The meta theory of the Incredible Proof Machine

Joachim Breitner   Denis Lohner

March 17, 2025

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

Contents

1 Introduction **2**

2 Auxiliary theories **4**
2.1 Entailment 4
2.2 Indexed_FSet 4
2.3 Rose_Tree 5
2.3.1 The rose tree data type 5
2.3.2 The set of paths in a rose tree 6
2.3.3 Indexing into a rose tree 6

3 Abstract formulas, rules and tasks **8**
3.1 Abstract_Formula 8
3.2 Abstract_Rules 9

4 Incredible Proof Graphs **13**
4.1 Incredible_Signatures 13
4.2 Incredible_Deduction 14
4.3 Abstract_Rules_To_Incredible 20

5 Natural Deduction **23**
5.1 Natural_Deduction 23

6 Correctness **25**
6.1 Incredible_Correctness 25

7 Completeness **28**
7.1 Incredible_Trees 28
7.2 Build_Incredible_Tree 34
7.3 Incredible_Completeness 35

8 Instantiations 41
8.1 Propositional_Formulas . . . 41
8.2 Incredible_Propositional . . . 41
8.3 Incredible_Propositional_Tasks . . . 43
8.3.1 Task 1.1 . . . 43
8.3.2 Task 2.11 . . . 45
8.4 Predicate_Formulas . . . 46
8.5 Incredible_Predicate . . . 50
8.6 Incredible_Predicate_Tasks . . . 51

1 Introduction

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

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

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

This formalization covers the following aspects:

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

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

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

Acknowledgements

We would like to thank Andreas Lochbihler for helpful comments.

References

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

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

2 Auxiliary theories

2.1 Entailment

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

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

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

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

end

2.2 Indexed_FSet

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

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

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

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

lemma *fset-from-list-of-fset*[*simp*]: *fset-from-list* (*list-of-fset s*) = *s*
  and *distinct-list-of-fset*[*simp*]: *distinct* (*list-of-fset s*)
  ⟨*proof*⟩

lemma *length-list-of-fset*[*simp*]: *length* (*list-of-fset s*) = *size s*
  ⟨*proof*⟩

lemma *nth-list-of-fset-mem*[*simp*]: *i* < *size s* ⟹ *list-of-fset s* ! *i* |∈| *s*
  ⟨*proof*⟩

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

lemma *indexed-fmember-is-fmember*: *x* $|\in|_i$ *s* ⟹ *x* |∈| *s*
⟨*proof*⟩

lemma *fmember-is-indexed-fmember*:
assumes x |∈| s
shows $\exists\, i.\ x$ $|\in|_i$ s
⟨*proof*⟩

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

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

lemma *mem-set-indexed-members*:
(*i*,*x*) ∈ *set* (*indexed-members s*) ⟷ $x\ |\in|_i\ s$
⟨*proof*⟩

lemma *mem-set-indexed-members'*[*simp*]:
t ∈ *set* (*indexed-members s*) ⟷ *snd t* $|\in|_{fst\ t}$ *s*
⟨*proof*⟩

definition *fnth* (**infixl** ‹|!|› *100*) **where**
s |!| *n* = *list-of-fset s* ! *n*
lemma *fnth-indexed-fmember*: *i* < *size s* ⟹ *s* |!| *i* $|\in|_i$ *s*
⟨*proof*⟩
lemma *indexed-fmember-fnth*: $x\ |\in|_i\ s$ ⟷ (*s* |!| *i* = *x* ∧ *i* < *size s*)
⟨*proof*⟩
end

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

lemma *fidx-eq*[*simp*]: $x\ |\in|_i\ s$ ⟹ *fidx s x* = *i*
⟨*proof*⟩

lemma *fidx-inj*[*simp*]: *x* |∈| *s* ⟹ *y* |∈| *s* ⟹ *fidx s x* = *fidx s y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *inj-on-fidx*: *inj-on* (*fidx vertices*) (*fset vertices*)
⟨*proof*⟩

end

2.3 Rose_Tree

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

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

2.3.1 The rose tree data type

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

2.3.2 The set of paths in a rose tree

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

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

inductive-cases *it-pathP-ConsE*: *it-pathsP t* (*i#is*)

inductive-cases *it-pathP-RNodeE*: *it-pathsP* (*RNode r ants*) *is*

definition *it-paths*:: *'a rose-tree* ⇒ *nat list set* **where**
it-paths t = *Collect* (*it-pathsP t*)

lemma *it-paths-eq* [*pred-set-conv*]: *it-pathsP t* = ($\lambda x.\ x \in$ *it-paths t*)
⟨*proof*⟩

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

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

lemma *it-paths-RNode-Nil*[*simp*]: *it-paths* (*RNode r* []) = {[]}
⟨*proof*⟩

lemma *it-paths-Union*: *it-paths t* ⊆ *insert* [] (*Union* (((λ (*i*,*t*). ((#) *i*) ‘ *it-paths t*) ‘ *set* (*List.enumerate* (*0*::*nat*) (*children t*)))))
⟨*proof*⟩

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

2.3.3 Indexing into a rose tree

fun *tree-at* :: *'a rose-tree* ⇒ *nat list* ⇒ *'a rose-tree* **where**
tree-at t [] = *t*
| *tree-at t* (*i#is*) = *tree-at* (*children t* ! *i*) *is*

lemma *it-paths-SnocE*[*elim-format*]:
assumes *is* @ [*i*] ∈ *it-paths t*
shows *is* ∈ *it-paths t* ∧ $i < length$ (*children* (*tree-at t is*))
⟨*proof*⟩

lemma *it-paths-strict-prefix*:
assumes *is* ∈ *it-paths t*
assumes *strict-prefix is' is*
shows *is'* ∈ *it-paths t*
⟨*proof*⟩

lemma *it-paths-prefix*:
assumes *is* ∈ *it-paths t*
assumes *prefix is' is*

shows *is'* ∈ *it-paths t*
⟨*proof*⟩

lemma *it-paths-butlast*:
assumes *is* ∈ *it-paths t*
shows *butlast is* ∈ *it-paths t*
⟨*proof*⟩

lemma *it-path-SnocI*:
assumes *is* ∈ *it-paths t*
assumes *i* < *length* (*children* (*tree-at t is*))
shows *is* @ [*i*] ∈ *it-paths t*
⟨*proof*⟩

end

3 Abstract formulas, rules and tasks

3.1 Abstract_Formula

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

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

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

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

assumes *freshenLC-eq-iff* [*simp*]: $freshenLC\ a\ v = freshenLC\ a'\ v' \longleftrightarrow a = a' \wedge v = v'$
assumes *lconsts-renameLCs*: $lconsts\ (renameLCs\ p\ f) = p\ `\ lconsts\ f$
assumes *rename-closed*: $lconsts\ f = \{\} \Longrightarrow renameLCs\ p\ f = f$
assumes *subst-closed*: $closed\ f \Longrightarrow subst\ s\ f = f$
assumes *closed-no-lconsts*: $closed\ f \Longrightarrow lconsts\ f = \{\}$
assumes *fv-subst*: $lconsts\ (subst\ s\ f) \subseteq lconsts\ f \cup subst\text{-}lconsts\ s$
assumes *rename-rename*: $renameLCs\ p1\ (renameLCs\ p2\ f) = renameLCs\ (p1 \circ p2)\ f$
assumes *rename-subst*: $renameLCs\ p\ (subst\ s\ f) = subst\ (subst\text{-}renameLCs\ p\ s)\ (renameLCs\ p\ f)$
assumes *renameLCs-cong*: $(\bigwedge x.\ x \in lconsts\ f \Longrightarrow f1\ x = f2\ x) \Longrightarrow renameLCs\ f1\ f = renameLCs\ f2\ f$
assumes *subst-renameLCs-cong*: $(\bigwedge x.\ x \in subst\text{-}lconsts\ s \Longrightarrow f1\ x = f2\ x) \Longrightarrow subst\text{-}renameLCs\ f1\ s = subst\text{-}renameLCs\ f2\ s$
assumes *subst-lconsts-subst-renameLCs*: $subst\text{-}lconsts\ (subst\text{-}renameLCs\ p\ s) = p\ `\ subst\text{-}lconsts\ s$
assumes *lconsts-anyP*: $lconsts\ anyP = \{\}$
assumes *empty-subst*: $\exists\ s.\ (\forall\ f.\ subst\ s\ f = f) \wedge subst\text{-}lconsts\ s = \{\}$
assumes *anyP-is-any*: $\exists\ s.\ subst\ s\ anyP = f$
begin
definition $freshen :: nat \Rightarrow 'form \Rightarrow 'form$ **where**
$freshen\ n = renameLCs\ (freshenLC\ n)$

definition $empty\text{-}subst :: 'subst$ **where**

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

lemma *empty-subst-spec*:
(∀ *f. subst empty-subst f* = *f*) ∧ *subst-lconsts empty-subst* = {}
⟨*proof*⟩
lemma *subst-empty-subst*[*simp*]: *subst empty-subst f* = *f*
⟨*proof*⟩
lemma *subst-lconsts-empty-subst*[*simp*]: *subst-lconsts empty-subst* = {}
⟨*proof*⟩

lemma *lconsts-freshen*: *lconsts* (*freshen a f*) = *freshenLC a* ' *lconsts f*
⟨*proof*⟩

lemma *freshen-closed*: *lconsts f* = {} ⟹ *freshen a f* = *f*
⟨*proof*⟩

lemma *closed-eq*:
assumes *closed f1*
assumes *closed f2*
shows *subst s1* (*freshen a1 f1*) = *subst s2* (*freshen a2 f2*) ⟷ *f1* = *f2*
⟨*proof*⟩

lemma *freshenLC-range-eq-iff*[*simp*]: *freshenLC a v* ∈ *range* (*freshenLC a'*) ⟷ *a* = *a'*
⟨*proof*⟩

