The meta theory of the Incredible Proof Machine

Joachim Breitner        Denis Lohner

August 16, 2018

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

Contents

1 Introduction 2

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

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

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

5 Natural Deduction 27
   5.1 Natural_Deduction 27

6 Correctness 29
   6.1 Incredible_Correctness 29

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

8 Instantiations 66
   8.1 Propositional_Formulas 66
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 proof 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 proof 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 Δ (Γ ⊢ c) = (Γ |∪| Δ ⊢ c)

end

2.2 Indexed_FSet

theory Indexed-FSet
imports
  HOL-Library.FSet
begin

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

context includes fset.lifting
begin

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

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

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

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

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

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

lemma indexed-fmember-is-fmember: x ∈· i s ⇒ x ∈ s
proof (induction rule: indexed-fmember.induct)
case (goal i s)
  hence i < length (list-of-fset s) by (metis length-list-of-fset)
  hence list-of-fset s ! i ∈ set (list-of-fset s) by (rule nth-mem)
  thus list-of-fset s ! i ∈ s by (metis mem-fset-from-list fset-from-list-of-fset)
qed

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-cov-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 ⇒ ('a × 'a) list where
indexed-members s = zip [0..<size s] (list-of-fset s)

lemma mem-set-indexed-members:
  (i,x) ∈ set (indexed-members s) ←→ x ∈| i s
  unfolding indexed-members-def indexed-fmember.simps
  by (force simp add: set-zip)

lemma mem-set-indexed-members[simp]:
  t ∈ set (indexed-members s) ←→ snd t ∈| 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 ∈| 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 ⇒ 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-inj[simp]: x ∈| i s ⇒ y ∈| i s ⇒ fidx s x = fidx s y ←→ x = y
  by (auto dest!: fmember-is-indexed-fmember simp add: indexed-fmember-unique)
lemma inj-on-fidx: inj-on (fidx vertices) (fset vertices)
  by (rule inj-onI) (auto simp: fmember.rep-eq [symmetric])
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-path-P-ConsE: it-pathsP t (i#is)
inductive-cases it-path-P-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-path-P-ConsE[to-set]
lemmas it-paths-RNodeE = it-path-P-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!: be1 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
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 \textit{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 lc onsts :: '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-lc onsts :: 'subst ⇒ 'var set
  -- A variable-changing function can be mapped over a substitution
  fixes subst-r enameLCs :: ('var ⇒ 'var) ⇒ ('subst ⇒ 'subst)
  -- A most generic formula, can be substituted to anything.
  fixes anyP :: 'form
  assumes freshenLC-e q-i [simp]: freshenLC a v = freshenLC a = v \iff a = a' ∧ v = v'
  assumes lc onsts-r enameLCs: lc onsts (renameLCs p f) = p • lc onsts f
  assumes rename-closed: lc onsts f = {} ⇒ renameLCs p f = f
  assumes subst-closed: closed f ⇒ subst s f = f
  assumes closed-no-lc onsts: closed f ⇒ lc onsts f = {}
  assumes fv-subst: lc onsts (subst s f) ⊆ lc onsts f ∪ subst-lc onsts s
  assumes rename-r ename: renameLCs p1 (rewriteLCs p2 f) = renameLCs (p1 ◦ p2) f
  assumes rename-subst: rewriteLCs p (subst s f) = subst (rewriteLCs p s) (renameLCs p f)
  assumes renameLCs-cong: (∀ x. x ∈ lc onsts f ⇒ f1 x = f2 x) ⇒ rewriteLCs f1 = rewriteLCs f2 f
  assumes subst-r enameLCs-cong: (∀ x. x ∈ subst-lc onsts s ⇒ f1 x = f2 x) ⇒ subst-r enameLCs f1 = subst-r enameLCs f2 s
  assumes subst-lc onsts-subst-r enameLCs: subst-lc onsts (subst-r enameLCs p s) = p • subst-lc onsts s
  assumes lc onsts-anyP: lc onsts anyP = {}
  assumes subst-empty: ∃ s. (∀ f. subst s f = f) ∧ subst-lc onsts s = {}
  assumes subst-anyP-is-empty: ∃ s. subst s anyP = f
begin
  definition freshen :: nat ⇒ 'form ⇒ 'form where
    freshen n = rewriteLCs (freshenLC n)
  definition empty-subst :: 'subst where

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

lemma empty-subst-spec:
(∀ f. subst empty-subst f = f) ∧ subst-consts 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-consts-empty-subst[simp]: subst-consts empty-subst = {}
by (metis empty-subst-spec)

lemma lemons-freshen: lemons (freshen a f) = freshenLC a ' lemons f

unfolding freshen-def by (rule lemons-renameLCs)

lemma freshen-closed: lemons f = {} ⇒ freshen a f = f

unfolding freshen-def by (rule rename-closed)

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

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

lemma rename-freshen[simp]: x ∈ V ⇒ rename V i (isidx is) f (freshenLC i x) = freshenLC (isidx is) x

unfolding rename-def by simp

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

lemma rename-noop:
  x ≠ freshenLC from ' V ⇒ rename V from to f x = f x
by (auto simp add: rename-def split: if-splits)

lemma rename-rename-noop:
  freshenLC from ' V ∩ lemons form = {} ⇒ renameLCs (rename V from to f) form = renameLCs f form
by (intro renameLCs-cong rename-noop) auto

lemma rename-subst-noop:
  freshenLC from ' V ∩ subst-consts s = {} ⇒ subst-renameLCs (rename V from to f) s = subst-renameLCs f s
by (intro subst-renameLCs-cong rename-noop) auto
end end
3.2 Abstract_Rules

theory Abstract-Rules
imports
    Abstract-Formula
begin

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

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

Each antecedent of a rule consists of

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

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

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

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

datatype ('form, 'var) antecedent =

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 ∈ set rules. consequent xs ≠ []

assumes no-local-consts-in-consequences: ∀ xs ∈ set rules. ∪ (lconsts · (set (consequent xs))) = {}
assumes no-multiple-local-consts:
    ⊓ r i i’ . r ∈ set rules ⟹ i < length (antecedent r) ⟹ i’ < length (antecedent r)
\[ a\text{-fresh } (\text{antecedent } r \ ! i) \cap a\text{-fresh } (\text{antecedent } r \ ! i') = \{ \} \lor i = i' \]

assumes all-local-cons-listed:
\[
\forall p. r \in \text{sset rules} \implies p \in \text{set } (\text{antecedent } r) \implies \text{leons } (a\text{-conc } p) \cup \bigcup (\text{leons } (\text{fset } (a\text{-hyps } p))) \subseteq a\text{-fresh } p
\]

begin
definition f-antecedent :: 'rule \implies ('form, 'var) antecedent fset
where f-antecedent r = fset-from-list (antecedent r)
definition f-consequent r = fset-from-list (consequent r)
end

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

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

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

fixes assumptions :: 'form list
fixes conclusions :: 'form list
assumes assumptions-closed: \\( \forall a. a \in \text{set assumptions} \implies \text{closed } a \)
assumes conclusions-closed: \\( \forall c. c \in \text{set conclusions} \implies \text{closed } c \)
begin
definition ass-forms where ass-forms = fset-from-list assumptions
definition conc-forms where conc-forms = fset-from-list conclusions

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

lemma mem-conc-forms[simp]: a \in\in 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-leons 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-leons freshen-def rename-closed subst-closed)

lemma subst-freshen-in-ass-forms1:
  assumes pf \in \text{set assumptions}
shows \( \text{subst}\ s\ (\text{freshen}\ a\ pf) \in ass\text{-forms} \)
using assms by simp

\textbf{lemma} \( \text{subst-freshen-in-conc-formsI} : \)
\textbf{assumes} \( pf \in \text{set conclusions} \)
\textbf{shows} \( \text{subst}\ s\ (\text{freshen}\ a\ pf) \in \text{conc-forms} \)
using assms by simp

end

end
4 Incredible Proof Graphs

4.1 Incredible_Signatures

theory Incredible-Signatures

imports
  Main
  HOL-Library.FSet
  HOL-Library.Stream
  Abstract-Formula

begin

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

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

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

locale Port-Graph-Signature-Scoped =
  Port-Graph-Signature +
  fixes hyps :: 'node ⇒ 'outPort ⇒ 'inPort
  assumes hyps-correct: hyps n p1 = Some p2 ⇒ p1 ∈| outPorts n ∧ p2 ∈| inPorts n

begin
  inductive-set hyps-for' :: 'node ⇒ 'inPort ⇒ 'outPort set for n p
    where hyps n h = Some p ⇒ h ∈ hyps-for' n p

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

context includes fset.lifting
end

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

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 Port-Graph-Signature Scoped-Vars = 
Port-Graph-Signature nodes inPorts outPorts +
Abstract Formulas freshenLC renameLCs leonst seq subst subst-leonst 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 leonst :: 'form => 'var set
and closed :: 'form => bool
and subst :: 'subst => 'form => 'form
and subst-leonst :: 'subst => 'var set
and subst-renameLCs :: ('var => 'var) => ('subst => 'subst)
and anyP :: 'form +

fixes local-vars :: 'node => 'inPort => 'var set

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 ps
and outPorts :: 'node ⇒ 'outPort ps
and vertices :: 'v ps
and nodeOf :: 'v ⇒ 'node +
fixes edges :: ('v, 'outPort, 'inPort) edge set

begin

fun edge-begin :: ((v × 'outPort) × (v × 'inPort)) ⇒ 'v where
edge-begin ((v1, p1), (v2, p2)) = v1

fun edge-end :: ((v × 'outPort) × (v × 'inPort)) ⇒ 'v where
edge-end ((v1, p1), (v2, p2)) = v2

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

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

inductive-simps path-cons-simp': path v v' (e#pth)
inductive-simps path-empty-simp[|simp|]: path v v []

lemma path-cons-simp: path v v' (e ≠ pth) ←→ fst (fst e) = v ∧ e ∈ edges ∧ path (fst (snd e)) v' pth
by (auto simp add: path-cons-simp', metis prod.collapse)

lemma path-append1: 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-end-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-end-tup intro: path-empty)

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

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

lemma snd-set-split:
assumes ps ∈ snd ' set pth
obtains pth1 pth2 e where pth = pth1@[e]@[pth2] and snd e = ps and ps ∉ snd ' set pth1
using assms
proof (atomize-elim, induction pth)
  case Nil thus ?case by simp
next
  case (Cons e pth)
show ?case
proof (cases snd e = ps)
  case True
  hence e # pth = [] @ [e] @ pth ∧ snd e = ps ∧ ps ≠ snd ' set [] by auto
  thus ?thesis by (intro exI)
next
  case False
  with Cons(2)
  have ps ∈ snd ' set pth by auto
  from Cons.3[OF this this]
  obtain pth1 e' pth2 where pth = pth1 @ [e'] @ pth2 ∧ snd e' = ps ∧ ps ≠ snd ' set pth1 by auto
  with False
  have e#pth = (e#pth1) @ [e'] @ pth2 ∧ snd e' = ps ∧ ps ≠ snd ' set (e#pth1) by auto
  thus ?thesis by blast
qed

qed

lemma scope-split:
  assumes v ∈ scope ps
  assumes path v v' pth
  assumes terminal-vertex v'
  obtains pth1 e pth2
    where pth = (pth1 @ [e]) @ pth2 and path v (fst ps) (pth1 @ [e]) and path (fst ps) v' pth2 and snd e = ps
    and ps ≠ snd ' set pth1
  proof
    from assms
    have ps ∈ snd ' set pth by (auto simp add: scope.simps)
    then obtain pth1 pth2 e
      where pth = pth1 @ [e] @ pth2 and path (fst ps) v' pth2 and snd e = ps
      and ps ≠ snd ' set pth1
    using ⟨path - - -⟩ ⟨pth = pth1 @ [e] @ pth2⟩ ⟨path v (fst ps) (pth1 @ [e])⟩ ⟨path (fst ps) v' pth2⟩ ⟨snd e = ps⟩ by (rule snd-set-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 (metis path-split)+
      show thesis
      qed
      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 notin-fset 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)
apply auto
apply (metis valid-out-port.elims(2) edge-begin.simps notin-fset case-prodD valid-edges)
done

end

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

locale Pruned-Port-Graph = Port-Graph +
  assumes pruned: \( \forall v. \ v \in \{ \text{vertices} \} \rightarrow (\exists \ pth \ v', \ \text{path} \ v \ v' \ \text{pth} \land \ \text{terminal-vertex} \ v') \)
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 \land least: \forall \ pth', \ \text{path} \ v \ t \ pth' \rightarrow \ \text{length \ pth} \leq \ \text{length \ pth'}
  
  by atomizeelim (auto simp del: terminal-vertex.simps elim: ex-has-least-nat)

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

from this(2) least
have length pth \leq length pth2 by auto
with (pth = -)
show False by auto
qed

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

lemma scopes-nest:
  fixes ps1 ps2
  shows scope ps1 \subseteq \text{scope} ps2 \lor \text{scope} ps2 \subseteq \text{scope} ps1 \lor \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} \} \text{using scope.simps by auto}
  then obtain pth t where \text{path} \ v \ t \ \text{pth} \land \text{terminal-vertex} \ t \text{using pruned by blast}
  from (\text{path} \ v \ t \ \text{pth}) \land \text{terminal-vertex} \ t \land v \in \text{scope} ps1 \cap \text{scope} ps2;
  obtain pth1a e1 pth1b where pth = (pth1a @ [e1]) @ pth1b \text{and} \text{path} v (\text{fst} ps1) (pth1a @ [e1]) \text{and} \text{snd} e1 = ps1 \text{and} ps1 \notin \text{snd} \set \text{set pth1a}
  by (auto elim: scope-split)

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

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

have \text{scope ps1} \subseteq \text{scope ps2}
proof
  fix v'
  assume v' \in \text{scope ps1}
  hence v' \in | \text{vertices} \text{ using \text{scope.simps} by auto}
  thus v' \in \text{scope ps2}
proof [rule \text{scope.intros}]
  fix pth' t'
  assume path v' t' pth' and terminal-vertex t'
  with (v' \in \text{scope ps1})
  obtain pth3a e3 pth3b where pth' = (pth3a@[e3])@pth3b and path (fst ps1) t' pth3b
    by (auto elim: \text{scope-split})
  have path v' t' ((pth1a@[e1]) @ pth3b) \text{ using \langle path v (fst ps1) (pth1a@[e1]) \rangle \text{ and \langle path (fst ps1) t' pth3b \rangle}}
    by (rule \text{path-append1})
  with \text{\langle terminal-vertex t' \rangle (v \in \cdot)}
  have ps2 \notin \text{snd \set (pth1a@[e1]) @ pth3b} \text{ by (meson IntD2 \text{scope.cases})}
  hence ps2 \notin \text{snd \set \text{set pth3b using \langle ps2 \notin \text{snd \set (pth1a@[e1])} \rangle \text{ by auto}}}
  thus ps2 \in \text{set pth' using \langle pth' \cdot \rangle \text{ by auto}}
  qed
qed
thus \text{?thesis by simp}
next
assume set pth2a \subseteq set pth1a with \text{\langle ps1 \notin \cdot \rangle}
have ps1 \notin \text{snd \set (pth2a@[e2])} \text{ using \langle ps1 \neq ps2 \rangle (\text{snd e2 = ps2}) by auto}

have \text{scope ps2} \subseteq \text{scope ps1}
proof
  fix v'
  assume v' \in \text{scope ps2}
  hence v' \in | \text{vertices} \text{ using \text{scope.simps} by auto}
  thus v' \in \text{scope ps1}
proof [rule \text{scope.intros}]
  fix pth' t'
  assume path v' t' pth' and terminal-vertex t'
  with (v' \in \text{scope ps2})
  obtain pth3a e3 pth3b where pth' = (pth3a@[e3])@pth3b and path (fst ps2) t' pth3b
    by (auto elim: \text{scope-split})
  have path v' t' ((pth2a@[e2]) @ pth3b) \text{ using \langle path v (fst ps2) (pth2a@[e2]) \rangle \text{ and \langle path (fst ps2) t' pth3b \rangle}}
    by (rule \text{path-append1})
  with \text{\langle terminal-vertex t' \rangle (v \in \cdot)}
  have ps1 \in \text{snd \set (pth2a@[e2]) @ pth3b} \text{ by (meson IntD1 \text{scope.cases})}
  hence ps1 \in \text{snd \set \text{set pth3b using \langle ps1 \notin \text{snd \set (pth2a@[e2])} \rangle \text{ by auto}}}
  thus ps1 \in \text{set pth' using \langle pth' \cdot \rangle \text{ by auto}}
  qed
qed
thus \text{?thesis by simp}
qed
)
thus \text{?thesis by blast}
qed simp
end
A well-scoped graph is one where a port marked to be a local hypothesis is only connected to the corresponding input port, either directly or via a path. It must not be, however, that there is a path from such a hypothesis to a terminal node that does not pass by the dedicated input port; this is expressed via scopes.

\[
\text{Scoped-Graph} = \text{Port-Graph} + \text{Port-Graph-Signature-Scoped}
\]

Assumes well-scoped: \((v_1,p_1),(v_2,p_2)\) \(\in\) \(\text{edges} \Rightarrow \text{hyps} (\text{nodeOf} v_1) p_1 = \text{Some} p' \Rightarrow (v_2,p_2) = (v_1,p') \lor v_2 \in \text{scope} (v_1,p')\)

Context Scoped-Graph

Definition hyps-free where
\[
\text{hyps-free pth} = (\forall v_1, v_2. ((v_1,p_1),(v_2,p_2)) \in \text{set pth} \Rightarrow \text{hyps} (\text{nodeOf} v_1) p_1 = \text{None})
\]

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

Lemma hyps-free-Cons simp: hyps-free (e#pth) \(\iff\) hyps-free pth \& hyps (nodeOf (fst (fst e))) (snd (fst e)) = None by (auto simp add: hyps-free-def) (metis prod.collapse)

