*
    show (*from-nat* ∘ *f* ∘ *to-nat*) (((*from-nat*::*nat* ⇒ *'a*) ∘ *inv-into* (*range* (*to-nat*::*'a* ⇒ *nat*)) *f* ∘ (*to-nat*::*'b* ⇒ *nat*)) *a*) = *a*
      by (*clarsimp simp*: *cd*)
  qed
  moreover have *inj* ((*from-nat*::*nat* ⇒ *'b*) ∘ *f* ∘ (*to-nat*::*'a* ⇒ *nat*))
    apply (*rule injI*)
    apply *auto*
    apply (*metis bij-betw-inv-into-left f-def f-inv-into-f f-range-trans from-nat-def image-eqI rangeI to-nat-split*)
    done
  ultimately show *?thesis* **by** (*blast intro*: *bijI*)
qed

Propositional formulas are either a variable from an infinite but countable set, or a function given by a name and the arguments.

datatype (*'var*,*'cname*) *pform* =
    Var 'var::{*countable*,*infinite*}

| *Fun* (*name*:*'cname*) (*params*: (*'var*,*'cname*) *pform list*)

Substitution on and closedness of propositional formulas is straight forward.

fun *subst* :: (*'var*::{*countable*,*infinite*} ⇒ (*'var*,*'cname*) *pform*) ⇒ (*'var*,*'cname*) *pform* ⇒ (*'var*,*'cname*) *pform*
  where *subst s* (*Var v*) = *s v*
  | *subst s* (*Fun n ps*) = *Fun n* (*map* (*subst s*) *ps*)

fun *closed* :: (*'var*::{*countable*,*infinite*},*'cname*) *pform* ⇒ *bool*
  where *closed* (*Var v*) ⟷ *False*
  | *closed* (*Fun n ps*) ⟷ *list-all closed ps*

Now we can interpret *Abstract-Formulas*. As there are no locally fixed constants in propositional formulas, most of the locale parameters are dummy values

interpretation *propositional*: *Abstract-Formulas*
  — No need to freshen locally fixed constants
  curry (*SOME f. bij f*):: *nat* ⇒ *'var* ⇒ *'var*
  — also no renaming needed as there are no locally fixed constants
  λ-. *id* λ-. {}
  — closedness and substitution as defined above
  closed :: (*'var*::{*countable*,*infinite*},*'cname*) *pform* ⇒ *bool subst*
  — no substitution and renaming of locally fixed constants
  λ-. {} λ-. *id*
  — most generic formula
  Var undefined
proof
  fix *a v a' v'*
  from *countable-infinite-ex-bij* **obtain** *f* **where** *bij* (*f*::*nat* × *'var* ⇒ *'var*) **by** *blast*
  then show (*curry* (*SOME f. bij* (*f*::*nat* × *'var* ⇒ *'var*)) (*a*::*nat*) (*v*::*'var*) = *curry* (*SOME f. bij f*) (*a'*::*nat*) (*v'*::*'var*)) =
      (*a* = *a'* ∧ *v* = *v'*)
  apply (*rule someI2* [**where** *Q*=λ*f. curry f a v* = *curry f a' v'* ⟷ *a* = *a'* ∧ *v* = *v'*])
  by *auto* (*metis bij-pointE prod.inject*)+
next
  fix *f s*
  assume *closed* (*f*::(*'var*, *'cname*) *pform*)
  then show *subst s f* = *f*
  proof (*induction s f rule*: *subst.induct*)
    case (*2 s n ps*)
    thus *?case* **by** (*induction ps*) *auto*
  qed *auto*
next
  have *subst Var f* = *f* **for** *f* :: (*'var*,*'cname*) *pform*
    by (*induction f*) (*auto intro*: *map-idI*)
  then show ∃ *s*. (∀ *f. subst s* (*f*::(*'var*,*'cname*) *pform*) = *f*) ∧ {} = {}
    by (*rule-tac x*=*Var* **in** *exI*; *clarsimp*)
qed *auto*

declare *propositional.subst-lconsts-empty-subst* [*simp del*]

end

8.2 Incredible_Propositional

theory *Incredible-Propositional* **imports**
  Abstract-Rules-To-Incredible

Propositional-Formulas
begin

Our concrete interpretation with propositional logic will cover conjunction and implication as well as constant symbols. The type for variables will be *string*.

datatype *prop-funs* = *and* | *imp* | *Const string*

The rules are introduction and elimination of conjunction and implication.

datatype *prop-rule* = *andI* | *andE* | *impI* | *impE*

definition *prop-rules* :: *prop-rule stream*
  where *prop-rules* = *cycle* [*andI*, *andE*, *impI*, *impE*]

lemma *iR-prop-rules* [*simp*]: *sset prop-rules* = {*andI*, *andE*, *impI*, *impE*}
  unfolding *prop-rules-def* **by** *simp*

Just some short notation.

abbreviation *X* :: (*string*,′*a*) *pform*
  where *X* ≡ *Var* ′′*X*′′
abbreviation *Y* :: (*string*,′*a*) *pform*
  where *Y* ≡ *Var* ′′*Y*′′