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

lemma *inj-freshenLC*[*simp*]: *inj* (*freshenLC i*)
⟨*proof*⟩

lemma *rerename-freshen*[*simp*]: *x* ∈ *V* ⟹ *rerename V i* (*isidx is*) *f* (*freshenLC i x*) = *freshenLC* (*isidx is*) *x*
⟨*proof*⟩

lemma *range-rerename*: *range* (*rerename V from to f*) ⊆ *freshenLC to* ' *V* ∪ *range f*
⟨*proof*⟩

lemma *rerename-noop*:
x ∉ *freshenLC from* ' *V* ⟹ *rerename V from to f x* = *f x*
⟨*proof*⟩

lemma *rerename-rename-noop*:
freshenLC from ' *V* ∩ *lconsts form* = {} ⟹ *renameLCs* (*rerename V from to f*) *form* = *renameLCs f form*
⟨*proof*⟩

lemma *rerename-subst-noop*:
freshenLC from ' *V* ∩ *subst-lconsts s* = {} ⟹ *subst-renameLCs* (*rerename V from to f*) *s* = *subst-renameLCs f s*
⟨*proof*⟩
end
end

3.2 Abstract_Rules

theory *Abstract-Rules*

imports
Abstract-Formula
begin

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

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

Each antecedent of a rule consists of

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

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

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

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

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

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

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

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

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

  assumes no-local-consts-in-consequences: ∀ xs∈sset rules. ⋃(lconsts ` (set (consequent xs))) = {}
  assumes no-multiple-local-consts:
    ⋀ r i i' . r ∈ sset rules ⟹
               i < length (antecedent r) ⟹
               i' < length (antecedent r) ⟹
               a-fresh (antecedent r ! i) ∩ a-fresh (antecedent r ! i') = {} ∨ i = i'
  assumes all-local-consts-listed:
    ⋀ r p. r ∈ sset rules ⟹ p ∈ set (antecedent r) ⟹
```

$lconsts\ (a\text{-}conc\ p) \cup (\bigcup (lconsts\ `\ fset\ (a\text{-}hyps\ p))) \subseteq a\text{-}fresh\ p$
begin
definition *f-antecedent* :: *'rule* ⇒ (*'form*, *'var*) *antecedent fset*
where *f-antecedent r* = *fset-from-list* (*antecedent r*)
definition *f-consequent r* = *fset-from-list* (*consequent r*)
end

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

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

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

fixes *assumptions* :: *'form list*
fixes *conclusions* :: *'form list*
assumes *assumptions-closed*: $\bigwedge a.\ a \in set\ assumptions \Longrightarrow closed\ a$
assumes *conclusions-closed*: $\bigwedge c.\ c \in set\ conclusions \Longrightarrow closed\ c$
begin
definition *ass-forms* **where** *ass-forms* = *fset-from-list assumptions*
definition *conc-forms* **where** *conc-forms* = *fset-from-list conclusions*

lemma *mem-ass-forms*[*simp*]: *a* |∈| *ass-forms* ⟷ *a* ∈ *set assumptions*
⟨*proof*⟩

lemma *mem-conc-forms*[*simp*]: *a* |∈| *conc-forms* ⟷ *a* ∈ *set conclusions*
⟨*proof*⟩

lemma *subst-freshen-assumptions*[*simp*]:
assumes *pf* ∈ *set assumptions*
shows *subst s* (*freshen a pf*) = *pf*
⟨*proof*⟩

lemma *subst-freshen-conclusions*[*simp*]:
assumes *pf* ∈ *set conclusions*
shows *subst s* (*freshen a pf*) = *pf*
⟨*proof*⟩

lemma *subst-freshen-in-ass-formsI*:
assumes *pf* ∈ *set assumptions*
shows *subst s* (*freshen a pf*) |∈| *ass-forms*
⟨*proof*⟩

lemma *subst-freshen-in-conc-formsI*:
assumes *pf* ∈ *set conclusions*

shows *subst s* (*freshen a pf*) |∈| *conc-forms*
⟨*proof*⟩
end

end

4 Incredible Proof Graphs

4.1 Incredible_Signatures

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

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

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

locale *Port-Graph-Signature* =
fixes *nodes* :: *'node stream*
fixes *inPorts* :: *'node* ⇒ *'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*)
⟨*proof*⟩

context includes *fset.lifting*
begin
lift-definition *hyps-for* :: *'node* ⇒ *'inPort* ⇒ *'outPort fset* **is** *hyps-for'*
⟨*proof*⟩
lemma *hyps-for-simp*[*simp*]: *h* |∈| *hyps-for n p* ⟷ *hyps n h* = *Some p*
⟨*proof*⟩
lemma *hyps-for-simp'*[*simp*]: *h* ∈ *fset* (*hyps-for n p*) ⟷ *hyps n h* = *Some p*
⟨*proof*⟩
lemma *hyps-for-collect*: *fset* (*hyps-for n p*) = {*h* . *hyps n h* = *Some p*}
⟨*proof*⟩
end
lemma *hyps-for-subset*: *hyps-for n p* |⊆| *outPorts n*
⟨*proof*⟩
end

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

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

fixes *local-vars* :: *'node* ⇒ *'inPort* ⇒ *'var set*

end

4.2 Incredible_Deduction

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

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

The following locale first defines graphs, without edges.

locale *Vertex-Graph* =
Port-Graph-Signature nodes inPorts outPorts
for *nodes* :: *'node stream*
and *inPorts* :: *'node* ⇒ *'inPort fset*
and *outPorts* :: *'node* ⇒ *'outPort fset* +
fixes *vertices* :: *'v fset*
fixes *nodeOf* :: *'v* ⇒ *'node*
begin
fun *valid-out-port* **where** *valid-out-port* (*v*,*p*) ⟷ *v* |∈| *vertices* ∧ *p* |∈| *outPorts* (*nodeOf v*)
fun *valid-in-port* **where** *valid-in-port* (*v*,*p*) ⟷ *v* |∈| *vertices* ∧ *p* |∈| *inPorts* (*nodeOf v*)

fun *terminal-node* **where**
terminal-node n ⟷ *outPorts n* = {||}
fun *terminal-vertex* **where**
terminal-vertex v ⟷ *v* |∈| *vertices* ∧ *terminal-node* (*nodeOf v*)
end

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

type-synonym (*'v*, *'outPort*, *'inPort*) *edge* = ((*'v* × *'outPort*) × (*'v* × *'inPort*))

locale *Pre-Port-Graph* =
Vertex-Graph nodes inPorts outPorts vertices nodeOf
for *nodes* :: *'node stream*

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

lemma *edge-begin-tup*: *edge-begin x* = *fst* (*fst x*) ⟨*proof*⟩
lemma *edge-end-tup*: *edge-end x* = *fst* (*snd x*) ⟨*proof*⟩

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

inductive-simps *path-cons-simp'*: *path v v'* (*e*#*pth*)
inductive-simps *path-empty-simp*[*simp*]: *path v v'* []
lemma *path-cons-simp*: *path v v'* (*e* # *pth*) ⟷ *fst* (*fst e*) = *v* ∧ *e* ∈ *edges* ∧ *path* (*fst* (*snd e*)) *v'* *pth*
⟨*proof*⟩

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

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

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

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

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

lemma *scope-find*:
assumes *v* ∈ *scope ps*
assumes *terminal-vertex v'*
assumes *path v v' pth*
shows *ps* ∈ *snd* ‘ *set pth*
⟨*proof*⟩

lemma *snd-set-split*:
assumes *ps* ∈ *snd* ‘ *set pth*
obtains *pth1 pth2 e* **where** *pth* = *pth1*@[*e*]@*pth2* **and** *snd e* = *ps* **and** *ps* ∉ *snd* ‘ *set pth1*
⟨*proof*⟩

lemma *scope-split*:
assumes *v* ∈ *scope ps*
assumes *path v v' pth*
assumes *terminal-vertex v'*

obtains *pth1 e pth2*
where $pth = (pth1@[e])@pth2$ **and** *path v (fst ps) (pth1@[e])* **and** *path (fst ps) v' pth2* **and** $snd\ e = ps$ **and** $ps \notin snd$ ‘ *set pth1*
⟨*proof*⟩
end

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

locale *Port-Graph* = *Pre-Port-Graph* +
assumes *valid-nodes*: *nodeOf* ‘ *fset vertices* ⊆ *sset nodes*
assumes *valid-edges*: ∀ (*ps1*,*ps2*) ∈ *edges*. *valid-out-port ps1* ∧ *valid-in-port ps2*
begin
lemma *snd-set-path-verties*: *path v v′ pth* ⟹ *fst* ‘ *snd* ‘ *set pth* ⊆ *fset vertices*
⟨*proof*⟩

