The meta theory of the Incredible Proof Machine

Joachim Breitner       Denis Lohner

September 13, 2023

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

Contents

1 Introduction                                           2

2 Auxiliary theories                                    4
   2.1 Entailment                                         4
   2.2 Indexed_FSet                                       4
   2.3 Rose_Tree                                          6
      2.3.1 The rose tree data type                        6
      2.3.2 The set of paths in a rose tree               6
      2.3.3 Indexing into a rose tree                     6

3 Abstract formulas, rules and tasks                   8
   3.1 Abstract_Formula                                  8
   3.2 Abstract_Rules                                    10

4 Incredible Proof Graphs                               13
   4.1 Incredible_Signatures                             13
   4.2 Incredible_Deduction                              14
   4.3 Abstract_Rules_To_Incredible                      23

5 Natural Deduction                                     27
   5.1 Natural_Deduction                                 27

6 Correctness                                           29
   6.1 Incredible_Correctness                            29

7 Completeness                                          39
   7.1 Incredible_Trees                                  39
   7.2 Build_Incredible_Tree                             50
   7.3 Incredible_Completeness                           53
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


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 ∆ (Γ ⊢ c) = (Γ |∪| ∆ ⊢ 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
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

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 some1_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 distinct-list-of-fset nth-list-of-fset-mem)

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

lemma indexed-fmember-is-fmember: x ∈ι s =⇒ x ∈ι s
proof (induction rule: indexed-fmember.induct)

end
lemma fmember-is-indexed-fmember:
  assumes x |∈| s
  shows ∃i. x |∈| i s
proof
  from assms have x ∈ 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 |∈| i s by (simp add: indexed-fmember.simps)
thus thesis..
qed

lemma indexed-fmember-unique: x |∈| i s =⇒ y |∈| j s =⇒ x = y ↔ 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[simp]:
t ∈ set (indexed-members s) ←→ snd t |∈| 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 s
unfolding fnth-def by (rule indexed-fmember.intros)
lemma indexed-fmember-fnth: x |∈| 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 |∈| i s)

lemma fidx-eq[simp]: x |∈| 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-injsimp: x |∈| s =⇒ y |∈| 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-on1 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 = (λx. 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 concrete 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 substitutions 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 substituted 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 subst-no-lconsts: closed f ⇒ lconsts f = { }
assumes fp-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 (renameLCs p s) (renameLCs p f)
assumes renameLCs-cong: (∀ x. x ∈ lconsts f ⇒ f1 x = f2 x) ⇒ renameLCs f1 = renameLCs f2
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)

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
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 =

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:
    \( \forall r i i' . r \in \text{sset rules} \implies i < \text{length (antecedent r)} \implies i' < \text{length (antecedent r)} \implies \)}
\[ a\text{-fresh (antecedent } r ! i) \cap a\text{-fresh (antecedent } r ! i') = \{\} \lor i = i' \]

**assumes** all-local-consts-listed:
\[ \land p.\ r \in \text{set rules} \implies p \in \text{set (antecedent } r ) \implies \text{lconst} (a\text{-conc } p) \cup (\bigcup (\text{lconst} \ fset (a\text{-hyps } p))) \subseteq a\text{-fresh } p \]

**begin**

**definition** \( f\text{-antecedent :: 'rule } \implies ('form, 'var) \text{ antecedent } f\text{set} \)

where \( f\text{-antecedent } r = f\text{set-from-list (antecedent } r ) \)

**definition** \( f\text{-consequent } r = f\text{set-from-list (consequent } r ) \)

**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 \( \implies 'var \implies 'var \)

and renameLCs :: ('var \( \implies 'var \)) \( \implies ('form \implies 'form) \)

and lconsts :: 'form \( \implies 'var \text{ set} \)

and closed :: 'form \( \implies \text{bool} \)

and subst :: 'subst \( \implies 'form \implies 'form \)

and subst-lconsts :: 'subst \( \implies 'var \text{ set} \)

and subst-renameLCs :: ('var \( \implies 'var \)) \( \implies (' subst \implies 'subst) \)

and anyP :: 'form

and antecedent :: 'rule \( \implies ('form, 'var) \text{ antecedent list} \)

and consequent :: 'rule \( \implies 'form \text{ list} \)

and rules :: 'rule \text{ stream} +

**fixes** assumptions :: 'form list

**fixes** conclusions :: 'form list

**assumes** assumptions-closed: \( \land a.\ a \in \text{set assumptions} \implies \text{closed } a \)

**assumes** conclusions-closed: \( \land c.\ c \in \text{set conclusions} \implies \text{closed } c \)

**begin**

**definition** ass-forms where ass-forms = fset-from-list assumptions

**definition** conc-forms where conc-forms = fset-from-list conclusions

**lemma** mem-ass-forms[simp]: \( a \mid\in\}\text{ ass-forms }\iff a \in \text{set assumptions} \)

by (auto simp add: ass-forms-def)

**lemma** mem-conc-forms[simp]: \( a \mid\in\}\text{ conc-forms }\iff a \in \text{set conclusions} \)

by (auto simp add: conc-forms-def)

**lemma** subst-freshen-assumptions[simp]:

**assumes** pf \( \in \text{set assumptions} \)

**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 \( \in \text{set conclusions} \)

**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 \( \in \text{set assumptions} \)
shows \( \text{subst } s (\text{freshen } a \text{ pf}) \mid \in \text{ ass-forms} \)
using \textbf{assms by simp}

\textbf{lemma} \textit{subst-freshen-in-conc-forms1}:
assumes \( pf \in \text{ set conclusions} \)
shows \( \text{subst } s (\text{freshen } a \text{ pf}) \mid \in \text{ conc-forms} \)
using \textbf{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-for_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 ∈ fset (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' (v#pth)

inductive-simps
  path-cons-simp': path v v' (e#pth)
| path-empty-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 ⇒ (λ 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-eq, induction pth)
    case Nil thus ?case by simp
    next
    case (Cons e pth)

end

15
show ?case
proof (cases snd e = ps)
  case True
  hence e # pth = [] @ [] @ pth ∧ snd e = ps ∧ ps ∉ snd ' set [] by auto
  thus ?thesis by (intro exI)
next
  case False
  have ps ∈ snd ' set pth by auto
  with Cons[2] have 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 (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 path v (fst ps) (pth1 @ e) and path (fst ps) v' pth2 and snd e = ps and ps ∉ snd ' set pth1 by (rule snd-set-split)
  from ‹path - - ‹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) by (simp add: edge-end-tup)
    show path (fst ps) v' pth2 using ‹path (edge-end e) v' pth2› 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 ⊆ set nodes
  assumes valid-edges: ∀ (ps1,ps2) ∈ edges. valid-out-port ps1 ∧ valid-in-port ps2
begin
  lemma snd-set-path-vertices: 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-vertices: path v v' pth =⇒ fst ' fst ' set pth ⊆ fset vertices
  apply (induction rule: path.induct)
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: \( \forall \ v . \ v \in \{\text{vertices}\} \implies (\exists \ pth \ v' . \ path \ v \ v' \ pth \wedge \text{terminal-vertex} \ v') \)
begin

lemma scopes-not-refl:
  assumes v \in \{\text{vertices}\}
shows v \notin \{\text{scope} \ (v,p)\}
proof (rule notI)
 assume v \in \{\text{scope} \ (v,p)\}
from pruned[OF assms] obtain pth t where terminal-vertex t and path v t pth
by atomize-elim (auto simp del: terminal-vertex.simps elim: ex-has-least-nat)
from scope-split[OF \( \forall \ v . \ v \in \{\text{scope} \ (v,p)\} \implies \\{\text{path} \ v \ t \ pth\} \implies \text{terminal-vertex} \ t \) obtain pth1 e pth2 where pth = (pth1 @ [e]) @ pth2 path v t pth2
by (metis fst-conv)
from this(2) least
have length pth \leq length pth2 by auto
with \( \_ \) show False by auto
qed

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

lemma scopes-nest:
  fixes ps1 ps2
shows \{\text{scope} \ ps1 \subseteq \{\text{scope} \ ps2 \} \vee \{\text{scope} \ ps2 \} \subseteq \{\text{scope} \ ps1 \} \wedge \{\text{scope} \ ps1 \} \cap \{\text{scope} \ ps2 \} = \{\}\}
proof (cases ps1 = ps2)
 assume ps1 \neq ps2
{ fix v
 assume v \in \{\text{scope} \ ps1 \cap \{\text{scope} \ ps2 \}
 hence v \in \{\text{vertices}\} using scope.simps by auto
 then obtain pth t where \{\text{path} \ v \ t \ pth\} \wedge \{\text{terminal-vertex} \ t\} using \{\text{pruned}\} by blast
from \{\text{path} \ v \ t \ pth\} \wedge \{\text{terminal-vertex} \ t\} \wedge \{\forall v . \ v \in \{\text{scope} \ ps1 \cap \{\text{scope} \ ps2 \}
 obtain pth1a e1 pth1b where pth = (pth1a @ [e1]) @ pth1b \wedge \{\text{path} \ v \ (fst \ ps1) \} \wedge \{\text{path} \ v \ (fst \ ps2) \} \wedge \{\text{ps1} \wedge \{\text{ps2} \notin \{\text{snd \ set \ pth1a} \} \wedge \{\text{ps1} \notin \{\text{snd \ set \ pth1a} \}
 by (auto elim: scope-split)
from \{\text{path} \ v \ t \ pth\} \wedge \{\text{terminal-vertex} \ t\} \wedge \{\forall v . \ v \in \{\text{scope} \ ps1 \cap \{\text{scope} \ ps2 \}
 obtain pth2a e2 pth2b where pth = (pth2a @ [e2]) @ pth2b \wedge \{\text{path} \ v \ (fst \ ps2) \} \wedge \{\text{path} \ v \ (fst \ ps2) \} \wedge \{\text{ps2} \wedge \{\text{ps2} \notin \{\text{snd \ set \ pth2a} \} \wedge \{\text{ps2} \notin \{\text{snd \ set \ pth2a} \}
 by (auto elim: scope-split)
from \{\text{ps1} \wedge \{\text{ps2} \notin \{\text{scope} \ ps2 \} \wedge \{\text{scope} \ ps2 \} \subseteq \{\text{scope} \ ps1 \}
proof
 assume set pth1a \subseteq set pth2a with \( \_ \in \)

have $ps_2 \notin \text{snd ' set (}pth_1[@[e_1]]\text{)}$ using $\langle ps_1 \neq ps_2 \rangle$ $\langle \text{snd } e_1 = ps_1 \rangle$ by auto

have scope $ps_1 \subseteq \text{scope } ps_2$
proof
  fix $v'$
  assume $v' \in \text{scope } ps_1$
  hence $v' \in \text{vertices using scope.simps by auto}$
  thus $v' \in \text{scope } ps_2$
proof (rule scope.intros)
  fix $pth'$ $t'$
  assume path $v' \leftrightarrow pth'$ and terminal-vertex $t'$
  with $(v' \in \text{scope } ps_1)$
  obtain $pth_3a \ e_3 \ pth_3b$ where $pth' = (pth_3a[\{e_3\}] \@ pth_3b$ and path $(fst \ ps_1) \ t' \ pth_3b$
    by (auto elim: scope-split)
  have path $v' \leftrightarrow ((pth_1[@[e_1]] \@ pth_3b)$ using $\langle \text{path } v (fst \ ps_1) \ (pth_1[@[e_1]]) \rangle$ and $\langle \text{path } (fst \ ps_1) \ t' \ pth_3b \rangle$
    by (rule path-appendI)
  with $\langle \text{terminal-vertex } t' \rangle$ $(v \in \text{scope } ps_1 \cap \text{scope } ps_2)$
  have $ps_2 \in \text{snd ' set (}pth_1[@[e_1]] \@ pth_3b\text{)}$ by (meson IntD2 scope.cases)
  hence $ps_2 \in \text{snd ' set (}pth_1[@[e_1]]\text{)}$ by auto
  thus $ps_2 \in \text{snd ' set } pth_3b$ using $\langle pth_3b \text{'} \rightarrow \text{ by auto}$
  qed
  qed
  thus $\neg \text{thesis by simp}$
next
assume set $pth_2a \subseteq \text{set } pth_1a$ with $\langle ps_1 \notin \rightarrow \rangle$
have $ps_1 \notin \text{snd ' set (}pth_2a[@[e_2]]\text{)}$ using $\langle ps_1 \neq ps_2 \rangle$ $\langle \text{snd } e_2 = ps_2 \rangle$ by auto

have scope $ps_2 \subseteq \text{scope } ps_1$
proof
  fix $v'$
  assume $v' \in \text{scope } ps_2$
  hence $v' \in \text{vertices using scope.simps by auto}$
  thus $v' \in \text{scope } ps_1$
proof (rule scope.intros)
  fix $pth'$ $t'$
  assume path $v' \leftrightarrow pth'$ and terminal-vertex $t'$
  with $(v' \in \text{scope } ps_2)$
  obtain $pth_3a \ e_3 \ pth_3b$ where $pth' = (pth_3a[\{e_3\}] \@ pth_3b$ and path $(fst \ ps_2) \ t' \ pth_3b$
    by (auto elim: scope-split)
  have path $v' \leftrightarrow ((pth_2a[@[e_2]] \@ pth_3b)$ using $\langle \text{path } v (fst \ ps_2) \ (pth_2a[@[e_2]]) \rangle$ and $\langle \text{path } (fst \ ps_2) \ t' \ pth_3b \rangle$
    by (rule path-appendI)
  with $\langle \text{terminal-vertex } t' \rangle$ $(v \in \text{scope } ps_1 \cap \text{scope } ps_2)$
  have $ps_1 \in \text{snd ' set (}pth_2a[@[e_2]] \@ pth_3b\text{)}$ by (meson IntD1 scope.cases)
  hence $ps_1 \in \text{snd ' set pth_3b using } \langle ps_1 \notin \text{snd ' set (}pth_2a[@[e_2]]\text{)} \rangle$ by auto
  thus $ps_1 \in \text{snd ' set } pth_3b$ using $\langle pth_3b \text{'} \rightarrow \text{ by auto}$
  qed
  qed
  thus $\neg \text{thesis by simp}$
  qed
}\)
  thus $\neg \text{thesis by blast}$
  qed simp
end
18
A well-scoped graph is one where a port marked to be a local hypothesis is only connected to the corresponding input port, either directly or via a path. It must not be, however, that there is a path from such a hypothesis to a terminal node that does not pass by the dedicated input port; this is expressed via scopes.