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

fun *consequent* :: *prop-rule* ⇒ (*string*, *prop-funs*) *pform list*
  where *consequent andI* = [*Fun and* [*X*, *Y*]]
  | *consequent andE* = [*X*, *Y*]
  | *consequent impI* = [*Fun imp* [*X*, *Y*]]
  | *consequent impE* = [*Y*]

fun *antecedent* :: *prop-rule* ⇒ ((*string*,*prop-funs*) *pform*,*string*) *antecedent list*
  where *antecedent andI* = [*plain-ant X*, *plain-ant Y*]
  | *antecedent andE* = [*plain-ant* (*Fun and* [*X*, *Y*])]
  | *antecedent impI* = [*Antecedent* {|*X*|} *Y* {}]
  | *antecedent impE* = [*plain-ant* (*Fun imp* [*X*, *Y*]), *plain-ant X*]

interpretation *propositional*: *Abstract-Rules*
  curry (*SOME f*. *bij f*):: *nat* ⇒ *string* ⇒ *string*
  λ-. *id*
  λ-. {}
  closed :: (*string*, *prop-funs*) *pform* ⇒ *bool*
  subst
  λ-. {}
  λ-. *id*
  Var undefined
  antecedent
  consequent
  prop-rules
proof
  show ∀ *xs*∈*sset prop-rules*. *consequent xs* ≠ []
    unfolding *prop-rules-def*
    using *consequent.elims* **by** *blast*
next
  show ∀ *xs*∈*sset prop-rules*. ⋃((λ-. {}) ‘ *set* (*consequent xs*)) = {}
    by *clarsimp*

next
  fix i' *r i ia*
  assume $r \in$ *sset prop-rules*
    and $ia <$ *length* (*antecedent r*)
    and $i' <$ *length* (*antecedent r*)
  then show *a-fresh* (*antecedent r* ! *ia*) ∩ *a-fresh* (*antecedent r* ! i') = {} ∨ $ia = i'$
    by (*cases* i'; *auto*)
next
  fix *p*
  show {} ∪ ⋃((λ-. {}) ‘ *fset* (*a-hyps p*)) ⊆ *a-fresh p* **by** *clarsimp*
qed

end

8.3 Incredible_Propositional_Tasks

theory *Incredible-Propositional-Tasks*
imports
  Incredible-Completeness
  Incredible-Propositional
begin

context *ND-Rules-Inst* **begin**
lemma *eff-NatRuleI*:
  nat-rule rule c ants
    ⟹ *entail* = (Γ ⊢ *subst s* (*freshen a c*))
    ⟹ *hyps* = ((λ*ant*. ((λ*p*. *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*
  by (*drule eff.intros*(*2*)) *simp-all*
end

context *Abstract-Task* **begin**
lemma *natEff-InstI*:
  rule = (*r*,*c*)
  ⟹ *c* ∈ *set* (*consequent r*)
  ⟹ *antec* = *f-antecedent r*
  ⟹ *natEff-Inst rule c antec*
  by (*metis natEff-Inst.intros*)
end

context begin

8.3.1 Task 1.1

This is the very first task of the Incredible Proof Machine: $A \longrightarrow A$

abbreviation *A* :: (*string*,*prop-funs*) *pform*
  where *A* ≡ *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*]
by *unfold-locales simp*

Then we show, that this task has a proof within our formalization of natural deduction by giving a concrete proof tree.

lemma *task1-1.solved*
unfolding *task1-1.solved-def*
apply *clarsimp*
apply (*rule-tac x*={|*A*|} **in** *exI*)
apply *clarsimp*
apply (*rule-tac x*=*Node* ({|*A*|} ⊢ *A*, *Axiom*) {||} **in** *exI*)
apply *clarsimp*
apply (*rule conjI*)
apply (*rule task1-1.wf*)
apply (*solves clarsimp*)
apply *clarsimp*
apply (*rule task1-1.eff.intros*(*1*))
apply (*solves simp*)
apply (*solves clarsimp*)
by (*auto intro*: *tfinite.intros*)

print-locale *Vertex-Graph*
interpretation *task1-1*: *Vertex-Graph task1-1.nodes task1-1.inPorts task1-1.outPorts* {|*0*::*nat*,*1*|}
undefined(*0* := *Assumption A*, *1* := *Conclusion A*)
.

print-locale *Pre-Port-Graph*
interpretation *task1-1*: *Pre-Port-Graph task1-1.nodes task1-1.inPorts task1-1.outPorts* {|*0*::*nat*,*1*|}
undefined(*0* := *Assumption A*, *1* := *Conclusion A*)
{((*0*,*Reg A*),(*1*,*plain-ant A*))}
.

print-locale *Instantiation*
interpretation *task1-1*: *Instantiation*
task1-1.inPorts
task1-1.outPorts
undefined(*0* := *Assumption A*, *1* := *Conclusion A*)
task1-1.hyps
task1-1.nodes
{((*0*,*Reg A*),(*1*,*plain-ant A*))}
{|*0*::*nat*,*1*|}
task1-1.labelsIn
task1-1.labelsOut
curry (*SOME f. bij f*):: *nat* ⇒ *string* ⇒ *string*
λ-. id
λ-. {}

closed :: (*string*, *prop-funs*) *pform* ⇒ *bool*
subst
λ-. {}
λ-. *id*
Var undefined
id
undefined
by *unfold-locales simp*

declare *One-nat-def* [*simp del*]

lemma *path-one-edge*[*simp*]:
task1-1.path v1 v2 pth ⟷
(*v1* = *0* ∧ *v2* = *1* ∧ *pth* = [((*0*,*Reg A*),(*1*,*plain-ant A*))] ∨
pth = [] ∧ *v1* = *v2*)
apply (*cases pth*)
apply (*auto simp add*: *task1-1.path-cons-simp′*)
apply (*rename-tac list*, *case-tac list*, *auto simp add*: *task1-1.path-cons-simp′*)+
done

Finally we can also show that there is a proof graph for this task.

interpretation *Tasked-Proof-Graph*
curry (*SOME f*. *bij f*):: *nat* ⇒ *string* ⇒ *string*
λ-. *id*
λ-. {}
closed :: (*string*, *prop-funs*) *pform* ⇒ *bool*
subst
λ-. {}
λ-. *id*
Var undefined
antecedent
consequent
prop-rules
[*A*]
[*A*]
{|*0*::*nat*,*1*|}
undefined(*0* := *Assumption A*, *1* := *Conclusion A*)
{((*0*,*Reg A*),(*1*,*plain-ant A*))}
id
undefined
apply *unfold-locales*
apply (*solves simp*)
apply (*solves clarsimp*)
apply (*solves clarsimp*)
apply (*solves clarsimp*)
apply (*solves fastforce*)
apply (*solves fastforce*)
apply (*solves* ‹*clarsimp simp add*: *task1-1.labelAtOut-def task1-1.labelAtIn-def*›)
apply (*solves clarsimp*)
apply (*solves clarsimp*)
done