lemma *fst-set-path-verties*: *path v v′ pth* ⟹ *fst* ‘ *fst* ‘ *set pth* ⊆ *fset vertices*
⟨*proof*⟩
end

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

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

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

lemma *scopes-nest*:
fixes *ps1 ps2*
shows *scope ps1* ⊆ *scope ps2* ∨ *scope ps2* ⊆ *scope ps1* ∨ *scope ps1* ∩ *scope ps2* = {}
⟨*proof*⟩
end

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

locale *Scoped-Graph* = *Port-Graph* + *Port-Graph-Signature-Scoped*
locale *Well-Scoped-Graph* = *Scoped-Graph* +
assumes *well-scoped*: $((v_1,p_1),(v_2,p_2)) \in edges \Longrightarrow hyps\ (nodeOf\ v_1)\ p_1 = Some\ p' \Longrightarrow (v_2,p_2) = (v_1,p') \lor v_2 \in scope\ (v_1,p')$

context *Scoped-Graph*
begin

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

lemma *hyps-free-Nil*[*simp*]: *hyps-free* [] ⟨*proof*⟩

lemma *hyps-free-Cons*[*simp*]: *hyps-free* (*e#pth*) ⟷ *hyps-free pth* ∧ *hyps* (*nodeOf* (*fst* (*fst e*))) (*snd* (*fst e*)) = *None*
⟨*proof*⟩

lemma *path-vertices-shift*:
assumes *path v v′ pth*
shows *map fst* (*map fst pth*)@[v'] = *v*#*map fst* (*map snd pth*)
⟨*proof*⟩

inductive *terminal-path* **where**
terminal-path-empty: *terminal-vertex v* ⟹ *terminal-path v v* [] |
terminal-path-cons: ((v_1,p_1),(v_2,p_2)) ∈ *edges* ⟹ *terminal-path* v_2 v' *pth* ⟹ *hyps* (*nodeOf* v_1) p_1 = *None* ⟹ *terminal-path* v_1 v' (((v_1,p_1),(v_2,p_2))#*pth*)

lemma *terminal-path-is-path*:
assumes *terminal-path v v′ pth*
shows *path v v′ pth*
⟨*proof*⟩

lemma *terminal-path-is-hyps-free*:
assumes *terminal-path v v′ pth*
shows *hyps-free pth*
⟨*proof*⟩

lemma *terminal-path-end-is-terminal*:
assumes *terminal-path v v′ pth*
shows *terminal-vertex v′*
⟨*proof*⟩

lemma *terminal-pathI*:
assumes *path v v′ pth*
assumes *hyps-free pth*
assumes *terminal-vertex v′*
shows *terminal-path v v′ pth*
⟨*proof*⟩
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'])
⟨*proof*⟩

lemma *hyps-free-vertices-distinct′*:
assumes *terminal-path v v′ pth*
shows *distinct* (*v* # *map fst* (*map snd pth*))
⟨*proof*⟩

lemma *hyps-free-limited*:
assumes *terminal-path v v′ pth*
shows *length pth* ≤ *fcard vertices*
⟨*proof*⟩

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

shows $v' \notin$ *scope* (v, p)
⟨*proof*⟩

end

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

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

These four conditions make up a well-shaped graph.

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

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

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

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

locale *Solution* =
  Instantiation - - - - - edges **for** *edges* :: ((*'vertex* × *'outPort*) × *'vertex* × *'inPort*) *set* +
  assumes *solved*: $((v_1,p_1),(v_2,p_2)) \in$ *edges* $\Longrightarrow$ *labelAtOut* v_1 p_1 = *labelAtIn* v_2 p_2

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

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

locale *Well-Scoped-Instantiation* =
  Pre-Port-Graph nodes inPorts outPorts vertices nodeOf edges +
  Instantiation inPorts outPorts nodeOf hyps nodes edges vertices labelsIn labelsOut freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP vidx inst +

Port-Graph-Signature-Scoped-Vars nodes inPorts outPorts freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP local-vars
for *freshenLC* :: *nat* ⇒ *'var* ⇒ *'var*
and *renameLCs* :: (*'var* ⇒ *'var*) ⇒ *'form* ⇒ *'form*
and *lconsts* :: *'form* ⇒ *'var set*
and *closed* :: *'form* ⇒ *bool*
and *subst* :: *'subst* ⇒ *'form* ⇒ *'form*
and *subst-lconsts* :: *'subst* ⇒ *'var set*
and *subst-renameLCs* :: (*'var* ⇒ *'var*) ⇒ (*'subst* ⇒ *'subst*)
and *anyP* :: *'form*
and *inPorts* :: *'node* ⇒ *'inPort fset*
and *outPorts* :: *'node* ⇒ *'outPort fset*
and *nodeOf* :: *'vertex* ⇒ *'node*
and *hyps* :: *'node* ⇒ *'outPort* ⇒ *'inPort option*
and *nodes* :: *'node stream*
and *vertices* :: *'vertex fset*
and *labelsIn* :: *'node* ⇒ *'inPort* ⇒ *'form*
and *labelsOut* :: *'node* ⇒ *'outPort* ⇒ *'form*
and *vidx* :: *'vertex* ⇒ *nat*
and *inst* :: *'vertex* ⇒ *'subst*
and *edges* :: (*'vertex*, *'outPort*, *'inPort*) *edge set*
and *local-vars* :: *'node* ⇒ *'inPort* ⇒ *'var set* +
assumes *well-scoped-inst*:
valid-in-port (*v*,*p*) ⟹
var ∈ *local-vars* (*nodeOf v*) *p* ⟹
v' |∈| *vertices* ⟹
freshenLC (*vidx v*) *var* ∈ *subst-lconsts* (*inst v'*) ⟹
v' ∈ *scope* (*v*,*p*)
begin
lemma *out-of-scope*: *valid-in-port* (*v*,*p*) ⟹ *v'* |∈| *vertices* ⟹ *v'* ∉ *scope* (*v*,*p*) ⟹ *freshenLC* (*vidx v*) ‘ *local-vars* (*nodeOf v*) *p* ∩ *subst-lconsts* (*inst v'*) = {}
⟨*proof*⟩
end

The following locale assembles all these conditions.

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

and *labelsIn* :: *'node* ⇒ *'inPort* ⇒ *'form*
and *labelsOut* :: *'node* ⇒ *'outPort* ⇒ *'form*
and *vidx* :: *'vertex* ⇒ *nat*
and *inst* :: *'vertex* ⇒ *'subst*
and *edges* :: (*'vertex*, *'outPort*, *'inPort*) *edge set*
and *local-vars* :: *'node* ⇒ *'inPort* ⇒ *'var set*

end

4.3 Abstract_Rules_To_Incredible

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

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

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

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

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

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

lemma *Helper-in-nodes*[*simp*]:
Helper ∈ *sset nodes* ⟨*proof*⟩
lemma *Assumption-in-nodes*[*simp*]:
Assumption a ∈ *sset nodes* ⟷ *a* ∈ *set assumptions* ⟨*proof*⟩
lemma *Conclusion-in-nodes*[*simp*]:
Conclusion c ∈ *sset nodes* ⟷ *c* ∈ *set conclusions* ⟨*proof*⟩
lemma *Rule-in-nodes*[*simp*]:
Rule r ∈ *sset nodes* ⟷ *r* ∈ *sset rules* ⟨*proof*⟩

fun *inPorts'* :: (*'form*, *'rule*) *graph-node* ⇒ (*'form*, *'var*) *in-port list* **where**
inPorts' (*Rule r*) = *antecedent r*
|*inPorts'* (*Assumption r*) = []
|*inPorts'* (*Conclusion r*) = [*plain-ant r*]
|*inPorts' Helper* = [*plain-ant anyP*]

fun *inPorts* :: (*'form*, *'rule*) *graph-node* ⇒ (*'form*, *'var*) *in-port fset* **where**
inPorts (*Rule r*) = *f-antecedent r*
|*inPorts* (*Assumption r*) = {||}

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

lemma *inPorts-fset-of*:
inPorts n = *fset-from-list* (*inPorts′ n*)
⟨*proof*⟩

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

lemma *Reg-in-outPortsRule*[*simp*]: *Reg c* |∈| *outPortsRule r* ⟷ *c* |∈| *f-consequent r*
⟨*proof*⟩
lemma *Hyp-in-outPortsRule*[*simp*]: *Hyp h c* |∈| *outPortsRule r* ⟷ *c* |∈| *f-antecedent r* ∧ *h* |∈| *a-hyps c*
⟨*proof*⟩

fun *outPorts* **where**
outPorts (*Rule r*) = *outPortsRule r*
|*outPorts* (*Assumption r*) = {|*Reg r*|}
|*outPorts* (*Conclusion r*) = {||}
|*outPorts Helper* = {| *Reg anyP* |}

fun *labelsIn* **where**
labelsIn - *p* = *a-conc p*

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

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

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

sublocale *Labeled-Signature nodes inPorts outPorts hyps labelsIn labelsOut*
⟨*proof*⟩

lemma *hyps-for-conclusion*[*simp*]: *hyps-for* (*Conclusion n*) *p* = {||}
⟨*proof*⟩
lemma *hyps-for-Helper*[*simp*]: *hyps-for Helper p* = {||}
⟨*proof*⟩
lemma *hyps-for-Rule*[*simp*]: *ip* |∈| *f-antecedent r* ⟹ *hyps-for* (*Rule r*) *ip* = (λ *h*. *Hyp h ip*) |'| *a-hyps ip*