Lemma path-vertices-shift:
\[
\text{assumes path v v' pth}
\]
\[
\text{shows map fst (map fst pth)@[v'] = v#map fst (map snd pth)}
\]
Using assms by induction auto

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

Lemma terminal-path-is-path:
\[
\text{assumes terminal-path v v' pth}
\]
\[
\text{shows path v v' pth}
\]
Using assms by induction (auto simp add: path-cons-simp)

Lemma terminal-path-is-hyps-free:
\[
\text{assumes terminal-path v v' pth}
\]
\[
\text{shows hyps-free pth}
\]
Using assms by induction (auto simp add: hyps-free-def)

Lemma terminal-path-end-is-terminal:
\[
\text{assumes terminal-path v v' pth}
\]
\[
\text{shows terminal-vertex v'}
\]
Using assms by induction

Lemma terminal-path1:
\[
\text{assumes path v v' pth}
\]
\[
\text{assumes hyps-free pth}
\]
\[
\text{assumes terminal-vertex v'}
\]
\[
\text{shows terminal-path v v' pth}
\]
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
  shows ?case by simp

next

  case (terminal-path-cons v1 p1 v2 p2 v' pth)

  note terminal-path-cons.
  IH

  moreover have v1 ∉ fst · fst · set pth

  proof

    assume v1 ∈ fst · fst · set pth

    then obtain pth1 e' pth2 where pth = pth1 @[e'] @ pth2 and v1 = fst (fst e')

    apply (atomize-elim)
    apply (induction pth)
    apply (solves simp)
    apply (auto)
    apply (solves (rule ext [where x = []]; simp))
    apply (metis Cons-eq-app endI image-eqI prod.sel(1))

    done

  with terminal-path-is-path[OF terminal-path v2 v' pth]

  have path v2 v1 pth1 by (simp add: path-split2 edge-begin-edge)

  with (((v1, p1), (v2, p2)) ∈ -)

  have path v1 v1 (((v1, p1), (v2, p2)) ≠ pth1) by (simp add: path-cons-simp)

  moreover

  from terminal-path-is-hyps-free[OF terminal-path v2 v' pth]

    (hyps (nodeOf v1) p1 = None)

    (pth = pth1@[e']@pth2)

  have hyps-free(((v1, p1), (v2, p2)) ≠ pth1)

    by (auto simp add: hyps-free-def)

    ultimately

    show False using hyps-free-acyclic by blast

  qed

  moreover

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

lemma hyps-free-path-not-in-scope:
assumes \( \text{terminal-path } \text{v } \text{t pth} \)
assumes \( \text{v',p') } \in \text{snd } \text{'} \text{ set pth} \)
shows \( \text{v' } \notin \text{ scope } \text{(v, p)} \)
proof
assume \( \text{v' } \in \text{ scope } \text{(v,p)} \)
from \( \text{(v',p'} ) \in \text{snd } \text{'} \text{ set pth} \)
obtain \( \text{pth1 pth2 e} \) where \( \text{pth } = \text{pth1 } \@\text{e} \@\text{pth2 and } \text{snd } \text{e } = \text{(v',p')} \)
by (rule snd-set-split)
from terminal-path-is-path[OF assms(1), unfolded \( \text{pth } = \text{- } \text{e} \)] \( \text{snd } \text{e } = \text{-} \)
have \( \text{path } \text{v' (pth1 } \@\text{e}) \) and \( \text{path } \text{v' t pth2 unfolding path-split by} \) (auto simp add: edge-end-tup)
from \( \text{v' } \in \text{ scope } \text{(v,p)} \): \( \text{terminal-path-end-is-terminal[OF assms(1), unfolding pth = - ]} \)
have \( \text{v' } \in \text{ scope } \text{(v, p)} \)
by (rule scope-find)
then obtain \( \text{pth2a e' pth2b} \) where \( \text{pth2 } = \text{pth2a } \@\text{e }' \@\text{pth2b and } \text{snd } \text{e' } = \text{(v, p)} \)
by (rule snd-set-split)
from \( \text{path } \text{v' t pth2: [unfolded pth2 = -]} \)
\( \text{snd } \text{e' } = \text{-} \)
have \( \text{path v' t pth2 unfolding path-split by } \) (auto simp add: edge-end-tup)
from \( \text{v' } \in \text{ scope } \text{(v,p)} \): \( \text{terminal-path-is-hyps-free[OF assms(1), unfolding pth = - \text{pth2} = -] \)
have \( \text{hyps-free } \) \( \text{(pth1 } \@\text{e}) \@\text{(pth2a } \@\text{e }') \)
by (auto simp add: hyps-free-def)
ultimately
have \( \text{((pth1 } \@\text{e}) \@\text{(pth2a } \@\text{e }')) \) \( \emptyset \)
by (rule hyps-free-acyclic)
thus \( \text{False} \) 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 \( \text{(v, p)} \) \( \Rightarrow \) \( \exists \text{ e } \in \text{edges } \) \( \text{snd } \text{e } = \text{(v, p)} \)

These four conditions make up a well-shaped graph.

locale Well-Shaped-Graph = Well-Scope-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 :: 'subt => 'form => 'form
and subst-lcOnsts :: 'subt => 'var set
and subst-renameLCs :: ('var => 'var) => ('subt => 'subt)
and anyP :: 'form +

fixes vidx :: 'vertex => nat
and inst :: 'vertex => 'subt

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 x 'outPort) x 'vertex x '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 :: 'subt => 'form => 'form
and subst-lcOnsts :: 'subt => 'var set
and subst-renameLCs :: ('var => 'var) => ('subt => 'subt)
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 labelsOut :: 'node ⇒ 'outPort ⇒ 'form
and vidx :: 'vertex ⇒ nat
and inst :: 'vertex ⇒ 'subst
and edges :: ('vertex, 'outPort, 'inPort) edge set
and local-vars :: 'node ⇒ 'inPort ⇒ 'var set +

assumes well-scoped-inst:
valid-in-port (v,p) ⇒
var ∈ local-vars (nodeOf v) p ⇒
v' ∈ vertices ⇒
freshenLC (vidx v) var ∈ subst-consts (inst v') ⇒
v' ∈ scope (v,p)

begin

lemma out-of-scope: valid-in-port (v,p) ⇒ v' ∈ vertices ⇒ v' ∉ scope (v,p) ⇒ freshenLC (vidx v) ′
local-vars (nodeOf v) p ∩ subst-consts (inst v') = {}

using well-scoped-inst by auto

end

The following locale assembles all these conditions.

locale Scoped-Proof-Graph =

Instantiation inPorts outPorts nodeOf hyps nodes edges vertices labelsIn labelsOut freshenLC renameLCs lcconsts closed subst subst-consts 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 lcconsts closed subst subst-consts subst-renameLCs anyP vidx inst edges +
Well-S Scoped-Instantiation freshenLC renameLCs lcconsts closed subst subst-consts 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 lcconsts :: '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
and inPorts :: 'node ⇒ 'inPort fset
and outPorts :: 'node ⇒ 'outPort fset
and nodeOf :: 'vertex ⇒ 'node
and hyps :: 'node ⇒ 'outPort ⇒ 'inPort option
and nodes :: 'node stream
and vertices :: 'vertex fset
and labelsIn :: 'node ⇒ 'inPort ⇒ 'form
and labelsOut :: 'node ⇒ 'outPort ⇒ 'form
and vidx :: 'vertex ⇒ nat
and inst :: 'vertex ⇒ 'subst
and edges :: ('vertex, 'outPort, 'inPort) edge set
and local-vars :: 'node ⇒ 'inPort ⇒ 'var set

end

4.3 Abstract_Rules_To_Incredible

theory Abstract_Rules-To-Incredible
imports
Main
HOL-Library.FSet
HOL-Library.Stream
Incredible-Deduction

Abstract-Rules

begin

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

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

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

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

type-synonym form reg-out-port = form

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

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

custom Abstratc-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 ∈ set 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: fmember.rep-eq f-antecedent-def)

definition outPortsRule where

outPortsRule r = fUnion ((λ a. (λ h. Hyp h a) 'a'-a-hyps a) 'a'-f-antecedent r) |∪| Reg |'a'-f-consequent r

lemma Reg-in-outPortsRule[simp]:

Reg c ∈| outPortsRule r ←→ c ∈| f-consequent r
by (auto simp add: outPortsRule-deq fmember.rep-eq fUnion.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-deq fmember.rep-eq fUnion.rep-eq)

fun outPorts where

outPorts (Rule r) = outPortsRule r
|outPorts (Assumption r) = \{ R e g r \}|
|outPorts (Conclusion r) = \{ || \}|
|outPorts Helper = \{ | Reg anyP | \}|

fun labelsIn where
  labelsIn - p = a-conc p

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

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

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

sublocale Labeled-Signature nodes inPorts outPorts hyps labelsIn labelsOut
proof (standard, goal-cases)
  case (I 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 = (λ h. Hyp h ip) \in| 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 leons closed subst subst-leons subst-renameLCs anyP antecedent consequent rules assumptions conclusions +
Scoped-Proof-Graph freshenLC renameLCs leons closed subst subst-leons subst-renameLCs anyP inPorts outPorts nodeOf hyps nodes vertices labelsIn labelsOut vids inst edges local-vars
for freshenLC :: nat \Rightarrow \'var \Rightarrow \'var
  and renameLCs :: (\'var \Rightarrow \'var) \Rightarrow \'form \Rightarrow \'form
  and leons :: \'form \Rightarrow \'var set
  and closed :: \'form \Rightarrow \text{bool}
  and subst :: \'sub \Rightarrow \'form \Rightarrow \'form
  and subst-leons :: \'sub \Rightarrow \'var set
  and subst-renameLCs :: (\'var \Rightarrow \'var) \Rightarrow (\'sub \Rightarrow \'sub)
  and anyP :: \'form

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

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

  and vertices :: \'vertex fset
and nodeOf :: 'vertex ⇒ ('form, 'rule) graph-node
and edges :: ('vertex, 'form, 'var) edge set
and vidx :: 'vertex ⇒ nat
and inst :: 'vertex ⇒ subst +
assumes conclusions-present: set (map Conclusion conclusions) ⊆ nodeOf \set 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 step in the derivation can either be the axiom rule, the cut rule, or one of the given rules in Incredible-Proof-Machine.Abstract-Rules.

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

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

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

fixes nat-rule :: 'rule ⇒ 'form ⇒ ('form, 'var) antecedent fset ⇒ bool
 and rules :: 'rule stream
begin

- An application of the Axiom rule is valid if the conclusion is among the assumptions.

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

- The Cut rule is again easy.

inductive eff :: 'rule NatRule ⇒ 'form entailment ⇒ 'form entailment fset ⇒ bool where
\[
\text{con } |\in| \Gamma \\
\Rightarrow \text{eff Axiom } (\Gamma \vdash \text{con}) \{||\}
\]

|nat-rule rule c ants
\[
\Rightarrow (\wedge \text{ant f. ant } |\in| \text{ants } \Rightarrow \text{f } |\in| \Gamma \Rightarrow \text{freshenLC a } \cdot (\text{a-fresh ant}) \cap \text{lecons f } = \{\})
\]

\[
\Rightarrow (\wedge \text{ant. ant } |\in| \text{ants } \Rightarrow \text{freshenLC a } \cdot (\text{a-fresh ant}) \cap \text{subst-lecons s } = \{\})
\]

\[
\Rightarrow \text{eff (NatRule rule)}
\]

\[
(\Gamma \vdash \text{subst s (freshen a c)})
\]

\[
((\lambda \text{ant. } ((\lambda p. \text{subst s (freshen a p)}) |\cdot| \text{a-hyps ant } |\cup| \Gamma \vdash \text{subst s (freshen a (a-conc ant)))))) |\cdot| \text{ants}
\]

\[
\Rightarrow \text{eff Cut } (\Gamma \vdash c') \{ | (\Gamma \vdash c')\}
\]

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

\[
\text{sublocale RuleSystem-Defs where}
\]

\[
\text{eff } = \text{eff and rules } = \text{Cut }##\text{ Axiom }## \text{smap NatRule rules.}
\]

end

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

context Abstract-Task
begin

\[
\text{inductive natEff-Inst where}
\]

\[
\text{c } \in \text{set (consequent r) } \Rightarrow \text{natEff-Inst (r,c) (f-antecedent r)}
\]

\[
\text{definition n-rules where}
\]

\[
\text{n-rules } = \text{flat } (\text{smap } (\lambda r. \text{map } (\lambda c. (r,c)) \text{ (consequent r))) rules)}
\]

\[
\text{sublocale ND-Rules-Inst - - - - - - natEff-Inst n-rules ..}
\]

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.

\[
\text{definition solved where}
\]

\[
\text{solved } \leftrightarrow (\forall c. c ) |\in| \text{conce-forms } \rightarrow (\exists t. \text{fst (root t) } = (\Gamma \vdash c) \wedge \Gamma |\subseteq| \text{ass-forms } \wedge \text{wf t } \wedge \text{finite t})
\]

end

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 fnmember_rep_eq 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.transferr]

  lemmas global-assms-simps = global-assms'.simpls[Transfer.transferr]

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 = fUnion (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 ∈ verticex and nodeOf v = Assumption pf f = labelAtOut v (Reg pf)
using assms
apply (auto simp add: fmember_rep-eq fUnion_rep-eq global-assms-simps[unfolded fmember_rep-eq])
apply (metis image_iff snd_conv)
done

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

princorec 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
| v h c where n = Rule v and p = Hyp h c
| r f where n = Rule v and p = Reg f
| n = Helper and p = Reg anyP
using assms by (atomize_elim, cases p; cases n) auto

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

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

theorem wf-tree:
assumes valid-in-port (v,p)
assumes terminal-path v t pth
shows wf (tree v p pth)
using assms
proof (coinduction arbitrary: v p pth)
case $$(\text{wf} \, v \, p \, \text{pth})$$

let $?t = \text{true} \, v \, p \, \text{pth}$

obtain $v' \, p'$

where $e:((v', p'),(v, p)) \in \text{edges}$ and \[\text{simp}: \text{adjacentTo} \, v \, p = (v', p')\]

by (auto simp add: adjacentTo-def, metis (no-types, lifting) eq-fst-iff tfl-some)

let $?c = ((v', p'),(v, p))

let $?pth' = ?c#pth

let $?\Gamma = \text{hyps-along} \, ?pth'$

let $?l = \text{labelAtIn} \, v \, p$

from $c$ valid-edges have $v' |\in| \text{vertices}$ and $p' |\in| \text{outPorts} \, (\text{nodeOf} \, v')$ by auto

hence $\text{nodeOf} \, v' |\in| \text{valid-nodes}$ using valid-nodes by (meson image-eqI notin-fset set-mp)

from $?e |\in| \text{edges}$

have $s: \text{labelAtOut} \, v' \, p' = \text{labelAtIn} \, v \, p$ by (rule solved)

from $p' |\in| \text{outPorts} \, (\text{nodeOf} \, v')$

show $?\text{case}$

proof (cases rule: out-port-cases)

next case (Hyp r h c)

from Hyp $p' |\in| \text{outPorts} \, (\text{nodeOf} \, v')$

have $h |\in| \text{a-hyps c}$ and $c |\in| \text{f-antecedent r}$ by auto

hence Hyp $$(\text{nodeOf} \, v') \, (\text{Hyp h c}) = \text{Some} \, c$$ using Hyp by simp

from well-scoped[OF $(- \in \text{edges})$[unfolded Hyp] this]

have $(v, p) = (v', c) \lor v \in \text{scope} \, (v', c)$,

hence $(v', c) \in \text{insert} \, (v, p)$ (snd set pth)

proof

assume $(v, p) = (v', c)$

thus $?\text{thesis}$ by simp

next assume $v \in \text{scope} \, (v', c)$

from this terminal-path-end-is-terminal[OF wf (2)] terminal-path-is-path[OF wf (2)]

have $(v', c) \in \text{snd set pth}$ by (rule scope-find)

thus $?\text{thesis}$ by simp

qed

moreover

from $\text{hyps} \, (\text{nodeOf} \, v') \, (\text{Hyp h c}) = \text{Some} \, c$

have Hyp $h \, c |\in| \text{hyps-for} \, (\text{nodeOf} \, v') \, c$ by simp

hence $\text{labelAtOut} \, v' \, (\text{Hyp h c}) |\in| \text{extra-assms} \, (v', c)$ by auto

ultimately

have $\text{labelAtOut} \, v' \, (\text{Hyp h c}) |\in| \, ?\Gamma$

by (fastforce simp add: fmember.rep-eq fUnion.rep-eq)

hence $\text{labelAtIn} \, v \, p |\in| \, ?\Gamma$ by (simp add: s[symmetric] Hyp fmember.rep-eq)

thus $?\text{thesis}$

using Hyp

apply (auto intro: exI[where $x = ?t$] simp add: eff.simps simp del: hyps-along.simps)

done

next case (Assumption $f$)
from \( v' \subseteq \text{vertices} \) (nodeOf \( v' = \text{Assumption} f \))

have \( \text{labelAtOut} v' (\text{Reg} f) \subseteq \text{global-assms} \ \text{TYPE}'\text{var} \)
  by (rule \text{global-assmsI})

hence \( \text{labelAtOut} v' (\text{Reg} f) \subseteq \Omega f \) by auto

hence \( \text{labelAtIn} \ p \subseteq \ ?\Gamma \) by (simp add: \text{symmetric} \ \text{Assumption fmember.rep-eq})

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

next

case \( \text{Rule} \ r \ f \)
  with \( \text{nodeOf} \ v' \in \text{sset nodes} \)
  have \( r \in \text{sset rules} \)
    by (auto simp add: \text{nodes-def} \ \text{stream.set-map})