Locale Scoped-Graph = Port-Graph + Port-Graph-Signature-Scoped
Locale Well-Scoped-Graph = Scoped-Graph +
    Assumes well-scoped: \(((v_1,p_1),(v_2,p_2)) \in \textit{edges} \implies \textit{hyps} (\textit{nodeOf} v_1) p_1 = \text{Some} p' \implies (v_2,p_2) = (v_1,p') \lor v_2 \in \textit{scope} (v_1,p')

Context Scoped-Graph
Begin

Definition hyps-free where
hyps-free pth = (∀ v_1 p_1 v_2 p_2. ((v_1,p_1),(v_2,p_2)) \in \textit{set} pth \implies \textit{hyps} (\textit{nodeOf} v_1) p_1 = \text{None}

Lemma hyps-free-Nil[simp]: hyps-free [] by (simp add: hyps-free-def)
Lemma hyps-free-Cons[simp]: hyps-free (e#pth) ←→ hyps-free pth ∧ hyps (\textit{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 snd (map snd pth)
Using assms by induction auto

Inductive terminal-path where
    Terminal-path-empty: terminal-vertex v \implies terminal-path v v []
    Terminal-path-cons: (\forall v_1 p_1 v_2 p_2. ((v_1,p_1),(v_2,p_2)) \in \textit{edges} \implies \textit{terminal-path} v_2 v' pth \implies \textit{hyps} (\textit{nodeOf} v_1) p_1 = \text{None}
\implies \textit{terminal-path} v_1 v' (((v_1,p_1),(v_2,p_2))#pth)

Lemma terminal-path-is-path:
    Assumes terminal-path v v' pth
    Shows path v v' pth
Using assms by induction (auto simp add: path-cons-simp)

Lemma terminal-path-is-hyps-free:
    Assumes terminal-path v v' pth
    Shows hyps-free pth
Using assms by induction (auto simp add: hyps-free-def)

Lemma terminal-path-end-is-terminal:
    Assumes terminal-path v v' pth
    Shows terminal-vertex v'
Using assms by induction

Lemma terminal-pathI:
    Assumes path v v' pth
    Assumes hyps-free pth
    Assumes terminal-vertex v'
    Shows terminal-path v v' pth
Using assms by induction (auto intro: terminal-path.intros)
End

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

locale Acyclic-Graph = Scoped-Graph +
  assumes hyps-free-acyclic: path v v pth ⇒ 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 v1 p1 v2 p2 v' pth)
  from terminal-path-cons
  have v1 ≠ 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 \ldots \leq card (fset vertices)
  proof (rule card mono[OF finite-fset])
    from assms (1)
    show set (map fst (map fst pth)) \subseteq fset vertices
      by (induction v v' pth) (auto, metis valid-edges case prod D valid out-port simps)
  qed
  also have \ldots = fcard vertices by (simp add: fcard rep eq)
  finally show \textasteriskcentered
  qed

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

  from (v',p') \in snd \cdot set pth
  obtain pth1 pth2 e where pth = pth1 \circ [e] \circ pth2 and snd e = (v',p')
    by (rule snd set split)

  from terminal-path-is-path[OF assms (1), unfolded (pth = - \cdot ) \cdot (snd e = \cdot )]
  have path v v' (pth1 \circ [e]) and path v' t pth2 unfolding path split by (auto simp add: edge end tap)

  have (pth1 \circ [e]) \cdot (pth2a \circ [e'])
    by (rule path append I)
  moreover
  from terminal-path-is-hyps-free[OF assms (1)] \cdot pth = \cdot \cdot (pth2 = \cdot )
  have hyps-free ((pth1 \circ [e]) \circ (pth2a \circ [e']))
    by (auto simp add: hyps-free def)
  ultimately
  have ((pth1 \circ [e]) \circ (pth2a \circ [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) \Rightarrow \exists e \in 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 :: 'var ⇒ 'form ⇒ 'form
and subst-lconsts :: 'var ⇒ 'var set
and subst-renameLCs :: ('var ⇒ 'var) ⇒ ('var ⇒ 'var)
and anyP :: 'form +
fixes vidx :: 'vertex ⇒ nat
and inst :: 'vertex ⇒ 'var
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 labeled with the same formula.

locale Solution =
  Instantiation - - - - - edges for edges :: (('vertex × 'outPort) × 'vertex × 'inPort) set +
  assumes solved: ((v1, p1), (v2, p2)) ∈ edges ⇒ labelAtOut v1 p1 = labelAtIn v2 p2

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 :: 'var ⇒ 'form ⇒ 'form
and subst-lconsts :: 'var ⇒ 'var set
and subst-renameLCs :: ('var ⇒ 'var) ⇒ ('var ⇒ 'var)
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

22
and \( \text{labelsOut} \) :: 'node ⇒ 'outPort ⇒ 'form
and \( \text{vidx} \) :: 'vertex ⇒ nat
and \( \text{inst} \) :: 'vertex ⇒ 'subst
and \( \text{edges} \) :: ('vertex, 'outPort, 'inPort) edge set
and \( \text{local-vars} \) :: 'node ⇒ 'inPort ⇒ 'var set +
assumes well-scoped-inst:
valid-in-port \((v, p)\) ⇒
  \( v' \mid \in \mid \text{vertices} \) ⇒
  freshenLC \((\text{vidx} \ v)\) \( \text{var} \in \text{subst-lconsts} \ (\text{inst} \ v') \) ⇒
  \( \text{v'} \in \text{scope} \ (v, p) \)
begin
  lemma out-of-scope: valid-in-port \((v, p)\) ⇒ \( v' \mid \in \mid \text{vertices} \) ⇒ \( v' \notin \text{scope} \ (v, p) \) ⇒
    freshenLC \((\text{vidx} \ v)\) \( \text{var} \in \text{local-vars} \ (\text{nodeOf} \ v) \) \( p = \) ⇒
  \( v' \in \text{scope} \ (v, p) \)
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-lconsts subst-renameLCs anyP vidx inst edges +
Well-Scooped-Instantiation freshenLC renameLCs lconsts closed 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 \( \text{vidx} \) :: '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 (v, 'form, 'var) edge = ((v × ('form, 'var) in-port) × ('v × ('form, 'var) in-port))

collection Abstract-Rules

begin

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) | tame-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
\[
\begin{align*}
\text{outPorts (Assumption } r) &= \{\text{Reg } r\} \\
\text{outPorts (Conclusion } r) &= \{\} \\
\text{outPorts Helper} &= \{\text{Reg anyP } r\}
\end{align*}
\]

fun labelsIn where
labelsIn - p = a-conc p

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

fun hyps where
hyps (Rule r) (Hyp h a) = (if a \in f-antecedent r \land h \in a-hyps a then Some a else None)
\mid hyps - _ = None

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

sublocale Labeled-Signature nodes inPorts outPorts hyps labelsIn labelsOut
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 \in f-antecedent r \Rightarrow hyps-for (Rule r) ip = (\lambda h. Hyp h ip) \mid 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 \Rightarrow 'var \Rightarrow 'var
and renameLCs :: ('var \Rightarrow 'var) \Rightarrow 'form \Rightarrow 'form
and lconsts :: 'form \Rightarrow 'var set
and closed :: 'form \Rightarrow bool
and subst :: 'subt \Rightarrow 'form \Rightarrow 'form
and subst-lconsts :: 'subt \Rightarrow 'var set
and subst-renameLCs :: ('var \Rightarrow 'var) \Rightarrow ('subt \Rightarrow 'subt)
and anyP :: 'form

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

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

and vertices :: 'vertex fset
and \( \text{nodeOf} :: \text{'vertex} \Rightarrow (\text{'form}, \text{'rule}) \text{graph-node} \)

and \( \text{edges} :: (\text{'vertex}, \text{'form}, \text{'var}) \text{edge} \Rightarrow \text{set} \)

and \( \text{vidx} :: \text{'vertex} \Rightarrow \text{nat} \)

and \( \text{inst} :: \text{'vertex} \Rightarrow \text{'subst} + \)

assumes conclusions-present: set (map \text{Conclusion} \text{conclusions}) \subseteq \text{nodeOf} \cdot \text{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-consts :: '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
\[
\begin{align*}
\text{con} & \in \Gamma \\
\implies & \text{eff Axiom (} \Gamma \vdash \text{con} \text{)} \\{||\}
\\text{|nat-rule rule c ants} \\
\implies & (\forall \text{ ant f. ant } \in \text{ ants } \implies f |\in| \Gamma \implies \text{freshenLC a }' \text{ (a-fresh ant)} \cap \text{lconsts f} = \{\}) \\\n\implies & (\forall \text{ ant. ant } \in \text{ ants } \implies \text{freshenLC a }' \text{ (a-fresh ant)} \cap \text{subst-lconsts s} = \{\}) \\\n\implies & \text{eff (NatRule rule)} \\
& (\Gamma \vdash \text{subst s} \text{(freshen a c)}) \\\n& ((\lambda \text{ant. } ((\lambda p. \text{subst s} \text{(freshen a p)}) |\vdash| \text{a-hyps ant} \text{|}\cup| \Gamma \vdash \text{subst s} \text{(freshen a (a-conc ant))}) |\vdash| \text{ants}) \\\n& \text{|eff Cut (} \Gamma \vdash c' \text{)} \\{\| (\Gamma \vdash c')\}
\end{align*}
\]

\text{inductive-simps eff-Cut-simps[simp]: eff Cut (} \Gamma \vdash c \text{) S}

\text{sublocale RuleSystem-Defs where}
\text{eff = eff and rules = Cut ## Axiom ## smap NatRule rules.}
\text{end}

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

\text{context Abstract-Task}
\text{begin}
\text{inductive natEff-Inst where}
\text{c} \in \text{set (consequent r)} \implies \text{natEff-Inst (r,c) (f-antecedent r)}
\text{definition n-rules where}
\text{n-rules = flat (smap (} \lambda r. \text{map (} \lambda c. (r,c)) \text{ (consequent r)}) \text{ rules)}
\text{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.

\text{definition solved where}
\text{solved} \iff (\forall c. c |\in| \text{conc-forms} \implies (\exists t \text{ t. fst (root t) = (} \Gamma \vdash c \text{) \land } \Gamma |\subseteq| \text{ass-forms \land } \text{wf t \land } \text{tfinite t})
\text{end}
\text{end}

28
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 assns
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 assns by (atomize-elim, cases p; cases n) auto

lemma hyps-for-finage: hyps-for (Rule r) x = (if x ∈ f-antecedent v then (λ f. Hyp f x) | | (a-hyps x) else {||})
apply (rule fset-eql)
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 assns
proof (coinduction arbitrary: v p pth)

30
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)) \in \ edges \ and \ 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 \ ?l = \ labelAtIn \ v \ p

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

from \ (\ ?e \in\ \edges)
have \ s: \ labelAtOut \ v' \ p' = \ labelAtIn \ v \ p \ by \ (rule \ solved)

from \ (?e \in\ \edges)
show \ ?case
proof \ (cases \ rule: \ out-port-cases)
  case (Hyp \ r \ h \ c)

  from \ Hyp \ (?e' \in\ \outPorts \ (nodeOf \ v'))
  have \ h \in\ \a-hyps \ c \ and \ c \in\ \f-antecedent \ r \ by \ auto
  hence \ hyps \ (nodeOf \ v') \ (Hyp \ h \ c) = \ Some \ c \ using \ Hyp \ by \ simp

  from \ well-scoped[\OF \ (- \in\ \edges) \ [unfolded \ Hyp] \ this]
  have \ (v, \ p) = \ (v', \ c) \ \\lor \ v \in\ \scope \ (v', \ c),
  hence \ (v', \ c) \in\ \insert \ (v, \ p) \ (\ snd \ ' \ set \ \ pth)
  proof
    assume \ (v, \ p) = \ (v', \ c)
    thus \ ?thesis \ by \ simp
  next
    assume \ v \in\ \scope \ (v', \ c)
    from \ this \ terminal-path-end-is-terminal[\OF \ \wf \ (2)] \ terminal-path-is-path[\OF \ \wf \ (2)]
    have \ (v', \ c) \in\ \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 \in\ \hyps-for \ (nodeOf \ v') \ c \ by \ simp
hence \ labelAtOut \ v' \ (Hyp \ h \ c) \in\ \extra-assms \ (v', c) \ by \ auto
ultimately

have \ labelAtOut \ v' \ (Hyp \ h \ c) \in\ \?T
  by \ (fastforce \ simp \ add: \ \ffUnion.\rep-eq)

hence \ labelAtIn \ v \ p \in\ \?T \ 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 \( v' \in \) vertices \( \langle \text{nodeOf} \ v' = \text{Assumption} \ f \rangle \)

have \( \text{labelAtOut} \ v' \ (\text{Reg} \ f) \in \) global-assms TYPE(\text{var})

by (rule global-assmsI)

hence \( \text{labelAtOut} \ v' \ (\text{Reg} \ f) \in \ ?T \) by auto

hence \( \text{labelAtIn} \ v \ p \in \ ?T \) by (simp add: \text{symmetric} \ Assumption)

thus \( ?\text{thesis} \) using Assumption

by (auto intro: exI[where \( x = \ ?t \]) simp add: \text{eff.simps})

next

case (\text{Rule} \ r \ f)

with \( \langle \text{nodeOf} \ v' \in \text{sset} \ nodes \rangle \)

have \( r \in \text{sset} \ rules \)

by (auto simp add: \text{sset-def \ stream.set-map})

from \text{Rule}

have \( \text{hyps} \ (\text{nodeOf} \ v') \ p' = \text{None} \) by simp

with \( e \ (\text{terminal-path} \ v \ t \ \text{pth}) \)

have \( \text{terminal-path} \ v' \ t \ ?\text{pth}'. ..\)

from \text{Rule} \ (\text{p'} \in \) \text{outPorts} \ (\text{nodeOf} \ v')

have \( f \in \) \text{sset} \ \text{f-consequent} \ \text{r} \ by \ simp

hence \( f \in \text{set} \ (\text{consequent} \ r) \) by (simp add: \text{f-consequent-def})

with \( r \in \text{sset} \ rules \)

have \( \text{NatRule} \ (r, f) \in \text{sset} \ (\text{smap} \ \text{NatRule} \ n\text{-rules}) \)