⟨*proof*⟩
end

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

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

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

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

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

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

end

5 Natural Deduction

5.1 Natural_Deduction

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

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

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

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

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

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

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

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

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

inductive *eff* :: *'rule NatRule* ⇒ *'form entailment* ⇒ *'form entailment fset* ⇒ *bool* **where**

con |∈| Γ
⟹ *eff Axiom* (Γ ⊢ *con*) {||}
|*nat-rule rule c ants*
⟹ (⋀ *ant f*. *ant* |∈| *ants* ⟹ *f* |∈| Γ ⟹ *freshenLC a* ‘ (*a-fresh ant*) ∩ *lconsts f* = {})
⟹ (⋀ *ant*. *ant* |∈| *ants* ⟹ *freshenLC a* ‘ (*a-fresh ant*) ∩ *subst-lconsts s* = {})
⟹ *eff* (*NatRule rule*)
(Γ ⊢ *subst s* (*freshen a c*))
((λ*ant*. ((λ*p*. *subst s* (*freshen a p*)) |‘| *a-hyps ant* |∪| Γ ⊢ *subst s* (*freshen a* (*a-conc ant*)))) |‘| *ants*)
|*eff Cut* (Γ ⊢ *c′*) {| (Γ ⊢ *c′*)|}

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

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

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

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

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

sublocale *ND-Rules-Inst* - - - - - - - - *natEff-Inst n-rules* ⟨*proof*⟩

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

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

end

6 Correctness

6.1 Incredible_Correctness

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

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

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*⟩

context includes *fset.lifting*
begin
lift-definition *global-assms* :: *'var itself* ⇒ *'form fset* **is** *global-assms'* ⟨*proof*⟩
lemmas *global-assmsI* = *global-assms'.intros*[*Transfer.transferred*]
lemmas *global-assms-simps* = *global-assms'.simps*[*Transfer.transferred*]
end

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

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

lemma *hyps-alongE*[*consumes 1*, *case-names Hyp Assumption*]:
assumes *f* |∈| *hyps-along pth*
obtains *v p h* **where** (*v*,*p*) ∈ *snd* ‘ *set pth* **and** *f* = *labelAtOut v h* **and** *h* |∈| *hyps-for* (*nodeOf v*) *p*
| *v pf* **where** *v* |∈| *vertices* **and** *nodeOf v* = *Assumption pf f* = *labelAtOut v* (*Reg pf*)
⟨*proof*⟩

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

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

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

lemma *out-port-cases*[*consumes 1*, *case-names Assumption Hyp Rule Helper*]:
assumes *p* |∈| *outPorts n*
obtains
a **where** *n* = *Assumption a* **and** *p* = *Reg a*
| *r h c* **where** *n* = *Rule r* **and** *p* = *Hyp h c*
| *r f* **where** *n* = *Rule r* **and** *p* = *Reg f*
| *n* = *Helper* **and** *p* = *Reg anyP*
⟨*proof*⟩

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

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

theorem *wf-tree*:
assumes *valid-in-port* (*v*,*p*)
assumes *terminal-path v t pth*
shows *wf* (*tree v p pth*)
⟨*proof*⟩

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

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

lemma *tfinite-map-tree*: *tfinite* (*map-tree f t*) ⟷ *tfinite t*
⟨*proof*⟩

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

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

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

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

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

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

theorem *finite-tree*:
assumes *valid-in-port* (*v*,*p*)
assumes *terminal-vertex* *v*
shows *tfinite* (*tree* *v* *p* *pth*)
⟨*proof*⟩

The main result of this theory.

theorem *solved*
⟨*proof*⟩

end

end

7 Completeness

7.1 Incredible_Trees

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

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

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

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

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

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

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

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

fun *iAnnot* **where** *iAnnot it* = *iAnnot′* (*root it*)
fun *iSubst* **where** *iSubst it* = *iSubst′* (*root it*)
fun *iAnts* **where** *iAnts it* = *children it*

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

context *Abstract-Task*
begin

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

inductive *iwf* :: ('*form*, '*rule*, '*subst*) *fresh-check* ⇒ ('*form*,'*rule*,'*subst*,'*var*) *itree* ⇒ '*form entailment* ⇒ *bool*

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

lemma *iwf-subst-freshen-outPort*:
iwf lc ts ent ⟹
snd ent = *subst* (*iSubst ts*) (*freshen* (*iAnnot ts*) (*iOutPort ts*))
⟨*proof*⟩

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

lemma *all-local-vars-Helper*[*simp*]:
all-local-vars Helper = {}
⟨*proof*⟩

lemma *all-local-vars-Assumption*[*simp*]:
all-local-vars (*Assumption c*) = {}
⟨*proof*⟩

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

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

abbreviation *local-iwf* ≡ *iwf local-fresh-check*

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

inductive *no-fresh-check* :: (′*form*, ′*rule*, ′*subst*) *fresh-check* **where**
no-fresh-check n i s (Γ ⊢ *c*)

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

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

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

$= subst\ (iSubst\ (tree\text{-}at\ t\ is))\ (freshen\ (iAnnot\ (tree\text{-}at\ t\ is))\ (a\text{-}conc\ (inPorts'\ (iNodeOf\ (tree\text{-}at\ t\ is))\ !\ i)))$
⟨*proof*⟩

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

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

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

lemma *iNodeOf-outPorts*:
$iwf\ fc\ t\ ent \Longrightarrow is \in it\text{-}paths\ t \Longrightarrow outPorts\ (iNodeOf\ (tree\text{-}at\ t\ is)) = \{||\} \Longrightarrow False$
⟨*proof*⟩

lemma *iNodeOf-tree-at*:
$iwf\ fc\ t\ ent \Longrightarrow is \in it\text{-}paths\ t \Longrightarrow iNodeOf\ (tree\text{-}at\ t\ is) \in sset\ nodes$
⟨*proof*⟩

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

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

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

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

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

shows *hyps-along t (i#is) =*
(λh. subst (iSubst t) (freshen (iAnnot t) (labelsOut (iNodeOf t) h))) ‘ fset (hyps-for (iNodeOf t) (inPorts′ (iNodeOf t) ! i))
∪ hyps-along (iAnts t ! i) is (**is** *?S1 = ?S2 ∪ ?S3*)
⟨*proof*⟩

lemma *iwf-hyps-exist*:
assumes *iwf lc it ent*
assumes *is ∈ it-paths it*