8.3.2 Task 2.11

This is a slightly more interesting task as it involves both our connectives: $P \wedge Q \longrightarrow R \Longrightarrow P \longrightarrow Q \longrightarrow R$

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*]]]
by *unfold-locales simp-all*

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*])
  unfolding *task2-11*.*n-rules-def* **by** (*simp add*: *prop-rules-def*)
lemma *n-andE1* [*simp*]: *n-andE1* = (*andE*, *X*)
  unfolding *task2-11*.*n-rules-def One-nat-def* **by** (*simp add*: *prop-rules-def*)
lemma *n-andE2* [*simp*]: *n-andE2* = (*andE*, *Y*)
  unfolding *task2-11*.*n-rules-def numeral-2-eq-2* **by** (*simp add*: *prop-rules-def*)
lemma *n-impI* [*simp*]: *n-impI* = (*impI*, *Fun imp* [*X*, *Y*])
  unfolding *task2-11*.*n-rules-def numeral-3-eq-3* **by** (*simp add*: *prop-rules-def*)
lemma *n-impE* [*simp*]: *n-impE* = (*impE*, *Y*)
proof −
  have *n-impE* = *task2-11*.*n-rules* !! *Suc 3* **by** *simp*
  also have ... = (*impE*, *Y*)
  unfolding *task2-11*.*n-rules-def numeral-3-eq-3* **by** (*simp add*: *prop-rules-def*)
  finally show *?thesis* **.**
qed

lemma *subst-Var-eq-id* [*simp*]: *subst Var* = *id*
  by (*rule ext*) (*induct-tac x*; *auto simp*: *map-idI*)

lemma *xy-update*: *f* = *undefined*(''*X*'' := *x*, ''*Y*'' := *y*) ⟹ *x* = *f* ''*X*'' ∧ *y* = *f* ''*Y*'' **by** *force*
lemma *y-update*: *f* = *undefined*(''*Y*'':=*y*) ⟹ *y* = *f* ''*Y*'' **by** *force*

declare *snth*.*simps*(*1*) [*simp del*]

By interpreting *Solved-Task* we show that there is a proof tree for the task. We get the existence of the proof graph for free by using the completeness theorem.

interpretation *task2-11*: *Solved-Task*
  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]]]
apply unfold-locales
  unfolding task2-11.solved-def
apply clarsimp
apply (rule-tac x={|Fun imp [Fun and [A,B],C]|} in exI)
apply clarsimp
— The actual proof tree for this task.
apply (rule-tac x=Node ({|Fun imp [Fun and [A, B], C]|} ⊢ Fun imp [A, Fun imp [B, C]],NatRule n-impI)
  {|Node ({|Fun imp [Fun and [A, B], C], A|} ⊢ Fun imp [B,C],NatRule n-impI)
    {|Node ({|Fun imp [Fun and [A, B], C], A, B|} ⊢ C,NatRule n-impE)
      {|Node ({|Fun imp [Fun and [A, B], C], A, B|} ⊢ Fun imp [Fun and [A,B], C],Axiom) {||},
        Node ({|Fun imp [Fun and [A, B], C], A, B|} ⊢ Fun and [A,B],NatRule n-andI)
          {|Node ({|Fun imp [Fun and [A, B], C], A, B|} ⊢ A,Axiom) {||},
            Node ({|Fun imp [Fun and [A, B], C], A, B|} ⊢ B,Axiom) {||}
          |}
      |}
    |}
  |} in exI)
apply clarsimp
apply (rule conjI)

 apply (rule task1-1.wf)
   apply (solves ‹clarsimp; metis n-impI snth-smap snth-sset›)
  apply clarsimp
  apply (rule task1-1.eff-NatRuleI [unfolded propositional.freshen-def, simplified]) apply simp-all[4]
    apply (rule task2-11.natEff-InstI)
      apply (solves simp)
     apply (solves simp)
    apply (solves simp)
   apply (intro conjI; simp; rule xy-update)
   apply (solves simp)
  apply (solves ‹fastforce simp: propositional.f-antecedent-def›)

 apply (rule task1-1.wf)
   apply (solves ‹clarsimp; metis n-impI snth-smap snth-sset›)
  apply clarsimp
  apply (rule task1-1.eff-NatRuleI [unfolded propositional.freshen-def, simplified]) apply simp-all[4]
    apply (rule task2-11.natEff-InstI)
      apply (solves simp)
     apply (solves simp)
    apply (solves simp)
   apply (intro conjI; simp; rule xy-update)
   apply (solves simp)
  apply (solves ‹fastforce simp: propositional.f-antecedent-def›)