by (auto \ simp \ add: \text{stream.set-map \ n-rules-def \ no-empty-conclusions})

moreover

\[
\begin{align*}
\text{from} \ (f |\in| \text{f-consequent} \ r) \\
\text{have} \ f \in \text{set} \ (\text{consequent} \ r) \ by \ (\text{simp add: f-consequent-def}) \\
\text{hence} \ \text{natEff-Inst} \ (r, f) \ f \ (\text{f-antecedent} \ r) \\
\text{by} \ (\text{rule natEff-Inst.intros}) \\
\text{hence} \ \text{eff} \ (\text{NatRule} \ (r, f)) \ (?T \vdash \text{subst} \ (\text{inst} \ v') \ (\text{freshen} \ (\text{vidx} \ v') \ \text{f})) \\
\quad ((\lambda \text{ant}. \ ((\lambda \text{p}. \text{subst} \ (\text{inst} \ v') \ (\text{freshen} \ (\text{vidx} \ v') \ \text{p}) |\vdash \text{a-hyps} \ \text{ant} |\cup| \ ?T \vdash \text{subst} \ (\text{inst} \ v') \ (\text{freshen} \ (\text{vidx} \ v') \ (\text{a-conc} \ \text{ant})))) |\vdash \text{f-antecedent} \ r) \\
\quad (\text{is eff - - ?ants}) \\
\text{proof} \ (\text{rule eff.intros}) \\
\quad \text{fix} \ \text{ant} \ f \\
\quad \text{assume} \ \text{ant} \ |\in| \text{f-antecedent} \ r \\
\quad \text{from} \ (v' |\in| \text{vertices} \langle \text{ant} |\in| \text{f-antecedent} \ r \rangle \\
\quad \text{have} \ \text{valid-in-port} \ (v', \text{ant}) \ by \ (\text{simp add: Rule})
\end{align*}
\]

assume \( f |\in| ?T \)

thus \( \text{freshenLC} \ (\text{vidx} \ v') \ (\text{a-fresh} \ \text{ant} \cap \ \text{lecons} \ f = \{\}) \)

proof(induct rule: \text{hyps-alongE})

\begin{align*}
\text{case} \ (\text{Hyp} \ v'' \ p'' \ \text{h''}) \\
\text{from} \ \text{Hyp(1)} \ \text{snd-set-vertices}[\text{OF} \ \text{terminal-path-is-path}[\text{OF} \ (\text{terminal-path} \ v' \ t \ ?\text{pth}')] \] \\
\text{have} \ v'' |\in| \text{vertices} \ by \ (\text{force simp add:})
\end{align*}

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

note \( \langle \text{terminal-path } v' \mid \in \langle \text{pth'} \rangle \) 
ultimately

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

\}
ultimately

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

next

case Helper 
from Helper 
have hyps \( (\text{nodeOf } v') \) \( p' = \text{None} \) by simp
with \( e \langle \text{terminal-path } v \text{ pth} \rangle \)
have \( \text{terminal-path } v' \text{ t } \langle \text{pth'} \rangle \)

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

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

note this(1)
moreover

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

note \( \langle \text{terminal-path } v' \text{ t } \langle \text{pth'} \rangle \) 
ultimately

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

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

qed

lemma global-in-ass: \( \text{global-assms } \text{TYPE}'(\text{var}) \mid \subseteq \langle \text{ass-forms} \) 
proof

fix \( x \)
assume \( x \mid \in \langle \text{global-assms } \text{TYPE}'(\text{var}) \rangle \)
then obtain \( v pf \) \( \text{where } v \mid \in \langle \text{vertices} \rangle \text{ and } \text{nodeOf } v = \text{Assumption } pf \text{ and } x = \text{labelAtOut } v \text{ (Reg } pf) \)
by (auto simp add: global-assms-simps)

qed

34
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 \((x = \text{labelAtOut} v \ (\text{Reg} pf))\)
  have \(x = pf\) by (auto simp add: labelAtOut-def freshen closed-no-lconsts freshen-closed subst-closed)
  hence \(pf \in\) set assumptions by (auto simp add: nodes-def stream.set-map)
  hence closed pf by (rule assumptions-closed)
with \(x = pf\) by (auto simp add: labelAtOut-def freshen closed-no-lconsts freshen-closed subst-closed)
thus \(x \in\) ass-forms using \(pf \in\) 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 op-split)
  thus thesis by (metis tfinite-map-tree)
qed

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

lemma path-is-forbidden:
  assumes valid-in-port (v, p)
  assumes ipath (edge-tree v p) es
  shows forbidden-path v es
using assms
proof (coinduction arbitrary: v p es)
  case forbidden-path
  let \(es' = \text{stl} es\)
from forbidden-path(2)
  obtain t' where root (edge-tree v p) = shd es and t' \in\) 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 ?thesis 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 iopath 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 \( p' = \text{Reg anyP} \) \( \langle \text{nodeOf} \ v' = \text{Helper} \rangle \)

have \( \text{hyps} \ (\text{nodeOf} \ v') \ p' = \text{None} \) by simp

moreover

from \( e \ \text{valid-edges} \) have \( v' \in \text{vertices} \) by auto

with \( \langle \text{nodeOf} \ v' = \text{Helper} \rangle \)

have \( \text{valid-in-port} \ (v', \text{plain-ant anyP}) \) by simp

moreover

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

ultimately

show \( \text{thesis} \) by metis

qed

qed

lemma forbidden-path-prefix-is-path:

assumes forbidden-path \( v \ es \)

obtains \( v' \) where \( \text{path} \ v' \ v \ (\text{rev} \ (\text{stake} \ ?n \ es)) \)

using \( \text{assms} \)

apply \( \langle \text{atomize-elim} \rangle \)

apply \( \langle \text{induction} \ n \ \text{arbitrary}; \ v \ es \rangle \)

apply \( \text{simp} \)

apply \( \langle \text{simp add: path-snoc} \rangle \)

apply \( \langle \text{subst (asm) (2) forbidden-path.simps} \rangle \)

apply \( \text{auto} \)

done

lemma forbidden-path-prefix-is-hyp-free:

assumes forbidden-path \( v \ es \)

shows \( \text{hyps-free} \ (\text{rev} \ (\text{stake} \ ?n \ es)) \)

using \( \text{assms} \)

apply \( \langle \text{induction} \ n \ \text{arbitrary}; \ v \ es \rangle \)

apply \( \langle \text{simp add: hyps-free-def} \rangle \)

apply \( \langle \text{subst (asm) (2) forbidden-path.simps} \rangle \)

apply \( \langle \text{force simp add: hyps-free-def} \rangle \)

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 \( \text{tfinite} \ (\text{tree} \ v \ p \ pth) \)

proof (rule ccontr)

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

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

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

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

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

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

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

from \( \text{forbidden-path-prefix-is-path[OF this]} \) forbidden-path-prefix-is-hyp-free[OF this]

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

by blast

from \( \text{this} \) terminal-vertex \( v \)

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

37
hence \( \text{length} \left( \text{rev} (\text{stake} \ ?n \ es) \right) \leq \text{fcard vertices} \)

by \( \text{(rule hyps-free-limited)} \)
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 \( c \in \text{set conclusions} \)
by (auto simp add: conc-forms-def)
from this(1) conclusions-present
obtain \( v \) where \( v \in \text{vertices} \) and \( \text{nodeOf} \ v = \text{Conclusion} \ c \)
by auto

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

have \( \text{terminal-vertex} \ v \) using \( v \in \text{vertices} \) (nodeOf \ v = Conclusion \ c) by auto

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

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

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

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

show \( \exists \Gamma. \ \text{fst} \ (\text{root} \ t) = (\Gamma \vdash \ c) \land \Gamma | \subseteq | \text{ass-forms} \land \text{wf} \ t \land \text{tfinite} \ t \) by blast

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

lemma iwf-subst-freshen-outPort:
\[ \text{iwf } lc \ ts \ ent \implies \text{snd } ent = \text{subst } (iSubst ts) \ (\text{freshen } (iAnnot ts) (iOutPort ts)) \]
by (auto elim: iwf_cases)

definition all-local-vars :: ('form, 'rule) graph-node \Rightarrow \text{var set where}
all-local-vars n = \bigcup (\text{local-vars } n \ ' \ fset \ (\text{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
[\bigwedge f. f \in \Gamma \implies \text{freshenLC } i \ (\text{all-local-vars } n) \cap \text{lconsts } f = \{\};
\text{freshenLC } i \ (\text{all-local-vars } n) \cap \text{subst-lconsts } s = \{\}
\] \implies \text{local-fresh-check } n i s (\Gamma \vdash c)

abbreviation local-iwf \equiv \text{iwf local-fresh-check}

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

inductive no-fresh-check :: ('form, 'rule, 'subst) fresh-check where
\text{no-fresh-check } n i s (\Gamma \vdash c)

abbreviation plain-iwf \equiv \text{iwf no-fresh-check}

fun isHNode where
\[ \text{isHNode } (\text{HNode } - - - ) = \text{True} \]
\text{isHNode } - - = \text{False}

lemma iwf-edge-match:
assumes iwf fc t ent
assumes is\@[i] \in \text{it-paths } t
shows subst (iSubst \ (\text{tree-at } t \ (is\@[i]))) \ (\text{freshen } (iAnnot \ (\text{tree-at } t \ (is\@[i]))) \ (iOutPort \ (\text{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 elim!: it-paths-SnocE dest list-all2-nthD2)[1]
using it-path-SnocI
apply (solves blast)
done

lemma iwf-length-inPorts:
assumes iwfc 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 iwfc 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:
iwfc 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:
iwfc 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 iwfc 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@[0|]) is \implies
i < length (inPorts' (iNodeOf (tree-at t is'))) \implies
hyps (iNodeOf (tree-at t is')) h = Some (inPorts' (iNodeOf (tree-at t is')) ! i) \implies
subst (iSubst (true-at t is')) (freshen (iAnnot (true-at t is')) (labelsOut (iNodeOf (tree-at t is')) h)) \in hyps-along t is

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

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

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

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

note \prefix (is''@[i']) is,
\langle i' < length (inPorts' (iNodeOf (true-at t is'))) \rangle

42
\(<\text{hyps } (\text{iNodeOf } (\text{tree-at } t \text{ is } ') ) h = \text{Some } (\text{inPorts'} (\text{iNodeOf } (\text{tree-at } t \text{ is } ') ) ! i')\),

\(\text{from this}\)\[\text{unfolded } (\text{tree-at } t \text{ is } ' = \text{tree-at } ?t' \text{ is } ')\]

\(\text{have subst } (\text{iSubst } (\text{tree-at } (\text{iAnts } t ! i) \text{ is } ') ) (\text{freshen } (\text{iAnnot } (\text{tree-at } (\text{iAnts } t ! i) \text{ is } ') ) (\text{labelsOut } (\text{iNodeOf } (\text{tree-at } (\text{iAnts } t ! i) \text{ is } ') ) h)) \in \text{hyps-along } (\text{iAnts } t ! i) \text{ is } \text{by } (\text{rule hyps-along.intros})\)

\(\text{hence } x \in ?S3 \text{ by simp}\)

\(\text{thus } ?\text{thesis}..\)

\(\text{qed}\)

\(\text{next}\)

\(\text{fix } x\)

\(\text{assume } x \in ?S2 \cup ?S3\)

\(\text{thus } x \in ?S1\)

\(\text{proof}\)

\(\text{have prefix } ([@i]) (i\#\text{is}) \text{ by simp}\)

\(\text{moreover}\)

\(\text{from } \langle \text{iwf - } t \rightarrow \rangle\)

\(\text{have length } (\text{iAnts } t) \leq \text{length } (\text{inPorts'} (\text{iNodeOf } (\text{tree-at } t [])))\)

\(\text{by cases } (\text{auto dest: list-all2-lengthD})\)

\(\text{with } (i < \rightarrow)\)

\(\text{have i < length } (\text{inPorts'} (\text{iNodeOf } (\text{tree-at } t []))) \text{ by simp}\)

\(\text{moreover}\)

\(\text{assume } x \in ?S2\)

\(\text{then obtain h where } h \in\)\[\text{hyps-for } (\text{iNodeOf } t) (\text{inPorts'} (\text{iNodeOf } t) ! i)\]

\(\text{and } [\text{simp}]: x = \text{subst } (\text{iSubst } t) (\text{freshen } (\text{iAnnot } t) (\text{labelsOut } (\text{iNodeOf } t) h)) \text{ by auto}\)

\(\text{from this}(1)\)

\(\text{have hyps } (\text{iNodeOf } (\text{tree-at } t [])) h = \text{Some } (\text{inPorts'} (\text{iNodeOf } (\text{tree-at } t [])) ! i) \text{ by simp}\)

\(\text{ultimately}\)

\(\text{have subst } (\text{iSubst } (\text{tree-at } t [])) (\text{freshen } (\text{iAnnot } (\text{tree-at } t [])) (\text{labelsOut } (\text{iNodeOf } (\text{tree-at } t [])) h)) \in\)\[\text{hyps-along } t (i \# \text{is})\]

\(\text{by } (\text{rule hyps-along.intros})\)

\(\text{thus } x \in \text{hyps-along } t (i \# \text{is}) \text{ by simp}\)

\(\text{next}\)

\(\text{assume } x \in ?S3\)

\(\text{thus } x \in ?S1\)

\(\text{apply } (\text{auto simp add: hyps-along.simps})\)

\(\text{apply } (\text{rule-tac } x = i\#\text{is'} \text{ in exI})\)

\(\text{apply auto}\)

\(\text{done}\)

\(\text{qed}\)

\(\text{qed}\)

\(\text{lemma iwf-hyps-exist}:\)

\(\text{assumes iwf lc it ent}\)

\(\text{assumes is } \in \text{it-paths it}\)

\(\text{assumes tree-at it is } = (\text{HNode } i \text{ s ants'})\)

\(\text{assumes fst ent } \subseteq \text{ass-forms}\)

\(\text{shows subst s (freshen i anyP) } \in \text{hyps-along it is}\)

\(\text{proof}\)

\(\text{from } \text{assms}(1, 2, 3)\)

\(\text{have subst s (freshen i anyP) } \in \text{hyps-along it is}\)

\(\text{∨ subst s (freshen i anyP) } \in\)\[\text{fst ent}\]

\(\text{∧ subst s (freshen i anyP) } \not\in \)\[\text{ass-forms}\)