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

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

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

have \( \text{NatRule} \ (r \in \text{sset}) \) (smap NatRule n-rules)
  by (auto simp add: \text{stream.set-map} \ \text{n-rules-def} \ \text{no-empty-conclusions})

moreover

\{
  from \( \{f \in \text{f-consequent} \ r\}

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

  hence \( \text{natEff-Inst} \ (r \ f) \ f \) (\text{f-antecedent} \( r \))
    by (rule \text{natEff-Inst.intros})

  hence \( \text{eff} \) (NatRule \( \ (r \ f) \)) \( \{?\Gamma \vdash \text{inst} v' \} \)
    by (simp add: \text{f-consequent-def})

  hence \( \text{f-antecedent} \ r \)

  (is \ \text{eff} - - ?\text{ants})

proof (rule \text{eff.intros})

  fix \( \text{ant} f \)

  assume \( \text{ant} \subseteq \text{f-antecedent} \ r \)

  from \( \{v' \subseteq \text{vertices} \} \) \( \text{ant} \subseteq \text{f-antecedent} \ r \)

  have \( \text{valid-in-port} \ (v' \text{.ant}) \) by (simp add: \text{Rule})

assume \( f \subseteq \ ?\Gamma \)

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

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

  case \( \text{Hyp} v'' p'' h'' \)

  from \( \text{Hyp}(1) \) \( \text{snd-set-path-vertices}[\text{OF} \text{terminal-path-is-path}[\text{OF} \text{terminal-path} v' t \ ?\text{pth}')] \)

  have \( v'' \subseteq \text{vertices} \) by (force simp add: \text{fmember.rep-eq})

  from \( \text{terminal-path} v' t \ ?\text{pth} \) \( \text{Hyp}(1) \)

  have \( v'' \notin \ \text{scope} \ (v', \text{ant}) \) by (rule \text{hypss-free-path-not-in-scope})

  with \( \text{valid-in-port} \ (v', \text{ant}) \) \( v'' \subseteq \text{vertices} \)

  have \( \text{freshenLC} \ (\text{vidx} v') \ ' \ \text{local-vars} \) (nodeOf \( v' \)) \( \text{ant} \cap \text{subst-leconsts} \) (inst \( v'' \)) = \{}
    by (rule \text{out-of-scope})

  moreover

from \( \text{hypss-free-vertices-distinct}[\text{OF} \text{terminal-path} v' t \ ?\text{pth}'] \) \( \text{Hyp}.\text{hypss}(1) \)

32
have \( v'' \neq v' \) by (metis distinct.simps(2) list.conv image-eq list.set-map)
hence \( v'' \neq v' \) using \((v' \in \mathit{vertices}) (v'' \in \mathit{vertices})\) by (meson vidx-inj inj-onD notin-fset)
hence \( \mathit{freshenLC} (\mathit{vidx} v') : \mathit{a-fresh} \text{ ant} \cap \mathit{freshenLC} (\mathit{vidx} v'') : \mathit{lecons} \) (labelsOut (nodeOf v'')) \( h'' = \) 
\( \{} \) by auto

moreover
have \( \mathit{lecons} f \subseteq \mathit{lecons} \) (freshen (vidx v'') (labelsOut (nodeOf v'') \( h'' \))) \( \cup \) \( \mathit{subst-\mathit{lecons}} (\mathit{inst v''}) \) using 
\( f = - \)
by (simp add: labelAtOut-def fo-subst)
ultimately
show ?thesis
by (fastforce simp add: \( \mathit{lecons} \)-freshen)

next

case (Assumption \( v\mathit{p}f \))
hence \( f = \mathit{subst} (\mathit{inst v'}) \) (freshen (vidx v) \( pf \)) by (simp add: labelAtOut-def)
moreover
from \( \mathit{Assumption} \) have \( \mathit{Assumption} \ \mathit{pf} \in \mathit{set} \ \mathit{nodes} \) using \( \mathit{valid-nodes} \) by (auto simp add: \( \mathit{fmember}.\mathit{rep-eq} \))
hence \( pf \in \mathit{set} \ \mathit{assumptions} \) unfolding nodes-def by (auto simp add: stream.set-map)
hence \( \mathit{closed pf} \) by (rule assumptions-closed)
ultimately
have \( \mathit{lecons} f = \{\} \) by (simp add: closed-no-lecons lecons-freshen subst-closed freshen-closed)
thus ?thesis by simp
qed

next

fix \( \mathit{ant} \)
assume \( \mathit{ant} \ |\ | f\-\mathit{antecedent} \ \mathit{r} \)
from \( (v' |\in \mathit{vertices}) (\mathit{ant} |\in f\-\mathit{antecedent} \ \mathit{r}) \)
have \( \mathit{valid-in-port} (v'\mathit{ant}) \) by (simp add: Rule)
moreover
note \( (v' |\in \mathit{vertices}) \)
moreover
hence \( v' \notin \mathit{scope} (v', \mathit{ant}) \) by (rule scopes-not-refl)
ultimately
have \( \mathit{freshenLC} (\mathit{vidx} v') : \mathit{local-vars} \ (\mathit{nodeOf v'}) \ \mathit{ant} \cap \mathit{subst-\mathit{lecons}} (\mathit{inst v'}) = \{\} \)
by (rule out-of-scope)
thus \( \mathit{freshenLC} (\mathit{vidx} v') : \mathit{a-fresh} \ \mathit{ant} \cap \mathit{subst-\mathit{lecons}} (\mathit{inst v'}) = \{\} \) by simp
qed
also
have \( \mathit{valid-nodes} (\mathit{inst v'}) (\mathit{freshen} (\mathit{vidx} v') f) = \mathit{labelAtOut} v' p' \) using Rule by (simp add: labelAtOut-def)
also
note \( (\mathit{labelAtOut} v' p' = \mathit{labelAtIn} v p) \)
also
have \( \mathit{ant} = (\lambda x . (\mathit{extr-assms} (v', x) |\cup| \mathit{hyps-along} ?p'th' \vdash \mathit{labelAtIn} v' x)) |\vdash f\-\mathit{antecedent} \ \mathit{r}) \)
by (rule \( \mathit{fmember}.\mathit{rep-eq} \))
(auto simp add: \( \mathit{labelAtIn-def} \mathit{labelAtOut-def} \mathit{Rule} \ \mathit{hyps-for-fimage} \ \mathit{fmember}.\mathit{rep-eq} \ \mathit{fUnion}.\mathit{rep-eq})
finally
have \( \mathit{eff} \) (NatRule \( (r, f) \))
\( (\mathit{\Gamma}, \mathit{\mathit{labelAtIn}} v p) \)
\( ((\lambda x . \mathit{extr-assms} (v', x) |\cup| \mathit{?\Gamma} \vdash \mathit{labelAtIn} v' x)) |\vdash f\-\mathit{antecedent} \ \mathit{r}) \).
}

moreover

\{ fix \( x \) \in \mathit{cont} \ ?t
then obtain \( \mathit{a} \) where \( x = \mathit{tree} v' \ ?p'th' \ \mathit{and} \ \mathit{a} |\in f\-\mathit{antecedent} \ \mathit{r} \)
by (auto simp add: Rule)
note this(1)
\}
moreover

from \( (v' \mid\in\) vertices) \& (a \mid\in\) f-antecedent \( r \) 
have valid-in-port \((v',a)\) by (simp add: Rule)
moreover

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

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

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

next

case Helper
from Helper
have hyps \( \langle\text{nodeOf } v' \rangle \ p' = \text{None} \) by simp
with e \( \langle\text{terminal-path } v \ t \ ?\text{pth} \rangle \)
have \( \langle\text{terminal-path } v' \ t \ ?\text{pth}' \rangle \)
have \( \langle\text{labelAtIn } v' \ (\text{plain-ant anyP}) \rangle = \text{labelAtIn } v \ p \)
unfolding s[symmetric]
using Helper by (simp add: labelAtIn-def labelAtOut-def)
moreover
\{ fix \( x \)
assume \( x \mid\in\) cont \( ?t \)

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

note this(1)
moreover

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

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

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

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

qed

qed

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

from this (1,2) valid-nodes

have Assumption pf ∈ set nodes by (auto simp add: fmember-rep-eq)

hence pf ∈ set assumptions by (auto simp add: nodes-def stream-set-map)

hence closed pf by (rule assumptions-closed)

with x = labelAtOut v (Reg pf)

have x = pf by (auto simp add: labelAtOut-def leonsts-freshen closed-no-consts freshen-closed subst-closed)

thus x ∈ | ass-forms using ⟨pf ∈ set assumptions⟩ by (auto simp add: ass-forms-def)

qed

primcorec edge-tree :: 'vertex ⇒ ('form, 'var) in-port ⇒ ('vertex, 'form, 'var) edge' tree where

root (edge-tree v p) = (adjacentTo v p, (v,p))

| cont (edge-tree v p) =

(case adjacentTo v p of (v', p') ⇒
(if isReg v' p' then ((λ p. edge-tree v' p) |? inPorts (nodeOf v')) else {||}))

lemma tfinite-map-tree: tfinite (map-tree f t) ←→ tfinite t

proof

assume tfinite (map-tree f t)

thus tfinite t

by (induction map-tree f t arbitrary; t rule: tfinite.induct)

(auto simp add: tree.map-sel simp add: tree.map-set)

next

assume tfinite t

thus tfinite (map-tree f t)

by (induction t rule: tfinite.induct)

(auto simp add: tree.map-sel simp add: tree.map-set)

qed

lemma finite-tree-edge-tree:

finite (tree v p pth) ←→ finite (edge-tree v p)

proof

have map-tree (λ z. ()) (tree v p pth) = map-tree (λ z. ()) (edge-tree v p)

by (coinduction arbitrary: v p pth)

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

thus thesis by (metis tfinite-map-tree)

qed

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

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

forbidden-path v2 (((v1,p1),(v2,p2))###pth)

lemma path-is-forbidden:

assumes valid-in-port (v,p)

assumes ipath (edge-tree v p) es

shows forbidden-path v es

using assms

proof (coinduction arbitrary: v p es)

case forbidden-path

let ?es' = stl es

from forbidden-path(2)

obtain t' where root (edge-tree v p) = shd es and t' ∈ | cont (edge-tree v p) and ipath t' ?es'

35
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 tl-some)
let ?e = ((v',p'),(v,p))

from e have p' ∈| outPorts (nodeOf v') using valid-edges by auto
thus ?case

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

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

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

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

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

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

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

  have ?e ∈ edges using e by simp
  moreover

36
from \( p' = \text{Reg anyP} \langle \text{nodeOf } v' = \text{Helper} \rangle \)
have hyps (nodeOf v') p' = None by simp

moreover

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

moreover

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

ultimately

show ?thesis by metis

qed

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

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

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

theorem finite-tree:
assumes valid-in-port (v,p)
assumes terminal-vertex v
shows tfinite (tree v p pth)
proof (rule cocontr)
let \(?n = \text{Suc (find vertices)}\)
 assume \(\neg \text{tfinite (tree v p pth)}\)
hence \(\neg \text{tfinite (edge-tree v p)}\) unfolding finite-tree-edge-tree.
then obtain es :: ('vertex, 'form, 'var) edge' stream
where ipath (edge-tree v p) es using Konig by blast
with (valid-in-port (v,p))
have forbidden-path v es by (rule path-is-forbidden)
from forbidden-path-prefix-is-path[OF this] forbidden-path-prefix-is-hyp-free[OF this]
obtain v' where path v' v (rev (stake ?n es)) and hyps-free (rev (stake ?n es))
by blast
from this (terminal-vertex v)
have terminal-path \( v' \) \( \text{rev} (\text{stake } \text{?n es}) \) by (rule terminal-pathI)
hence length \( \text{rev} (\text{stake } \text{?n es}) \) \( \leq \) \text{fc ard vertices}
by (rule hyps-free-limited)
thus \( \text{False} \) by simp
qed

The main result of this theory.

\textbf{theorem} solved
\textbf{unfolding} solved-def
\textbf{proof} (intro ballI allI conjI impl)
\begin{itemize}
  \item fix \( c \)
  \item assume \( c \in \text{conc-forms} \)
  \item hence \( c \in \text{set conclusions} \) by (auto simp add: conc-forms-def)
  \item from this(1) conclusions-present
  \item obtain \( v \) where \( v \in \text{vertices} \) and \( \text{nodeOf} \ v = \text{Conclusion} \ c \)
  \item by (auto, metis (no-types, lifting) image-iff image-subset-iff notin-fset)
\end{itemize}

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

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

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

have global-assms TYPE('var) \( \subseteq \) ass-forms by (rule global-in-ass)
moreover

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

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

show \( \exists \Gamma \ t. \ \text{fst} \ (\text{root} \ t) = (\Gamma \vdash c) \land \Gamma \ [\subseteq] \ \text{ass-forms} \land \text{wf} \ t \land \text{finite} \ t \) by blast
qed
de
end
7 Completeness

7.1 Incredible_Trees

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

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

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

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

abbreviation INode n p i s ants ≡ RNode (I n p i s) ants
abbreviation HNode i s ants ≡ RNode (H i s) ants

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

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

context Abstract-Formulas begin
fun iOutPort where
  iOutPort (INode n p i s ants) = p
  | iOutPort (HNode i s ants) = anyP
end

fun iAnnot where iAnnot it = iAnnot' (root it)
fun iSubst where iSubst it = iSubst' (root it)
fun iAnts where iAnts it = children it

type-synonym ('form,'rule,'subst) fresh-check = ('form,'rule) graph-node ⇒ nat ⇒ 'subst ⇒ 'form entailment ⇒ bool

context Abstract-Task
begin

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

  inductive iwf :: ('form,'rule,'subst) fresh-check ⇒ ('form,'rule,'subst,'var) itree ⇒ 'form entailment ⇒ bool
for fc
where
iwf: \[
\begin{align*}
& n \in \text{sset nodes;} \\
& \text{Reg } p \mid \in \text{outPorts } n; \\
& \text{list-all2 } (\lambda \text{ip t. iwf fc t } ((\lambda h . \text{subst } s \text{ (freshen } i \text{ (labelsOut } n \ h))) \mid | \text{ hyps-for } n \text{ ip } \bigcup \Gamma \vdash \text{subst } s \text{ (freshen } i \text{ (labelsIn } n \text{ ip}))))) \\
& (\text{inPorts'} n) \text{ ants}; \\
& \text{fc } n \text{ i } s \text{ (} \Gamma \vdash \text{c}) ; \\
& c = \text{subst } s \text{ (freshen } i \text{ p)} \\
\end{align*}
\]
\[\Rightarrow iwf fc (\text{INode } n \ p \ i \ s \ a n t s) (\Gamma \vdash c)\]

| iwf H: \[
\begin{align*}
& c \not\in \text{ass-forms}; \\
& c \in \Gamma; \\
& c = \text{subst } s \text{ (freshen } i \text{ anyP)} \\
\end{align*}
\]
\[\Rightarrow iwf fc (\text{HNode } i \ s \ []) (\Gamma \vdash c)\]

lemma iwf-subst-fr eshen-outPort:
\[
\begin{align*}
& \text{iwf fc ts ent } \Rightarrow \\
& \text{snd ent } = \text{subst } (\text{iSubst } ts) \text{ (freshen } \text{iAnnot } ts \text{) (iOutPort } ts) \\
\end{align*}
\]

by \((\text{auto elim: iwf.cases})\)

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

\[\text{lemma all-local-vars-Helper[simp]:}\]
\[\text{all-local-vars Helper } = \{\}\]

\[\text{unfolding all-local-vars-def by simp}\]

\[\vspace{0.1cm}\]
\[\text{lemma all-local-vars-Assumption[simp]:}\]
\[\text{all-local-vars (Assumption } c) = \{\}\]

\[\text{unfolding all-local-vars-def by simp}\]

Local freshness side-conditions, corresponding what we have in the theory \text{Natural-Deduction}.

\[\text{inductive local-fresh-check :: (form, 'rule, 'subst) fresh-check where}\]
\[\bigwedge f . f \in \Gamma \Rightarrow \text{freshenLC } i \text{ (all-local-vars } n) \cap \text{loconsts } f = \{\}; \\
\text{freshenLC } i \text{ (all-local-vars } n) \cap \text{subconsts } s = \{\} \\
\] \[\Rightarrow \text{local-fresh-check } n \text{ i } s \text{ (} \Gamma \vdash \text{c})\]

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

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

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

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

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

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

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

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

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

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

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

lemma iwf-outPort:
assumes iwf fc t ent
assumes is ∈ it-paths t
shows Reg (iOutPort (tree-at t is)) |∈| outPorts (iNodeOf (tree-at t is))
using assms
by (induction arbitrary: is rule: iwf.induct)
(auto 4 4 elim: i Paths-RNodeE dest: list-all2-lengthD list-all2-nthD2)

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

prefix (is@[i]) is $\Rightarrow$

$\begin{align*}
i &< \text{length (inPorts' (tree-at t is'))} \\
i &\Rightarrow \\
\text{hyps (inNodeOf (tree-at t is')) h = Some (inPorts' (iNodeOf (tree-at t is')) ! i)} \\
\text{subst (iSubst (tree-at t is')) (freshen (iAnnot (tree-at t is')) (labelsOut (iNodeOf (tree-at t is')) h))} &\in \text{hyps-along t is}
\end{align*}$

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

**lemma** iPath-xapp-Cons-elim:
assumes prefix (xs@[y]) (z#zs)
obtains $\begin{align*}xs &\in [] \text{ and } y = z \\
xs' &\text{ where } xs = z#xs' \text{ and } prefix (xs@[y]) zs
\end{align*}$
using assms by (cases xs) auto

**lemma** iPath-xalongs-
assumes iwf (t ent)
assumes i#is $\in i$Paths t
shows hyps-along t (i#is) =
$(\lambda h. \text{subst (iSubst t) (freshen (iAnnot t) (labelsOut (iNodeOf t) h)) \set (\text{hyps-for (iNodeOf t) (inPorts'} (iNodeOf t) ! i))} \\
\cup \text{hyps-along (iAnts t ! i)} (is ?S1 = ?S2 \cup ?S3)\end{align*}$