 apply (rule task1-1.wf)
   apply (solves ‹clarsimp; metis n-impE snth-smap snth-sset›)
```

apply *clarsimp*
apply (*rule task1-1.eff-NatRuleI* [*unfolded propositional.freshen-def*, *simplified*, **where** *s=undefined(''Y'':=C,''X'':=Fun and* [*A*,*B*])]) **apply** *simp-all*[*4*]
apply (*rule task2-11.natEff-InstI*)
apply (*solves simp*)
apply (*solves simp*)
apply (*solves simp*)
apply (*solves* ‹*intro conjI*; *simp*›)
apply (*solves* ‹*simp add*: *propositional.f-antecedent-def*›)
apply (*erule disjE*)

apply (*auto intro*: *task1-1.wf intro*!: *task1-1.eff.intros*(*1*))[*1*]

apply (*rule task1-1.wf*)
apply (*solves* ‹*clarsimp*; *metis n-andI snth-smap snth-sset*›)
apply *clarsimp*
apply (*rule task1-1.eff-NatRuleI* [*unfolded propositional.freshen-def*, *simplified*]) **apply** *simp-all*[*4*]
apply (*rule task2-11.natEff-InstI*)
apply (*solves simp*)
apply (*solves simp*)
apply (*solves simp*)
apply (*intro conjI*; *simp*; *rule xy-update*)
apply (*solves simp*)
apply (*solves* ‹*simp add*: *propositional.f-antecedent-def*›)
apply *clarsimp*

apply (*erule disjE*)
apply (*solves* ‹*rule task1-1.wf*; *auto intro*: *task1-1.eff.intros*(*1*)›)
apply (*solves* ‹*rule task1-1.wf*; *auto intro*: *task1-1.eff.intros*(*1*)›)

by (*rule tfinite.intros*; *auto*)+

interpretation *Tasked-Proof-Graph*
curry (*SOME f. bij f*):: *nat* ⇒ *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*]]]
task2-11.vertices
task2-11.nodeOf
task2-11.edges
task2-11.vidx
task2-11.inst
by *unfold-locales*

end

end

8.4 Predicate_Formulas

theory *Predicate-Formulas*
imports
HOL−Library.Countable
HOL−Library.Infinite-Set
HOL−Eisbach.Eisbach
Abstract-Formula
begin

This theory contains an example instantiation of *Abstract-Formulas* with an formula type with local constants. It is a rather ad-hoc type that may not be very useful to work with, though.

type-synonym *var* = *nat*
type-synonym *lconst* = *nat*

We support higher order variables, in order to express ∀ *x*.*?P x*. But we stay first order, i.e. the parameters of such a variables will only be instantiated with ground terms.

datatype *form* =
Var (*var*:*var*) (*params*: *form list*)
| *LC* (*var*:*lconst*)
| *Op* (*name*:*string*) (*params*: *form list*)
| *Quant* (*name*:*string*) (*var*:*nat*) (*body*: *form*)

type-synonym *schema* = *var list* × *form*

type-synonym *subst* = (*nat* × *schema*) *list*

fun *fv* :: *form* ⇒ *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* ⇒ *var* **where**
fresh-for V = (*SOME n*. *n* ∉ *V*)

lemma *fresh-for-fresh*: *finite V* ⟹ *fresh-for V* ∉ *V*
unfolding *fresh-for-def*
apply (*rule someI2-ex*)
using *infinite-nat-iff-unbounded-le*
apply *auto*
done

Free variables

fun *fv-schema* :: *schema* ⇒ *var set* **where**
fv-schema (*ps*,*f*) = *fv f* − *set ps*

definition *fv-subst* :: *subst* ⇒ *var set* **where**
fv-subst s = ⋃(*fv-schema* ‘ *ran* (*map-of s*))

definition *fv-subst1* **where**
fv-subst1 s = ⋃(*fv* ‘ *snd* ‘ *set s*)

lemma *fv-subst-Nil*[*simp*]: *fv-subst1* [] = {}
unfolding *fv-subst1-def* **by** *auto*

Local constants, separate from free variables.

fun *lc* :: *form* ⇒ *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* ⇒ *lconst set* **where**
lc-schema (*ps*,*f*) = *lc f*

definition *lc-subst1* **where**
lc-subst1 s = ⋃(*lc* ' *snd* ' *set s*)

fun *lc-subst* :: *subst* ⇒ *lconst set* **where**
lc-subst s = ⋃(*lc-schema* ' *snd* ' *set s*)

fun *map-lc* :: (*lconst* ⇒ *lconst*) ⇒ *form* ⇒ *form* **where**
map-lc f (*Var v xs*) = *Var v* (*map* (*map-lc f*) *xs*)
| *map-lc f* (*LC n*) = *LC* (*f n*)
| *map-lc f* (*Op n xs*) = *Op n* (*map* (*map-lc f*) *xs*)
| *map-lc f* (*Quant n v f'*) = *Quant n v* (*map-lc f f'*)

lemma *fv-map-lc*[*simp*]: *fv* (*map-lc p f*) = *fv f*
by (*induction f*) *auto*

lemma *lc-map-lc*[*simp*]: *lc* (*map-lc p f*) = *p* ' *lc f*
by (*induction f*) *auto*

lemma *map-lc-map-lc*[*simp*]: *map-lc p1* (*map-lc p2 f*) = *map-lc* (*p1* ∘ *p2*) *f*
by (*induction f*) *auto*

fun *map-lc-subst1* :: (*lconst* ⇒ *lconst*) ⇒ (*var* × *form*) *list* ⇒ (*var* × *form*) *list* **where**
map-lc-subst1 f s = *map* (*apsnd* (*map-lc f*)) *s*

fun *map-lc-subst* :: (*lconst* ⇒ *lconst*) ⇒ *subst* ⇒ *subst* **where**
map-lc-subst f s = *map* (*apsnd* (*apsnd* (*map-lc f*))) *s*

lemma *map-lc-noop*[*simp*]: *lc f* = {} ⟹ *map-lc p f* = *f*
by (*induction f*) (*auto simp add*: *map-idI*)

lemma *map-lc-cong*[*cong*]: (⋀*x*. *x* ∈ *lc f* ⟹ *f1 x* = *f2 x*) ⟹ *map-lc f1 f* = *map-lc f2 f*
by (*induction f*) *auto*

lemma [*simp*]: *fv-subst1* (*map* (*apsnd* (*map-lc p*)) *s*) = *fv-subst1 s*
unfolding *fv-subst1-def*
by *auto*

lemma *map-lc-subst-cong*[*cong*]:
assumes (⋀*x*. *x* ∈ *lc-subst s* ⟹ *f1 x* = *f2 x*)
shows *map-lc-subst f1 s* = *map-lc-subst f2 s*
by (*force intro*!: *map-lc-cong assms*)

In order to make the termination checker happy, we define substitution in two stages: One that substitutes only ground terms for variables, and the real one that can substitute schematic terms (or lambda expression, if you want).

fun *subst1* :: (*var* × *form*) *list* ⇒ *form* ⇒ *form* **where**