\(\text{proof}(\text{induction arbitrary: is rule: iwf.induct})\)

\(\text{case } (\text{iwf n p s' a' } \Gamma \text{ ants c is})\)

\(\text{have iwf lc (INode n p a' s' ants) (} \Gamma \vdash c)\)
using iwf(1,2,3,4,5)
by (auto intro!: iwf.intros elim!: list-all2-mono)

show ?case
proof (cases is)
  case Nil
  with \textup{tree-at} (INode n p a' s' ants) is = HNode i s ants'
  show \textup{thesis} by auto
next
  case (Cons i' is')
  with \textup{is} \in \textup{it-paths} (INode n p a' s' ants)
  have i' < length ants and \textup{is} \in \textup{it-paths} (ants ! i')
    by (auto elim: \textup{it-paths-ConsE})
  let \(\Lambda' = (\lambda h. \text{subst} s' \text{(freshen} a' \text{(labelsOut n h)}) \mid^? \text{hyps-for} n \text{(inPorts'} n \mid i')\)
  from \textup{tree-at} (INode n p a' s' ants) is = HNode i s ants'
  have \textup{tree-at} (ants ! i') is' = HNode i s ants' using Cons by simp
from iwf.H \textup{\langle i' < length} ants\rangle \textup{\langle is' \in \text{it-paths} (ants ! i')\rangle this}
have \textup{subst} s \text{(freshen} i \text{anyP}) \in \text{hyps-along} \text{(ants ! i')} is'
  \lor \text{subst} s \text{(freshen} i \text{anyP}) \mid^? \Lambda' \mid^? \Gamma \land \text{subst} s \text{(freshen} i \text{anyP}) \mid^? \emptyset \text{ass-forms}
  by (auto dest: list-all2-nthD2)
moreover
from \textup{\langle is \in \text{it-paths} (INode n p a' s' ants)\rangle}
have \text{hyps-along} \text{(INode n p a' s' ants) is} = fset \Lambda' \cup \text{hyps-along} \text{(ants ! i')} is'
  using \textup{\langle is = \_\rangle}
  by (simp add: \text{hyps-along-Cons[OF iwf lc (INode n p a' s' ants) (\Gamma \vdash c)\]])
ultimately
  show \textup{\langle thesis \rangle by auto}
qed

next
case (iwf H c \Gamma s' i' \_)
  hence \textup{\langle simp\rangle: is = \[ i' = i s' = s \text{ by simp-all}\rangle}
  from \textup{\langle c = subst} s' \text{(freshen} i' \text{anyP)} \text{\langle c \in}\mid^? \Gamma \text{\langle c \mid^? \emptyset \text{ass-forms}\rangle}
  show \textup{\langle case2 by simp\rangle}
qed
with \textup{assms(4)}
show \textup{\langle thesis \rangle by blast}
qed

definition hyp-port-for' :: ('form, 'rule, 'subst, 'var) \textit{tree} \Rightarrow \textit{nat list} \Rightarrow 'form \Rightarrow \textit{nat list} \times \textit{nat} \times ('form, 'var) \textit{out-port} where
hyp-port-for' t is f = (\text{SOME} x.
  (case x of (is', i, h) \Rightarrow
    \text{prefix} \text{(is' \text{\&} [i])} \text{ is \&}
    i < \text{length} \text{(\text{inPorts'} (iNodeOf (\text{tree-at} t \text{ is'})))} \land
    \text{hyps} \text{(iNodeOf (\text{tree-at} t \text{ is'}))} \text{ h} = \text{Some} \text{(\text{inPorts'} (iNodeOf (\text{tree-at} t \text{ is'})) \mid i) \land}
    f = \text{subst} \text{(iSubst (\text{tree-at} t \text{ is'})} \text{(freshen} \text{(iAnnot (\text{tree-at} t \text{ is'}) \text{(labelsOut (iNodeOf (\text{tree-at} t \text{ is'}))) h})
  ))
)

lemma hyp-port-for-spec';
assumes f \in \text{hyps-along} \text{t is}
shows (case hyp-port-for' t is f of (is', i, h) \Rightarrow
  \text{prefix} \text{(is' \text{\&} [i])} \text{ is \&}
  i < \text{length} \text{(\text{inPorts'} (iNodeOf (\text{tree-at} t \text{ is'})))} \land
  \text{hyps} \text{(iNodeOf (\text{tree-at} t \text{ is'}))} \text{ h} = \text{Some} \text{(\text{inPorts'} (iNodeOf (\text{tree-at} t \text{ is'})) \mid i) \land}
\[ f = \text{subst} (\text{iSubst} (\text{tree-at} \ t \ i)) (\text{freshen} (\text{iAnnot} (\text{tree-at} \ t \ i)) (\text{labelsOut} (\text{iNodeOf} (\text{tree-at} \ t \ i)) \ h))) \]

**Using** `assms` **unfolding** `hyps-along`.simp `hyp-port-for`-def by ```(rule someI-ex, blast)```  

**Definition** `hyp-port-path-for` :: `('form, 'rule, 'subbst, 'var)` \( \text{tree-at} \Rightarrow \text{nat list} \Rightarrow \text{form} \Rightarrow \text{nat list} \)

**Where** `hyp-port-path-for` \( t \) is \( f = \text{fst} (\text{hyp-port-for'} t \ i) \)

**Definition** `hyp-port-i-for` :: `('form, 'rule, 'subbst, 'var)` \( \text{tree-at} \Rightarrow \text{nat list} \Rightarrow \text{form} \Rightarrow \text{nat}`

**Where** `hyp-port-i-for` \( t \) is \( f = \text{fst} (\text{snd} (\text{hyp-port-for'} t \ i)) \)

**Definition** `hyp-port-h-for` :: `('form, 'rule, 'subbst, 'var)` \( \text{tree-at} \Rightarrow \text{nat list} \Rightarrow \text{form} \Rightarrow \text{('form, 'var) out-port} \)

**Where** `hyp-port-h-for` \( t \) is \( f = \text{snd} (\text{snd} (\text{hyp-port-for'} t \ i)) \)

**Lemma** `hyp-port-prefix`:

- **Assumes** \( f \in \text{hyps-along} \ t \)
- **Shows** `prefix` (\( \text{hyp-port-path-for} t \) is \( f \in [\text{hyp-port-i-for} t \] \) is)

**Using** `hyp-port-for-spec`\( [\text{OF} \ assms] \)  **unfolding** `hyp-port-path-for-def` `hyp-port-i-for-def` by ```auto```  

**Lemma** `hyp-port-strict-prefix`:

- **Assumes** \( f \in \text{hyps-along} \ t \)
- **Shows** `strict-prefix` (\( \text{hyp-port-path-for} t \) is \( f \))

**Using** `hyp-port-prefix`\( [\text{OF} \ assms] \)  **by** ```(simp add: strict-prefixI prefix-order.dual-order.strict-trans1)```  

**Lemma** `hyp-port-it-paths`:

- **Assumes** \( i \in \text{it-paths} \ t \)
- **Assumes** \( f \in \text{hyps-along} \ t \)
- **Shows** `hyp-port-path-for t` is \( f \in \text{it-paths} t \)

**Using** `assms` **by** ```(rule it-paths-strict-prefix \text{OF} - \text{hyp-port-strict-prefix})```  

**Lemma** `hyp-port-hyps`:

- **Assumes** \( f \in \text{hyps-along} \ t \)
- **Shows** `hyps` (\( \text{iNodeOf} (\text{tree-at} \ t \ (\text{hyp-port-path-for} t \ i)) \) (\( \text{hyp-port-h-for} t \) is \( f \)) = \text{Some} \ (\text{inPorts'} \ (\text{iNodeOf} (\text{tree-at} \ t \ (\text{hyp-port-path-for} t \ i)))) \) \( \text{hyp-port-i-for} t \) is \( f \))

**Using** `hyp-port-for-spec`\( [\text{OF} \ assms] \)  **unfolding** `hyp-port-path-for-def` `hyp-port-i-for-def` `hyp-port-h-for-def` by ```auto```  

**Lemma** `hyp-port-outPort`:

- **Assumes** \( f \in \text{hyps-along} \ t \)
- **Shows** (\( \text{hyp-port-h-for} t \) is \( f \)) \( \in \text{outPorts} \ (\text{iNodeOf} (\text{tree-at} \ t \ (\text{hyp-port-path-for} t \ i))) \)

**Using** `hyps-correct`\( [\text{OF} \ hyp-port-hyps] [\text{OF} \ assms] \)

**Lemma** `hyp-port-eq`:

- **Assumes** \( f \in \text{hyps-along} \ t \)
- **Shows** \( f = \text{subst} (\text{iSubst} (\text{tree-at} \ t \ (\text{hyp-port-path-for} t \ i))) (\text{freshen} (\text{iAnnot} (\text{tree-at} \ t \ (\text{hyp-port-path-for} t \ i))) (\text{labelsOut} (\text{iNodeOf} (\text{tree-at} \ t \ (\text{hyp-port-path-for} t \ i))) (\text{hyp-port-h-for} t \) is \( f)))\)

**Using** `hyp-port-for-spec`\( [\text{OF} \ assms] \)  **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 x` = to-nat (Some \( x \))

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

**Lemma** `isidx-inj[simp]`; `isidx x` = `isidx y` ⇔ `x` = `y`

**Unfolding** `isidx-def` by ```simp```  

**Lemma** `isidx-v-away[simp]`; `isidx x` ≠ `v-away`

**Unfolding** `isidx-def v-away-def` by ```simp```  

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

**Lemma** `mapWithIndex-cong` ```[fundef-cong]```:
\[ xs = ys \implies (\forall x. x \in set \ ys \implies f \ i \ x = g \ i \ x) \implies mapWithIndex f \ xs = mapWithIndex g \ ys \]

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

\begin{verbatim}
fun globalize-node :: nat list \Rightarrow ('var \Rightarrow 'var) \Rightarrow ('form,'rule,'subst,'var) itnode \Rightarrow ('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 \Rightarrow ('var \Rightarrow 'var) \Rightarrow ('form,'rule,'subst,'var) itree \Rightarrow ('form,'rule,'subst,'var) itree
   where
   globalize is f (RNode rants) = RNode
   (globalize-node is f r)
   (mapWithIndex (\lambda i \ t. globalize (is@i[]) (\mapWithIndex f xs) (\mapWithIndex g ys))
     (isAnnot (RNode rants)) (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' \in 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-localconsts-listed':
   assumes n \in set nodes
   assumes p \in set inPorts n
   shows \iscons (a-conc p) \cup (\cup (\iscons \setminus fset (a-hyps p))) \subseteq a-fresh p
   using assms
   by (auto simp add: nodes-def stream.set-map \iscons-anyP closed-no-\iscons conclusions-closed f-antecedent-def dest!: all-localconsts-listed)

lemma no-localconsts-in-consequences':
   n \in set nodes \implies Reg p \in\set outPorts n \implies \iscons p = {}
   using no-localconsts-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 (renameLCs f |τ [γ] Γ ⊢ renameLCs f c)
using assms
proof (induction t Γ ⊢ c arbitrary: is f c rule: iwf.induct)
case (iwf n p s Γ ants c is f)

note \( n \in \text{sset nodes} \)

note (Reg p |∈| outPorts n)

\{ fix i' \}
  \begin{itemize}
  
  \item let \( ?V = \text{a-fresh} \ (\text{inPorts'} n \setminus i') \)
  \item let \( ?f' = \text{rename} \ ?V i \ (\text{isidx} \ \text{is}) \ f \)
  \item let \( ?t = \text{globalize} \ (\text{is} @ [i']) \ ?f' \ (\text{ants} \ i') \)
  \item let \( ?ip = \text{inPorts'} n \setminus i' \)
  \item let \( ?t' = (\lambda h. \ \text{subst} \ (\text{subt-renameLCs} f s) \ (\text{freshen} \ (\text{isidx} \ \text{is}) \ (\text{labelsOut} n \ h)) \ (\text{hyps-for} \ n \ ?ip) \)
  \item let \( ?c' = \text{subt} \ (\text{subt-renameLCs} f s) \ (\text{freshen} \ (\text{isidx} (\text{isIn} n \ ?ip)) \)
  
  \end{itemize}

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

\begin{itemize}
  
  \item hence \((\text{inPorts'} n \setminus i') |∈| \text{inPorts} n \) by (simp add: inPorts-fset-of)
  
  \item from \((i' < \text{length} \ (\text{inPorts'} n)) \)
  
  \item have \( \text{subset-V} \ ?V \subseteq \text{all-local-vars} \ n \)
  
  \item unfolding \text{all-local-vars-def}
  
  \item by (auto simp add: inPorts-fset-of)
  
  \item from \(\text{local-fresh-check} n i s \ (\Gamma \vdash c)\)
  
  \item have \( \text{freshenLC} i' \ \text{all-local-vars} \ n \cap \text{subt-lconsts} s = \{\} \)
  
  \item by (rule local-fresh-check.cases) simp
  
  \item hence \(\text{freshenLC} i' \ ?V \cap \text{subt-lconsts} s = \{\} \)
  
  \item using \text{subset-V} by auto
  
  \item hence \(\text{rename-subst}: \text{subt-renameLCs} \ ?f' s = \text{subt-renameLCs} f s \)
  
  \item by (rule rename-subst-noop)
  
  \item from \(\text{all-local-consts-listed'}[\text{OF} \ (n \in \text{sset nodes}) \ (\text{inPorts'} n \setminus i') |∈| \text{inPorts} n] \}
  
  \item have \(\text{subset-conc}: \text{lconsts} \ (\text{a-conc} \ (\text{inPorts'} n \setminus i')) \subseteq \ ?V \)
  
  \item and \(\text{subset-hyp} \ (\text{hyp} : \text{hyp} |∈| \text{a-hyps} \ (\text{inPorts'} n \setminus i') \Rightarrow \text{lconsts} \ \text{hyp} \subseteq \ ?V \)
  
  \item by auto
  
  \item from \(\text{List.list-all2-nthD}[\text{OF} \ (\text{list-all2} \ \cdots \ \cdot i' < \text{length} \ (\text{inPorts'} n))],\text{simplified} \)
  