assumes *tree-at it is = (HNode i s ants′)*
assumes *fst ent |⊆| ass-forms*
shows *subst s (freshen i anyP) ∈ hyps-along it is*
⟨*proof*⟩

definition *hyp-port-for′* :: (*′form, ′rule, ′subst, ′var) itree ⇒ nat list ⇒ ′form ⇒ nat list × nat × (′form, ′var) out-port* **where**
hyp-port-for′ t is f = (SOME x.
(case x of (is′, i, h) ⇒
prefix (is′ @ [i]) is ∧
i < length (inPorts′ (iNodeOf (tree-at t is′))) ∧
hyps (iNodeOf (tree-at t is′)) h = Some (inPorts′ (iNodeOf (tree-at t is′)) ! i) ∧
f = subst (iSubst (tree-at t is′)) (freshen (iAnnot (tree-at t is′)) (labelsOut (iNodeOf (tree-at t is′)) h))
))

lemma *hyp-port-for-spec′*:
assumes *f ∈ hyps-along t is*
shows *(case hyp-port-for′ t is f of (is′, i, h) ⇒*
prefix (is′ @ [i]) is ∧
i < length (inPorts′ (iNodeOf (tree-at t is′))) ∧
hyps (iNodeOf (tree-at t is′)) h = Some (inPorts′ (iNodeOf (tree-at t is′)) ! i) ∧
f = subst (iSubst (tree-at t is′)) (freshen (iAnnot (tree-at t is′)) (labelsOut (iNodeOf (tree-at t is′)) h)))
⟨*proof*⟩

definition *hyp-port-path-for* :: (*′form, ′rule, ′subst, ′var) itree ⇒ nat list ⇒ ′form ⇒ nat list*
where *hyp-port-path-for t is f = fst (hyp-port-for′ t is f)*
definition *hyp-port-i-for* :: (*′form, ′rule, ′subst, ′var) itree ⇒ nat list ⇒ ′form ⇒ nat*
where *hyp-port-i-for t is f = fst (snd (hyp-port-for′ t is f))*
definition *hyp-port-h-for* :: (*′form, ′rule, ′subst, ′var) itree ⇒ nat list ⇒ ′form ⇒ (′form, ′var) out-port*
where *hyp-port-h-for t is f = snd (snd (hyp-port-for′ t is f))*

lemma *hyp-port-prefix*:
assumes *f ∈ hyps-along t is*
shows *prefix (hyp-port-path-for t is f@[hyp-port-i-for t is f]) is*
⟨*proof*⟩

lemma *hyp-port-strict-prefix*:
assumes *f ∈ hyps-along t is*
shows *strict-prefix (hyp-port-path-for t is f) is*
⟨*proof*⟩

lemma *hyp-port-it-paths*:
assumes *is ∈ it-paths t*
assumes *f ∈ hyps-along t is*
shows *hyp-port-path-for t is f ∈ it-paths t*
⟨*proof*⟩

lemma *hyp-port-hyps*:
  assumes $f \in$ *hyps-along t is*
  shows *hyps* (*iNodeOf* (*tree-at t* (*hyp-port-path-for t is f*))) (*hyp-port-h-for t is f*) = *Some* (*inPorts'* (*iNodeOf* (*tree-at t* (*hyp-port-path-for t is f*))) ! *hyp-port-i-for t is f*)
⟨*proof*⟩

lemma *hyp-port-outPort*:
  assumes $f \in$ *hyps-along t is*
  shows (*hyp-port-h-for t is f*) |∈| *outPorts* (*iNodeOf* (*tree-at t* (*hyp-port-path-for t is f*)))
⟨*proof*⟩

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

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

definition *mapWithIndex* **where** *mapWithIndex f xs* = *map* (λ (*i*,*t*) . *f i t*) (*List.enumerate 0 xs*)
lemma *mapWithIndex-cong* [*fundef-cong*]:
  xs = *ys* ⟹ (⋀*x i*. *x* ∈ *set ys* ⟹ *f i x* = *g i x*) ⟹ *mapWithIndex f xs* = *mapWithIndex g ys*
⟨*proof*⟩

lemma *mapWithIndex-Nil*[*simp*]: *mapWithIndex f* [] = []
  ⟨*proof*⟩

lemma *length-mapWithIndex*[*simp*]: *length* (*mapWithIndex f xs*) = *length xs*
  ⟨*proof*⟩

lemma *nth-mapWithIndex*[*simp*]: *i* < *length xs* ⟹ *mapWithIndex f xs* ! *i* = *f i* (*xs* ! *i*)
  ⟨*proof*⟩

lemma *list-all2-mapWithIndex2E*:
  assumes *list-all2 P as bs*
  assumes ⋀ *i a b* . *i* < *length bs* ⟹ *P a b* ⟹ *Q a* (*f i b*)
  shows *list-all2 Q as* (*mapWithIndex f bs*)
⟨*proof*⟩

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

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

fun *globalize* :: *nat list* ⇒ (*'var* ⇒ *'var*) ⇒ (*'form*,*'rule*,*'subst*,*'var*) *itree* ⇒ (*'form*,*'rule*,*'subst*,*'var*) *itree* **where**
  globalize is f (*RNode r ants*) = *RNode*
    (*globalize-node is f r*)

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

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

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

lemma *all-local-consts-listed′*:
assumes $n \in sset\ nodes$
assumes $p\ |\in|\ inPorts\ n$
shows $lconsts\ (a\text{-}conc\ p) \cup (\bigcup(lconsts\ `\ fset\ (a\text{-}hyps\ p))) \subseteq a\text{-}fresh\ p$
⟨*proof*⟩

lemma *no-local-consts-in-consequences′*:
$n \in sset\ nodes \Longrightarrow Reg\ p\ |\in|\ outPorts\ n \Longrightarrow\ lconsts\ p = \{\}$
⟨*proof*⟩

lemma *iwf-globalize*:
assumes $local\text{-}iwf\ t\ (\Gamma \vdash c)$
shows $plain\text{-}iwf\ (globalize\ is\ f\ t)\ (renameLCs\ f\ |`|\ \Gamma \vdash renameLCs\ f\ c)$
⟨*proof*⟩

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

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

lemma *fresh-at-snoc*[*simp*]:
$fresh\text{-}at\ t\ (is@[i]) = freshenLC\ (iAnnot\ (tree\text{-}at\ t\ is))\ `\ (a\text{-}fresh\ (inPorts'\ (iNodeOf\ (tree\text{-}at\ t\ is))\ !\ i))$
⟨*proof*⟩

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

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

definition *fresh-at-path* **where**

fresh-at-path t is = $\bigcup$(*fresh-at t* ‘ *set* (*prefixes is*))

lemma *fresh-at-path-Nil*[*simp*]:
fresh-at-path t [] = {}
⟨*proof*⟩

lemma *fresh-at-path-Cons*[*simp*]:
fresh-at-path t (*i*#*is*) = *fresh-at t* [*i*] ∪ *fresh-at-path* (*iAnts t* ! *i*) *is*
⟨*proof*⟩

lemma *globalize-local-consts*:
assumes *is′* ∈ *it-paths* (*globalize is f t*)
shows *subst-lconsts* (*iSubst* (*tree-at* (*globalize is f t*) *is′*)) ⊆
fresh-at-path (*globalize is f t*) *is′* ∪ *range f*
⟨*proof*⟩

lemma *iwf-globalize′*:
assumes *local-iwf t ent*
assumes ⋀ *x*. *x* |∈| *fst ent* ⟹ *closed x*
assumes *closed* (*snd ent*)
shows *plain-iwf* (*globalize is* (*freshenLC v-away*) *t*) *ent*
⟨*proof*⟩
end

end

7.2 Build_Incredible_Tree

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

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

lemma *image-eq-to-f*:
assumes *f1* ‘ *S1* = *f2* ‘ *S2*
obtains *f* **where** ⋀ *x*. *x* ∈ *S2* ⟹ *f x* ∈ *S1* ∧ *f1* (*f x*) = *f2 x*
⟨*proof*⟩

context includes *fset.lifting*
begin
lemma *fimage-eq-to-f*:
assumes *f1* |‘| *S1* = *f2* |‘| *S2*
obtains *f* **where** ⋀ *x*. *x* |∈| *S2* ⟹ *f x* |∈| *S1* ∧ *f1* (*f x*) = *f2 x*
⟨*proof*⟩
end

context *Abstract-Task*
begin

lemma *build-local-iwf*:
fixes *t* :: (*′form entailment* × (*′rule* × *′form*) *NatRule*) *tree*
assumes *tfinite t*
assumes *wf t*
shows ∃ *it*. *local-iwf it* (*fst* (*root t*))
⟨*proof*⟩

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

lemma *iwf-to-it*:
assumes *tfinite t* **and** *wf t*
shows *local-iwf* (*to-it t*) (*fst* (*root t*))
⟨*proof*⟩
end
end

7.3 Incredible_Completeness

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

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

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

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

Let us get our hand on concrete trees.

definition *ts* :: *'form* ⇒ ((*'form entailment*) × (*'rule* × *'form*) *NatRule*) *tree* **where**
ts c = (*SOME t*. *snd* (*fst* (*root t*)) = *c* ∧ *fst* (*fst* (*root t*)) |⊆| *ass-forms* ∧ *wf t* ∧ *tfinite t*)

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

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

lemma *iwf-it*:
  assumes $c \in$ *set conclusions*
  shows *plain-iwf* (*it′ c*) (*fst* (*root* (*ts c*)))
  ⟨*proof*⟩

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

lemma *mem-vertices*: *v* |∈| *vertices* ⟷ (*fst v* ∈ *set conclusions* ∧ (*snd v* = [] ∨ *snd v* ∈ ((#) *0*) ‘ *it-paths* (*it′* (*fst v*))))
  ⟨*proof*⟩

lemma *prefixeq-vertices*: (*c*,*is*) |∈| *vertices* ⟹ *prefix is′ is* ⟹ (*c*, *is′*) |∈| *vertices*
  ⟨*proof*⟩

lemma *none-vertices*[*simp*]: (*c*, []) |∈| *vertices* ⟷ *c* ∈ *set conclusions*
  ⟨*proof*⟩