```
  subst1 s (Var v []) = (case map-of s v of Some f ⇒ f | None ⇒ Var v [])
| subst1 s (Var v xs) = Var v xs
| subst1 s (LC n) = LC n
| subst1 s (Op n xs) = Op n (map (subst1 s) xs)
| subst1 s (Quant n v f) =
    (if v ∈ fv-subst1 s then
    (let v′ = fresh-for (fv-subst1 s)
    in Quant n v′ (subst1 ((v, Var v′ [])#s) f))
    else Quant n v (subst1 s f))

lemma subst1-Nil[simp]: subst1 [] f = f
  by (induction []::(var × form) list f rule:subst1.induct)
    (auto simp add: map-idI split: option.splits)

lemma lc-subst1: lc (subst1 s f) ⊆ lc f ∪ ⋃(lc ‘ snd ‘ set s)
  by (induction s f rule: subst1.induct)
    (auto split: option.split dest: map-of-SomeD simp add: Let-def)

lemma apsnd-def′: apsnd f = (λ(k, v). (k, f v))
  by auto

lemma map-of-map-apsnd:
  map-of (map (apsnd f) xs) = map-option f ∘ map-of xs
  unfolding apsnd-def′ by (rule map-of-map)

lemma map-lc-subst1[simp]: map-lc p (subst1 s f) = subst1 (map-lc-subst1 p s) (map-lc p f)
  apply (induction s f rule: subst1.induct)
  apply (auto split: option.splits simp add: map-of-map-apsnd Let-def)
  apply (subst subst1.simps, auto split: option.splits)[1]
  apply (subst subst1.simps, auto split: option.splits)[1]
  apply (subst subst1.simps, auto split: option.splits)[1]
  apply (subst subst1.simps, auto split: option.splits)[1]
  apply (subst subst1.simps, auto split: option.splits, simp only: Let-def map-lc.simps)[1]
  apply (subst subst1.simps, auto split: option.splits)
  done


fun subst′ :: subst ⇒ form ⇒ form where
  subst′ s (Var v xs) =
    (case map-of s v of None ⇒ (Var v (map (subst′ s) xs))
               | Some (ps,rhs) ⇒
                   if length ps = length xs
                   then subst1 (zip ps (map (subst′ s) xs)) rhs
                   else (Var v (map (subst′ s) xs)))
| subst′ s (LC n) = LC n
| subst′ s (Op n xs) = Op n (map (subst′ s) xs)
| subst′ s (Quant n v f) =
    (if v ∈ fv-subst s then
    (let v′ = fresh-for (fv-subst s)
     in Quant n v′ (subst′ ((v,([], Var v′ []))#s) f))
    else Quant n v (subst′ s f))

lemma subst′-Nil[simp]: subst′ [] f = f
  by (induction f) (auto simp add: map-idI fv-subst-def)