  \item have \(\text{plain-uf} \ ?t \)
    
    \begin{itemize}
      
      \item ((renameLCs \ ?f' |τ [γ] (\lambda h. \ \text{subt} s \ (\text{freshen} i \ (\text{labelsOut} n \ h))))) \ (\text{hyps-for} \ n \ ?ip |∪| \Gamma) \vdash \text{renameLCs} \ ?f' \ (\text{subt} s \ (\text{freshen} i \ (\text{a-conc} \ ?ip))) \)
    
    \item by simp
    
    \item also have \text{renameLCs} \ ?f' |τ [γ] (\lambda h. \ \text{subt} s \ (\text{freshen} i \ (\text{labelsOut} n \ h))) \ (\text{hyps-for} \ n \ ?ip |∪| \Gamma) \)
      
      \begin{itemize}
        
        \item = (\lambda s. \ \text{subt} \ (\text{subt-renameLCs} \ ?f' s) \ (\text{renameLCs} \ ?f' \ (\text{freshen} i \ (\text{labelsOut} n \ x))) \ (\text{hyps-for} \ n \ ?ip |∪| \text{renameLCs} \ ?f' |τ [γ] \))
      
      \item by \(\text{simp add: fimage-fimage fimage-funion comp-def rename-subst})
    
    \item also have \(\text{renameLCs} \ ?f' |τ [γ] = \text{renameLCs} f |τ [γ] \)
    
    \item \text{proof}(\text{rule fimage-cong}[\text{OF refl}])
  
    \begin{itemize}
      
      \item fix \( x \)
      
      \item assume \( x |∈| \Gamma \)
      
      \item with \(\text{local-fresh-check} n i s \ (\Gamma \vdash c)\)
      
      \item have \(\text{freshenLC} i' \ \text{all-local-vars} \ n \cap \text{lconsts} x = \{\} \)
  
  \end{itemize}
\end{itemize}
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 rename-rename-rename-noop)

qed

also have \((λx. subst (renameLCs ?f' s) (renameLCs ?f' (freshen i (labelsOut n x)))) \mid \| hyps-for n \forall p = ?T')

proof(rule fimage-cong[OF refl])

fix hyp

assume hyp \in hyps-for n (inPorts' n ! i')

hence labelsOut n hyp \in 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 (renameLCs ?f' s) (renameLCs ?f' (freshen i (labelsOut n hyp))) = 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 \(λx. subst (renameLCs ?f' s) (renameLCs ?f' (freshen i (a-conc ?ip)))) = subst (renameLCs ?f' s) (renameLCs ?f' (freshen i (a-conc ?ip)))\) by (simp add: rename-subst)

also have \(\ldots = ?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 (\| \| renameLCs f | Γ ⌦ \?c').

\}
with list-all2-lengthD[OF list-all2 - - -]

have list-all2

(Aip t. plain-iwf t ((λh. subst (renameLCs f s) (freshen (isidx is) (labelsOut n h))) \mid \| hyps-for n \forall p | \| renameLCs f | Γ ⌦ subst (renameLCs f s) (freshen (isidx is) (labelsIn n ip))))

(inPorts' n)

(mapWithIndex (λ i' t. globalize (is@i') (rename (a-fresh (inPorts' n ! i')) i (isidx is) f) t) ants)

by (auto simp add: list-all2-conv-all-nth)

moreover

have no-fresh-check n (isidx is) (renameLCs f s) (renameLCs f | Γ ⌦ renameLCs f c)...

moreover

from \(\in sset nodes) ⊆ Reg p | \| outPorts n\)

have lconsts p = {} by (rule no-local-consts-in-consequences')

with \(c = subst s (freshen i p)\)

have renameLCs f c = subst (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-consts lconsts-renameLCs rename-closed
  by fastforce

moreover
from (c ∈ Γ)
have renameLCs f c ∈ renameLCs f ⊢ renameLCs f c
  by auto

moreover
from (HNode i s [] (renameLCs f c))
ultimately
show plain-iwf (globalize is f (HNode i s [])) (renameLCs f ⊢ 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 (tree-at t (butlast is))) ' (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)
done

lemma iwf-globalize':
  assumes local-iwf t 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

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 \( \forall x. x \in S2 \Rightarrow f x \in S1 \land f1 (f x) = f2 x \)
proof (atomize-elim)
  from assms
  have \( \forall x. x \in S2 \Rightarrow (\exists y. y \in S1 \land f1 y = f2 x) \) by (metis image-iff)
  thus \( \exists f. \forall x. x \in S2 \Rightarrow f x \in S1 \land f1 (f x) = f2 x \) by metis
qed

context includes fset.lifting
lemma fimage-eq-to-f:
  assumes f1 |- ?S1 = f2 |- ?S2
  obtains f where \( \forall x. x \in ?S2 \Rightarrow f x \in ?S1 \land f1 (f x) = f2 x \)
  using assms apply transfer using image-eq-to-f by metis
end

context Abstract-Task
begin

lemma build-local-iwf:
  fixes t :: (form entailment \times ('rule \times 'form) NatRule) tree
  assumes tfinite t
  assumes wf t
  shows \( \exists it. local-iwf it (fst (root t)) \)
  using assms
  proof (induction)
    case (tfinite t)
    from \( \langle wf t \rangle \)
    have snd (root t) \in R using wf.simps by blast
    from \( \langle wf t \rangle \)
    have \( \forall t'. t' \in| cont t \Rightarrow wf t' \)
    using wf.simps by blast
    hence IH: \( \forall \Gamma' t'. t' \in| cont t \Rightarrow (\exists it'. local-iwf it' (fst (root t'))) \)
    using tfinite(2) by blast
    then obtain its where its: \( \forall t', t' \in| cont t \Rightarrow local-iwf (its t') (fst (root t')) \)
    by metis
    from \( \langle eff \langle \ldots \rangle \rangle \)
    show ?case
    proof (cases rule: eff.cases[case-names Axiom NatRule Cut])
      case (Axiom c \( \Gamma \))
      show ?thesis
      proof
        then have c \in set assumptions by (auto simp add: ass-forms-def)
        case True
        then have c \in set assumptions by (auto simp add: ass-forms-def)
        let \( ?it = INode (Assumption c) c undefined undefined [] :: (form, 'rule, 'subst, 'var) itree \)
        from \( \langle c \in set assumptions \rangle \)
        have local-iwf ?it (\( \Gamma \vdash c \))
          by (auto intro: iwf local-fresh-check.intros)
        thus ?thesis unfolding Axiom.. next
      case False
      obtain s where subst s anyP = c by atomize-elim (rule anyP-is-any)
      hence [simp]: subst s (freshen undefined anyP) = c by (simp add: lconsts-anyP freshen-closed)
      let \( ?it = HNode undefined s [] :: (form, 'rule, 'subst, 'var) itree \)
      from \( \langle c \in \Gamma \rangle \)
      have local-iwf ?it (\( \Gamma \vdash c \)) by (auto intro: iwfH)
      thus ?thesis unfolding Axiom..
      qed
    next
  qed
case (NatRule rule c ants Γ i s)
from (natEff-Inst rule c ants)
have snd rule = c \text{ and simp:} ants = f-antecedent (fst rule) \text{ and} c \in \text{ set (consequent (fst rule))}
by (auto simp add: natEff-Inst.simps)

from (fst o root) \text{ \sqsubset cont} t = (\lambda t. (\lambda p. \text{ subst s (freshen i p)}) \sqsubset \text{ a-hyps ant} \sqcup \Gamma \vdash \text{ subst s (freshen i (a-conc ant))}) \sqsubset \text{ ants)
obtain to-t where \text{ \sqsubset cont} t \sqsubseteq \text{ to-t ant} \sqsubseteq \text{ ants} \Rightarrow \text{ to-t ant} \sqsubseteq \text{ cont} t \wedge (\text{ fst o root}) \sqsubseteq \text{ to-t ant} = ((\lambda p. \text{ subst s (freshen i p)}) \sqsubset \text{ a-hyps ant} \sqcup \Gamma \vdash \text{ subst s (freshen i (a-conc ant))})
by (rule \text{ simps})

hence \text{ to-t-in-cont:} \sqsubseteq \text{ ants} \Rightarrow \text{ to-t ant} \sqsubseteq \text{ cont} t
\text{ and} \text{ to-t-root:} \sqsubseteq \text{ ants} \Rightarrow \text{ fst (root (to-t ant))} = ((\lambda p. \text{ subst s (freshen i p)}) \sqsubset \text{ a-hyps ant} \sqcup \Gamma \vdash \text{ subst s (freshen i (a-conc ant))})
by auto

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

from (snd (root t)) \in R
have \text{ fst rule} \in \text{ sset rules}
unfolding NatRule
by (auto simp add: stream.set-map n-rules-def no-empty-conclusions )
moreover
from \in \text{ set (consequent (fst rule))}

have c \in \text{ f-consequent (fst rule)} \text{ by simp add: f-consequent-def)
moreover

\{ fix ant
  assume ant \in \text{ set (antecedent (fst rule))}
  hence ant \in \text{ ants} by (simp add: f-consequent-def)
  from \in \text{ its(OF to-t-in-cont[OF this])}
  have local-iwf (\text{ its (to-t ant)}) (\text{ fst (root (to-t ant))}).
  also have \text{ fst (root (to-t ant))} =
    ((\lambda p. \text{ subst s (freshen i p)}) \sqsubset \text{ a-hyps ant} \sqcup \Gamma \vdash \text{ subst s (freshen i (a-conc ant))})
  by (rule \text{ rule that})
  also have \ldots =
    ((\lambda h. \text{ subst s (freshen i (labelsOut (Rule (fst rule)) h))}) \sqsubset \text{ hyps-for (Rule (fst rule))} \text{ ant} \sqcup \Gamma \vdash \text{ subst s (freshen i (a-conc ant))})
  using \text{ ant} \in \text{ ants}
by auto
finally
have local-iwf (\text{ its (to-t ant)})
  ((\lambda h. \text{ subst s (freshen i (labelsOut (Rule (fst rule)) h))}) \sqsubset \text{ hyps-for (Rule (fst rule))} \text{ ant} \sqcup \Gamma \vdash \text{ subst s (freshen i (a-conc ant))}).
\} \text{ moreover
from NatRule(5.6)

have local-fresh-check (Rule (fst rule)) i s \in \text{ (\Gamma \vdash \text{ subst s (freshen i c)})}
  by (fastforce intro: local-fresh-check.intros simp add: all-local-vars-def)
ultimately
have local-iwf \text{ ?it ((\Gamma \vdash \text{ subst s (freshen i c))})}
  by (intro iwf ) (auto simp add: list-all2-map2 list-all2-same)
thus \text{ ?thesis unfolding NatRule..}
next
\text{ case (Cut \Gamma \text{ con})}
obtain s where \text{ subst s anyP = con by atomize-elim (rule anyP-is:any)}
hence [simp]: \text{ subst s (freshen undefined anyP) = con by simp add: Iconst-\text{anyP freshen-closed)
from \((fst \circ root) \mid \Gamma \vdash \text{cont } t = \{\Gamma \vdash \text{con}\}\) 

obtain \(t'\) where \(t' \in \text{cont } t\) and \([\text{simpl}: \text{fst } (root t') = (\Gamma \vdash \text{con})\]

by (cases \text{cont } t\) auto

from \(t' \in \text{cont } t\) obtain \(it'\) where local-iwf \(it'\) \(\Gamma \vdash \text{con}\) using IH by force

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

from \(\text{local-iwf } it'\) \(\Gamma \vdash \text{con}\) have \(\text{local-iwf } ?it\) \(\Gamma \vdash \text{con}\) by (auto intro!: iwf local-fresh-check.intro) thus \(?thesis unfolding Cut..\)

qed

definition to-it \:: (\text{form entailment} \times (\text{rule} \times \text{form} \text{NatRule}) \text{tree} \Rightarrow (\text{form}, \text{rule}, \'subst', \'var\) \text{itree where}

to-it t = \(\text{SOME } it. \text{local-iwf } it\) \(\text{fst } (\text{root } t)\))

lemma iwf-to-it:

assumes \(tfinite t\) and \(wf t\)

shows \(\text{local-iwf } (\text{to-it } t) \Rightarrow (\text{fst } (\text{root } t))\)

unfolding to-it-def using build-local-iwf[OF assms] by (rule someI2-ex)

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 \(\text{form vertex} = (\text{form} \times \text{nat list})\)
type-synonym \(\text{edge}'' = (\text{form vertex}, \text{form}, \'var\) edge\')

locale Solved_Task =
Abstract_Task freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP antecedent consequent rules assumptions conclusions

for freshenLC :: nat \Rightarrow 'var \Rightarrow 'var

and renameLCs :: ('var \Rightarrow 'var) \Rightarrow 'form \Rightarrow 'form

and lconsts :: 'form \Rightarrow 'var set

and closed :: 'form \Rightarrow bool

and subst :: 'subst \Rightarrow 'form \Rightarrow 'form

and subst-lconsts :: 'subst \Rightarrow 'var set

and subst-renameLCs :: ('var \Rightarrow 'var) \Rightarrow ('subst \Rightarrow 'subst)

and anyP :: 'form

and antecedent :: 'rule \Rightarrow ('form, 'var) antecedent list

and consequent :: 'rule \Rightarrow 'form list

and rules :: 'rule stream

and assumptions :: 'form list

and conclusions :: 'form list +

assumes solved: solved

begin

Let us get our hand on concrete trees.

definition ts :: 'form \Rightarrow ((\text{form entailment}) \times (\text{rule} \times \text{form} \text{NatRule}) \text{tree where}
\[ ts \ c = (SOME \ t. \ \text{snd} \ (\text{fst} \ (\text{root} \ t))) = c \land \ \text{fst} \ (\text{fst} \ (\text{root} \ t)) \mid \subseteq \ \text{ass-forms} \land \ \text{wf} \ t \land \ \text{tfinite} \ t \]

**Lemma**

- **Assumes**: \( c \mid \subseteq \ \text{conc-forms} \)
- **Shows**: \( ts-\text{conc}: \ \text{snd} \ (\text{fst} \ (\text{root} \ (ts \ c))) = c \)
- **And**: \( ts-\text{context}: \ \text{fst} \ (\text{fst} \ (\text{root} \ (ts \ c))) \mid \subseteq \ \text{ass-forms} \)
- **And**: \( ts-\text{wf}: \ \text{wf} \ (ts \ c) \)
- **And**: \( ts-\text{finite}[\text{simp}]: \ \text{tfinite} \ (ts \ c) \)