**proof**
from assms
have $i < \text{length (iAnts t)} \text{ and } is \in i$Paths (iAnts t ! i)
by (auto elim: iPaths-ConsE)
let $?t' = iAnts t ! i$

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

next
fix is''
assume [simp]: $is' = i \text{ # is'' and prefix (is'' @ [i'])}$ is
have tree-at t is' = tree-at ?t' is'' by simp

note 'prefix (is'' @ [i']) is:
$i' < \text{length (inPorts' (iNodeOf (tree-at t is'))})$;
⟨hyps (iNodeOf (tree-at t is')) h = Some (inPorts' (iNodeOf (tree-at t is')) ! i') ⟩
from this unfolded 'tree-at t is' = tree-at (?t is''')
have subst (iSubst (tree-at (iAnts t ! i) is'')) (freshen (iAnnot (tree-at (iAnts t ! i) is'')) (labelsOut (iNodeOf (tree-at (iAnts t ! i) is''')) h)) ∈ hyps-along (iAnts t ! i) is by (rule hyps-along.intros)

hence x ∈ ?S3 by simp
thus ⊢thesis...

qed

next

fix x
assume x ∈ ?S2 ∪ ?S3
thus x ∈ ?S1
proof

have prefix ([@i]) (i#is) by simp
moreover
from (iwf.1 t)
have length (iAnts t) ≤ length (inPorts' (iNodeOf (tree-at t [])))
by cases (auto dest: list-all2-length D)
with (i {i < -})
have i < length (inPorts' (iNodeOf (tree-at t []))) by simp
moreover
assume x ∈ ?S2
then obtain h where h |∈| hyps-for (iNodeOf t) (inPorts' (iNodeOf t) ! i)
and [simp]: x = subst (iSubst t) (freshen (iAnnot t) (labelsOut (iNodeOf t) h)) by auto
from this (1)
have hyps (iNodeOf (tree-at t [])) h = Some (inPorts' (iNodeOf (tree-at t []))) ! i) by simp
ultimately
have subst (iSubst (tree-at t [])) (freshen (iAnnot (tree-at t [])) (labelsOut (iNodeOf (tree-at t [])) h)) ∈ hyps-along t (i # is)
by (rule hyps-along.intros)
thus x ∈ hyps-along t (i # is) by simp
next

assume x ∈ ?S3
thus x ∈ ?S1
apply (auto simp add: hyps-along.simps)
apply (rule-tac x = i#is' in exI)
apply auto
done

qed

qed

lemma iwf-hyps-exist:
assumes iwf lc it ent
assumes is ∈ it-paths it
assumes tree-at it is = (INode is ants')
assumes fst ent ⊑ ass-forms
shows subst s (freshen i anyP) ∈ hyps-along it is
proof

from assms(1,2,3)
have subst s (freshen i anyP) ∈ hyps-along it is
∨ subst s (freshen i anyP) |∈| fst ent
∧ subst s (freshen i anyP) |∉| ass-forms
proof (induction arbitrary; is rule: iwf.induct)
case (iwf n p s' a' Γ ants c is)

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

43
using \( \text{inf}(1, 2, 3, 4, 5) \)
by (auto intro!: \text{inf}.intros elim!: \text{list-all2-mono})

show \(?\)case
proof (cases is)
  case Nil
  with \( \text{t} \text{ree-at}(\text{IN}\text{ode}\ n\ p\ a'\ s'\ \text{ants}) = \text{H}\text{Node}\ i\ s\ \text{ants}' \)
  show \(?\)thesis by auto
next
  case (Cons \( i'\ \text{is}' \))
  with \( \text{is}' \in\text{it-pats}(\text{IN}\text{ode}\ n\ p\ a'\ s'\ \text{ants}) \)
  have \( i' < \text{length\ ants}\) and \( \text{is}' \in\text{it-pats}(\text{ants}!i') \)
  by (auto elim: \text{it-pats-ConsE})

let \( \Gamma' = (\lambda h. \text{subst\ s'}(\text{freshen\ a'}(\text{labelsOut\ n\ h})))\ [\]?\text{hyp-for\ n}\ (\text{inPorts}'\ n\ !\ i')\)

from \( \text{t} \text{ree-at}(\text{IN}\text{ode}\ n\ p\ a'\ s'\ \text{ants}) = \text{H}\text{Node}\ i\ s\ \text{ants}' \)
have \( \text{t} \text{ree-at}(\text{ants}!i')\ \text{is}' = \text{H}\text{Node}\ i\ s\ \text{ants}'\ \text{using Cons by simp} \)
from \( \text{inf.HI} (i' < \text{length\ ants})\ ;\ \text{is}' \in\text{it-pats}(\text{ants}!i')\); this
have \( \text{subt\ s}(\text{freshen\ i\ anyP})\ \in\ \text{hyp}\text{-along}(\text{ants}!i')\ \text{is}' \)
  \( \lor\ \text{subt\ s}(\text{freshen\ i\ anyP})\ [\in\ ?\Gamma']\ [\cup\] \\\Sigma\ \text{subt\ s}(\text{freshen\ i\ anyP})\ [\notin\ \text{ass-forms}] \)
  by (auto dest: \text{list-all2-nthD2})
moreover
from \( \text{is}' \in\text{it-pats}(\text{IN}\text{ode}\ n\ p\ a'\ s'\ \text{ants}) \)
have \( \text{hyp}\text{-along}(\text{IN}\text{ode}\ n\ p\ a'\ s'\ \text{ants}) = \text{fset}\ ?\Gamma'\ \cup\ \text{hyp}\text{-along}(\text{ants}!i')\ \text{is}' \)
  \( \text{using is} = \cdot \)
  by (simp add: \text{hyp}\text{-along-Cons}[\text{OF} \text{inf.lc}(\text{IN}\text{ode}\ n\ p\ a'\ s'\ \text{ants})(\Sigma \Gamma\ c)])
ultimately
show \(?\)thesis by auto
qed

next
  case (\text{inf.H c} \Gamma\ s'\ i'\ \text{is})
  hence (\text{simp}): \text{is} = []\ i' = i\ s' = s\ by\ \text{simp-all}
from \( \text{c} = \text{subt\ s}\ (\text{freshen\ i}'\ \text{anyP})\ \langle\ c\ [\in\ ]\ \Sigma\ \langle\ \text{c}\ [\notin\ \text{ass-forms}]\) \)
show \(?\)case by simp
qed
with \text{assms(4)}
show \(?\)thesis by blast
qed

definition \text{hyp}\text{-port-for}':: (\text{form},\ \text{rule},\ \text{subt},\ \text{var})\ \text{t}ree\ \Rightarrow\ \text{nat\ list}\ \Rightarrow\ \text{form}\ \Rightarrow\ \text{nat\ list}\ \times\ \text{nat}\ \times\ (\text{form},\ \text{var})\ \text{out-port\ where} \)
\text{hyp}\text{-port-for}'\ \text{t}\ \text{is}\ \text{f} = (\text{SOME} \ x). \)
  (case \text{x}\ of\ \text{(is}',\ i,\ h)\ \Rightarrow\ 
    \text{prefix}\ (\text{is}' \oplus [i])\ \text{is}\ \land
    i < \text{length}\ (\text{inPorts}'(\text{INodeOf}\ \text{(tree-at} t\ \text{is}'))))\ \land
    \text{hyps}(\text{INodeOf}\ (\text{tree-at} t\ \text{is}'))\ h = \text{Some\ (inPorts}'(\text{INodeOf}\ (\text{tree-at} t\ \text{is}')\ !i)\ \land
    \text{f} = \text{subt}\ (\text{INodeOf}\ (\text{tree-at} t\ \text{is}'))\ (\text{freshen}\ (\text{INode}\ \text{(tree-at} t\ \text{is}')\ (\text{labelsOut}\ (\text{INodeOf}\ (\text{tree-at} t\ \text{is}')\ h)))\ )\ )\)

lemma \text{hyp}\text{-port-for-spec}';
  assumes \text{f} \in \text{hyp}\text{-along\ t}\ \text{is}
  shows (case \text{hyp}\text{-port-for}'\ \text{t}\ \text{is}\ \text{f}\ \text{of}\ \text{(is}',\ i,\ h)\ \Rightarrow\ 
    \text{prefix}\ (\text{is}' \oplus [i])\ \text{is}\ \land
    i < \text{length}\ (\text{inPorts}'(\text{INodeOf}\ (\text{tree-at} t\ \text{is}'))))\ \land
    \text{hyps}(\text{INodeOf}\ (\text{tree-at} t\ \text{is}'))\ h = \text{Some\ (inPorts}'(\text{INodeOf}\ (\text{tree-at} t\ \text{is}')\ !i)\ \land

\[ f = \text{subst} (\text{iSubst} (\text{tree-at\,t\,is})) (\text{freshen} (\text{iAnnot} (\text{tree-at\,t\,is}))) (\text{labelsOut} (\text{iNodeOf} (\text{tree-at\,t\,is}))) \) \\
\text{using} \quad \text{assms} \quad \text{unfolding} \quad \text{hyps-along}, \text{simp} \quad \text{hyp-port-for'-def} \quad \text{by} \quad -(\text{rule someI-ex}, \text{blast})

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

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

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

\text{lemma} \quad \text{hyp-port-prefix}: \\
\text{assumes} \quad f \in \text{hyps-along\,t is} \\
\text{shows} \quad \text{prefix} (\text{hyp-port-path-for\,t is f} \circ [\text{hyp-port-i-for\,t is f}]) \text{ is}

\text{using} \quad \text{hyp-port-for-spec'}(\text{OF\ assms}) \quad \text{unfolding} \quad \text{hyp-port-path-for-def} \quad \text{hyp-port-i-for-def} \quad \text{by} \quad \text{auto}

\text{lemma} \quad \text{hyp-port-strict-prefix}: \\
\text{assumes} \quad f \in \text{hyps-along\,t is} \\
\text{shows} \quad \text{strict-prefix} (\text{hyp-port-path-for\,t is f}) \text{ is}

\text{using} \quad \text{hyp-port-prefix}'(\text{OF\ assms}) \quad \text{by} \quad (\text{simp\ add:\ strict-prefix',\ prefix-order.dual-order.strict-trans})

\text{lemma} \quad \text{hyp-port-it-paths}: \\
\text{assumes} \quad \text{is } \in \text{it-paths\,t} \\
\text{assumes} \quad f \in \text{hyps-along\,t is} \\
\text{shows} \quad \text{hyp-port-path-for\,t is f} \in \text{it-paths\,t}

\text{using} \quad \text{assms} \quad \text{by} \quad (\text{rule\ it-paths-strict-prefix}(\text{OF\ -\ hyp-port-strict-prefix})

\text{lemma} \quad \text{hyp-port-hyps}: \\
\text{assumes} \quad f \in \text{hyps-along\,t is} \\
\text{shows} \quad \text{hyps} (\text{iNodeOf} (\text{tree-at\,t\,(hyp-port-path-for\,t\,is\,f)})) (\text{hyp-port-h-for\,t\,is\,f}) = \text{Some} (\text{inPorts'} (\text{iNodeOf} (\text{tree-at\,t\,(hyp-port-path-for\,t\,is\,f)}))) ! \text{hyp-port-i-for\,t\,is\,f})

\text{using} \quad \text{hyp-port-for-spec'}(\text{OF\ assms}) \quad \text{unfolding} \quad \text{hyp-port-path-for-def} \quad \text{hyp-port-i-for-def} \quad \text{hyp-port-h-for-def} \quad \text{by} \quad \text{auto}

\text{lemma} \quad \text{hyp-port-outPort}: \\
\text{assumes} \quad f \in \text{hyps-along\,t is} \\
\text{shows} \quad (\text{hyp-port-h-for\,t\,is\,f}) \subset \text{outPorts} (\text{iNodeOf} (\text{tree-at\,t\,(hyp-port-path-for\,t\,is\,f)}))

\text{using} \quad \text{hyps-correc'}(\text{OF\ hyp-port-hyps}(\text{OF\ assms})),

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

\text{using} \quad \text{hyp-port-for-spec'}(\text{OF\ assms}) \quad \text{unfolding} \quad \text{hyp-port-path-for-def} \quad \text{hyp-port-i-for-def} \quad \text{hyp-port-h-for-def} \quad \text{by} \quad \text{auto}

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

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

\text{lemma} \quad \text{isidx-}\text{inj}[\text{simp}]: \quad \text{isidx\,xs = isidx\ ys} \iff \text{xs = ys}

\text{unfolding} \quad \text{isidx-def\ by\ simp}

\text{lemma} \quad \text{isidx-}\text{v-away}[\text{simp}]: \quad \text{isidx\,xs} \neq \text{v-away}

\text{unfolding} \quad \text{isidx-def\ v-away-def\ by\ simp}

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

\text{lemma} \quad \text{mapWithIndex-cong}\ (\text{fundef-cong})
\[ xs = ys \implies (\forall x. i. x \in \text{set} \ ys \implies f \ i \ x = g \ i \ x) \implies \text{mapWithIndex} \ f \ xs = \text{mapWithIndex} \ g \ ys \]

**unfolding** \text{mapWithIndex-Nil} [simp]: \text{mapWithIndex} \ f \ [] = []

**unfolding** \text{mapWithIndex-def} by simp

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

**unfolding** \text{mapWithIndex-def} by simp

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

**unfolding** \text{mapWithIndex-def} by (auto simp add: \text{nth-enumerate-eq})

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

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

where

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

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

where

- \text{globalize} is \text{f} \ (\text{RNode} \ r \ \text{ants}) = \text{RNode}
- \text{globalize} is \text{f} \ r
- \text{mapWithIndex} \ (\lambda \ i. t.
  \text{globalize} \ (is@i'))
  \ (\text{rerename} \ (\text{a-fresh} \ (\text{inPorts} \ (\text{iNodeOf} \ (\text{RNode} \ r \ \text{ants})))) \ i')
  \ (\text{iAnnot} \ (\text{RNode} \ r \ \text{ants})) \ (\text{isidx} \ is) \ f)

) \ \text{ants}

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

by (cases \ n) auto

**lemma** \text{iAnnot-globalize}:
- \text{assumes} is' \in \text{it-path} (\text{globalize} is \text{f} \ t)
- \text{shows} \text{iAnnot} \ (\text{tree-at} (\text{globalize} is \text{f} \ t) \ \text{is}') = \text{isidx} \ (is@is')

using \text{assms}

by (induction \ t \ arbitrary; \ f \ is \ is') (auto elim!: \text{it-path-RNodeE})

**lemma** \text{all-consts-listed}:
- \text{assumes} \ n \in \text{sset nodes}
- \text{assumes} p \in \text{inPorts} \ n
- \text{shows} \text{lc const} (a\text{-conc} \ p) \cup (\bigcup \text{lc const} \ (a\text{-hyp} \ p)) \subseteq \text{a-fresh} \ p

using \text{assms}

by (auto simp add: \text{nodes-def stream.set-map lc const anyP closed-no-consts conclusions-closed fmember.rep-eq f-antecedent-def dest!: all-consts-listed)

**lemma** \text{no-local-consts-in-consequences}:
- \text{n} \in \text{sset nodes} \implies \text{Rep} \ p \in \text{outPorts} \ n \implies \text{lc const} \ p = \{}

using \text{no-local-consts-in-consequences}

46
lemma iwf-globalize:

assumes local-iwf t (Γ ⊢ c)

shows plain-iwf (globalize is f t) (renameLCs f |·| Γ ⊢ renameLCs f c)

using assms

proof (induction t Γ ⊢ c arbitrary: is f Γ c rule: iwf.induct)

case (iwf n p s i Γ ants c is f)

note (n ∈ sset nodes)

moreover

note (Reg p |∈| outPorts n)

moreover

{ fix i'

  let ?V = a-fresh (inPorts' n ! i')
  let $?f' = rename ?V i (isidx is) f
  let $?t = globalize (is @ [i']) $?f' (ants ! i')
  let $?ip = inPorts' n ! i'

  let $?f'' = (λh. subst (subt renameLCs f s) (freshen (isidx is) (labelsOut n h))) |·| hyps-for n $?ip

  let $?c' = subst (subt renameLCs f s) (freshen (isidx is) (labelsIn n $?ip))

  assume i' < length (inPorts' n)

  hence (inPorts' n ! i') |∈| inPorts n by (simp add: inPorts-set-of)

  from i' < length (inPorts' n)

  have subset-V: ?V ⊆ all-local-vars n

  unfolding all-local-vars-def

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

  from local-fresh-check n i s (Γ ⊢ c)

  have freshenLC i · all-local-vars n ∩ subst-consts s = {}

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

  hence freshenLC i · ?V ∩ subst-consts s = {}

  using subset-V by auto

  hence rename-subst: subst renameLCs ?f' s = subst renameLCs f s

  by (rule rename-subst-ssnoop)

  from all-local-consts-listed' [OF ∧ n ∈ sset nodes] ((inPorts' n ! i') |∈| inPorts n]

  have subst-conc: consts (a-conc (inPorts' n ! i')) ⊆ ?V

  and subset-hyp: Λ hyp . hyp |∈| a-hyps (inPorts' n ! i') ⇒ consts hyp ⊆ ?V

  by (auto simp add: fmember.rep-eq)

  from List.List-all2-nthD [OF ∧ list-all2 · · · (i' < length (inPorts' n)), simplified]

  have plain-iwf ?t

  (renameLCs ?f' |·| ((λh. subst s (freshen i (labelsOut n h))) |·| hyps-for n ?ip |∪| Γ) ≈ renameLCs ?f' (subst s (freshen i (a-conc ?ip))))

  by simp

  also have renameLCs ?f' |·| ((λh. subst s (freshen i (labelsOut n h))) |·| hyps-for n ?ip |∪| Γ) ≈ (λs. subst (subt renameLCs ?f' s) (renameLCs ?f' (freshen i (labelsOut n x)))) |·| hyps-for n ?ip |∪|

  renameLCs ?f' |·| Γ

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

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

  proof (rule fimage-cong (OF refl))

  fix x

  assume x |∈| Γ

  with local-fresh-check n i s (Γ ⊢ c):

  have freshenLC i · all-local-vars n ∩ leasts x = {}
by (elim local-fresh-check.cases) simp
hence freshenLC i ?V ∩ lconsts x = {}
using subset-V by auto
thus renameLCs ?f' x = renameLCs f x
by (rule rename-name-noop)

qed
also have (λx. subst (renameLCs ?f' s) (renameLCs ?f' (freshen i (labelsOut n x)))) ∈ hyps-for n
by (rule image-cong[OF refl])

fix hyp
assume hyp ∈ hyps-for n (inPorts n x i)
hence labelsOut n hyp ∈ a-hyps (inPorts n x i)
apply (cases hyp)
apply (cases n)
apply (auto split: if-splits)
done

from subset-hyp [OF this]
have subset-hyp: lconsts (labelsOut n hyp) ⊆ ?V.