lemma lc-subst′: lc (subst′ s f) ⊆ lc f ∪ lc-subst s
  apply (induction s f rule: subst′.induct)
```

apply (*auto split*: *option.splits dest*: *map-of-SomeD dest*!: *subsetD*[*OF lc-subst1*] *simp add*: *fv-subst-def*)
apply (*fastforce dest*!: *set-zip-rightD*)+
done

lemma *ran-map-option-comp*[*simp*]:
ran (*map-option f* ∘ *m*) = *f* ‘ *ran m*
unfolding *comp-def* **by** (*rule ran-map-option*)

lemma *fv-schema-apsnd-map-lc*[*simp*]:
fv-schema (*apsnd* (*map-lc p*) *a*) = *fv-schema a*
by (*cases a*) *auto*

lemma *fv-subst-map-apsnd-map-lc*[*simp*]:
fv-subst (*map* (*apsnd* (*apsnd* (*map-lc p*))) *s*) = *fv-subst s*
unfolding *fv-subst-def*
by (*auto simp add*: *map-of-map-apsnd*)

lemma *map-apsnd-zip*[*simp*]: *map* (*apsnd f*) (*zip a b*) = *zip a* (*map f b*)
by (*simp add*: *apsnd-def′ zip-map2*)

lemma *map-lc-subst′*[*simp*]: *map-lc p* (*subst′ s f*) = *subst′* (*map-lc-subst p s*) (*map-lc p f*)
apply (*induction s f rule*: *subst′.induct*)
apply (*auto split*: *option.splits dest*: *map-of-SomeD simp add*: *map-of-map-apsnd Let-def*)
apply (*solves* ‹(*subst subst′.simps*, *auto split*: *option.splits*)[*1*]›)
apply (*solves* ‹(*subst subst′.simps*, *auto split*: *option.splits cong*: *map-cong*)[*1*]›)
apply (*solves* ‹(*subst subst′.simps*, *auto split*: *option.splits*)[*1*]›)
apply (*solves* ‹(*subst subst′.simps*, *auto split*: *option.splits*)[*1*]›)
apply (*solves* ‹(*subst subst′.simps*, *auto split*: *option.splits*)[*1*]›)
apply (*solves* ‹(*subst subst′.simps*, *auto split*: *option.splits*, *simp only*: *Let-def map-lc.simps*)[*1*]›)
apply (*solves* ‹(*subst subst′.simps*, *auto split*: *option.splits*)[*1*]›)
done

Since subst’ happily renames quantified variables, we have a simple wrapper that ensures that the substitution is minimal, and is empty if *f* is closed. This is a hack to support lemma *subst-noop*.

fun *subst* :: *subst* ⇒ *form* ⇒ *form* **where**
subst s f = *subst′* (*filter* (λ (*v*,*s*). *v* ∈ *fv f*) *s*) *f*

lemma *subst-Nil*[*simp*]: *subst* [] *f* = *f*
by *auto*

lemma *subst-noop*[*simp*]: *fv f* = {} ⟹ *subst s f* = *f*
by *simp*

lemma *lc-subst*: *lc* (*subst s f*) ⊆ *lc f* ∪ *lc-subst s*
by (*auto dest*: *subsetD*[*OF lc-subst′*])

lemma *lc-subst-map-lc-subst*[*simp*]: *lc-subst* (*map-lc-subst p s*) = *p* ‘ *lc-subst s*
by *force*

lemma *map-lc-subst*[*simp*]: *map-lc p* (*subst s f*) = *subst* (*map-lc-subst p s*) (*map-lc p f*)
unfolding *subst.simps*
by (*auto simp add*: *filter-map intro*!: *arg-cong*[*OF filter-cong*])

fun *closed* :: *form* ⇒ *bool* **where**
closed f ⟷ *fv f* = {} ∧ *lc f* = {}

interpretation *predicate*: *Abstract-Formulas*

curry to-nat :: *nat* ⇒ *var* ⇒ *var*
map-lc
lc
closed
subst
lc-subst
map-lc-subst
Var 0 []
apply *unfold-locales*
apply (*solves fastforce*)
apply (*solves fastforce*)
apply (*solves fastforce*)
apply (*solves fastforce*)
apply (*solves fastforce*)
apply (*solves* ‹*rule lc-subst*›)
apply (*solves fastforce*)
apply (*solves fastforce*)
apply (*solves fastforce*)
apply (*solves* ‹*metis map-lc-subst-cong*›)
apply (*solves* ‹*rule lc-subst-map-lc-subst*›)
apply (*solves simp*)
apply (*solves* ‹*rule exI*[*where x* = []], *simp*›)
apply (*solves* ‹*rename-tac f*, *rule-tac x* = [(*0*, ([],*f*))] *in exI*, *simp*›)
done

declare *predicate.subst-lconsts-empty-subst* [*simp del*]

end

8.5 Incredible_Predicate

theory *Incredible-Predicate* **imports**
Abstract-Rules-To-Incredible
Predicate-Formulas
begin

Our example interpretation with predicate logic will cover implication and the universal quantifier.

The rules are introduction and elimination of implication and universal quantifiers.

datatype *prop-rule* = *allI* | *allE* | *impI* | *impE*

definition *prop-rules* :: *prop-rule stream*
where *prop-rules* = *cycle* [*allI*, *allE*, *impI*, *impE*]

lemma *iR-prop-rules* [*simp*]: *sset prop-rules* = {*allI*, *allE*, *impI*, *impE*}
unfolding *prop-rules-def* **by** *simp*

Just some short notation.

abbreviation *X* :: *form*
where *X* ≡ *Var 10* []
abbreviation *Y* :: *form*
where *Y* ≡ *Var 11* []
abbreviation *x* :: *form*
where *x* ≡ *Var 9* []
abbreviation *t* :: *form*

where $t \equiv Var\ 13\ []$
abbreviation $P :: form \Rightarrow form$
where $P\ f \equiv Var\ 12\ [f]$
abbreviation $Q :: form \Rightarrow form$
where $Q\ f \equiv Op\ ''Q''\ [f]$
abbreviation $imp :: form \Rightarrow form \Rightarrow form$
where $imp\ f1\ f2 \equiv Op\ ''imp''\ [f1, f2]$
abbreviation $ForallX :: form \Rightarrow form$
where $ForallX\ f \equiv Quant\ ''all''\ 9\ f$

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
show ∀ *xs*∈*sset prop-rules*. *consequent xs* ≠ []
unfolding *prop-rules-def*
using *consequent.elims* **by** *blast*
next
show ∀ *xs*∈*sset prop-rules*. ⋃(*lc* ‘ *set* (*consequent xs*)) = {}
by *auto*
next
fix *i′ r i ia*
assume *r* ∈ *sset prop-rules*
and *ia* < *length* (*antecedent r*)
and *i′* < *length* (*antecedent r*)
then show *a-fresh* (*antecedent r* ! *ia*) ∩ *a-fresh* (*antecedent r* ! *i′*) = {} ∨ *ia* = *i′*
by (*cases i′*; *auto*)
next
fix *r p*
assume *r* ∈ *sset prop-rules*
and *p* ∈ *set* (*antecedent r*)
thus *lc* (*a-conc p*) ∪ ⋃(*lc* ‘ *fset* (*a-hyps p*)) ⊆ *a-fresh p* **by** *auto*

qed

end

8.6 Incredible_Predicate_Tasks

theory *Incredible-Predicate-Tasks*
imports
  Incredible-Completeness
  Incredible-Predicate
  HOL−Eisbach.Eisbach
begin

declare *One-nat-def* [*simp del*]

context *ND-Rules-Inst* **begin**
lemma *eff-NatRuleI*:
  nat-rule rule c ants
    $\Longrightarrow$ *entail* = (Γ ⊢ *subst s* (*freshen a c*))
    $\Longrightarrow$ *hyps* = ((λ*ant*. ((λ*p*. *subst s* (*freshen a p*)) |'| *a-hyps ant* |∪| Γ ⊢ *subst s* (*freshen a* (*a-conc ant*)))) |'| *ants*)
    $\Longrightarrow$ (⋀ *ant f*. *ant* |∈| *ants* $\Longrightarrow$ *f* |∈| Γ $\Longrightarrow$ *freshenLC a* ' (*a-fresh ant*) ∩ *lconsts f* = {})
    $\Longrightarrow$ (⋀ *ant*. *ant* |∈| *ants* $\Longrightarrow$ *freshenLC a* ' (*a-fresh ant*) ∩ *subst-lconsts s* = {})
    $\Longrightarrow$ *eff* (*NatRule rule*) *entail hyps*
  by (*drule eff.intros*(*2*)) *simp-all*
end

context *Abstract-Task* **begin**
lemma *natEff-InstI*:
  rule = (*r*,*c*)
  $\Longrightarrow$ *c* ∈ *set* (*consequent r*)
  $\Longrightarrow$ *antec* = *f-antecedent r*
  $\Longrightarrow$ *natEff-Inst rule c antec*
  by (*metis natEff-Inst.intros*)
end

context begin

A typical task with local constants:: $\forall x.\ Q\ x \longrightarrow Q\ x$

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

interpretation *task*: *Abstract-Task*
  curry to-nat :: *nat* ⇒ *var* ⇒ *var*
  map-lc
  lc
  closed
  subst
  lc-subst
  map-lc-subst
  Var 0 []
  antecedent
  consequent
  prop-rules
  []
  [*ForallX* (*imp* (*Q x*) (*Q x*))]
by *unfold-locales auto*

Then we show, that this task has a proof within our formalization of natural deduction by giving a concrete proof tree.

abbreviation *lx* :: *nat* **where** *lx* ≡ *to-nat* (*1*::*nat*,*0*::*nat*)

abbreviation *base-tree* :: ((*form fset* × *form*) × (*prop-rule* × *form*) *NatRule*) *tree* **where**
base-tree ≡ *Node* ({|*Q* (*LC lx*)|} ⊢ *Q* (*LC lx*), *Axiom*) {||}

abbreviation *imp-tree* :: ((*form fset* × *form*) × (*prop-rule* × *form*) *NatRule*) *tree* **where**
imp-tree ≡ *Node* ({||} ⊢ *imp* (*Q* (*LC lx*)) (*Q* (*LC lx*)), *NatRule* (*impI*, *imp X Y*)) {|*base-tree*|}

abbreviation *solution-tree* :: ((*form fset* × *form*) × (*prop-rule* × *form*) *NatRule*) *tree* **where**
solution-tree ≡ *Node* ({||} ⊢ *ForallX* (*imp* (*Q x*) (*Q x*)), *NatRule* (*allI*, *ForallX* (*P x*))) {|*imp-tree*|}

abbreviation *s1* **where** *s1* ≡ [(*12*, ([*9*], *imp* (*Q x*) (*Q x*)))]
abbreviation *s2* **where** *s2* ≡ [(*10*, ([], *Q* (*LC lx*))), (*11*, ([], *Q* (*LC lx*)))]

lemma *fv-subst-s1*[*simp*]: *fv-subst s1* = {}
by (*simp add*: *fv-subst-def*)

lemma *subst1-simps*[*simp*]:
subst s1 (*P* (*LC n*)) = *imp* (*Q* (*LC n*)) (*Q* (*LC n*))
subst s1 (*ForallX* (*P x*)) = *ForallX* (*imp* (*Q x*) (*Q x*))
by *simp-all*

lemma *subst2-simps*[*simp*]:
subst s2 X = *Q* (*LC lx*)
subst s2 Y = *Q* (*LC lx*)
subst s2 (*imp X Y*) = *imp* (*subst s2 X*) (*subst s2 Y*)
by *simp-all*

lemma *substI1*: *ForallX* (*imp* (*Q x*) (*Q x*)) = *subst s1* (*predicate.freshen 1* (*ForallX* (*P x*)))
by (*auto simp add*: *predicate.freshen-def Let-def*)

lemma *substI2*: *imp* (*Q* (*LC lx*)) (*Q* (*LC lx*)) = *subst s2* (*predicate.freshen 2* (*imp X Y*))
by (*auto simp add*: *predicate.freshen-def Let-def*)

declare *subst.simps*[*simp del*]

lemma *task.solved*
unfolding *task.solved-def*
apply *clarsimp*
apply (*rule-tac x*={||} **in** *exI*)
apply *clarsimp*
apply (*rule-tac x*=*solution-tree* **in** *exI*)
apply *clarsimp*
apply (*rule conjI*)

apply (*rule task.wf*)
apply (*solves* ‹(*auto simp add*: *stream.set-map task.n-rules-def*)[*1*]›)
apply *clarsimp*
apply (*rule task.eff-NatRuleI*)
apply (*solves* ‹*rule task.natEff-Inst.intros*;*simp*›)
apply *clarsimp*
apply (*rule conjI*)
apply (*solves* ‹*simp*›)
apply (*solves* ‹*rule substI1*›)
apply (*simp add*: *predicate.f-antecedent-def predicate.freshen-def*)