**Unfolding** atomize-conj conj-assoc ts-def

**Apply** (rule somel-ex)

**Using** solved assms

**By** (force simp add: solved-def)

**Abbreviation** \( it' \) where

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

**Lemma** \( \text{iwf-it} \)

- **Assumes**: \( c \in \text{set conclusions} \)
- **Shows**: \( \text{plain-iwf} \ (it' \ c) \ (\text{fst} \ (\text{root} \ (ts \ c))) \)
- **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** \( \text{vertices} :: \ {'\text{form} \ \text{vertex} \ \text{fset} \ '} \)

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

**Lemma** \( \text{mem-vertices}: \ v \mid \subseteq \ \text{vertices} \longleftrightarrow (\text{fst} \ v \in \text{set conclusions} \land (\text{snd} \ v = [] \lor \text{snd} \ v \in (\# \ 0) \land '\text{it-paths} \ (it' \ (\text{fst} \ v)))) \)

**Unfolding** vertices-def ffUnion.rep-eq

**By** (cases v)(auto simp add: Abs-fset-inverse Bex-def)

**Lemma** \( \text{prefixeq-vertices}: (c, is) \mid \subseteq \ \text{vertices} \Longrightarrow \text{prefix} \ is' \Longrightarrow (c, ts') \mid \subseteq \ \text{vertices} \)

**By** (cases is')(auto simp add: mem-vertices intro!: imageI elim: it-paths-prefix)

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

**By** (simp add: mem-vertices)

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

**By** (auto simp add: mem-vertices)

**Lemma** \( \text{prefixes-cases}[\text{consumes} \ 1, \ \text{case-names} \ None \ Some]: \)

**Assumes**: \( v \mid \subseteq \ \text{vertices} \)

**Obtains**: \( c \ \text{where} \ c \in \text{set conclusions} \land v = (c, []) \)

**Using** assms by (cases v; rename-tac is; case-tac is; auto)

**Lemma** \( \text{prefixes-induct}[\text{consumes} \ 1, \ \text{case-names} \ None \ Some]: \)

**Assumes**: \( v \mid \subseteq \ \text{vertices} \)

**Assumes**: \( c, c \in \text{set conclusions} \Longrightarrow P (c, []) \)

**Assumes**: \( c, c \in \text{set conclusions} \Longrightarrow i \in \text{it-paths} \ (it' \ c) \Longrightarrow P (c, 0\#is) \)

**Shows**: \( P \ v \)

**Using** assms by (cases v; rename-tac is; case-tac is; auto)

**Fun** \( \text{nodeOf} :: \ {'\text{form} \ \text{vertex} \ \Rightarrow \ ('\text{form}, \ '\text{rule}) \ \text{graph-node} \ '} \)

\[ \text{nodeOf} (pf, []) = \text{Conclusion} \ pf \]

\[ \text{nodeOf} (pf, i\#is) = \text{iNodeOf} \ (\text{tree-at} \ (it' \ pf) \ \text{is}) \]
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-inf (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 \( \Rightarrow \) nat list \( \Rightarrow \) nat \( \Rightarrow \) subst \( \Rightarrow \) ('form, 'var) out-port)  

**Where**  
\[ \text{hyp-edge-from } c \text{ is } n s = \{(c, 0 \# \text{hyp-port-path-for } (it' c) \text{ is } (\text{subst } s (\text{freshen } n \text{ anyP})), \ hyp-port-k-for (it' c) \text{ is } (\text{subst } s (\text{freshen } n \text{ anyP}))\} \]

**Definition** hyp-edge-at :: 'form \( \Rightarrow \) nat list \( \Rightarrow \) nat \( \Rightarrow \) subst \( \Rightarrow \) ('form, 'var) edge''  

**Where**  
\[ \text{hyp-edge-at } c \text{ is } n s = (\text{hyp-edge-from } c \text{ is } n s, \text{hyp-edge-to } c \text{ is}) \]

**Lemma** fst-hyp-edge-at[simp]:  
\[ \text{fst } (\text{hyp-edge-at } c \text{ is } n s) = \text{hyp-edge-from } c \text{ is } n s \text{ by (simp add: hyp-edge-at-def)} \]

**Lemma** snd-hyp-edge-at[simp]:  
\[ \text{snd } (\text{hyp-edge-at } c \text{ is } n s) = \text{hyp-edge-to } c \text{ is by (simp add: hyp-edge-at-def)} \]

**Inductive-set** edges where  
\[ \text{regular-edge: } c \in \text{ set conclusions } \Rightarrow is \in \text{it-paths } (it' c) \Rightarrow \text{edge-at } c \text{ is } c \in \text{edges} \]
\[ \text{| hyp-edge: } c \in \text{ set conclusions } \Rightarrow is \in \text{it-paths } (it' c) \Rightarrow \text{tree-at } (it' c) \text{ is } \text{HNode } n s \text{ ants } \Rightarrow \text{hyp-edge-at } c \text{ is } n s \in \text{edges} \]

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

**Lemma** edge-from-valid-out-port:  
\[ \text{assumes } p \in \text{it-paths } (it' c) \]
\[ \text{assumes } c \in \text{set conclusions} \]
\[ \text{shows } \text{valid-out-port } (\text{edge-from } c \text{ p}) \]
\[ \text{using } \text{assms} \]
\[ \text{by (auto simp add: edge-from-def intro: iwf-outPort iwf-it)} \]

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

**Lemma** hyp-edge-from-valid-out-port:  
\[ \text{assumes } is \in \text{it-paths } (it' c) \]
\[ \text{assumes } c \in \text{set conclusions} \]
\[ \text{assumes } \text{tree-at } (it' c) \text{ is } \text{HNode } n s \text{ ants} \]
\[ \text{shows } \text{valid-out-port } (\text{hyp-edge-from } c \text{ is } n s) \]
\[ \text{using } \text{assms} \]
\[ \text{by (auto simp add: hyp-edge-from-def intro: iwf-outPort hyps-exist')} \]

**Lemma** hyp-edge-to-valid-in-port:  
\[ \text{assumes } is \in \text{it-paths } (it' c) \]
\[ \text{assumes } c \in \text{set conclusions} \]
\[ \text{assumes } \text{tree-at } (it' c) \text{ is } \text{HNode } n s \text{ ants} \]
\[ \text{shows } \text{valid-in-port } (\text{hyp-edge-to } c \text{ is}) \]
\[ \text{using } \text{assms by (auto simp add: hyp-edge-to-def)} \]
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-import:
  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, θ ≠ 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 (bb@[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 (bb@[?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)
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)
  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

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

from (scope' \(v' p' v\))
have \((c, is) \in\) vertices unfolding \(v = \_\) by (rule scope-valid)
hence \((c, is) \in\) scope \(((c, is'), (p')\)
proof (rule scope.intros)
fix \(pth t\)
assume terminal-vertex \(t\)
hence \(\text{snd } t = [\_] \) by auto
from path-has-prefixes[OF \langle path (c, is) t pth \rangle \langle \text{snd } t = [\_], simplified, OF \langle \text{prefix} (is' \@ [i]) \text{ is} \rangle \rangle\]
show \(((c, is'), (p')) \in\) snd \('\ set pth\ unfolding \(p' = \_\) \(v' = \_\).
qed
thus \(v \in\) scope \((v', p')\) using \((v = \_\) \(v' = \_\) by simp
qed

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

have \(\forall \ e \in\) edges. valid-out-port \((\text{fst } e)\) \& valid-in-port \((\text{snd } e)\)
by (auto elim!: edges.cases simp add: edge-at-def
dest: edge-from-valid-out-port edge-to-valid-in-port
dest: hyp-edge-from-valid-out-port hyp-edge-to-valid-in-port)
thus \(\forall (ps1, ps2) \in\) 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 \((\text{snd } v')\) = length \((\text{snd } v)\)
using assms by induction (auto elim!: edge-step)

fun vidx :: 'form vertex \Rightarrow \text{nat} where
vidx \((c, [])\) = isidx \[\text{fidx conc-forms } c\]
\[\text{vidx} (c, \#is) = \text{iAnnot} (\text{tree-at} (it' c) \text{ is})\]

lemma mg-vidx-inj: inj-on \(vidx\) \(\text{fset vertices}\)
by (rule inj-on1)
(auto simp add: mem-vertices iAnnot-globalize simp del: iAnnot.simps)
lemma vidx-not-v-away[simp]: \( \forall v \mid\in\) vertices \(\Rightarrow\) \(\exists v \neq v\)-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: \(((v_1, p_1), (v_2, p_2)) \in\) 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 \mid\in\) vertices
using valid-edges by auto
from assms
have \(\exists i. \\text{fst v_1 = \text{fst v_2}} \land \text{prefix (snd v_1 @\{i\}) (snd v_2)} \land p' = \text{in-port-at v_1} \ i\)
by (cases v_1; cases v_2; auto elim!: edge-step)
hence scope' v_1 p' v_2
unfolding scope-valid-inport[OF \((v_2 \mid\in\) vertices)\].
hence v_2 \in scope (v_1, p')
unfolding in-scope[OF \((v_2 \mid\in\) vertices\)].
thus \((v_2, p_2) = (v_1, p') \lor v_2 \in\) 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 \(\exists e \in\) edges. \(\text{snd e = (v, p)}\)
proof(induction v)
fix c cis
assume valid-in-port ((c, cis), p)
hence c \in set conclusions by (auto simp add: mem-vertices)
show \(\exists e \in\) edges. \(\text{snd e = ((c, cis), p)}\)
proof(cases cis)
case Nil
with \(\text{valid-in-port ((c, cis), p)}\)
have [simp]: \(p = \text{plain-ant c}\) by simp
have [] \in \text{it-paths (it' c)} by simp
with \(\langle c \in\) set conclusions\)
have edge-at c [] \in\) edges by (rule regular-edge)
moreover
have \(\text{snd (edge-at c []) = ((c, []), \text{plain-ant c})}\)
by (simp add: edge-to-def)
ultimately

show ?thesis by (auto simp add: Nil simp del: snd-edge-at)

next
case (Cons c’ is)
with (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)

qed

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 [∈ 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

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)
proof

lconsts closed subst subst-lconsts subst-renameLCs anyP vidx inst edges

sol

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 c 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 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

qed
using regular-edge snoc by (simp add: labelAtIn-def edge-at-def)
finally show \(?\)thesis.

qed

next

case \((\text{hyp-edge } c \text{ is } n \text{ s } \text{ants})\)
\begin{itemize}
  \item let \(\exists f = \text{subst } s (\text{freshen } n \text{ anyP})\)
  \item let \(\exists h = \text{hyp-port-h-for } (it' c) \text{ is } \exists f\)
  \item let \(\exists his = \text{hyp-port-path-for } (it' c) \text{ is } \exists f\)
  \item let \(\exists t1 = \text{tree-at } (it' c) \exists his\)
  \item let \(\exists t2 = \text{tree-at } (it' c) \exists is\)
\end{itemize}

from \((c \in \text{set conclusions}) \text{ is } \exists i \text{-paths } (it' c) \text{ (tree-at } (it' c) \text{ is } HNode n \text{ s } \text{ants})\)
have \(\exists f \in \text{hyps-along } (it' c) \text{ is}\)
by \((\text{rule hyps-exist})\)

from \(((v_1, p_1), v_2, p_2) = \text{hyp-edge-at } c \text{ is } n \text{ s}\)
have \((v_1, p_1) = \text{hyp-edge-from } c \text{ is } n \text{ s}\) using \(\text{fst-hyp-edge-at by (metis fast-force)}\)
hence \([\text{simp}] v_1 = (c, 0 \# \exists his)\) by \((\text{simp add: hyp-edge-from-def})\)

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

qed

lemma node-disjoint-fresh-vars:
\begin{itemize}
  \item assumes \(n \in \text{sset nodes}\)
  \item assumes \(i < \text{length } (\text{inPorts'} n)\)
  \item assumes \(i' < \text{length } (\text{inPorts'} n)\)
  \item shows \(a\text{-fresh } (\text{inPorts'} n \mid i) \cap \text{a-fresh } (\text{inPorts'} n \mid i') = \{\} \lor i = i'\)
  \item using \(\text{assms no-multiple-local-consts}\)
  \item by \((\text{fastforce simp add: nodes-def stream.set-map})\)
\end{itemize}

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
\begin{itemize}
  \item fix \(v p \text{ var } v'\)
  \item assume valid-in-port \((v, p)\)
  \item hence \(v \in\mid \text{vertices by simp}\)
\end{itemize}

obtain \(c \text{ is where } v = (c, \text{is})\) by \((\text{cases } v, \text{auto})\)

from \((\text{valid-in-port } (v, p)) : (v = \cdot)\)
have \((c, \text{is}) \in\mid \text{vertices and } p \in\mid \text{inPorts } (\text{nodeOf } (c, \text{is}))\) by \((\text{simp-all})\)

hence \(c \in \text{set conclusions by } (\text{simp add: mem-vertices})\)

from \((p \in\mid \rightarrow) \text{ obtain } i \text{ where }\)
\begin{itemize}
  \item \(i < \text{length } (\text{inPorts'} (\text{nodeOf } (c, \text{is})))\) and
  \item \(p = \text{inPorts'} (\text{nodeOf } (c, \text{is})) \mid i\) by \((\text{auto simp add: inPorts-fset-of in-set-conv-nth})\)
\end{itemize}
hence \( p = \text{in-port-at} \ (c, \text{is}) \ i \text{ by (cases is)} \text{ auto} \)

**assume** \( v' \mid \text{vertices} \)

**then obtain** \( c' \text{ is'} \text{ where} \ v' = (c',is') \text{ by (cases v', auto)} \)

**assume** \( \text{var} \in \text{local-vars} \ (\text{nodeOf v}) \ p \)

**hence** \( \text{var} \in \text{a-fresh} \ p \text{ by simp} \)

**assume** \( \text{freshenLC} \ (\text{vidx v}) \ \text{var} \in \text{subst-lconsts} \ (\text{inst v'}) \)

then **obtain** \( i' \text{ where} \ is' = 0 \# is'' \text{ and} is''' \in \text{it-paths} \ (i' \ c') \)

using \( \langle v' \mid \text{vertices} \rangle \text{ by (cases is') (auto simp add: (v'='))} \)