lemma *some-vertices*[*simp*]: (*c*, *i*#*is*) |∈| *vertices* ⟷ *c* ∈ *set conclusions* ∧ *i* = *0* ∧ *is* ∈ *it-paths* (*it′ c*)
  ⟨*proof*⟩

lemma *vertices-cases*[*consumes 1*, *case-names None Some*]:
  assumes *v* |∈| *vertices*
  obtains *c* **where** *c* ∈ *set conclusions* **and** *v* = (*c*, [])
    | *c is* **where** *c* ∈ *set conclusions* **and** *is* ∈ *it-paths* (*it′ c*) **and** *v* = (*c*, *0*#*is*)
⟨*proof*⟩

lemma *vertices-induct*[*consumes 1*, *case-names None Some*]:
  assumes *v* |∈| *vertices*
  assumes ⋀ *c*. *c* ∈ *set conclusions* ⟹ *P* (*c*, [])
  assumes ⋀ *c is* . *c* ∈ *set conclusions* ⟹ *is* ∈ *it-paths* (*it′ c*) ⟹ *P* (*c*, *0*#*is*)
  shows *P v*
⟨*proof*⟩

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

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

lemma *terminal-is-nil*[*simp*]: *v* |∈| *vertices* ⟹ *outPorts* (*nodeOf v*) = {||} ⟷ *snd v* = []
 ⟨*proof*⟩

sublocale *Vertex-Graph nodes inPorts outPorts vertices nodeOf*⟨*proof*⟩

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

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

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

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

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

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

definition *edge-at* :: *'form* ⇒ *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* ⟨*proof*⟩
lemma *snd-edge-at*[*simp*]: *snd* (*edge-at c is*) = *edge-to c is* ⟨*proof*⟩

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*⟩

definition *hyp-edge-to* :: *'form* ⇒ *nat list* => (*'form vertex* × (*'form*,*'var*) *in-port*) **where**
hyp-edge-to c is = ((*c*, *0* # *is*), *plain-ant anyP*)

definition *hyp-edge-from* :: *'form* ⇒ *nat list* => *nat* ⇒ *'subst* ⇒ (*'form vertex* × (*'form*,*'var*) *out-port*)
where
hyp-edge-from c is n s =
((*c*, *0* # *hyp-port-path-for* (*it' c*) *is* (*subst s* (*freshen n anyP*))),
hyp-port-h-for (*it' c*) *is* (*subst s* (*freshen n anyP*)))

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

lemma *fst-hyp-edge-at*[*simp*]:
fst (*hyp-edge-at c is n s*) = *hyp-edge-from c is n s* ⟨*proof*⟩
lemma *snd-hyp-edge-at*[*simp*]:
snd (*hyp-edge-at c is n s*) = *hyp-edge-to c is* ⟨*proof*⟩

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

sublocale *Pre-Port-Graph nodes inPorts outPorts vertices nodeOf edges*⟨*proof*⟩

lemma *edge-from-valid-out-port*:
assumes *p* ∈ *it-paths* (*it' c*)
assumes *c* ∈ *set conclusions*
shows *valid-out-port* (*edge-from c p*)
⟨*proof*⟩

lemma *edge-to-valid-in-port*:
  assumes $p \in$ *it-paths* (*it′ c*)
  assumes $c \in$ *set conclusions*
  shows *valid-in-port* (*edge-to c p*)
  ⟨*proof*⟩

lemma *hyp-edge-from-valid-out-port*:
  assumes *is* $\in$ *it-paths* (*it′ c*)
  assumes $c \in$ *set conclusions*
  assumes *tree-at* (*it′ c*) *is* = *HNode n s ants*
  shows *valid-out-port* (*hyp-edge-from c is n s*)
⟨*proof*⟩

lemma *hyp-edge-to-valid-in-port*:
  assumes *is* $\in$ *it-paths* (*it′ c*)
  assumes $c \in$ *set conclusions*
  assumes *tree-at* (*it′ c*) *is* = *HNode n s ants*
  shows *valid-in-port* (*hyp-edge-to c is*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *scope-valid-inport*:
  v′ |∈| *vertices* ⟹ *scope′ v ip v′* ⟷ (∃ *i*. *fst v* = *fst v′* ∧ *prefix* (*snd v*@[*i*]) (*snd v′*) ∧ *ip* = *in-port-at v i*)
⟨*proof*⟩

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* []]
  ⟨*proof*⟩

lemma *terminal-path-from-Snoc*[*simp*]:
  terminal-path-from c (*is* @ [*i*]) = *edge-at c* (*is*@[*i*]) # *terminal-path-from c is*
  ⟨*proof*⟩

lemma *path-terminal-path-from*:
  c ∈ *set conclusions* ⟹
  is ∈ *it-paths* (*it′ c*) ⟹
  path (*c*, *0* # *is*) (*c*, []) (*terminal-path-from c is*)
⟨*proof*⟩

lemma *edge-step*:
assumes $(((a,\ b),\ ba),\ ((aa,\ bb),\ bc)) \in$ *edges*
obtains
i **where** $a = aa$ **and** $b = bb@[i]$ **and** $bc =$ *in-port-at* (aa,bb) *i* **and** *hyps* (*nodeOf* $(a,\ b))$ *ba* = *None*
| *i* **where** $a = aa$ **and** *prefix* $(b@[i])$ *bb* **and** *hyps* (*nodeOf* $(a,\ b))$ *ba* = *Some* (*in-port-at* (a,b) *i*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *in-scope*: *valid-in-port* $(v',\ p') \Longrightarrow v \in$ *scope* $(v',\ p') \longleftrightarrow$ *scope'* $v'\ p'\ v$
⟨*proof*⟩

sublocale *Port-Graph nodes inPorts outPorts vertices nodeOf edges*
⟨*proof*⟩

sublocale *Scoped-Graph nodes inPorts outPorts vertices nodeOf edges hyps*⟨*proof*⟩

lemma *hyps-free-path-length*:
assumes *path* $v\ v'$ *pth*
assumes *hyps-free pth*
shows *length pth* + *length* (*snd* v') = *length* (*snd* v)
⟨*proof*⟩

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

lemma *my-vidx-inj*: *inj-on vidx* (*fset vertices*)
⟨*proof*⟩

lemma *vidx-not-v-away*[*simp*]: v |∈| *vertices* $\Longrightarrow$ *vidx* $v \neq$ *v-away*
⟨*proof*⟩

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

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

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

sublocale *Saturated-Graph nodes inPorts outPorts vertices nodeOf edges*
⟨*proof*⟩

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

sublocale *Well-Shaped-Graph nodes inPorts outPorts vertices nodeOf edges hyps*⟨*proof*⟩

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

lemma *node-disjoint-fresh-vars*:
assumes $n \in$ *sset nodes*
assumes $i <$ *length* (*inPorts′ n*)
assumes $i' <$ *length* (*inPorts′ n*)
shows *a-fresh* (*inPorts′ n* ! *i*) ∩ *a-fresh* (*inPorts′ n* ! *i′*) = {} ∨ $i = i'$
⟨*proof*⟩

sublocale *Well-Scoped-Instantiation freshenLC renameLCs lconsts closed subst subst-lconsts subst-renameLCs anyP inPorts outPorts nodeOf hyps nodes vertices labelsIn labelsOut vidx inst edges local-vars*
⟨*proof*⟩

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

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

end

end

8 Instantiations

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

8.1 Propositional_Formulas

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

lemma *countable-infinite-ex-bij*: ∃*f*::(′*a*::{*countable*,*infinite*}⇒′*b*::{*countable*,*infinite*}). *bij f*
⟨*proof*⟩

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

datatype (′*var*,′*cname*) *pform* =
    Var ′*var*::{*countable*,*infinite*}
  | *Fun* (*name*:′*cname*) (*params*: (′*var*,′*cname*) *pform list*)

Substitution on and closedness of propositional formulas is straight forward.

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

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

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

interpretation *propositional*: *Abstract-Formulas*
  — No need to freshen locally fixed constants
  curry (*SOME f*. *bij f*):: *nat* ⇒ ′*var* ⇒ ′*var*
  — also no renaming needed as there are no locally fixed constants
  λ-. *id* λ-. {}
  — closedness and substitution as defined above
  closed :: (′*var*::{*countable*,*infinite*},′*cname*) *pform* ⇒ *bool subst*
  — no substitution and renaming of locally fixed constants
  λ-. {} λ-. *id*
  — most generic formula
  Var undefined
⟨*proof*⟩

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

end

8.2 Incredible_Propositional

theory *Incredible-Propositional* **imports**

Abstract-Rules-To-Incredible
Propositional-Formulas
begin

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

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

The rules are introduction and elimination of conjunction and implication.

