Abstract

We present in this paper a formalisation of the chapter *Logical Relations and a Case Study in Equivalence Checking* by Karl Crary from the book on *Advanced Topics in Types and Programming Languages*, MIT Press 2005. We use a fully nominal approach to deal with binders. The formalisation has been performed within the Isabelle/HOL proof assistant using the Nominal Package.
Contents

1 Introduction

2 Definition of the language 4
   2.1 Definition of the terms and types ........................................ 4
   2.2 Size functions ................................................................. 4

3 Capture-avoiding substitutions 5
   3.1 Parallel substitution .......................................................... 6
   3.2 Substitution ............................................................................ 6
   3.3 Lemmas about freshness and substitutions ................................. 6

4 Typing 9
   4.1 Typing contexts ..................................................................... 9
   4.2 Definition of the typing relation ............................................... 11
   4.3 Inversion lemmas for the typing relation ................................... 11

5 Definitional Equivalence 12

6 Type-driven equivalence algorithm 12
   6.1 Weak head reduction .............................................................. 12
   6.1.1 Inversion lemma for weak head reduction ............................ 12
   6.2 Weak head normalization ......................................................... 13
   6.3 Algorithmic term equivalence and algorithmic path equivalence .... 14
       6.3.1 Inversion lemmas for algorithmic term and path equivalences .. 15
   6.4 Definition of the logical relation ............................................. 17
   6.5 Fundamental theorems ............................................................. 22
   6.6 Completeness ........................................................................ 24

7 About soundness 24
1 Introduction

For several reasons, proof assistants can be useful for proving properties of programming languages. Indeed, often the proofs consist in inductions involving cases, many of which are trivial. But it hard to guess in advance which case is trivial and even a small error can invalidate a result. Even more the use of a proof assistant can also help the researcher: it is possible to quickly check after a modification of the definitions if the proof is still valid. But in practice, the formalisation of proofs about programming has to address many troubles. The main problem, which is well known in the community is the representation of binders. Informal proofs contains arguments such as 'by renaming of the variables' or 'reasoning modulo alpha conversion'. These arguments are very hard to formalise. Several solutions have been proposed to try to solve this problem. On solution to represent binder is by using De-Bruijn indices. This alleviates such problems about too many details and in some cases leads to very slick proofs. Unfortunately, by using De-Bruijn indices the "symbol-pushing" involves a rather large amount of arithmetic on indices which is not present in informal descriptions. Another method of representing binders is by using higher-order abstract-syntax (HOAS) where the meta-language provides binding-constructs. The disadvantage with HOAS is that one has to encode the language at hand and use the reasoning infrastructure the theorem prover, for example Twelf, provides. In practice this means often that reasoning does not proceed as one would expect from the informal reasoning on paper.

These solution tend to force the user of the system to modify his proofs. We think that this should be the opposite, the system should be modified.

That is why we are currently developing a package for the Isabelle/HOL proof assistant [3], which provides an infrastructure in the theorem prover Isabelle/HOL for representing binders as named $\alpha$-equivalence classes [1, 5, 4].

In this paper, we formalise the chapter about Logical Relation and a Case Study in Equivalence Checking by Karl Crary of the book Advanced Topics in Types and Programming Languages[2]. This example is interesting because logical relations are a fundamental technique for proving properties of programming languages. The purpose of this formalisation is to test and improve the Nominal Package in the context of a 'real life' example. Indeed, this chapter is not an exception, the problem of binders is treated informally, on the first page the reader can find the following sentence: 'As usual, we will identify terms that differ only in the names of bound variables, and our substitution is capture avoiding'.

The formalisation we provide has been realized withing the Isar language [6] within the Isabelle/HOL proof assistant[3]. The definitions and proofs given in this paper have been generated automatically from the formal proofs.
2 Definition of the language

2.1 Definition of the terms and types

First we define the type of atom names which will be used for binders. Each atom type is infinitely many atoms and equality is decidable.

```text
atom-decl name
```

We define the datatype representing types. Although, it does not contain any binder we still use the `nominal_datatype` command because the Nominal datatype package will provide permutation functions and useful lemmas.

```text
nominal-datatype ty =
  TBase
  | TUnit
  | Arrow ty ty (\mapsto [100,100] 100)
```

The datatype of terms contains a binder. The notation `≪name≫ trm` means that the name is bound inside `trm`.

```text
nominal-datatype trm =
  Unit
  | Var name
  | Lam ≪name≫ trm (Lam [\_\_\_ \_\_ [100,100] 100])
  | App trm trm
  | Const nat
```