have 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-name rename-subst)
apply (rule arg-cong[OF renameLCs-cong])
apply (auto dest: subsetD[OF subset-hyp])
done

qed
also have renameLCs ?f' (renameLCs (freshen i (a-conc ?ip))) = subst (renameLCs ?f' s) (renameLCs f (freshen i (a-conc ?ip))) by (simp add: rename-subst)
also have ... = ?c'
apply (simp add: freshen-def rename-name rename-subst)
apply (rule arg-cong[OF renameLCs-cong])
apply (auto dest: subsetD[OF subset-conc])
done

finally
have plain-iwf ?t (?Γ' |∪| renameLCs f |'| Γ ⊢ ?c').

with list-all2-lengthD[OF list-all2 - - -]
have list-all2
(A ip t. plain-iwf t ((λh. subst (renameLCs f s) (freshen (isidx is) (labelsOut n h))) |'| hyps-for n ip |∪| renameLCs f |'| Γ ⊢ subst (renameLCs f s) (freshen (isidx is) (labelsIn n ip))))
apply (auto simp add: list-all2-conv-all-nth)

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

have (n ∈ sset nodes) (Reg p |∈| outPorts n)
apply (rule no-localconsts-in-consequences)

have lconsts p = {} by (rule rename-no-localconsts-in-consequences)

ultimately
show true
unfolding globalize.simps globalize-node.simps iNodeOf.simps iAnnot.simps iNodeSel.rename-tree Sel Let-def
by (rule iwf.intros(1))

next

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

definition fresh-at where
  fresh-at t xs = (case rev xs of [] ⇒ {} | (i#is′) ⇒ freshenLC (iAnnot (tree-at t (rev is′))(a-fresh (inPorts′ (iNodeOf (tree-at t (rev is′)))) ! i)))

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

lemma fresh-at-snoc[simp]:
  fresh-at t (is@i) = freshenLC (iAnnot (tree-at t is)) \ (a-fresh (inPorts′ (iNodeOf (tree-at t is))) ! i))
  unfolding fresh-at-def by simp

lemma fresh-at-def':
  fresh-at t is = case rev is of [] ⇒ {} | (i#is′) ⇒ freshenLC (iAnnot (tree-at t (rev is′))) \ (a-fresh (inPorts′ (iNodeOf (tree-at t is))) ! i))
  unfolding fresh-at-def by (auto split: list.split)

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

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

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

lemma fresh-at-path-Cons[simp]:
  fresh-at-path t (i#is) = fresh-at t [i] ∪ fresh-at-path (iAnts t ! i) is
  unfolding fresh-at-path-def
  by (fastforce split: if-splits)
lemma globalize-local-cons:
  assumes is' ∈ it-paths (globalize is f t)
  shows subst-cons (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 rants is ')
  apply (case-tac r)
  apply (auto simp add: subst-cons subst-renameLCs elim: it-paths-RNodeE)
  apply (solves [force dest!: subsetD [OF range-rename]!])
  apply (solves [force dest!: subsetD [OF range-rename]!])
  done

lemma iwf-globalize':
  assumes local-iiw 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) 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-consts rename-closed)
    also from Pair(2)
    have renameLCs (freshenLC v-away) c = Γ = Γ
      by (auto simp add: closed-no-consts rename-closed fmember.rep-eq image-i)
    finally show ?case.
  qed
end

7.2 Build_Incredible_Tree

description "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-i)
    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
begin
lemma image-eq-to-f:
  assumes \( f_1 \mid S_1 = f_2 \mid S_2 \)
  obtains \( f \) where \( \forall x. x \in S_2 \implies f x \in S_1 \land f_1 (f x) = f_2 x \)
using assms apply transfer using image-eq-to-f by metis
end

context Abstract-Task
begin

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

  from (wft)
  have eff (snd (root \( t \))) (fst (root \( t \))) ((fst \( \circ \) root) \mid | cont \( t \)) using wf.simps by blast

  from (wft)
  have \( \forall t'. t' \in | cont \( t \) \implies wft t' \) using wf.simps by blast
  hence III: \( \forall t'. t' \in | cont \( t \) \implies (\exists it'. local-iwf it' (fst (root \( t' \)))) \) using tfinite(2) by blast
  then obtain its where its: \( \forall t', t' \in | cont \( t \) \implies local-iwf (its t') (fst (root \( t' \))) \) by metis

  from (eff - - -)
  show ?case
  proof (cases rule: eff.cases [case-names Axiom NatRule Cut])
    case (Axiom \( c \) \( \Gamma \))
    show ?thesis
    proof (cases c \mid | ass-forms)
      case True
      then have c \( \in \) set assumptions by (auto simp add: ass-forms-def)
      let \( ?it = INode (Assumption c) c \) undefined undefined \( [] :: ('form, 'rule, 'subst, 'var) iTree \)
      from (c \in set assumptions)
      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: leoms-t-anyP freshen-closed)
    let \( ?it = INode undefined s \) \( [] :: ('form, 'rule, 'subst, 'var) iTree \)
    from (c \mid | \( \Gamma \) False
    have local-iwf \( ?it (\Gamma \vdash c) \) by (auto intro: iwfH)
    thus ?thesis unfolding Axiom..
    qed
  next

51
case (NatRule rule c ants Γ i s)
  from (natEff-Inst rule c ants)
  have snd rule = c and [simp]: ants = f-antecedent (fst rule) and c ∈ set (consequent (fst rule))
  by (auto simp add: natEff-Inst.simps)

  from (fst o root) [\ant. (\p. subst s (freshen i p)) \ant. (\p. substit s (freshen i p)) \ant. (\p. subst s (freshen i (a-conc ant)))] [\ant. (\p. substit s (freshen i p)) \ant. (\p. subst s (freshen i (a-conc ant)))]
  obtain to-t where \ant. ant \ant. ants \ant. to-t ant \ant. cont t \ant. (fst o root) (to-t ant) = (\ant. subst s (freshen i p)) \ant. a-hyps ant \ant. (\ant. subst s (freshen i (a-conc ant)))
  by (rule image-eq-to-f) (rule that)
  hence to-t-in-cont: \ant. ant \ant. ants \ant. to-t ant \ant. cont t
  and to-t-root: \ant. ant \ant. ants \ant. to-t ant = (\ant. subst s (freshen i p)) \ant. a-hyps ant
  by auto

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

  from (snd (root t)) ∈ R
  have fst rule ∈ sset rules
    unfolding NatRule
    by (auto simp add: stream.set-map n-rules-def no-empty-conclusions)
  moreover
  have c ∈ set (consequent (fst rule))
  by (simp add: f-consequent-def)
  moreover
  { fix ant
    assume ant ∈ set (antecedent (fst rule))
    hence ant ∈ ants by (simp add: f-antecedent-def)
    from its[OF to-t-in-cont[OF this]]
    have local-iwf (its (to-t ant)) (fst (mot (to-t ant))).
    also have fst (mot (to-t ant)) = (\ant. subst s (freshen i p)) \ant. a-hyps ant \ant. (\ant. subst s (freshen i (a-conc ant)))
    by (rule to-t-root[OF ant \ant. ants])
    also have \ldots = (\ant. subst s (freshen i (labelsOut (Rule (fst rule)) h))) \ant. hyps-for (Rule (fst rule)) ant \ant. (\ant. subst s (freshen i (a-conc ant)))
    by auto
    finally
    have local-iwf (its (to-t ant)) (\ant. subst s (freshen i (labelsOut (Rule (fst rule)) h))) \ant. hyps-for (Rule (fst rule)) ant \ant. (\ant. subst s (freshen i (a-conc ant))).
  }
  moreover
  from NatRule(5.6)
  have local-fresh-check (Rule (fst rule)) i s (\ant. subst s (freshen i c))
  by (fastforce intro!: local-fresh-check.intros simp add: all-local-vars-def fmember.rep-eq)
  ultimately
  have local-iwf ?it ((\ant. subst s (freshen i c)))
  by (intro iwf ) (auto simp add: list-all2-map2 list-all2-same)
  thus \thesis unfolding NatRule.
next
  case (Cut Γ con)
  obtain s where subst s anyP = con by atomize-eq (rule anyP-is-any)
  hence [simp]: subst s (freshen undefined anyP) = con by (simp add: leconsts-anyP freshen-closed)
from \((f_{st} \circ \text{root}) |\) \(\vdash\) cont \(t = |\Gamma \vdash \text{con}||\)

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

by (cases cont \(t\) auto)

from \(t' \in\) cont \(t\) obtain \(it'\) where local-iwf \(it'\) \((\Gamma \vdash \text{con})\) by (auto intro: iwf local-fresh-check intros)

thus \(\text{thesis unfolding Cut.}\)

qed

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

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

lemma iwf-to-it:

assumes tfinite \(t\) and uf \(t\)

shows local-iwf \(\vdash (\text{to-it } t) (f_{st} (\text{root } t))\)

unfolding to-it-def using build-local-iwf \([OF \text{assms}]\) 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 \(\text{form} \times (\text{var}) \text{ edge''} = (\text{form vertex}, \text{form} \times (\text{var}) \text{ edge}'\)

locale Solved-Task =

Abstract-Task freshenLC renameLCs lecons closed subst subst-lecons subst-renameLCs anyP antecedent

consequent rules assumptions conclusions

for freshenLC :: nat \Rightarrow \text{var} \Rightarrow \text{var}

and renameLCs :: (\text{var} \Rightarrow \text{var}) \Rightarrow \text{form} \Rightarrow \text{form}

and lecons :: \text{form} \Rightarrow \text{var set}

and closed :: \text{form} \Rightarrow \text{bool}

and subst :: subst \Rightarrow \text{form} \Rightarrow \text{form}

and subst-lecons :: subst \Rightarrow \text{var set}

and subst-renameLCs :: (\text{var} \Rightarrow \text{var}) \Rightarrow (\text{subst} \Rightarrow \text{subst})

and anyP :: \text{form}

and antecedent :: \text{rule} \Rightarrow (\text{form} \times \text{var}) \text{ antecedent list}

and consequent :: \text{rule} \Rightarrow \text{form list}

and rules :: \text{rule stream}

and assumptions :: \text{form list}

and conclusions :: \text{form list} +

assumes solved: solved

begin

Let us get our hand on concrete trees.

definition ts :: \((\text{form entailment}) \times (\text{rule} \times \text{form}) \text{NatRule})\) \text{tree where}
\[ \text{ts } c = (\text{SOME } t. \text{ snd } (\text{fst } (\text{root } t))) = c \land \text{fst } (\text{fst } (\text{root } t)) \subseteq \text{ass-forms } \land \text{wf } t \land \text{finite } t \]

**Lemma**

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

**Unfolding**

- \( \text{atomize-conj conj-assoc ts-def} \)
- **Apply** (rule solved-assms)
  - Using solved assms
  - By (force simp add: solved-def)

**Abbreviation** \( \text{it' where} \)

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

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

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

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

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

**Unfolding**

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

**Lemma** \( \text{prefixeq-vertices: } (c, is) \in \text{vertices} \Rightarrow \text{prefix is'} is \Rightarrow (c, is') \in \text{vertices} \)

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

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

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

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

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

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

- **Assumes** \( v \in \text{vertices} \)
  - **Obtains** \( c \text{ where } c \in \text{set conclusions and } v = (c, []) \)
    - **Where** \( c \text{ is where } c \in \text{set conclusions and } is \in \text{it-paths } (\text{it'} c) \text{ and } v = (c, 0\#is) \)
  - **Using** assms by (cases v; rename-tac is; case-tac is; auto)

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

- **Assumes** \( v \in \text{vertices} \)
  - **Assumes** \( \land c. c \in \text{set conclusions } \Rightarrow P (c, []) \)
  - **Assumes** \( \land c. c \in \text{set conclusions } \Rightarrow is \in \text{it-paths } (\text{it'} c) \Rightarrow P (c, 0\#is) \)
  - **Shows** \( P v \)
  - **Using** assms by (cases v; rename-tac is; case-tac is; auto)

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

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

\( \text{nodeOf } (pf; i \# is) = \text{iNodeOf } (\text{tree-at } (\text{it'} pf) ) \text{ is} \)
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] subst)

sublocale Vertex-Graph nodes inPorts outPorts vertices nodeOf,
definition edge-from :: 'form ⇒ nat list ⇒ (nat list × ('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 ⇒ (nat list × ('form,'var) in-port) where
  edge-to c is =
    (case rev is of  [] ⇒ ((c, []), in-port-at (c, [])), in-port-at (c, []))
      | 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[i]) = ((c, 0 # is), in-port-at ((c, 0 # is)) i)
by (simp add: edge-to-def)

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

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

lemma hyps-exist':
  assumes c ∈ set conclusions
  assumes is ∈ it-paths (it' c)
  assumes tree-at (it' c) is = (HNode i s ants)
  shows subst s (freshen i anyP) ∈ hyps-along (it' c) is
proof −
  from assms(1)
  have plain-iwf (it' c) (fst (root (ts c))) by (rule iwf-it)
  moreover
  note assms(2,3)
  moreover
  have fst (fst (root (ts c))) ⊆ ass-forms
    by (simp add: assms(1) to-context)
  ultimately
  show thesis by (rule iwf-hyps-exist)
qed
definition \texttt{hyp-edge-to} :: \texttt{form} $\Rightarrow$ \texttt{nat list} $\Rightarrow$ (\texttt{form vertex} $\times$ (\texttt{form}, \texttt{var}) \texttt{in-port}) where
\texttt{hyp-edge-to} \texttt{c} is $= ((c, \theta \neq \texttt{is}), \texttt{plain-ant anyP})$

definition \texttt{hyp-edge-from} :: \texttt{form} $\Rightarrow$ \texttt{nat list} $\Rightarrow$ \texttt{nat} $\Rightarrow$ (\texttt{form vertex} $\times$ (\texttt{form}, \texttt{var}) \texttt{out-port}) where
\texttt{hyp-edge-from} \texttt{c} is \texttt{n s} $= ((c, \theta \neq \texttt{hyp-port-path-for} \ (\texttt{it'} c)) \texttt{is (subst s (freshen n anyP))}),$
\texttt{hyp-port-h-for} \ (\texttt{it'} c) \texttt{is (subst s (freshen n anyP))})$

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

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

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

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

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

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

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

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

lemma \texttt{hyp-edge-to-valid-in-port}:
\texttt{assumes is \in it-paths (it' c)}
\texttt{assumes c \in set conclusions}
\texttt{assumes tree-at (it' c) is = HNode n s ants}
\texttt{shows valid-in-port (hyp-edge-to c is)}
using \texttt{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' ℓ→ (∃ 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'.sims 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-nilE[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 rule: rev-induct)
  (auto simp add: path-cons-simp intro!: regular-edge elim: it-paths-nilE)

lemma edge-step:
  assumes (((a, b), ba), ((aa, bb), bc)) ∈ edges
  obtains
  i where a = aa and b = bb@[i] and bc = in-port-at (aa,bb) i and hyps (nodeOf (a,b)) ba = None
  i where a = aa and prefix (b@[i]) bb and hyps (nodeOf (a,b)) ba = Some (in-port-at (a,b) i)
using assms
proof(cases rule: edges.cases{consumes 1, case-names Reg Hyp})
  case (Reg c is)
  then obtain i where a = aa and b = bb@[i] and bc = in-port-at (aa,bb) i and hyps (nodeOf (a,b)) ba = None
  by (auto elim!: edges.cases simp add: edge-at-def edge-from-def edge-to-def split: list.split list.split-asm)
  thus thesis by (rule that)
next
  case (Hyp c is n s)
  let ?i = hyp-port-i-for (it' c) is (subst s (freshen n anyP))
  from Hyp have a = aa and prefix (b@[?i]) bb and
  hyps (nodeOf (a,b)) ba = Some (in-port-at (a,b) ?i)
  by (auto simp add: edge-at-def edge-from-def edge-to-def hyp-edge-at-def hyp-edge-to-def hyp-edge-from-def
lemma path-has-prefixes:
assumes path v v' pth
assumes snd v' = []
assumes prefix (is' @ [i]) (snd v)
shows ((fst v, is'), (in-port-at (fst v, is') i)) \in snd ' set pth
using assms
by (induction rule: path.induct)(auto elim!: edge-step dest: prefix-snocD)

lemma in-scop-e: valid-in-port (v', p') \Longrightarrow v \in scop-e (v', p') \Longleftrightarrow scop-e' v' p' v
proof
assume v \in scop-e (v', p')
hence v \in vertices and \ \land pth t. path v t pth \Longrightarrow terminal-vertex t \Longrightarrow (v', p') \in snd ' set pth
by (auto simp add: scop-e.simps)
from this
show scop-e' v' p' v
proof (induction rule: vertices-induct)
case (None c)
from None(2)(of (c, [])) [], simp; [1], simpfied, OF None(1)]
have False.
thus scop-e' v' p'(c, []).
next
case (Some c is)
from (c \in set conclusions) \is \in it-paths (it' c):
have path (c, 0 #is) (c, []) (terminal-path-from c is)
by (rule path-terminal-path-from)
moreover
from (c \in set conclusions)
have terminal-vertex (c, []) by simp
ultimately
have (v', p') \in snd ' set (terminal-path-from c is)
by (rule Some(3))
hence (v',p') \in set (map (edge-to c) (prefixes is))
unfolding terminal-path-from-def by auto
then obtain is' where prefix is' is and (v',p') = edge-to c is'
by auto
show scop-e' 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 \in set conclusions) (is \in it-paths (it' c))
show ?thesis by (auto intro!: scop-e'.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 \in set conclusions) (is \in it-paths (it' c)) (prefix is' is)[unfolded snoc]
show ?thesis
by (auto intro!: scop-e'.intros)
qed
qed

next

assume valid-in-port (v', p')
assume scope' v' p' v

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

from (scope' v' p' v).
have (c, is) ∈ vertices unfolding (v = -) by (rule scope-valid)

hence (c, is) ∈ scope ((c, is'), p')

proof (rule scope.intros)
  fix pth t
  assume path (c, is) t pth

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

  from path-has-prefixes[OF (path (c, is) t pth) (snd t = [])], simplified, OF 'prefix (is' @ [t]) is]
  show ((c, is'), p') ∈ snd t set unfolding (p' = -) (v' = -).