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

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

lemma *iR-prop-rules* [*simp*]: *sset prop-rules* = {*andI*, *andE*, *impI*, *impE*}
⟨*proof*⟩

Just some short notation.

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

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

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

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

interpretation *propositional*: *Abstract-Rules*
curry (*SOME f*. *bij f*):: *nat* ⇒ *string* ⇒ *string*
λ-. *id*
λ-. {}
closed :: (*string*, *prop-funs*) *pform* ⇒ *bool*
subst
λ-. {}
λ-. *id*
Var undefined
antecedent
consequent
prop-rules
⟨*proof*⟩

end

8.3 Incredible_Propositional_Tasks

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

context *ND-Rules-Inst* **begin**
lemma *eff-NatRuleI*:
  $nat\text{-}rule\ rule\ c\ ants$
    $\Longrightarrow entail = (\Gamma \vdash subst\ s\ (freshen\ a\ c))$
    $\Longrightarrow hyps = ((\lambda ant.\ ((\lambda p.\ subst\ s\ (freshen\ a\ p))\ |`|\ a\text{-}hyps\ ant\ |\cup|\ \Gamma \vdash subst\ s\ (freshen\ a\ (a\text{-}conc\ ant))))\ |`|\ ants)$
    $\Longrightarrow (\bigwedge ant\ f.\ ant\ |\in|\ ants \Longrightarrow f\ |\in|\ \Gamma \Longrightarrow freshenLC\ a\ `\ (a\text{-}fresh\ ant) \cap lconsts\ f = \{\})$
    $\Longrightarrow (\bigwedge ant.\ ant\ |\in|\ ants \Longrightarrow freshenLC\ a\ `\ (a\text{-}fresh\ ant) \cap subst\text{-}lconsts\ s = \{\})$
    $\Longrightarrow eff\ (NatRule\ rule)\ entail\ hyps$
  ⟨*proof*⟩
end

context *Abstract-Task* **begin**
lemma *natEff-InstI*:
  $rule = (r,c)$
  $\Longrightarrow c \in set\ (consequent\ r)$
  $\Longrightarrow antec = f\text{-}antecedent\ r$
  $\Longrightarrow natEff\text{-}Inst\ rule\ c\ antec$
  ⟨*proof*⟩
end

context begin

8.3.1 Task 1.1

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

abbreviation A :: (*string*,*prop-funs*) *pform*
  where $A \equiv Fun\ (Const\ ''A'')\ []$

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

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

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

lemma *task1-1.solved*
⟨*proof*⟩

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

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

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

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

lemma *path-one-edge*[*simp*]:
task1-1.path v1 v2 pth ⟷
(*v1* = *0* ∧ *v2* = *1* ∧ *pth* = [((*0*,*Reg A*),(*1*,*plain-ant A*))] ∨
pth = [] ∧ *v1* = *v2*)
⟨*proof*⟩

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

interpretation *Tasked-Proof-Graph*
curry (*SOME f*. *bij f*):: *nat* ⇒ *string* ⇒ *string*
λ-. *id*
λ-. {}
closed :: (*string*, *prop-funs*) *pform* ⇒ *bool*
subst
λ-. {}
λ-. *id*
Var undefined
antecedent

consequent
prop-rules
[*A*]
[*A*]
{|*0*::*nat*,*1*|}
undefined(*0* := *Assumption A*, *1* := *Conclusion A*)
{((*0*,*Reg A*),(*1*,*plain-ant A*))}
id
undefined
⟨*proof*⟩

8.3.2 Task 2.11

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

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

interpretation *task2-11*: *Abstract-Task*
curry (*SOME f*. *bij f*):: *nat* ⇒ *string* ⇒ *string*
λ-. *id*
λ-. {}
closed :: (*string*, *prop-funs*) *pform* ⇒ *bool*
subst
λ-. {}
λ-. *id*
Var undefined
antecedent
consequent
prop-rules
[*Fun imp* [*Fun and* [*A*,*B*],*C*]]
[*Fun imp* [*A*,*Fun imp* [*B*,*C*]]]
⟨*proof*⟩

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

lemma *n-andI* [*simp*]: *n-andI* = (*andI*, *Fun and* [*X*,*Y*])
⟨*proof*⟩
lemma *n-andE1* [*simp*]: *n-andE1* = (*andE*, *X*)
⟨*proof*⟩
lemma *n-andE2* [*simp*]: *n-andE2* = (*andE*, *Y*)
⟨*proof*⟩
lemma *n-impI* [*simp*]: *n-impI* = (*impI*, *Fun imp* [*X*,*Y*])
⟨*proof*⟩
lemma *n-impE* [*simp*]: *n-impE* = (*impE*, *Y*)
⟨*proof*⟩

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

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

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

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

interpretation *task2-11*: *Solved-Task*
  curry (*SOME f. bij f*):: *nat* ⇒ *string* ⇒ *string*
  λ-. *id*
  λ-. {}
  closed :: (*string*, *prop-funs*) *pform* ⇒ *bool*
  subst
  λ-. {}
  λ-. *id*
  Var undefined
  antecedent
  consequent
  prop-rules
  [*Fun imp* [*Fun and* [*A*,*B*],*C*]]
  [*Fun imp* [*A*,*Fun imp* [*B*,*C*]]]
⟨*proof*⟩

interpretation *Tasked-Proof-Graph*
  curry (*SOME f. bij f*):: *nat* ⇒ *string* ⇒ *string*
  λ-. *id*
  λ-. {}
  closed :: (*string*, *prop-funs*) *pform* ⇒ *bool*
  subst
  λ-. {}
  λ-. *id*
  Var undefined
  antecedent
  consequent
  prop-rules
  [*Fun imp* [*Fun and* [*A*,*B*],*C*]]
  [*Fun imp* [*A*,*Fun imp* [*B*,*C*]]]
  task2-11.vertices
  task2-11.nodeOf
  task2-11.edges
  task2-11.vidx
  task2-11.inst
⟨*proof*⟩

end

end

8.4 Predicate_Formulas

theory *Predicate-Formulas*
imports
  HOL−Library.Countable
  HOL−Library.Infinite-Set

HOL−Eisbach.Eisbach
Abstract-Formula
begin

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

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

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

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

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

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

fun *fv* :: *form* ⇒ *var set* **where**
  fv (*Var v xs*) = *insert v* (*Union* (*fv* ‘ *set xs*))
| *fv* (*LC v*) = {}
| *fv* (*Op n xs*) = *Union* (*fv* ‘ *set xs*)
| *fv* (*Quant n v f*) = *fv f* − {*v*}

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

lemma *fresh-for-fresh*: *finite V* ⟹ *fresh-for V* ∉ *V*
⟨*proof*⟩

Free variables

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

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

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

lemma *fv-subst-Nil*[*simp*]: *fv-subst1* [] = {}
⟨*proof*⟩

Local constants, separate from free variables.

fun *lc* :: *form* ⇒ *lconst set* **where**
  lc (*Var v xs*) = *Union* (*lc* ‘ *set xs*)
| *lc* (*LC c*) = {*c*}
| *lc* (*Op n xs*) = *Union* (*lc* ‘ *set xs*)
| *lc* (*Quant n v f*) = *lc f*

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

definition *lc-subst1* **where**
lc-subst1 s = $\bigcup$(*lc* ' *snd* ' *set s*)

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

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

lemma *fv-map-lc*[*simp*]: *fv* (*map-lc p f*) = *fv f*
⟨*proof*⟩

lemma *lc-map-lc*[*simp*]: *lc* (*map-lc p f*) = *p* ' *lc f*
⟨*proof*⟩

lemma *map-lc-map-lc*[*simp*]: *map-lc p1* (*map-lc p2 f*) = *map-lc* (*p1* ∘ *p2*) *f*
⟨*proof*⟩

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

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

lemma *map-lc-noop*[*simp*]: *lc f* = {} ⟹ *map-lc p f* = *f*
⟨*proof*⟩

lemma *map-lc-cong*[*cong*]: (⋀*x*. *x* ∈ *lc f* ⟹ *f1 x* = *f2 x*) ⟹ *map-lc f1 f* = *map-lc f2 f*
⟨*proof*⟩

lemma [*simp*]: *fv-subst1* (*map* (*apsnd* (*map-lc p*)) *s*) = *fv-subst1 s*
⟨*proof*⟩

lemma *map-lc-subst-cong*[*cong*]:
assumes (⋀*x*. *x* ∈ *lc-subst s* ⟹ *f1 x* = *f2 x*)
shows *map-lc-subst f1 s* = *map-lc-subst f2 s*
⟨*proof*⟩

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

fun *subst1* :: (*var* × *form*) *list* ⇒ *form* ⇒ *form* **where**
subst1 s (*Var v* []) = (*case map-of s v of Some f* ⇒ *f* | *None* ⇒ *Var v* [])
| *subst1 s* (*Var v xs*) = *Var v xs*
| *subst1 s* (*LC n*) = *LC n*