**note** \( \text{freshenLC} \ (\text{vidx v}) \ \text{var} \in \text{subst-lconsts} \ (\text{inst v'}) \)

also **have** subst-lconsts \( (\text{inst v'}) = \text{subst-lconsts} \ (\text{iSubst (tree-at (it' c') is''))} \)

by \( \text{(simp add: (v'=') (is'='))} \)

also from \( \langle i'' \in \text{it-paths} \ (i' \ c') \rangle \)

**have** \( \ldots \subseteq \text{fresh-at-path} \ (i' \ c') \text{ is''} \cup \text{range} \ (\text{freshenLC v-away}) \)

by \( \text{(rule globalize-local consts)} \)

finally **have** freshenLC \( (\text{vidx v}) \ \text{var} \in \text{fresh-at-path} \ (i' \ c') \text{ is'''} \)

using \( \langle v \mid \text{vertices} \rangle \text{ by auto} \)

then **obtain** is'''' where prefix is'''' is'''' and freshenLC \( (\text{vidx v}) \ \text{var} \in \text{fresh-at} \ (i' \ c') \text{ is''''} 

unfolding fresh-at-path-def by auto

then **obtain** \( i' \text{ is'''' where} \prefix (i''''\@[i']) \text{ is''''} \)

and freshenLC \( (\text{vidx v}) \ \text{var} \in \text{fresh-at} \ (i' \ c') \text{ (is'''''@[i'])} \)

using append-butlast-last-id[\text{where} xs = is''''', symmetric]

apply \( \text{(cases is'''' = [])} \)

apply \( \text{(auto simp del: fresh-at-snoc append-butlast-last-id)} \)

apply metis

done

from \( \langle i'' \in \text{it-paths} \ (i' \ c') \rangle \langle \text{prefix} (i''''\@[i']) \text{ is''''} \rangle \)

**have** \( \langle i''''\@[i'] \rangle \in \text{it-paths} \ (i' \ c') \text{ by (rule it-paths-prefix)} \)

**hence** is''''' \in \text{it-paths} \ (i' \ c') \text{ using append-prefixD it-paths-prefix by blast} \)

from this freshenLC \( (\text{vidx v}) \ \text{var} \in \text{fresh-at} \ (i' \ c') \text{ (is''''\@[i'])} \)

**have** \( c = c' \land is = 0 \# is''''' \land \text{var} \in \text{a-fresh (inPorts' (iNodeOf (tree-at (it' c') is''''')))} \text{ (is'''''@i')} \)

unfolding fresh-at-def using \( \langle v \mid \text{vertices} \rangle \langle v' \mid \text{vertices} \rangle \)

apply \( \text{(cases is)} \)

apply \( \text{(auto split: if-splits simp add: iAnnot-globalize it-paths-butlast (v=} v'=' is'=} simp del: iAnnot-simps)} \)

done

**hence** \( c' = c \) \text{ and} \ is = 0 \# is''''' \text{ and} \text{ var} \in \text{a-fresh (inPorts' (iNodeOf (tree-at (it' c') is'''''}) \text{ (is''''')} \text{ } \text{ (is''''')} \text{ (i')} \text{ by simp-all}} \)

from \( \langle (i''''\@[i']) \rangle \in \text{it-paths} \ (i' \ c') \rangle \)

**have** \( \text{i' < length (inPorts' (nodeOf (c, is))} \)

using \( \text{iwf-length-inPorts[OF iwf-it[OF (c \in set conclusions)]]} \)

by \( \text{(auto elim!: it-paths-SnocE simp add: (is=} c'=} \text{ order.strict-trans2)} \)

**have** nodeOf \( (c, \text{is}) \in \text{set nodes} \text{ unfolding (is=} c'=} \text{ nodeOf.simps} \)

by \( \text{(rule iNodeOf-tree-at[OF iwf-it[OF (c \in set conclusions)] (is'''''} \text{ (is'''''} \text{ (i')} \text{ unfolded (c'=} \text{})})} \)

64
from \( \{ \text{var} \in a\text{-fresh} \ (\text{inPorts}' \ (\text{tree-at} \ (\text{it}' \ c') \ is''') \ ! \ i') \} \)
\( \{ \text{var} \in a\text{-fresh} \ p' \ p = \text{inPorts}' \ (\text{nodeOf} \ (c, is)) \ ! \ i' \} \)
\text{node-disjoint-fresh-vars}(OF)
\( \{ \text{nodeOf} \ (c, is) \in \text{sset nodes} \}
\langle i < \text{length} \ (\text{inPorts}' \ (\text{nodeOf} \ (c, is))) \rangle \langle i' < \text{length} \ (\text{inPorts}' \ (\text{nodeOf} \ (c, is))) \rangle \}
\text{have} \ i' = i \text{ by} \ (\text{auto simp add} \ : \ { \text{is} = -} \langle c' = c \rangle )
\text{from} \ (\text{prefix} \ (\text{is}'@[i']) \ is'')
\text{have} \ \text{prefix} \ (\text{is}@[i']) \ is' \text{ by} \ (\text{simp add} \ : \ { \text{is}' = -} \langle \text{is} = - \rangle )
\text{from} \ (c \in \text{set conclusions}) \ (\text{is}'' \in \text{it-paths} \ (\text{it}' c') \ (\text{prefix} \ (\text{is}@[i']) \ is')
\langle p = \text{in-port-at} \ (c, is) \ i \}
\text{have} \ \text{scope} \ v \ p \ v'
\text{unfolding} \ (v\Rightarrow \langle v'\Rightarrow \langle c' = \Rightarrow \langle is' = \Rightarrow \langle i'\Rightarrow \text{by} \ (\text{auto intro: scope'.intros})
\text{thus} \ v' \in \text{scope} \ (v, p) \text{ using} \ \text{valid-in-port} \ (v, p) \text{ by} \ (\text{simp add: in-scope})
\text{qed}

\text{sublocale} \ \text{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..}

\text{sublocale} \ \text{tpg:Tasked-Proof-Graph freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP antecedent consequent rules assumptions conclusions vertices nodeOf edges vidx inst}
\text{proof}
\text{show} \ \text{set} \ (\text{map} \ \text{Conclusion} \ \text{conclusions}) \subseteq \text{nodeOf} \ v \ \text{fset} \ \text{vertices}
\text{proof–}
\{ \text{fix} \ c
\text{assume} \ c \in \text{set conclusions}
\text{hence} \ (c, []) \in\ | \ \text{vertices} \text{ by} \ \text{simp}
\text{hence} \ \text{nodeOf} \ (c, []) \in \text{nodeOf} \ v \ \text{fset} \ \text{vertices}
\text{ by} \ (\text{rule imageI})
\text{hence} \ \text{Conclusion} \ c \in \text{nodeOf} \ v \ \text{fset} \ \text{vertices} \text{ by} \ \text{simp}
\} \text{ thus} \ ?\text{thesis} \text{ by} \ \text{auto}
\text{qed}
\text{qed}
\text{end}
\text{end}
8 Instantiations

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

8.1 Propositional_Formulas

theory Propositional_Formulas
imports
  Abstract-Formula
  HOL-Library.Countable
  HOL-Library.Infinite-Set
begin

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

instance nat :: infinite
  by (intro-classes) simp
instance prod :: (infinite, type) infinite
  by intro-classes (simp add: finite-prod infinite-UNIV)
instance list :: (type) infinite
  by intro-classes (simp add: infinite-UNIV-listI)

lemma countable-infinite-ex-bij: \( \exists f::('a::{countable, infinite})\Rightarrow('b::{countable, infinite}).\ bij f \)
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 \( \exists f.\ bij-betw f (range (to-nat::'a ⇒ nat)) (range (to-nat::'b ⇒ nat)) \)
    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 [simp]: from-nat c = (d::'a) by blast
      with \( c \in range (to-nat::') \) 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)

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

```plaintext
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.

```plaintext
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)
```

```plaintext
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

```plaintext
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
```

```plaintext
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) (a::nat) (v::'var) = curry (SOME f. bij f) (a':nat) (v':'var)) =
  (a = a' ∧ v = v')
apply (rule someI2 [where Q=λf. 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
```

67
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

68
8.3 Incredible_Propositional_Tasks

theory Incredible-Propositional-Tasks
imports
   Incredible-Completeness
   Incredible-Propositional
begin
context ND-Rules-Inst begin
lemma eff-NatRuleI:
  nat-rule rule c ants
  ⇒ entail = (Γ ⊢ subst s (freshen a c))
  ⇒ hyps = ((λant. ((λp. subst s (freshen a p)) | ^ a-hyps ant) | ^ □ | Γ ⊢ subst s (freshen a (a-conc ant)))) | ^ ants)
  ⇒ (Λ ant f. ant ∈ ants ⇒ f ∈ □ | Γ ⇒ freshenLC a • (a-fresh ant) ∩ lconsts f = {})
  ⇒ (Λ 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: \( A \rightarrow A \)

**abbreviation** \( A :: (\text{string,prop-funs}) \text{pform} \)

where \( A \equiv \text{Fun (Const "A")} \)

First the task is defined as an Abstract-Task.

**interpretation** task1-1: Abstract-Task

\[ \text{curry (SOME f. bij f)} :: \text{nat} \Rightarrow \text{string} \Rightarrow \text{string} \]

\[ \lambda \cdot \text{id} \]

\[ \lambda \cdot \{\} \]

\[ \text{closed :: (string, prop-funs) pform} \Rightarrow \text{bool} \]

\[ \text{subst} \]

\[ \lambda \cdot \{\} \]

\[ \lambda \cdot \text{id} \]

\[ \text{Var undefined} \]

\[ \text{antecedent} \]

\[ \text{consequent} \]

\[ \text{prop-rules} \]

\[ [A] \]

\[ [A] \]

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 \( x=\text{Node (\{\{A\}\} \vdash A, Axiom)\{||} \) in exI}

apply clarsimp

apply (rule conjI)

apply (rule task1-1.wf)

apply (solves clarsimp)

apply (solves 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
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
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
```

```
```
```
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: \( P \land Q \Rightarrow R \Rightarrow P \Rightarrow Q \Rightarrow R \)

abbreviation \( B :: \) (string, prop-funs) pform
  where \( B \equiv \text{Fun} \ (\text{Const} \ "B") \) []
abbreviation \( C :: \) (string, prop-funs) pform
  where \( C \equiv \text{Fun} \ (\text{Const} \ "C") \) []

interpretation task2-11: Abstract-Task
  curry \((\text{SOME} \ f \ \text{bij} \ f): \text{nat} \Rightarrow \text{string} \Rightarrow \text{string}\)
  \(\lambda\) - id
  \(\lambda\) - {}
  closed \((\text{string}, \text{prop-funs}) \text{pform} \Rightarrow \text{bool}\)
  subst
  \(\lambda\). id
  Var undefined
  antecedent
  consequent
  prop-rules
  \[\text{Fun imp} \ [\text{Fun and} [A,B],C]\]
  \[\text{Fun imp} \ [A,\text{Fun imp} [B,C]]\]
  by unfold-locales simp-all

abbreviation \( n-andI \equiv \) task2-11.n-rules !! 0
abbreviation \( n-andE1 \equiv \) task2-11.n-rules !! 1
abbreviation \( n-andE2 \equiv \) task2-11.n-rules !! 2
abbreviation \( n-impl \equiv \) task2-11.n-rules !! 3
abbreviation \( n-impE \equiv \) task2-11.n-rules !! 4

lemma \( n-andI \) [simp]: \( n-andI = (\text{andI}, \text{Fun} \ [X,Y]) \)
  unfolding task2-11.n-rules-def by (simp add: prop-rules-def)
lemma \( n-andE1 \) [simp]: \( n-andE1 = (\text{andE}, X) \)
  unfolding task2-11.n-rules-def One-nat-def by (simp add: prop-rules-def)
lemma \( n-andE2 \) [simp]: \( n-andE2 = (\text{andE}, Y) \)
lemma \( n-impl \) [simp]: \( n-impl = (\text{impl}, \text{Fun imp} [X,Y]) \)
  unfolding task2-11.n-rules-def numeral-3-eq-3 by (simp add: prop-rules-def)
lemma \( n-impE \) [simp]: \( n-impE = (\text{impE}, Y) \)
proof
  have \( n-impE = \) task2-11.n-rules !! Suc 3 by simp
  also have \( \ldots = (\text{impE}, Y) \)
    unfolding task2-11.n-rules-def numeral-3-eq-3 by (simp add: prop-rules-def)
  finally show \( \text{thesis} \).
  qed

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

lemma xy-update: \( f = \text{undefined}("X\!":= x, "Y\!":= y) \implies x = f "X" \land y = f "Y" \) by force
lemma y-update: \( f = \text{undefined}("Y\!":= y) \implies y = f "Y" \) 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 \Rightarrow string \Rightarrow string
\lambda. id
\lambda. \{\}
closed :: (string, prop-funs) pform \Rightarrow bool
  subst
\lambda. \{\}
\lambda. id
Var undefined
antecedent
consequent
  prop-rules
  [Fun imp [Fun and [A,B],[C]]]
  [Fun imp [A,Fun imp [B,C]]]
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]]]|) \implies Fun imp [A, Fun imp [B,C]],NatRule n-impI)
  (|[Node (|[Fun imp [Fun and [A,B],[C]], A]|) \implies Fun imp [B,C],NatRule n-impI])
  (|[Node (|[Fun imp [Fun and [A,B],[C]], A]|) \implies C,NatRule n-impE])
  (|[Node (|[Fun imp [Fun and [A,B],[C]], A]|) \implies Fun imp [Fun and [A,B],[C],Axiom] |])
  (Node (|[Fun imp [Fun and [A,B],[C]], A]|) \implies Fun and [A,B],NatRule n-andI)
  (|[Node (|[Fun imp [Fun and [A,B],[C]], A]|) \implies A,Axiom |])
  (Node (|[Fun imp [Fun and [A,B],[C]], A]|) \implies B,Axiom |])
  |)
  |)
in exI)
apply clarsimp
apply (rule conjI)
apply (rule task1-1.wf)
apply (solves \langle clarsimp; metis n-impI snth-smap snth-sset\rangle)
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 \langle fastforce simp: propositional.f-antecedent-def\rangle)
apply (rule task1-1.wf)
apply (solves \langle clarsimp; metis n-impI snth-smap snth-sset\rangle)
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 simp add: propositional.f-antecedent-def))
apply clarsimp
apply (erule disjE)
apply (auto intro: task1-1.wf intro!: task1-1.eff.intros(1))[1]
apply (rule task1-1.wf)
apply (solves (clarsimp simp add: propositional.f-antecedent-def))
apply clarsimp
apply (erule disjE)
apply (solves (erule task1-1.wf; auto intro: task1-1.eff.intros(1);))
apply (solves (erule 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

74
This theory contains an example instantiation of Abstract-Formulas with an formula type with local constants. It is a rather ad-hoc type that may not be very useful to work with, though.