qed

thus v ∈ scope (v', p') using (v =-) (v' =-) by simp

qed

sublocale Port-Graph nodes inPorts outPorts vertices nodeOf edges
proof
  show nodeOf 'f set vertices ⊆ set nodes
    apply (auto simp add: fnmember.revp-eq[symmetric] mem-vertices)
  apply (auto simp add: stream.set-map dest: iNodeOf-tree-at[OF iwf-it])
  done

next

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

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

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

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

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

lemma my-vidx-inj: inj-on vidx (fset vertices)
  by (rule inj-on1)
lemma vidx-not-v-away [simp]: v ∈ vertices ↯ vidx v ≠ v-away
by (cases v rule:vidx.cases) (auto simp add: Annont-globalize simp del: Annont.simps)

sublocale Instantiation inPorts outPorts nodeOf hyps nodes edges vertices labelsIn labelsOut freshenLC renameLCs lc insts 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-Gmph nodes inPorts outPorts vertices nodeOf edges hyps
proof
  fix v1 p1 v2 p2 p'
  assume assms: ((v1, p1), (v2, p2)) ∈ edges hyps (nodeOf v1) p1 = Some p'
  from assms(1) hyps-correct[OF assms(2)]
  have valid-out-port (v1, p1) and valid-in-port (v2, p2) and valid-in-port (v1, p') and v2 ∈| vertices
    using valid-edges by auto
  from assms
  have ∃ i. fst v1 = fst v2 ∧ prefix (snd v1 @@ i) (snd v2) ∧ p' = in-port-at v1 i
    by (cases v1; cases v2; auto elim!: edge-step)
  hence scope' v1 p' v2
    unfolding scope-valid-inport[OF v2 ∈| vertices].
  hence v2 ∈ scope (v1, p')
    unfolding in-scope[OF valid-in-port (v1, p')].
  thus (v2, p2) = (v1, p') ∨ v2 ∈ scope (v1, p') ..
qed

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

sublocale Saturated-Graph nodes inPorts outPorts vertices nodeOf edges
proof
  fix v p
  assume valid-in-port (v, p)
  thus ∃ e ∈ edges. snd e = (v, p)
  proof (induction v)
  fix c cis
    assume valid-in-port ((c, cis), p)
    hence c ∈ set conclusions by (auto simp add: mem-vertices)
    show ∃ e ∈ edges. snd e = ((c, cis), p)
    proof (cases cis)
    case Nil
    with valid-in-port ((c, cis), p)
    have simp]: p = plain-ant c by simp
    have [] ∈ it-paths (it' c) by simp
    with c ∈ set conclusions
    have edge-at c [] ∈ edges by (rule regular-edge)
    moreover

60
have \( \text{snd} \ (\text{edge-at} \ c \ []) = ((c, \ [])', \text{plain-ant} \ c) \)

by (simp add: \text{edge-to-def})

ultimately

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

next

case \((\text{Cons} \ c' \ is)\)

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

have \([\text{simp}]: \ c' = 0 \ and \ is \in \text{it-paths} \ (it' \ c)\)

and \(p \in \text{inPorts} \ (\text{iNodeOf} \ \text{tree-at} \ (it' \ c) \ is)\) by auto

from this(3) obtain \(i\) where

\(i < \text{length} \ (\text{inPorts}' \ (\text{iNodeOf} \ \text{tree-at} \ (it' \ c) \ is))\) and

\(p = \text{inPorts}' \ (\text{iNodeOf} \ \text{tree-at} \ (it' \ c) \ is) \ ! \ i\)

by (auto simp add: \text{inPorts-set-of in-set-conv-nth})

show \(?thesis\)

proof (cases \text{tree-at} \ (it' \ c) \ is)

case \([\text{simp}]: \ (\text{RNode} \ r \ ants)\)

show \(?thesis\)

proof (cases \(r\))

case \(I\)

hence \(\neg \text{isHNode} \ \text{tree-at} \ (it' \ c) \ is\) by simp

from \(\text{iwf-length-inPorts-not-HNode}[OF \text{iwf-it}[OF \ c \in \text{set conclusions}]] \ \text{(is \in \text{it-paths} \ (it' \ c) \ this)}\)

\(i < \text{length} \ (\text{children} \ (\text{tree-at} \ (it' \ c) \ is))\) by simp

with \(\text{is} \in \text{it-paths} \ (it' \ c)\),

have \(\text{is} \in \text{it-paths} \ (it' \ c) \ by \ \text{(rule it-path-Snoc})\)

from \(\text{c} \in \text{set conclusions} \ )\ this

have \(\text{hyd-\text{at} c} \ (\text{is} \in \text{set conclusions}) \) \text{by edges by (rule regular-edge)\}

moreover

have \(\text{snd} \ (\text{edge-at} \ c \ (\text{is} \in \text{set conclusions}) \) = ((c, 0 \# \ is), \ \text{inPorts}' \ (\text{iNodeOf} \ \text{tree-at} \ (it' \ c) \ is) \ ! \ i)\)

by (simp add: \text{edge-to-def})

ultimately

show \(?thesis\) by (auto simp add: \text{Cons} \ \text{p} = \# \ simp del: \text{snd-edge-at})

next

case \((\text{H} \ n \ s)\)

hence \(\text{tree-at} \ (it' \ c) \ is = \text{HNode} \ n \ s \ ants\) by simp

from \(\text{c} \in \text{set conclusions} \ )\ this

have \(\text{hyd-\text{at} c} \ is n \ s \in \text{edges}\)

moreover

from \(\text{H} \ (p \in \text{inPorts} \ (\text{iNodeOf} \ \text{tree-at} \ (it' \ c) \ is))\):

have \([\text{simp}]: \ p = \text{plain-ant} \ \text{anyP} \ by \ simp\)

have \(\text{snd} \ (\text{hyd-\text{at} c} \ is n s) = ((c, 0 \# \ is), \ p)\)

by (simp add: \text{hyd-\text{at}-to-def})

ultimately

show \(?thesis\) by (auto simp add: \text{Cons} \ \text{simp del: hyd-hyd-edge-at})

qed

qed

qed

sublocale \text{Pruned-Port-Graph nodes inPorts outPorts vertices nodeOf edges}

proof

fix \(v\)

assume \(v \in \text{vertices}\)
thus $\exists ptv'. \text{path } v v' \land \text{terminal-vertex } v'$

**proof (induct rule: vertices-induct)**

- **case (None c)**
  - hence $\text{terminal-vertex } (c,[])$ by simp
  - with path.intros(I)
  - show ?case by blast

- **next**
  - **case (Some c is)**
    - hence $\text{path } (c, 0 \# is) (c, [])$ (terminal-path-from c is)
      - by (rule path-terminal-path-from)
    - moreover
      - have $\text{terminal-vertex } (c,[])$ using Some(I) by simp
    - ultimately
      - show ?case by blast

qed

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

**sublocale** sol:Solution inPorts outPorts nodeOf hyps nodes vertices labelsIn labelsOut freshenLC renameLCs lcconsts closed subst subst-consts subst-renameLCs anyP vidx inst edges proof

- fix $v_1 p_1 v_2 p_2$
- assume $((v_1, p_1), (v_2, p_2)) \in \text{edges}$
- thus $\text{labeledAtOut } v_1 p_1 = \text{labeledAtIn } v_2 p_2$

**proof (cases rule: edges cases)**

- **case (regular-edge c is)**

  - from $((v_1, p_1), (v_2, p_2)) = \text{edge-at } c$ is
  - have $(v_1, p_1) = \text{edge-from } c$ is using fst-edge-at by (metis fst-conv)
  - hence [simp]: $v_1 = (c, 0 \# is)$ by (simp add: edge-from-def)

  - show ?thesis

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

  - **case Nil**
    - let $?t' = it' c$
    - have $\text{labeledAtOut } v_1 p_1 = \text{subst } (\text{iSubst } ?t') (\text{freshen } (\text{vidx } v_1)) (\text{iOutPort } ?t')$
      - using regular-edge Nil by (simp add: labeledAtOut-def edge-at-def edge-from-def)
    - also have $\text{vidx } v_1 = \text{iAnnot } ?t' \text{' by (simp add: Nil)}$
    - also have $\text{subst } (\text{iSubst } ?t') (\text{freshen } (\text{iAnnot } ?t')) (\text{iOutPort } ?t')) = \text{snd } (\text{fst } (\text{root } (\text{ts } c)))$
      - unfolding iwf-subst-freshen-outPort[OF iwf-it[OF c \in \text{set conclusions}]]
    - also have $... = c$ using $(c \in \text{set conclusions})$ by (simp add: ts-conc)
    - also have $... = \text{labelAtIn } v_2 p_2$
      - using $(c \in \text{set conclusions})$ regular-edge Nil
      - by (simp add: labeledAtIn-def edge-at-def freshen-closed conclusions-closed closed-no-consts)

  - finally show ?thesis.

- **next**

  - **case (snoc is i)**
    - let $?t1 = \text{tree-at } (it' c) (is'@[i])$
    - let $?t2 = \text{tree-at } (it' c)' is'
      - have $\text{labeledAtOut } v_1 p_1 = \text{subst } (\text{iSubst } ?t1) (\text{freshen } (\text{vidx } v_1)) (\text{iOutPort } ?t1)$
        - using regular-edge snoc by (simp add: labeledAtOut-def edge-at-def edge-from-def)
      - also have $\text{vidx } v_1 = \text{iAnnot } ?t1$ using snoc regular-edge(3) by simp
      - also have $\text{subst } (\text{iSubst } ?t1) (\text{freshen } (\text{iAnnot } ?t1)) (\text{iOutPort } ?t1))$
        - $= \text{subst } (\text{iSubst } ?t2) (\text{freshen } (\text{iAnnot } ?t2)) (\text{a-conc } (\text{inPorts }' (\text{iNodeOf } ?t2) \# i))$
        - by (rule iwf-edge-match[OF iwf-it[OF c \in \text{set conclusions}] is \in \text{it-paths } (it' c)|unfolded snoc])
      - also have $\text{iAnnot } ?t2 = \text{vidx } (c, 0 \# is')$ by simp

62
also have subst (iSubst ?t2) (freshen (vidx (c, 0 ≠ i)) (a-conc (inPorts' (iNodeOf ?t2) ! i))) = labelAtIn
v \_2 p \_2
using regular-edge snoc by (simp add: labelAtIn-def edge-at-def)
finally show ?thesis.
qed

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

from \( c \in \text{set conclusions} \) \( \exists \text{it-paths (it' c)} \) \( \langle \text{tree-at (it' c)} \rangle \) is = HNode n s ant
have \( \exists \in \text{hyps-along (it' c)} \) is
by (rule hyps-exist')
from \((v \_1, p \_1), v \_2, p \_2) = \text{hyp-edge-at c is n s}
have \((v \_1, p \_1) = \text{hyp-edge-from c is n s}\) using fst-hyp-edge-at by (metis fst-conv)

hence \( \text{simp add: hyp-edge-from-def} \)

have labelAtOut v \_1 p \_1 = subst (iSubst ?t1) (freshen (vidx v \_1) (labelsOut (iNodeOf ?t1) ?h))
using hyp-edge by (simp add: hyp-edge-at-def hyp-edge-from-def labelAtOut-def)
also have v \_1 = iAnnot ?t1 by simp
also have subst (iSubst ?t1) (freshen (iAnnot ?t1) (labelsOut (iNodeOf ?t1) ?h)) = ?if using ?if ∈ hyps-along (it' c) is
by (rule local.hyp-port-eq[symmetric])
also have ... = subst (iSubst ?t2) (freshen (iAnnot ?t2) anyP) using hyp-edge by simp
also have subst (iSubst ?t2) (freshen (iAnnot ?t2) anyP) = labelAtIn v \_2 p \_2
using hyp-edge by (simp add: labelAtIn-def hyp-edge-at-def hyp-edge-to-def)
finally show ?thesis.
qed

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

sublocale Well-Scoped-Instantiation freshenLC renameLCs lecons closed subst subst-lecons subst-renameLCs anyP inPorts outPorts nodeOf hyps nodes vertices labelsIn labelsOut vidx inst edges local-vars
proof
fix v p var v'
assume valid-in-port (v, p)

hence v \in |vertices| by simp

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

from (valid-in-port (v, p)) \( v = - \)

have (c, is) \in |vertices| and p \in |inPorts (nodeOf (c, is))| by simp-all

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

from p \in |\cdot| obtain i where
assume \( v' \mid \in \text{vertices} \)
then obtain \( c' \mid \text{is'} \mid \text{where} \ v' = (c', \text{is'}) \) by \((\text{cases} \ v', \ \text{auto})\)

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

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

assume \( \text{freshenLC} \ (\text{vidx} \ v) \) \( \var \in \text{subst-consts (inst} \ v') \)
then obtain \( \text{is'} = 0 \# \text{is}'' \) and \( \text{is}'' \in \text{it-paths (it'} \ c') \)
using \( (v' \mid \in \text{vertices}) \)
by \((\text{cases} \ \text{is'}) \) \((\text{auto} \ \text{simp add:} \ v' = -\cdot)\)

\( \text{note} \) \( \text{freshenLC} \ (\text{vidx} \ v) \) \( \var \in \text{substage-consts (inst} \ v') \)
also
have \( \text{subst-consts (inst} \ v') = \text{subst-consts (iSubst (tree-at (it'} \ c') \ 'i'))} \)
by \((\text{simp add:} \ v' = -\cdot; \ \text{is'} = -\cdot)\)
also
from \( \text{is}'' \in \text{it-paths (it'} \ c') \)

have \ldots \subseteq \text{fresh-at-path (it'} \ c') \text{is''} \cup \text{range (freshenLC} \ v\text{-away})
by \((\text{rule} \ \text{globalize-local-consts})\)

finally
have \( \text{freshenLC} \ (\text{vidx} \ v) \) \( \var \in \text{fresh-at-path (it'} \ c') \text{is''} \)
using \( (v \mid \in \text{vertices}) \) by \( \text{auto} \)
then obtain \( \text{is}'' '' \) \( \text{where} \ \text{prefix} \ \text{is}'' '' \text{and} \ \text{freshenLC} \ (\text{vidx} \ v) \) \( \var \in \text{fresh-at (it'} \ c') \text{is''''} \)

\( \text{unfolding} \ \text{fresh-at-path-def} \) \( \text{by} \ \text{auto} \)
then obtain \( i' \text{is}''''' \) \( \text{where} \ \text{prefix} \ (\text{is}''''(\hat{i'})) \text{is''''} \)
and \( \text{freshenLC} \ (\text{vidx} \ v) \) \( \var \in \text{fresh-at (it'} \ c') \) \( \text{is''''(\hat{i'})} \)
using \( \text{append-ballast-last-id} \) \( \text{where} \ xs = \text{is}''''' \), \text{symmetric} \)
apply \((\text{cases} \ \text{is}''''' = [])\)
apply \((\text{auto} \ \text{simp del:} \ \text{fresh-at-snoc} \ \text{append-ballast-last-id})\)
apply \( \text{metis} \)
done

from \( \text{is}'''' \in \text{it-paths (it'} \ c') \) \( \text{prefix (is}''''(\hat{i'})) \text{is'''} \)
have \( \text{is}''''(\hat{i'}) \in \text{it-paths (it'} \ c') \) \( \text{by} \ (\text{rule} \ \text{it-paths-prefix}) \)

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

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

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

\( \text{unfolding} \ \text{fresh-at-def} \) \( \text{using} \) \( (v \mid \in \text{vertices}) \) \( (v' \mid \in \text{vertices}) \)
apply \((\text{cases} \ \text{is})\)
apply \((\text{auto} \ \text{split:} \ \text{if}-\text{splits} \ \text{simp add:} \ \text{iAnnot}-\text{globalize} \ \text{it-paths-ballast} \ (v = -) \ (v' = -) \ (\text{is} = -) \ \text{simp del:} \ \text{iAnnot}\text{-simp})\)
done

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

from \( \text{(is}''''(\hat{i'})) \in \text{it-paths (it'} \ c') \)