The datatype of types does not contain any binder, the application of a permutation is the identity function. In the future, this should be automatically derived by the package.

```text
lemma perm-ty[simp]:
  fixes T::ty
  and pi::name prm
  shows pi·T = T
  by (induct T rule: ty.weak-induct) (simp-all)
```

```text
lemma fresh-ty[simp]:
  fixes x::name
  and T::ty
  shows x#T
  by (simp add: fresh-def supp-def)
```

```text
lemma ty-cases:
  fixes T::ty
  shows (\exists T_1, T_2. T=T_1\rightarrow T_2) \lor T=TUnit \lor T=TBase
  by (induct T rule:ty.weak-induct) (auto)
```

2.2 Size functions

We define size functions for types and terms. As Isabelle allows overloading we can use the same notation for both functions.
These functions are automatically generated for non nominal datatypes. In the future, we need to extend the package to generate size functions automatically for nominal datatypes as well.

The definition of a function using the nominal package generates four groups of proof obligations.

The first group are goal of the form $\text{finite}(\text{supp } ())$, these often be solve using the $\text{finite\_guess}$ tactic. The second group of goals corresponds to the invariant. If the user has not chosen to setup an invariant, then it just true and hence can easily be solved.

\begin{verbatim}
instance ty :: size ..

nominal-primrec
  size (TBase) = 1
  size (TUnit) = 1
  size (T1→T2) = size T1 + size T2
by (rule TrueI)+

lemma ty-size-greater-zero[simp]:
  fixes T::ty
  shows size T > 0
by (nominal-induct rule:ty.induct) (simp-all)
\end{verbatim}

\section{Capture-avoiding substitutions}

In this section we define parallel substitution. The usual substitution will be derived as a special case of parallel substitution. But first we define a function to lookup for the term corresponding to a type in an association list. Note that if the term does not appear in the list then we return a variable of that name.

\begin{verbatim}
fun
  lookup :: Subst ⇒ name ⇒ trm
where
  lookup [] x       = Var x
  | lookup ((y,T)#θ) x = (if x=y then T else lookup θ x)

lemma lookup-eqvt[eqvlt]:
  fixes pi::name prm
  shows pi·(lookup θ x) = lookup (pi·θ) (pi·x)
by (induct θ) (auto simp add: perm-bij)

lemma lookup-fresh:
  fixes z::name
  assumes a: z#θ z#x
  shows z# lookup θ x
using a
by (induct rule: lookup.induct)
  (auto simp add: fresh-list-cons)

lemma lookup-fresh':
\end{verbatim}
assumes $a : z \# \theta$
shows $\text{lookup} \ \theta \ z = \text{Var} \ z$
using $a$
by (induct rule: lookup.induct)
(auto simp add: fresh-list-cons fresh-prod fresh-atm)

3.1 Parallel substitution

consts
$\text{psubst} :: \text{Subst} \Rightarrow \text{trm} \Rightarrow \text{trm} \ (\cdot\cdot\cdot) [60,100] 100$

nominal-primrec
$\theta<(\text{Var} \ x)> = (\text{lookup} \ \theta \ x)$
$\theta<(\text{App} \ t_1 \ t_2)> = \text{App} \ (\theta< t_1>) \ (\theta< t_2>)$
$x\#\theta \ \Rightarrow \ \theta<(\text{Lam} \ [x].t)> = \text{Lam} \ [x].(\theta< t>)$
$\theta<(\text{Const} \ n)> = \text{Const} \ n$
$\theta<(\text{Unit})> = \text{Unit}$
apply(finite-guess)+
apply(rule TrueI)+
apply(simp add: abs-fresh)+
apply(fresh-guess)+
done

3.2 Substitution

The substitution function is defined just as a special case of parallel substitution.

abbreviation
$subst :: \text{trm} \Rightarrow \text{name} \Rightarrow \text{trm} \Rightarrow \text{trm} \ (\cdot\cdot\cdot) [100,100,100] 100$
where
$t[x::=t'] \equiv (((x,t'))<t>$

lemma subst[simp]:
shows $(\text{Var} \ x)[y::=t'] = (\text{if} \ x=y \ \text{then} \ t' \ \text{else} \ (\text{Var} \ x))$
and $(\text{App} \ t_1 \ t_2)[y::=t'] = \text{App} \ (t_1[y::=t']) \ (t_2[y::=t'])$
and $x\#(y,t') \ \Rightarrow \ (\text{Lam} \ [x].t)[y::=t'] = \text{Lam} \ [x].(t[y::=t'])$
and $\text{Const} \ n[y::=t'] = \text{Const} \ n$
and $\text{Unit}[y::=t'] = \text{Unit}$
by (simp-all add: fresh-list-cons fresh-list-nil)

lemma subst-eqvt[eqvt]:
fixes $c :: \text{name}$
shows $\pi \cdot (t[x::=t']) = (\pi \cdot t)[(\pi \cdot x)::=(\pi \cdot t')]$
by (nominal-induct t avoiding: x t' rule: trm.induct)
(perm-simp add: fresh-bij)+