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

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

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

type-synonym schema = var list × 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

\[ \text{fun } \text{fv-schema} :: \text{schema} \Rightarrow \text{var set where} \]
\[ \text{fv-schema } (\text{ps},f) = \text{fv } f - \text{ set ps} \]

\[ \text{definition } \text{fv-subst} :: \text{subst} \Rightarrow \text{var set where} \]
\[ \text{fv-subst } s = \bigcup (\text{fv-schema } \circ \text{ ran (map-of s)}) \]

\[ \text{definition } \text{fv-subst1} \text{ where} \]
\[ \text{fv-subst1 } s = \bigcup (\text{fv } \circ \text{ snd } \circ \text{ set } s) \]

\[ \text{lemma } \text{fv-subst-Nil[simp]}: \text{fv-subst1 } [] = \{ \} \]

\[ \text{unfolding } \text{fv-subst1-def} \text{ by auto} \]

Local constants, separate from free variables.

\[ \text{fun } \text{lc} :: \text{form} \Rightarrow \text{lconst set where} \]
\[ \text{lc } (\text{Var } v \text{ xs}) = \text{Union } (\text{lc } \circ \text{ set } \text{xs}) \]
\[ | \text{lc } (\text{LC } c) = \{ c \} \]
\[ | \text{lc } (\text{Op } n \text{ xs}) = \text{Union } (\text{lc } \circ \text{ set } \text{xs}) \]
\[ | \text{lc } (\text{Quant } n v f) = \text{lc } f \]

\[ \text{fun } \text{lc-schema} :: \text{schema} \Rightarrow \text{lconst set where} \]
\[ \text{lc-schema } (\text{ps},f) = \text{lc } f \]

\[ \text{definition } \text{lc-subst1} \text{ where} \]
\[ \text{lc-subst1 } s = \bigcup (\text{lc } \circ \text{ snd } \circ \text{ set } s) \]

\[ \text{fun } \text{lc-subst} :: \text{subst} \Rightarrow \text{lconst set where} \]
\[ \text{lc-subst } s = \bigcup (\text{lc-schema } \circ \text{ set } s) \]

\[ \text{fun } \text{map-lc} :: (\text{lconst} \Rightarrow \text{lconst}) \Rightarrow \text{form} \Rightarrow \text{form where} \]
\[ \text{map-lc } f (\text{Var } v \text{ xs}) = \text{Var } v (\text{map } (\text{map-lc } f) \text{ xs}) \]
\[ | \text{map-lc } f (\text{LC } n) = \text{LC } (f n) \]
\[ | \text{map-lc } f (\text{Op } n \text{ xs}) = \text{Op } n (\text{map } (\text{map-lc } f) \text{ xs}) \]
\[ | \text{map-lc } f (\text{Quant } n v f') = \text{Quant } n v (\text{map-lc } f f') \]

\[ \text{lemma } \text{fv-map-lc}[\text{simp}]: \text{fv } (\text{map-lc } p f) = \text{fv } f \]
\[ \text{by (induction } f \text{) auto} \]

\[ \text{lemma } \text{lc-map-lc}[\text{simp}]: \text{lc } (\text{map-lc } p f) = p \circ \text{lc } f \]
\[ \text{by (induction } f \text{) auto} \]

\[ \text{lemma } \text{map-lc-map-lc}[\text{simp}]: \text{map-lc } p1 (\text{map-lc } p2 f) = \text{map-lc } (p1 \circ p2) f \]
\[ \text{by (induction } f \text{) auto} \]

\[ \text{fun } \text{map-lc-subst1} :: (\text{lconst} \Rightarrow \text{lconst}) \Rightarrow (\text{var } \times \text{ form}) \text{ list} \Rightarrow (\text{var } \times \text{ form}) \text{ list where} \]
\[ \text{map-lc-subst1 } f \text{ s} = \text{map } (\text{apsnd } (\text{map-lc } f)) \text{ s} \]

\[ \text{fun } \text{map-lc-subst} :: (\text{lconst} \Rightarrow \text{lconst}) \Rightarrow \text{subst} \Rightarrow \text{subst where} \]
\[ \text{map-lc-subst } f \text{ s} = \text{map } (\text{apsnd } (\text{apsnd } (\text{map-lc } f))) \text{ s} \]

\[ \text{lemma } \text{map-lc-noop}[\text{simp}]: \text{lc } f = \{} \Rightarrow \text{map-lc } p f = f \]
\[ \text{by (induction } f \text{) (auto simp add: map-idL)} \]

\[ \text{lemma } \text{map-lc-cong}[\text{cong}]: (\forall x. x \in \text{lc } f \Rightarrow f1 x = f2 x) \Rightarrow \text{map-lc } f1 f = \text{map-lc } f2 f \]
\[ \text{by (induction } f \text{) auto} \]
lemma [simp]: \( \text{fv-subst1} (\text{map} (\text{apsnd} (\text{map}-\text{lc}\ p)) s) = \text{fv-subst1} s \)

unfolding \text{fv-subst1-def}

by auto

lemma \text{map-lc-subst-cong}: 

assumes (\( \forall x. x \in \text{lc-subst} s \Rightarrow f_1 x = f_2 x \) )

shows \( \text{map-lc-subst} f_1 s = \text{map-lc-subst} f_2 s \)

by (force \text{intro!}: \text{map-lc-cong assms})

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

fun \text{subst1} :: \((\text{var} \times \text{form})\ \text{list} \Rightarrow \text{form} \Rightarrow \text{form}\) where

\text{subst1} s (Var v []) = (case \text{map-of} s v of Some f \Rightarrow f | None \Rightarrow \text{Var} v [])

| \text{subst1} s (Var v xs) = \text{Var} v xs |

| \text{subst1} s (LC n) = LC n |

| \text{subst1} s (Op n xs) = Op n (map (\text{subst1} s) xs) |

| \text{subst1} s (Quant n v f) = |

(if v \in \text{fv-subst1} s then

(let v' = fresh-for (fv-subst1 s)

in Quant n v' (\text{subst1} ((v, \text{Var} v' [])\#s) f))

else Quant n v (\text{subst1} s f))

lemma \text{substit1-Nil[simp]}: \text{subst1} [] f = f

by (induction []::(\text{var} \times \text{form})\ \text{list} \ f \ \text{rule:subst1.induct})

(auto simp add: \text{map-id} \text{I} \text{split: option.splits})

lemma \text{lc-subst1}: \text{lc} (\text{subst1} s f) \subseteq \text{lc} f \cup \bigcup \{\text{lc} \circ \text{snd} \text{'} \text{set} s\} |

by (induction s f rule: \text{subst1.induct})

(auto split: \text{option.split dest: map-of-someD simp add: Let-def})

lemma \text{apsnd-def'}: \text{apsnd} f = (\lambda (k, v). (k, f v))

by auto

lemma \text{map-of-map-apsnd}: 

map-of (map (\text{apsnd} f) xs) = map-option f \circ map-of xs

unfolding \text{apsnd-def'} by (rule \text{map-of-map})

lemma \text{map-lc-subst1[simp]}: \text{map-lc} p (\text{subst1} s f) = \text{subst1} (\text{map-lc-subst1} p s) (\text{map-lc} p f)

apply (induction s f rule: \text{subst1.induct})

apply (auto split: \text{option.split simp add: map-of-map-apsnd Let-def})

apply (\text{subst subst1.simps}, auto split: \text{option.split})

apply (\text{subst subst1.simps}, auto split: \text{option.split})

apply (\text{subst subst1.simps}, auto split: \text{option.split})

apply (\text{subst subst1.simps}, auto split: \text{option.split})

apply (\text{subst subst1.simps}, auto split: \text{option.split}, simp only: \text{Let-def map-lc.simps})

apply (\text{subst subst1.simps}, auto split: \text{option.split})

done

fun \text{subst'} :: \text{subst} \Rightarrow \text{form} \Rightarrow \text{form} where

\text{subst'} s (Var v xs) = |

(case \text{map-of} s v of None \Rightarrow (\text{Var} v (\text{map} (\text{subst'} s) xs)) |

| Some (ps, rhs) \Rightarrow |

if length ps = length xs

then \text{subst1} (\text{zip} ps (\text{map} (\text{subst'} s) xs)) rhs

else (\text{Var} v (\text{map} (\text{subst'} s) xs))

77
| 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 run-map-option-comp[simp]:**
  ran (map-option f ∘ m) = f · ran m
unfolding comp-def by (rule run-map-option)

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

**Lemma fu-subst-map-apsnd-map-lc[simp]:**
  fu-subst (map (apsnd (apsnd (map-lc p)))) s = fu-subst s
unfolding fu-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'.simp, auto split: option.splits cong: map-cong)[1])
  apply (solves (subst subst'.simp, auto split: option.splits cong: map-cong)[1])
  apply (solves (subst subst'.simp, auto split: option.splits cong: map-cong)[1])
  apply (solves (subst subst'.simp, auto split: option.splits cong: map-cong)[1])
  apply (solves (subst subst'.simp, auto split: option.splits cong: map-cong)[1])
  apply (solves (subst subst'.simp, auto split: option.splits cong: map-cong)[1])
  apply (solves (subst subst'.simp, auto split: option.splits cong: map-cong)[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 fastforce)
apply (solves (metis map-lc-subst-cong))
apply (solves (srule lc-subst))
apply (solves fastforce)
apply (solves fastforce)
apply (solves fastforce)
apply (solves fastforce)
apply (solves fastforce)
apply (solves fastforce)
apply (solves (srule exI[where x = []], simp))
apply (solves (rename_tac f, rule_tac x = [([], f)] in exI, simp))
done

declare predicatesubst-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 ≡ Var 13 []
abbreviation P :: form ⇒ form
  where P f ≡ Op "P" [f]
abbreviation Q :: form ⇒ form
  where Q f ≡ Op "Q" [f]
abbreviation imp :: form ⇒ form ⇒ form
  where imp f1 f2 ≡ Op "imp" [f1, f2]
abbreviation forallX :: form ⇒ form
  where forallX f ≡ Quant "all" 9 f

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

fun consequent :: prop-rule ⇒ form list
  where consequent allI = [forallX (P x)]
        | consequent allE = [P t]
        | consequent impI = [imp X Y]
        | consequent impE = [Y]

abbreviation allI-input where allI-input ≡ Antecedent {||} (P (LC 0)) {0}
abbreviation impI-input where impI-input ≡ Antecedent {X} Y {}

fun antecedent :: prop-rule ⇒ (form, lconst) antecedent list
  where antecedent allI = [allI-input]
        | antecedent allE = [plain-ant (forallX (P x))]
        | antecedent impI = [impI-input]
        | antecedent impE = [plain-ant (imp X Y), plain-ant X]

interpretation predicate: Abstract-Rules
  curry to-nat :: nat ⇒ var ⇒ var
map-lc
lc
closed
subst
lc-subst
map-lc-subst
Var 0 []
antecedent
consequent
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

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

A typical task with local constants:: ∀ x. Q x → Q x

First the task is defined as an Abstract-Task.

interpretation task: Abstract-Task
  curry to-nat :: nat ⇒ var ⇒ var

81
Then we show, that this task has a proof within our formalization of natural deduction by giving a concrete proof tree.

**abbreviation** \( lx :: \text{nat} \) where \( lx \equiv \text{to-nat} (1::\text{nat},0::\text{nat}) \)

**abbreviation** \( \text{base-tree} :: ( (\text{form fset} \times \text{form}) \times (\text{prop-rule} \times \text{form}) \text{NatRule} ) \text{ tree} \) where
\[ \text{base-tree} \equiv \text{Node} (\{\{Q (LC lx)\}\} \vdash Q (LC lx), \text{Axiom} ) \{||\} \]

**abbreviation** \( \text{imp-tree} :: ( (\text{form fset} \times \text{form}) \times (\text{prop-rule} \times \text{form}) \text{NatRule} ) \text{ tree} \) where
\[ \text{imp-tree} \equiv \text{Node} (\{||\} \vdash \text{imp} (Q (LC lx)) (Q (LC lx)), \text{NatRule} (\text{impI}, \text{imp} X Y)) \{||\} \]

**abbreviation** \( \text{solution-tree} :: ( (\text{form fset} \times \text{form}) \times (\text{prop-rule} \times \text{form}) \text{NatRule} ) \text{ tree} \) where
\[ \text{solution-tree} \equiv \text{Node} (\{||\} \vdash \forall X (\text{imp} (Q x) (Q x)), \text{NatRule} (\text{allI}, \forall X (P x))) \{||\} \]

**abbreviation** \( s1 \) where \( s1 \equiv [(12, [[9], \text{imp} (Q x) (Q x)])] \)

**abbreviation** \( s2 \) where \( s2 \equiv [(10, [[], Q (LC lx)), (11, [||], Q (LC lx))] \]

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

**lemma** \( \text{subst1-simps[simp]} : \)
\[ \text{subst} s1 (P (LC n)) = \text{imp} (Q (LC n)) (Q (LC n)) \]
\[ \text{subst} s1 (\forall X (P x)) = \forall X (\text{imp} (Q x) (Q x)) \]
by \( \text{simp-all} \)

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

**lemma** \( \text{substI1} : \forall X (\text{imp} (Q x) (Q x)) = \text{subst} s1 (\text{predicate.freshen} 1 (\forall X (P x))) \)
by \( \text{(auto simp add: predicate.freshen-def Let-def)} \)

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

**declare** \( \text{subst.simps[simp del]} \)

**lemma** \( \text{task.solved} \)
unfolding \( \text{task.solved-def} \)
apply clarsimp
apply \( \text{(rule-tac x=|| in exI)} \)
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.wf)
apply (solves (simp add: predicate.f-antecedent-def predicate.freshen-def))
apply (solves simp)
apply (rule conjI)
apply (solves simp)
apply (solves substI1)
apply (simp add: predicate.f-antecedent-def predicate.freshen-def)
apply (subst antecedent.sel(2))
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.nEff.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)
apply (solves simp)
apply simp

apply (solves (auto intro: task.uf 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] ]

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
carry 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