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

using \( \text{uwf-length}\text{-inPorts}[O \text{uwf-it}[O \text{c in set conclusions}]] \)
by \((\text{auto} \ \text{elim!} \ : \ \text{it-paths-Snoc} \text{E} \ \text{simp add:} \ (\text{is} = -) \ (c' = -) \ \text{order.strict}\text{-trans2})\)

have \( \text{nodeOf (c, is)} \in \text{sset nodes} \)

\( \text{unfolding} \) \( (\text{is} = -) \ (c' = -) \ \text{nodeOf}\text{-simp} \)
by (rule iNodeOf-tree-at[OF iwf-it[OF ```(c ∈ set conclusions)] (is'′′′ ∈ it-paths (it' c') [unfolded (c' = -)])

from (i ∈ a-fresh (inPorts' (iNodeOf {it' c'} is'′′′)) ! i')
    (i ∈ a-fresh p) (p = inPorts' (nodeOf (c, is)) ! i)

node-disjoint-fresh-vars[OF
  (nodeOf (c, is) ∈ sset nodes)
  i < length (inPorts' (nodeOf (c, is))); i' < length (inPorts' (nodeOf (c, is))))]

have i' = i by (auto simp add: (is = -) (c' = -))

from (prefix (is'′′′@[i'] !))) is'''

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

from (c ∈ set conclusions) (is'′′′ ∈ it-paths (it' c') (prefix (is @ [i']) is'))

have scope' v p v'
  unfolding (v = -) (v' = -) (c' = -) (is' = -) (i' = -)
  by (auto intro: scope'.intros)

thus v' ∈ scope (v, p) using (valid-in-port (v, p)) by (simp add: in-scope)

qed

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

sublocale tpg:Tasked-Proof-Graph freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP antecedent consequent rules assumptions conclusions vertices nodeOf edges vidx inst

proof
  show set (map Conclusion conclusions) ⊆ nodeOf ` fset vertices

  proof
    { fix c
      assume c ∈ set conclusions
      hence (c, []) ∈ vertices by simp
      hence nodeOf (c, []) ∈ nodeOf ` fset vertices
        unfolding fnmember rep-eq by (rule imageI)
      hence Conclusion c ∈ nodeOf ` fset vertices by simp
    } thus ?thesis by auto
    qed
  qed

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: infinite-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 \Rightarrow nat))
    using finite-imageD infinite-UNIV by blast
  moreover have infinite (range (to-nat::'b \Rightarrow nat))
    using finite-imageD infinite-UNIV by blast
  ultimately have \exists f. bij-btw f (range (to-nat::'a \Rightarrow nat)) (range (to-nat::'b \Rightarrow nat))
    by (meson bij-btw-inv bij-btw-trans bij-enumerate)
  then obtain f where f-def: bij-btw f (range (to-nat::'a \Rightarrow nat)) (range (to-nat::'b \Rightarrow nat)) ..
  then have f-range-trns: f `(range (to-nat::'a \Rightarrow nat)) = range (to-nat::'b \Rightarrow nat)
    unfolding bij-btw-def by simp
  have surj ((from-nat::nat \Rightarrow 'b) \circ f \circ (to-nat::'a \Rightarrow nat))
    proof (rule surj)
      fix a
      obtain b where [simp]: to-nat (a::'b) = b by blast
      hence b \in range (to-nat::'b \Rightarrow nat) by blast
      with f-range-trns have b \in f `(range (to-nat::'a \Rightarrow nat)) by simp
      from imageE [OF this] obtain c where [simp]:f c = b and c \in range (to-nat::'a \Rightarrow nat)
        by auto
      with f-def have [simp]: inv-into (range (to-nat::'a \Rightarrow nat)) f b = c
        by (meson bij-btw-inv inv-into-f-f)
      then obtain d where cd: 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 \circ f \circ to-nat) (((from-nat::nat \Rightarrow 'a) \circ inv-into (range (to-nat::'a \Rightarrow nat)) f \circ (to-nat::'b \Rightarrow nat)) a) = a
        by (clarsimp simp: cd)
    qed
  moreover have inj ((from-nat::nat \Rightarrow 'b) \circ f \circ (to-nat::'a \Rightarrow nat))
    apply (rule injf)
    apply auto
    apply (metis bij-btw-inv-into-left f-def f-inv-into-f f-range-trns from-nat-def image_eqI rangeI to-nat-split)

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

```
datatype ('var,'cname) pform =
  Var 'var::[countable, infinite]
  | Fun (name:'cname) (params: ('var,'cname) pform list)
```

Substitution on and closedness of propositional formulas is straightforward.

```
fun subst :: ('var::[countable, infinite] ⇒ ('var,'cname) pform) ⇒ ('var,'cname) pform ⇒ ('var,'cname) pform
  where subst (Var v) = s v
  | subst (Fun n ps) = Fun n (map (subst s) ps)
```