3.3 Lemmas about freshness and substitutions

lemma subst-rename:
fixes $c :: \text{name}$
assumes $a : c\#t_1$
shows $t_1[x::=t_2] = ((c,a)\cdot t_1)[c::=t_2]$
using $a$
apply (nominal-induct $t_1$ avoiding: $a \ c \ t_2$ rule: trm.induct)
apply (simp add: trm.inject calc-atm fresh-atm abs-fresh perm-nat-def)+ done

lemma fresh-psubst:
  fixes $::=$
  assumes $a$: $z \# t \ z \# \theta$
  shows $z \# (\theta<\theta>)$
  using $a$
  by (nominal-induct $t$ avoiding: $z \ \theta \ t$ rule: trm.induct)
    (auto simp add: abs-fresh lookup-fresh)

lemma fresh-subst":
  fixes $::=$
  assumes $z\#t_2$
  shows $z\#t_1[z::=t_2]$
  using assms
  by (nominal-induct $t_1$ avoiding: $y \ t_2 \ z$ rule: trm.induct)
    (auto simp add: abs-fresh fresh-nat fresh-atm)

lemma fresh-subst':
  fixes $::=$
  assumes $a$: $z\#t_1 \ z\#t_2$
  shows $z\#t_1[y::=t_2]$
  using $a$
  by (auto simp add: fresh-subst' abs-fresh)

lemma fresh-psubst-simp:
  assumes $x\#t$
  shows $(x,u)\# \theta<\theta> = \theta<\theta>$
  using assms
  proof (nominal-induct $t$ avoiding: $x \ u \ \theta$ rule: trm.induct)
    case (Lam $y \ t \ x \ u$
      have fs: $y \# \theta \ y \# x \ y \# u$ by fact
      moreover have $x \# \text{Lam } [y].t$ by fact
      ultimately have $x \# t$ by (simp add: abs-fresh fresh-atm)
      moreover have $ih:\\n n \ T. \ n \# t \Longrightarrow ((n,T)\# \theta)<\theta> = \theta<\theta>$ by fact
      ultimately have $(x,u)\# \theta<\theta> = \theta<\theta>$ by auto
      moreover have $(x,u)\# \theta<\text{Lam } [y].t> = \text{Lam } [y]. ((x,u)\# \theta<\theta>)$ using fs
      by (simp add: fresh-list-cons fresh-prod)
      moreover have $\theta<\text{Lam } [y].t> = \text{Lam } [y]. (\theta<\theta>)$ using fs by simp
      ultimately show $(x,u)\# \theta<\text{Lam } [y].t> = \theta<\text{Lam } [y].t>$ by auto
    qed (auto simp add: fresh-atm abs-fresh)

7
lemma forget:
  fixes x::name
  assumes a: x#t
  shows t[x:=t'] = t
  using a
by (nominal-induct t avoiding: x t' rule: trm.induct)
(auto simp add: fresh-atm abs-fresh)

lemma subst-fun-eq:
  fixes u::trm
  assumes h: [x].t1 = [y].t2
  shows t1[x:=u] = t2[y:=u]
proof -
  { assume x=y and t1=t2
    then have ?thesis using h by simp
  }
moreover
  { assume h1: x ≠ y and h2: t1 = [x,y] · t2 and h3: x # t2
    then have ([(x,y)] · t2)[x:=u] = t2[y:=u] by (simp add: subst-rename)
    then have ?thesis using h2 by simp
  }
ultimately show ?thesis using alpha h by blast
qed

lemma psubst-empty[simp]:
  shows []<t> = t
by (nominal-induct t rule: trm.induct)
(auto simp add: fresh-list-nil)

lemma psubst-subst-psubst:
  assumes h:c#θ
  shows θ<[(x,y)]·t2>[c:=s] = (c,s)#θ<t>
  using h
by (nominal-induct t avoiding: θ c s rule: trm.induct)
(auto simp add: fresh-list-cons fresh-atm forget lookup-fresh lookup-fresh' fresh-psubst)

lemma subst-fresh-simp:
  assumes a: x#θ
  shows θ<Var x> = Var x
  using a
by (induct θ arbitrary: x, auto simp add: fresh-list-cons fresh-prod fresh-atm)