```
    apply (subst antecedent.sel(2))
    apply (solves ‹simp›)
   apply (solves ‹simp›)
  apply (solves ‹simp›)
 apply simp

 apply (rule task.wf)
   apply (solves ‹(auto simp add: stream.set-map task.n-rules-def)[1]›)
  apply clarsimp
  apply (rule task.eff-NatRuleI)
      apply (solves ‹rule task.natEff-Inst.intros; simp›)
     apply clarsimp
     apply (rule conjI)
      apply (solves ‹simp›)
     apply (solves ‹rule substI2›)
    apply (solves ‹simp add: predicate.f-antecedent-def predicate.freshen-def›)
   apply (solves ‹simp›)
  apply (solves ‹simp add: predicate.f-antecedent-def›)
 apply simp

 apply (solves ‹(auto intro: task.wf intro!: task.eff.intros(1))[1]›)
apply (solves ‹(rule tfinite.intros, simp)+›)
done

abbreviation vertices where vertices ≡ {|0::nat,1,2 |}
fun nodeOf where
  nodeOf n = [Conclusion (ForallX (imp (Q x) (Q x))),
              Rule allI,
              Rule impI] ! n

fun inst where
  inst n = [[],s1,s2] ! n


interpretation task: Vertex-Graph task.nodes task.inPorts task.outPorts vertices nodeOf.