| *subst1 s* (*Op n xs*) = *Op n* (*map* (*subst1 s*) *xs*)
| *subst1 s* (*Quant n v f*) =
(*if v* ∈ *fv-subst1 s then*
(*let v'* = *fresh-for* (*fv-subst1 s*)
in Quant n v' (*subst1* ((*v*, *Var v'* [])#*s*) *f*))
else Quant n v (*subst1 s f*))

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

⟨*proof*⟩

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

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

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

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

fun $subst' :: subst \Rightarrow form \Rightarrow form$ **where**
$subst'\ s\ (Var\ v\ xs) =$
$(case\ map\text{-}of\ s\ v\ of\ None \Rightarrow (Var\ v\ (map\ (subst'\ s)\ xs))$
$\mid Some\ (ps,rhs) \Rightarrow$
$if\ length\ ps = length\ xs$
$then\ subst1\ (zip\ ps\ (map\ (subst'\ s)\ xs))\ rhs$
$else\ (Var\ v\ (map\ (subst'\ s)\ xs)))$
$\mid subst'\ s\ (LC\ n) = LC\ n$
$\mid subst'\ s\ (Op\ n\ xs) = Op\ n\ (map\ (subst'\ s)\ xs)$
$\mid subst'\ s\ (Quant\ n\ v\ f) =$
$(if\ v \in fv\text{-}subst\ s\ then$
$(let\ v' = fresh\text{-}for\ (fv\text{-}subst\ s)$
$in\ Quant\ n\ v'\ (subst'\ ((v,([],\ Var\ v'\ []))\#s)\ f))$
$else\ Quant\ n\ v\ (subst'\ s\ f))$

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

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

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

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

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

lemma *map-apsnd-zip*[*simp*]: $map\ (apsnd\ f)\ (zip\ a\ b) = zip\ a\ (map\ f\ b)$
⟨*proof*⟩

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

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

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

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

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

lemma *lc-subst*: *lc* (*subst s f*) ⊆ *lc f* ∪ *lc-subst s*
⟨*proof*⟩

lemma *lc-subst-map-lc-subst*[*simp*]: *lc-subst* (*map-lc-subst p s*) = *p* ‘ *lc-subst s*
⟨*proof*⟩

lemma *map-lc-subst*[*simp*]: *map-lc p* (*subst s f*) = *subst* (*map-lc-subst p s*) (*map-lc p f*)
⟨*proof*⟩

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

interpretation *predicate*: *Abstract-Formulas*
curry to-nat :: *nat* ⇒ *var* ⇒ *var*
map-lc
lc
closed
subst
lc-subst
map-lc-subst
Var 0 []
⟨*proof*⟩

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

end

8.5 Incredible_Predicate

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

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

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

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

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

lemma *iR-prop-rules* [*simp*]: *sset prop-rules* = {*allI*, *allE*, *impI*, *impE*}
⟨*proof*⟩

Just some short notation.

abbreviation *X* :: *form*
  where *X* ≡ *Var 10* []
abbreviation *Y* :: *form*
  where *Y* ≡ *Var 11* []
abbreviation *x* :: *form*
  where *x* ≡ *Var 9* []
abbreviation *t* :: *form*
  where *t* ≡ *Var 13* []
abbreviation *P* :: *form* ⇒ *form*
  where *P f* ≡ *Var 12* [*f*]
abbreviation *Q* :: *form* ⇒ *form*
  where *Q f* ≡ *Op ''Q''* [*f*]
abbreviation *imp* :: *form* ⇒ *form* ⇒ *form*
  where *imp f1 f2* ≡ *Op ''imp''* [*f1*, *f2*]
abbreviation *ForallX* :: *form* ⇒ *form*
  where *ForallX f* ≡ *Quant ''all'' 9 f*

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

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

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

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

interpretation *predicate*: *Abstract-Rules*
  curry to-nat :: *nat* ⇒ *var* ⇒ *var*
  map-lc
  lc
  closed
  subst
  lc-subst
  map-lc-subst
  Var 0 []
  antecedent
  consequent
  prop-rules
⟨*proof*⟩

end

8.6 Incredible_Predicate_Tasks

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

begin

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

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

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

context begin

A typical task with local constants:: ∀ *x*. *Q x* ⟶ *Q x*

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

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

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

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

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

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

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

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

lemma *fv-subst-s1*[*simp*]: *fv-subst s1* = {}
⟨*proof*⟩

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

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

lemma *substI1*: *ForallX* (*imp* (*Q x*) (*Q x*)) = *subst s1* (*predicate.freshen 1* (*ForallX* (*P x*)))
⟨*proof*⟩

lemma *substI2*: *imp* (*Q* (*LC lx*)) (*Q* (*LC lx*)) = *subst s2* (*predicate.freshen 2* (*imp X Y*))
⟨*proof*⟩

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

lemma *task.solved*
⟨*proof*⟩

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

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

interpretation *task*: *Vertex-Graph task.nodes task.inPorts task.outPorts vertices nodeOf*⟨*proof*⟩

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

interpretation *task*: *Scoped-Graph task.nodes task.inPorts task.outPorts vertices nodeOf task-edges task.hyps*
⟨*proof*⟩

interpretation *task*: *Instantiation*
task.inPorts

task.outPorts
nodeOf
task.hyps
task.nodes
task-edges
vertices
task.labelsIn
task.labelsOut
curry to-nat :: *nat* ⇒ *var* ⇒ *var*
map-lc
lc
closed
subst
lc-subst
map-lc-subst
Var 0 []
id
inst
⟨*proof*⟩

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
⟨*proof*⟩

lemma *no-path-01*[*simp*]: *task.path 0 v pth* ⟷ (*pth* = [] ∧ *v* = *0*)
⟨*proof*⟩
lemma *no-path-12*[*simp*]: ¬ *task.path 1 2 pth*
⟨*proof*⟩

interpretation *Acyclic-Graph*
task.nodes
task.inPorts
task.outPorts
vertices
nodeOf
task-edges
task.hyps
⟨*proof*⟩

interpretation *Saturated-Graph*
task.nodes
task.inPorts
task.outPorts
vertices
nodeOf
task-edges
⟨*proof*⟩

interpretation *Pruned-Port-Graph*
task.nodes

task.inPorts
task.outPorts
vertices
nodeOf
task-edges
⟨*proof*⟩

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

interpretation *Solution*
task.inPorts
task.outPorts
nodeOf
task.hyps
task.nodes
vertices
task.labelsIn
task.labelsOut
curry to-nat :: *nat* ⇒ *var* ⇒ *var*
map-lc
lc
closed
subst
lc-subst
map-lc-subst
Var 0 []
id
inst
task-edges
⟨*proof*⟩

interpretation *Proof-Graph*
task.nodes
task.inPorts
task.outPorts
vertices
nodeOf
task-edges
task.hyps
task.labelsIn
task.labelsOut
curry to-nat :: *nat* ⇒ *var* ⇒ *var*
map-lc
lc
closed
subst
lc-subst
map-lc-subst
Var 0 []
id

inst
⟨*proof*⟩

lemma *path-20*:
assumes *task.path 2 0 pth*
shows (*1*, *allI-input*) ∈ *snd* ‘ *set pth*
⟨*proof*⟩

lemma *scope-21*: *2* ∈ *task.scope* (*1*, *allI-input*)
⟨*proof*⟩

interpretation *Scoped-Proof-Graph*
curry to-nat :: *nat* ⇒ *var* ⇒ *var*
map-lc
lc
closed
subst
lc-subst
map-lc-subst
Var 0 []
task.inPorts
task.outPorts
nodeOf
task.hyps
task.nodes
vertices
task.labelsIn
task.labelsOut
id
inst
task-edges
task.local-vars
⟨*proof*⟩

interpretation *Tasked-Proof-Graph*
curry to-nat :: *nat* ⇒ *var* ⇒ *var*
map-lc
lc
closed
subst
lc-subst
map-lc-subst
Var 0 []
antecedent
consequent
prop-rules
[]
[*ForallX* (*imp* (*Q x*) (*Q x*))]
vertices
nodeOf
task-edges
id
inst
⟨*proof*⟩

end

end