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

```
interpretation propositional: Abstract-Formulas
  — No need to freshen locally fixed constants
  — also no renaming needed as there are no locally fixed constants
  — closedness and substitution as defined above
  — no substitution and renaming of locally fixed constants
  — most generic formula
  Var undefined

proof
  fix a v a' v'
  from countable-infinite-ex-bij obtain f where bij (f::nat × 'var ⇒ 'var) by blast
  then show (carry (SOME (f. bij f)::nat ⇒ 'var ⇒ 'var)) (a::nat) (v::'var) = carry (SOME (f. bij f) (a'::nat))
    (v::'var) = (a = a' ∧ v = v')
    apply (rule someI2 [where Q=λf. carry f a v = carry 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 ex1; 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 \textit{string}.

datatype \textit{prop-funs} = \textit{and} | \textit{imp} | Const \textit{string}

The rules are introduction and elimination of conjunction and implication.

datatype \textit{prop-rule} = \textit{andI} | \textit{andE} | \textit{impI} | \textit{impE}

definition \textit{prop-rules} :: \textit{prop-rule} stream
  where \textit{prop-rules} = cycle \{ \textit{andI}, \textit{andE}, \textit{impI}, \textit{impE} \}

lemma \textit{iR-prop-rules} \[ \text{simp} \]: \textit{sset prop-rules} = \{ \textit{andI}, \textit{andE}, \textit{impI}, \textit{impE} \}

unfolding \textit{prop-rules-def} by \textit{simp}

Just some short notation.

abbreviation \textit{X} :: (\textit{string},'a) pform
  where \textit{X} \equiv \textit{Var} "X"
abbreviation \textit{Y} :: (\textit{string},'a) pform
  where \textit{Y} \equiv \textit{Var} "Y"

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

fun \textit{consequent} :: \textit{prop-rule} \Rightarrow (\textit{string}, \textit{prop-funs}) pform list
  where \textit{consequent \textit{andI}} = \{ \textit{Fun and [X, Y]} \}
    | \textit{consequent \textit{andE}} = \{ \textit{X, Y} \}
    | \textit{consequent \textit{implI}} = \{ \textit{Fun imp [X, Y]} \}
    | \textit{consequent \textit{implE}} = \{ \textit{Y} \}

fun \textit{antecedent} :: \textit{prop-rule} \Rightarrow ((\textit{string}, \textit{prop-funs}) pform, \textit{string}) antecedent list
  where \textit{antecedent \textit{andI}} = \{ \textit{plain-ant X, plain-ant Y} \}
    | \textit{antecedent \textit{andE}} = \{ \textit{plain-ant (Fun and [X, Y])} \}
    | \textit{antecedent \textit{implI}} = \{ \textit{Antecedent \{X\} Y \{} \}
    | \textit{antecedent \textit{implE}} = \{ \textit{plain-ant (Fun imp [X, Y]), plain-ant X} \}

interpretation propositional: Abstract-Rules
  curry (SOME f. bij f):: nat \Rightarrow \textit{string} \Rightarrow \textit{string}
lambda. id
lambda. {}
closed :: (\textit{string}, \textit{prop-funs}) pform \Rightarrow \textit{bool}
subst
lambda. {}
lambda. id
Var undefined
antecedent

68
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. \bigcup ((\lambda- \{} \cdot \set \text{consequent} xs) = {})
by clarsimp

next

fix i ′ r i a

assume r ∈ sset prop-rules
and ia < length \text{antecedent} r
and i ′ < length \text{antecedent} r

then show a-fresh \text{antecedent} r ! ia \cap a-fresh \text{antecedent} r ! i ′ = {} ∨ ia = i ′
by (cases i ′; auto)

next

fix p

show \{} ∪ \bigcup ((\lambda- \{} \cdot \fset \text{a-hyps p}) \subseteq \text{a-fresh p} by clarsimp

qed

end

8.3 Incredible _Propositional_ Tasks

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

context ND-Rules-Inst begin

lemma eff-NatRuleI :
nat-rule rule c ants

⇒ entail = (Γ ⊢ subst s \text{freshen a c})

⇒ hyps = ((\lambdaant. ((\lambdap. subst s \text{freshen a p}) ∨ \text{a-hyps ant} |∪| Γ ⊢ subst s \text{freshen a (a-conc ant)})) |∈| ants)

⇒ (\lambdaant. ant |∈| ants ⇒ f |∈| Γ ⇒ \text{freshenLC a} \cdot (a-fresh ant) \cap \text{constfs f} = {})

⇒ (\lambdaant. ant |∈| ants ⇒ \text{freshenLC a} \cdot (a-fresh ant) \cap subst-constfs 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 \text{consequent} r

⇒ antec = f-\text{antecedent} r

⇒ natEff-Inst rule c antec
by (metis natEff-Inst.intros)

end

context begin

end
8.3.1 Task 1.1

This is the very first task of the Incredible Proof Machine: \( A \rightarrow A \)

**abbreviation** \( A::\text{(string,prop-funs)}pform \)
  
  **where** \( A \equiv \text{Fun} (\text{Const "A"}) \) [1]

First the task is defined as an Abstract-Task.

**interpretation** task1-1: Abstract-Task
  
  **curry** (SOME f, bij f):: \( \text{nat} \Rightarrow \text{string} \Rightarrow \text{string} \)
  
  **\( \lambda \)- id**
  
  **\( \lambda \)- []**
  
  **closed:: (string, prop-funs) pform \Rightarrow bool**
  
  **subst**
  
  **\( \lambda \)- []**
  
  **\( \lambda \)- id**
  
  **Var undefined**
  
  **antecedent**
  
  **consequent**
  
  **prop-rules**
  
  [A]
  
  [A]

by unfold-locales simp

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

**lemma** task1-1.solved
  
  **unfolding** task1-1.solved-def
  
  **apply** clarsimp
  
  **apply** (rule-tac \( x=[|A|] \) \ in \ exI)
  
  **apply** clarsimp
  
  **apply** (rule-tac \( x=\text{Node} \ (\{|A|\) \vdash A, Axiom} \) \|) \ in \ exI)
  
  **apply** clarsimp
  
  **apply** (rule conjI)
  
  **apply** (rule task1-1.wf)
  
  **apply** (solves clarsimp)
  
  **apply** clarsimp
  
  **apply** (rule task1-1.eff.intros(1))
  
  **apply** (solves simp)
  
  **apply** (solves clarsimp)

by (auto intro: finite.intros)

**print-locale** Vertex-Grph

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

**print-locale** Pre-Port-Grph

**interpretation** task1-1: Pre-Port-Grph task1-1.nodes task1-1.inPorts task1-1.outPorts \( \{|0::\text{nat},1|\} \)
  
  **undefined(0 := Assumption A, 1 := Conclusion A)

  \{((0,\text{Reg} A),(1,\text{plain-ant} A))\}

**print-locale** Instantiation

**interpretation** task1-1: Instantiation
unspecified(\(0 := \text{Assumption A}, 1 := \text{Conclusion A}\))

\(\lambda\). id

\(\lambda\). {}

closed :: (string, prop-funs) pform \(\Rightarrow\) bool

\(\lambda\). {}

\(\lambda\). id

Var undefined

id

undefined

by unfold-locales simp

\textbf{declare} One-nat-def [simp del]

\textbf{lemma} path-one-edge[simp]:

\textbf{task1-1}.path \(v1\ v2\ pth \longleftrightarrow \)

\((v1 = 0 \land v2 = 1 \land pth = [((0,\text{Reg A}),(1,\text{plain-ant A}))]) \lor\)

\(pth = [] \land v1 = v2)\)

\textbf{apply} (cases pth)

\textbf{apply} (auto simp add: task1-1.path-cons-simp')

\textbf{apply} (rename-tac list, case-tac list, auto simp add: task1-1.path-cons-simp')+

done

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

\textbf{interpretation} Tasked-Proof-Graph

carry (SOME \(f\), bij \(f\)) :: nat \(\Rightarrow\) string \(\Rightarrow\) string

\(\lambda\). id

\(\lambda\). {}

closed :: (string, prop-funs) pform \(\Rightarrow\) bool

\(\lambda\). {}

\(\lambda\). id

Var undefined

antecedent

consequent

prop-rules

\([A]\)

\([A]\)

\{\(\theta::\text{nat},1]\}\)

unspecified(\(\theta := \text{Assumption A}, 1 := \text{Conclusion A}\))

\{\((0,\text{Reg A}),(1,\text{plain-ant A}))\}

id

undefined

\textbf{apply} unfold-locales

\textbf{apply} (solves simp)

\textbf{apply} (solves clarsimp)

\textbf{apply} (solves clarsimp)

\textbf{apply} (solves clarsimp)
apply (solves fastforce)
apply (solves fastforce)
apply (solves clarsimp simp add: task1-1.labelAtOut-def task1-1.labelAtIn-def)
apply (solves clarsimp)
apply (solves clarsimp)
done

8.3.2 Task 2.11

This is a slightly more interesting task as it involves both our connectives: \( P \land Q \rightarrow R \Rightarrow P \rightarrow Q \rightarrow R \)

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

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

abbreviation n-andI \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 = (andI, Fun and [X,Y])
  unfolding task2-11.n-rules-def by (simp add: prop-rules-def)
lemma n-andE1 [simp]: n-andE1 = (andE, X)
  unfolding task2-11.n-rules-def One-nat-def by (simp add: prop-rules-def)
lemma n-andE2 [simp]: n-andE2 = (andE, Y)
  unfolding task2-11.n-rules-def One-nat-def by (simp add: prop-rules-def)
lemma n-impl [simp]: n-impl = (impl1, Fun imp [X,Y])
lemma n-impE [simp]: n-impE = (impE, Y)
  proof -
    have n-impE = task2-11.n-rules !! Suc 3 by simp
    also have ... = (impE, Y)
    unfolding task2-11.n-rules-def numeral-3-eq-3 by (simp add: prop-rules-def)
    finally show ?thesis .
  qed

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

lemma xy-update: f = undefined("X":= x, "Y":= y) \implies x = f "X" \land y = f "Y" by force
lemma y-update: f = 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
  carry (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]\}\} \cap Fun imp [A, Fun imp [B, C]].NatRule n-impI)
\{\Node (\{\{Fun imp [Fun and [A,B],C], A\}\} \cap Fun imp [B, C].NatRule n-impI)
\{\Node (\{\{Fun imp [Fun and [A,B],C], A\}\} \cap C.NatRule n-impE)
\{\Node (\{\{Fun imp [Fun and [A,B],C], A\}\} \cap Fun imp [Fun and [A,B], C].Axiom \{\},
\Node (\{\{Fun imp [Fun and [A,B],C], A\}\} \cap Fun imp [Fun and [A,B], C].Axiom \{\},
\Node (\{\{Fun imp [Fun and [A,B],C], A\}\} \cap Fun imp [Fun and [A,B], C].Axiom \{\},
\Node (\{\{Fun imp [Fun and [A,B],C], A\}\} \cap Fun imp [Fun and [A,B], C].Axiom \{\}
\})
\})
\}) in exI)
apply clarsimp
apply (rule conjI)
apply (rule task1-1.wf)
apply (solves \{clarsimp; metis n-impI snth-smap snth-sset\})
apply clarsimp
apply (rule task1-1.eff-NatRuleI [unfolded propositional; freshen-def, simplified]) apply simp-all[4]
apply (rule task2-11.natEff-InstI)
apply (solves simp)
apply (solves simp)
apply (solves simp)
apply (solves simp)
apply (intro conjI; simp; rule xy-update)
apply (solves simp)
apply (solves (fastforce simp: propositional.f-antecedent-def))
apply clarsimp
apply (rule task1-1.wf)
by (rule tfinite.intros; auto)+

interpretation Tasked-Proof-Graph

\textbf{interpretation} Tasked-Proof-Graph

carry (SOME f, bij f) :: nat \Rightarrow \text{string} \Rightarrow \text{string}

\lambda\cdot \text{id}

\lambda\cdot \{\}

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

subst

\lambda\cdot \{\}

\lambda\cdot \text{id}

\text{Var undefined}

\text{antecedent}
consequent
prop-rules
\[ \text{Fun imp} \{ \text{Fun and} \ [A,B], C \} \]
\[ \text{Fun imp} \{ A, \text{Fun imp} \ [B,C] \} \]
task2-11.vertices
task2-11.nodeOf
task2-11.edges
task2-11.vidx
task2-11.inst
by unfold-locales

end
end

8.4 Predicate Formulas

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

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

type-synonym var = nat
type-synonym lcOnst = nat

We support higher order variables, in order to express \( \forall x.\exists P \ x \). 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 v set xs))
| fv (LC v) = {}
| fv (Op n xs) = Union (fv v set xs)
| fv (Quant n v f) = fv f - {v}

definition fresh-for :: var set ⇒ var where
  fresh-for V = (SOME n. n \notin V)

lemma fresh-for-fresh: finite V ⇒ fresh-for V \notin V
unfolding fresh-for-def
apply (rule someI2-ex)
using infinite-nat-iff-unbounded-le
apply auto
done

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

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

definition \( \text{fv-subst1} \) where
\( \text{fv-subst1} \, s = \bigcup (\text{fv} \, \cdot \, \text{snd} \, \cdot \, \text{set} \, \text{s}) \)

lemma \( \text{fv-subst-Nil[simp]}: \text{fv-subst1} \, [] = \{\} \)
  unfolding \( \text{fv-subst1-def} \) by auto

Local constants, separate from free variables.
fun \( \text{lc} :: \text{form} \Rightarrow \text{lc onst set} \) where
\begin{align*}
\text{lc} \, (\text{Var} \, v \, x) &= \text{Union} \, (\text{lc} \, \cdot \, \text{set} \, x) \\
\text{lc} \, (\text{LC} \, c) &= \{ c \} \\
\text{lc} \, (\text{Op} \, n \, x) &= \text{Union} \, (\text{lc} \, \cdot \, \text{set} \, x) \\
\text{lc} \, (\text{Quant} \, n \, v \, f) &= \text{lc} \, f
\end{align*}

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

definition \( \text{lc-subst1} \) where
\( \text{lc-subst1} \, s = \bigcup (\text{lc} \, \cdot \, \text{snd} \, \cdot \, \text{set} \, \text{s}) \)

fun \( \text{lc-subst} :: \text{subst} \Rightarrow \text{lc onst set} \) where
\( \text{lc-subst} \, s = \bigcup (\text{lc-schema} \, \cdot \, \text{snd} \, \cdot \, \text{set} \, \text{s}) \)

fun \( \text{map-lc} :: (\text{lc onst} \Rightarrow \text{lc onst}) \Rightarrow \text{form} \Rightarrow \text{form} \) where
\begin{align*}
\text{map-lc} \, f \, (\text{Var} \, v \, x) &= \text{Var} \, v \, (\text{map} \, (\text{map-lc} \, f) \, x) \\
\text{map-lc} \, f \, (\text{LC} \, n) &= \text{LC} \, (f \, n) \\
\text{map-lc} \, f \, (\text{Op} \, n \, x) &= \text{Op} \, n \, (\text{map} \, (\text{map-lc} \, f) \, x) \\
\text{map-lc} \, f \, (\text{Quant} \, n \, v \, f') &= \text{Quant} \, n \, v \, (\text{map-lc} \, f \, f')
\end{align*}

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

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

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

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

fun \( \text{map-lc-subst} :: (\text{lc onst} \Rightarrow \text{lc onst}) \Rightarrow \text{subst} \Rightarrow \text{subst} \) where
\( \text{map-lc-subst} \, f \, s = \text{map} \, (\text{apsnd} \, (\text{map-lc} \, f)) \, s \)

lemma \( \text{map-lc-noop[simp]}: \text{lc} \, f = \{\} \Rightarrow \text{map-lc} \, p \, f = f \)
  by (induction \( f \)) (auto simp add: \text{map-idI})
In order to make the termination check happy, we define substitution in two stages: One that substitutes only ground terms for variables, and the real one that can substitute schematic terms (or lambda expression, if you want).

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

\[
\begin{align*}
\text{ subst1 } & s \ (\text{Var } v \ []) = (\text{case } \text{map-of} \ s \ v \ \text{of} \ f \ ⇒ f \ | \ \text{None} ⇒ \text{Var } v \ []) \\
\text{ subst1 } & s \ (\text{Var } v \ xs) = \text{Var } v \ xs \\
\text{ subst1 } & s \ (\text{LC } n) = \text{LC } n \\
\text{ subst1 } & s \ (\text{Op } n \ xs) = \text{Op } n \ (\text{map } \text{subst1 } s \ xs) \\
\text{ subst1 } & s \ (\text{Quant } n \ v \ f) = \\
& \quad \text{if } v \in \text{fv-subst1 } s \ \text{then} \\
& \quad \text{let } v' = \text{fresh-for } (\text{fv-subst1 } s) \\
& \quad \text{in } \text{Quant } n \ v' \ (\text{subst1 } ((v, \text{Var } v' \ []) \# s) \ f) \\
& \quad \text{else } \text{Quant } n \ v \ v \ (\text{subst1 } s \ f)
\end{align*}
\]

**lemma subst1-nil [simp] :: subst1 [] f = f**

by (induction []::(var × form) list f rule: subst1.induct)

(auto simp add: map-idI split: option.splits)

**lemma lc-subst1 :: lc (subst1 s f) ⊆ lc f ∪ (lc ' snd ' set s)**

by (induction s f rule: subst1.induct)

(auto split: option.split dest: map-of-SomeD simp add: Let-def)

**lemma map-of-map-apsnd ::**

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

by auto

**lemma 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: subst1.induct)

apply (auto split: option.splits simp add: map-of-map-apsnd Let-def)

apply (subst subst1.simps, auto split: option.splits)[1]

apply (subst subst1.simps, auto split: option.splits)[1]

apply (subst subst1.simps, auto split: option.splits)[1]

apply (subst subst1.simps, auto split: option.splits, simp only: Let-def map-lc.simps)[1]

apply (subst subst1.simps, auto split: option.splits)

done

**fun subst' :: subst ⇒ form ⇒ form where**

\[
\begin{align*}
\text{ subst'} & s \ (\text{Var } v \ xs) = \\
& \quad (\text{case } \text{map-of } s \ v \ \text{of} \ \text{None} ⇒ (\text{Var } v \ (\text{map } (\text{subst'} \ s) \ xs)) \\
& \quad | \ \text{Some } (\text{ps}, \text{rhs}) ⇒ \\
\end{align*}
\]

77
if length ps = length xs
    then subst1 (zip ps (map (subst' s) xs)) rhs
else (Var v (map (subst' s) xs))

| subst' s (LC n) = LC n |
| subst' s (Op n xs) = Op n (map (subst' s) xs) |
| subst' s (Quant n v f) = |
| (if v ∈ fv-subst s then |
| (let v' = fresh-for (fv-subst s) |
| in Quant n v' (subst' ((v,[[],[Var v']]))#s) f) |
| else Quant n v (subst' s f) |

lemma subst'-Nil[simp]: subst' [] f = f
  by (induction f) (auto simp add: map-id subst-def)

lemma le-subst': le (subst' s f) ⊆ le f ∪ le-subst s
  apply (induction s f rule: subst',induct)
  apply (fastforce dest!: set-zip-rightD)+
  done

lemma ran-map-option-comp[simp]:
  ran (map-option f o m) = f·ran m
unfolding comp-def by (rule ran-map-option)

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

lemma fv-subst-map-apsnd-map-le[simp]:
  fv-subst (map (apsnd (map-le p))) s = fv-subst s
unfolding fv-subst-def
by (auto simp add: map-of-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-le-subst[simp]: map-le p (subst' s f) = subst' (map-le-subst p s) (map-le 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'.sims, auto split: option.splits)[1])
apply (solves (subst subst'.sims, auto split: option.splits cong; map-cong)[1])
apply (solves (subst subst'.sims, auto split: option.splits)[1])
apply (solves (subst subst'.sims, auto split: option.splits)[1])
apply (solves (subst subst'.sims, auto split: option.splits only: Let-def map-le substitution)[1])
apply (solves (subst subst'.sims, auto split: option.splits)[1])
apply (solves (subst subst'.sims, auto split: option.splits)[1])
done

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

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

lemma subst-nil[simp]: subst [] f = f
  by auto

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

lemma lc-subst: \(\text{lc}\ (\text{subst}\ s\ f) \subseteq \text{lc}\ f \cup \text{lc-subst}\ s\)
by (auto dest: set-map[OF lc-subst])

lemma lc-subst-map-lc-subst[simp]: \(\text{lc-subst}\ (\text{map-lc-subst}\ p\ s) = p \cdot \text{lc-subst}\ s\)
by force

lemma map-lc-subst[simp]: \(\text{map-lc}\ p\ (\text{subst}\ s\ f) = \text{subst}\ (\text{map-lc-subst}\ p\ s)\ (\text{map-lc}\ p\ f)\)
unfolding subst_simps
by (auto simp add: filter-map intro!: arg-cong[OF filter-cong])

fun closed :: form \Rightarrow bool where
closed f \iff fv f = \{} \land le f = \{}

interpretation predicate: Abstract-Formulas
carry-to-nat :: nat \Rightarrow var \Rightarrow var
map-lc
le
subst
lc-subst
map-lc-subst
Var 0 []
apply unfold-locals
apply (solves fastforce)
apply (solves fastforce)
apply (solves fastforce)
apply (solves fastforce)
apply (solves fastforce)
apply (solves fastforce)
apply (solves fastforce)
apply (metis map-lc-cong)
apply (solves rule lc-subst-map-lc-subst)
apply (solves simp)
apply (solves rule exI[where x = []], simp)
apply (solves rename-tac f, rule-tac x = [(0, ([] f))] in exI, simp)
done

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

8.5 Incredible.Predicate

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

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

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

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

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

Just some short notation.

abbreviation X :: form
  where X ≡ Var 10 []
abbreviation Y :: form
  where Y ≡ Var 11 []
abbreviation x :: form
  where x ≡ Var 9 []
abbreviation t :: form
  where t ≡ Var 13 []
abbreviation P :: form ⇒ form
  where P f ≡ Var 12 [f]
abbreviation Q :: form ⇒ form
  where Q f ≡ Op "Q" [f]
abbreviation imp :: form ⇒ form ⇒ form
  where imp f1 f2 ≡ Op "imp" [f1, f2]
abbreviation ForallX :: form ⇒ form
  where ForallX f ≡ Quant "all" 9 f

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

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

abbreviation allI-input where allI-input ≡ Antecedent {||} (P (LC 0)) {0}
abbreviation impI-input where impI-input ≡ Antecedent {||} 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
  carry to-nat :: nat ⇒ var ⇒ var
  map-λ
  le
  closed
  subst
  le-subst
  map-le-subst
  Var 0 []
  antecedent
  consequent
  prop-rules

proof
  show ∀ xs ∈ sset prop-rules. consequent xs ≠ []
    unfolding prop-rules-def
using consequent.elims by blast

next
show \forall xs : set prop-rules. \bigcup \{ lc \set (consequent xs) \} = {} 
  by auto

next
fix i' \ r \ ia
assume r : set prop-rules
  and ia < length \{ antecedent r \}
  and i' < length \{ antecedent r \}
then show a-fresh (antecedent r ! ia) \cap a-fresh (antecedent r ! i') = {} \lor ia = i'
  by (cases i'; auto)

next
fix r \ p
assume r : set prop-rules
and p : set \{ antecedent r \}
thus \{ le (a-conc p) \bigcup \{ lc \set (a-hyps p) \} \} \subseteq a-fresh p by auto
qed

end

8.6 Incredible_Predicate_Tasks

theory Incredible-Predicate-Tasks

imports
  Incredible-Completeness
  Incredible-Predicate
  HOL-Isabelle.Isabelle

begin

declare One-nat-def [simp del]

context ND-Rules-Inst begin

lemma eff-NatRuleI:
  nat-rule rule c ants
  \implies entail = (\Gamma \vdash \subst s (freshen a c))
  \implies hyps = ((\lambda ant. ((\lambda p. \subst s (freshen a p)) |\cdot| a-hyps ant |\union| \Gamma \vdash \subst s (freshen a (a-conc ant)))) |\cdot| ants)
  \implies (\Lambda \cdot ant. \cdot a \in ants \implies f \in \Gamma \implies freshenLC a \cdot (a-fresh ant) \cap \text{lecons f} = {})
  \implies (\Lambda \cdot ant. \cdot a \in ants \implies freshenLC a \cdot (a-fresh ant) \cap \text{subst-lecons s} = {})
  \implies eff (NatRule rule) entail hyps
  by (drule eff.intros(2)) simp-all
end

context Abstract-Task begin

lemma natEff-InstI:
  rule = \{ r,c \}
  \implies c \in set \{ consequent r \}
  \implies antec = f-antecedent r
  \implies natEff-Inst rule c antec
  by (metis natEff-Inst.intros)
end

context begin

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

First the task is defined as an Abstract-Task.
interpretation task: Abstract-Task
carry to-nat :: nat ⇒ var ⇒ var
map-lc
lc
closed
subst
lc-subst
map-lc-subst
Var 0 []
antecedent
consequent
prop-rules
[]
[ForallX (imp (Q x) (Q x))]
by unfold-locales auto

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

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

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

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

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

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

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

lemma subst1-simps[simp]:
substitution
substitution

lemma subst2-simps[simp]:
substitution
substitution

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

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

declare subst.simps[simp def]

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

apply (rule taskwf)
apply (solves \((\text{auto simp add: stream.set-map task.n-rules-def})[1]\))
apply clarsimp
apply (rule task.eff-NatRuleI)
apply (solves \((\text{rule task.nEff-Inst.intro; simp})\))
apply clarsimp
apply (rule conjI)
apply (solves \((\text{simp})\))
apply (solves \((\text{rule substI1})\))
apply (solves \((\text{simp add: predicate.f-antecedent-def predicate.freshen-def})\))
apply (substit antecedent.set(2))
apply (solves \((\text{simp})\))
apply (solves \((\text{simp})\))
apply (solves \((\text{simp})\))
apply simp

apply (solves \((\text{auto intro: task.wf intro: task.eff.intro(1)})[1]\))
apply (solves \((\text{rule tfinite.intro, simp}+)\))
done

abbreviation vertices where vertices \( \equiv \{ |0::\text{nat},1,2 | \} \)

fun node0 where
node0 n = \[\text{Conclusion} \ (\text{ForallX} \ (\text{imp} \ (Q x) \ (Q x))), \ Rule \ allI, \ Rule \ impI] \ n

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

interpretation task: \( \text{Vertex-Graph task.nodes task.inPorts task.outPorts vertices node0} \).

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

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

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

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

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

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

**interpretation** task: Instanitation
  task.inPorts
  task.outPorts
  nodeOf
  task.hyps
  task.nodes
  task-edges
  vertices
  task.labelsIn
  task.labelsOut
  curry to-nat :: \text{nat} \Rightarrow \text{var} \Rightarrow \text{var}
  map-lc
  lc
  closed
  subst
  lc-subst
  map-lc-subst
  \text{Var} \ 0 \ []
  id
  inst
  by unfold-locals 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 $\leftrightarrow$ (pth $=$ [] $\land$ v $=$ 0)
  by (cases pth) (auto simp add: task.path-cons-simp)
**lemma** no-path-12[simp]: $\neg$ task.path 1 2 pth
  by (cases pth) (auto simp add: task.path-cons-simp)

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

qed

**interpretation** Saturated-Graph

- task.nodes
- task.inPorts
- task.outPorts
- vertices
- nodeOf
- task-edges

by standard

(auto simp add: predicate.f-consequent-def predicate.f-antecedent-def)

**interpretation** Pruned-Port-Graph

- task.nodes
- task.inPorts
- task.outPorts
- vertices
- nodeOf
- task-edges

**proof**

fix v

assume v ∈ vertices

hence ∃ pth. task.path v 0 pth

apply auto

apply (rule exI [where x = [e1]], auto simp add: task.path-cons-simp)

apply (rule exI [where x = [e2,e1]], auto simp add: task.path-cons-simp)

done

moreover

have task.terminal-vertex 0 by auto

ultimately

show ∃ pth v. task.path v v' pth ∧ task.terminal-vertex v' by blast

qed

**interpretation** Well-Shaped-Graph

- task.nodes
- task.inPorts
- task.outPorts
- vertices
- nodeOf task-edges
- task.hyps

..

**interpretation** Solution

- task.inPorts
- task.outPorts
- nodeOf
- task.hyps
- task.nodes
- vertices
- task.labelsIn
- task.labelsOut

- curry to-nat :: nat ⇒ var ⇒ var
- map-le
- le
- closed
- subst
\[\begin{align*}
\text{interpretation} & \quad \text{Proof-Graph} \\
\text{task.nodes} & \quad \text{task.inPorts} \\
\text{task.outPorts} & \quad \text{vertices} \\
\text{nodeOf} & \quad \text{task-edges} \\
\text{task.hyps} & \quad \text{task.labelsIn} \\
\text{task.labelsOut} & \\
\text{curry to-nat} & : \text{nat} \Rightarrow \text{var} \Rightarrow \text{var} \\
\text{map-lc} & \\
\text{lc} & \\
\text{closed} & \\
\text{subst} & \\
\text{lc-subst} & \\
\text{map-lc-subst} & \\
\text{Var 0} & [] \\
\text{id} & \\
\text{inst} & \\
\text{task-edges} & \end{align*}\]

\text{lemma path-20:} \\
\text{assumes task.path 2 0 pth} \\
\text{shows} (1, \text{allI-input}) \in \text{snd} \setminus \text{pth} \\
\text{proof} – \\
\{ \text{fix} v \\
\quad \text{assume task.path v 0 pth} \\
\quad \text{hence} v = 0 \lor v = 1 \lor (1, \text{allI-input}) \in \text{snd} \setminus \text{pth} \\
\quad \text{by (induction v 0::nat pth rule: task.path.induct) auto} \\
\} \\
\text{from this[OF assms]} \\
\text{show } \text{thesis by auto} \\
\text{qed} \\

\text{lemma scope-21:} 2 \in \text{task.scope (1, allI-input)} \\
\text{by (auto intro!: task.scope.intro\$ elim: path-20 simp add: task.outPortsRule-def predicate.f-antecedent-def predicate.f-consequent-def)} \\

\text{interpretation} \quad \text{Scoped-Proof-Graph} \\
\text{curry to-nat} & : \text{nat} \Rightarrow \text{var} \Rightarrow \text{var} \\
\text{map-lc} & \\
\text{lc} & \\
\text{closed} & \\
\text{subst} & \\
\text{lc-subst} & \\
\text{map-lc-subst} & \\
\text{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
carry to-nat :: nat ⇒ var ⇒ var
map-lc
lc
closed
subst
lc-subst
map-lc-subst
Var 0 []
antecedent
consequent
prop-rules
[]
[∀X (imp (Q x) (Q x))]
vertices
nodeOf
task-edges
id
inst

by unfold-locales auto

end

end