abbreviation e1 :: (nat, form, nat) edge'
  where e1 ≡ ((1,Reg (ForallX (P x))), (0,plain-ant (ForallX (imp (Q x) (Q x)))))
abbreviation e2 :: (nat, form, nat) edge'
  where e2 ≡ ((2,Reg (imp X Y)), (1,allI-input))
abbreviation e3 :: (nat, form, nat) edge'
  where e3 ≡ ((2,Hyp X (impI-input)), (2,impI-input))
abbreviation task-edges :: (nat, form, nat) edge' set where task-edges ≡ {e1, e2, e3}


interpretation task: Scoped-Graph task.nodes task.inPorts task.outPorts vertices nodeOf task-edges task.hyps
  by standard (auto simp add: predicate.f-consequent-def predicate.f-antecedent-def)

interpretation task: Instantiation
  task.inPorts
  task.outPorts
  nodeOf
  task.hyps
  task.nodes
  task-edges
  vertices
  task.labelsIn
```

task.labelsOut
curry to-nat :: *nat* ⇒ *var* ⇒ *var*
map-lc
lc
closed
subst
lc-subst
map-lc-subst
Var 0 []
id
inst
by *unfold-locales simp*

Finally we can also show that there is a proof graph for this task.

interpretation *Well-Scoped-Graph*
task.nodes
task.inPorts
task.outPorts
vertices
nodeOf
task-edges
task.hyps
by *standard* (*auto split*: *if-splits*)

lemma *no-path-01*[*simp*]: *task.path 0 v pth* ⟷ (*pth* = [] ∧ *v* = *0*)
by (*cases pth*) (*auto simp add*: *task.path-cons-simp*)
lemma *no-path-12*[*simp*]: ¬ *task.path 1 2 pth*
by (*cases pth*) (*auto simp add*: *task.path-cons-simp*)

interpretation *Acyclic-Graph*
task.nodes
task.inPorts
task.outPorts
vertices
nodeOf
task-edges
task.hyps
proof
fix *v pth*
assume *task.path v v pth* **and** *task.hyps-free pth*
thus *pth* = []
by (*cases pth*) (*auto simp add*: *task.path-cons-simp predicate.f-antecedent-def*)
qed

interpretation *Saturated-Graph*
task.nodes
task.inPorts
task.outPorts
vertices
nodeOf
task-edges
by *standard*
(*auto simp add*: *predicate.f-consequent-def predicate.f-antecedent-def*)

interpretation *Pruned-Port-Graph*
task.nodes
task.inPorts

task.outPorts
vertices
nodeOf
task-edges
proof
fix *v*
assume *v* |∈| *vertices*
hence ∃ *pth. task.path v 0 pth*
apply *auto*
apply (*rule exI*[**where** *x* = [*e1*]], *auto simp add*: *task.path-cons-simp*)
apply (*rule exI*[**where** *x* = [*e2*,*e1*]], *auto simp add*: *task.path-cons-simp*)
done
moreover
have *task.terminal-vertex 0* **by** *auto*
ultimately
show ∃ *pth v′. task.path v v′ pth* ∧ *task.terminal-vertex v′* **by** *blast*
qed

interpretation *Well-Shaped-Graph*
task.nodes
task.inPorts
task.outPorts
vertices
nodeOf task-edges
task.hyps
..

interpretation *Solution*
task.inPorts
task.outPorts
nodeOf
task.hyps
task.nodes
vertices
task.labelsIn
task.labelsOut
curry to-nat :: *nat* ⇒ *var* ⇒ *var*
map-lc
lc
closed
subst
lc-subst
map-lc-subst
Var 0 []
id
inst
task-edges
by *standard*
(*auto simp add*: *task.labelAtOut-def task.labelAtIn-def predicate.freshen-def*, *subst antecedent.sel*, *simp*)

interpretation *Proof-Graph*
task.nodes
task.inPorts
task.outPorts
vertices
nodeOf
task-edges

task.hyps
task.labelsIn
task.labelsOut
curry to-nat :: *nat* ⇒ *var* ⇒ *var*
map-lc
lc
closed
subst
lc-subst
map-lc-subst
Var 0 []
id
inst
..

lemma *path-20*:
assumes *task.path 2 0 pth*
shows (*1*, *allI-input*) ∈ *snd* ' *set pth*
proof−
{ fix *v*
assume *task.path v 0 pth*
hence *v* = *0* ∨ *v* = *1* ∨ (*1*, *allI-input*) ∈ *snd* ' *set pth*
by (*induction v 0*::*nat pth rule*: *task.path.induct*) *auto*
}
from *this*[*OF assms*]
show *?thesis* **by** *auto*
qed

lemma *scope-21*: *2* ∈ *task.scope* (*1*, *allI-input*)
by (*auto intro*!: *task.scope.intros elim*: *path-20 simp add*: *task.outPortsRule-def predicate.f-antecedent-def predicate.f-consequent-def*)

interpretation *Scoped-Proof-Graph*
curry to-nat :: *nat* ⇒ *var* ⇒ *var*
map-lc
lc
closed
subst
lc-subst
map-lc-subst
Var 0 []
task.inPorts
task.outPorts
nodeOf
task.hyps
task.nodes
vertices
task.labelsIn
task.labelsOut
id
inst
task-edges
task.local-vars
by *standard* (*auto simp add*: *predicate.f-antecedent-def scope-21*)

interpretation *Tasked-Proof-Graph*
curry to-nat :: *nat* ⇒ *var* ⇒ *var*

map-lc
lc
closed
subst
lc-subst
map-lc-subst
Var 0 []
antecedent
consequent
prop-rules
[]
[*ForallX* (*imp* (*Q x*) (*Q x*))]
vertices
nodeOf
task-edges
id
inst
by *unfold-locales auto*

end

end