lemma psubst-subst-propagate:
  assumes x#θ
  shows θ<[(x,u)]> = θ<[(x:=u)]>
  using assms
proof (nominal-induct t avoiding: x u θ rule: trm.induct)
case (Var n x u θ)
  { assume x=n
  moreover have x#θ by fact
  }
ultimately have $\theta < \text{Var} \ n[x::=u]> = \theta < \text{Var} \ n[x::=\theta <u>]$ using subst-fresh-simp by auto 
}
moreover 
{ assume $h\not\in x$ 
 then have $x\not\in \text{Var} \ n$ by (auto simp add: fresh-atm) 
 moreover have $x\not\in \theta$ by fact 
 ultimately have $x\not\in \theta < \text{Var} \ n>$ using fresh-psubst by blast 
 then have $\theta < \text{Var} \ n[x::=\theta <u>]> = \theta < \text{Var} \ n[x::=\theta <u>]> >$ using forget by auto 
 then have $\theta < \text{Var} \ n[x::=u]> = \theta < \text{Var} \ n[x::=\theta <u>]> <$ using $h$ by auto 
} 
ultimately show $\text{case}$ by auto 
next 
case $(\text{Lam} \ n \ t \ x \ u \ \theta)$ 
 have $fs:n\not\in x$ $n\not\in u$ $n\not\in \theta$ $x\not\in \theta$ by fact 
 have $\theta : (\text{Lam} \ [n], t)[x::=u]> = \theta < \text{Lam} \ [n]. (t [x::=u]>)$ using $fs$ by auto 
 then have $\theta < (\text{Lam} \ [n], t)[x::=u]> = \text{Lam} \ [n]. \theta < t [x::=u]> >$ using $fs$ by auto 
 moreover have $\theta < t [x::=u]> = \theta < t [x::=\theta <u>]> >$ using $ih$ by blast 
 ultimately have $\theta < (\text{Lam} \ [n], t)[x::=u]> = \text{Lam} \ [n]. (\theta < t [x::=\theta <u>]>)$ by auto 
 moreover have $\text{Lam} \ [n]. (\theta < t [x::=\theta <u>]>)$ by auto 
 ultimately have $\theta < (\text{Lam} \ [n], t)[x::=u]> = (\text{Lam} \ [n], \theta < t > [x::=\theta <u>]>)$ using $fs$ by auto 
 then show $\theta < (\text{Lam} \ [n], t)[x::=u]> = \theta < (\text{Lam} \ [n], t)[x::=\theta <u>]> >$ using $fs$ by auto 
qed (auto) 

4 Typing 

4.1 Typing contexts 

This section contains the definition and some properties of a typing context. As the concept of context often appears in the litterature and is general, we should in the future provide these lemmas in a library. 

Definition of the Validity of contexts 

First we define what valid contexts are. Informally a context is valid is it does not contains twice the same variable. 

We use the following two inference rules: 

\[
\begin{align*}
\text{valid} & \quad \text{[V_NIL]} \\
\text{valid} \quad \text{valid} & \quad \Gamma \quad \ a \ # \ \Gamma \quad \text{[V_CONS]}
\end{align*}
\]

We need to derive the equivariance lemma for the relation valid. If all the constants which appear in the inductive definition have previously been shown to be equivariant and the lemmas have been tagged using the equiariant attribute then this proof can automated using the nominal.inductive command. 

equivariance valid 

We obtain the following lemma under the name valid.eqvt:
If valid \( x \) then valid \((pi \cdot x)\).

Now, we generate the inversion lemma for non empty lists. We add the \texttt{elim} attribute to tell the automated tactics to use it.

\texttt{inductive-cases2 valid-cons-elim-auto[elim]} \((x,T)\#\Gamma\)

The generated theorem is the following:

\[
\begin{array}{c}
\text{valid } ((x, T) \# \Gamma); [\text{valid } \Gamma; x \# \Gamma] \Rightarrow P \Rightarrow P
\end{array}
\]

\textbf{Definition of sub-contexts} The definition of sub context is standard. We do not use the subset definition to prevent the need for unfolding the definition. We include validity in the definition to shorten the statements.

\texttt{abbreviation sub-context :: Ctxt \Rightarrow Ctxt \Rightarrow bool \ (- \subseteq - [55,55] 55)}

\texttt{where}

\[
\Gamma_1 \subseteq \Gamma_2 \equiv \forall a T. (a,T) \in set \Gamma_1 \longrightarrow (a,T) \in set \Gamma_2
\]

\textbf{Lemmas about valid contexts} Now, we can prove two useful lemmas about valid contexts.

\texttt{lemma valid-monotonicity[elim] :}

\texttt{assumes a \( \Gamma \subseteq \Gamma' \) and b \( x\#\Gamma' \) shows \((x,T_1)\#\Gamma \subseteq (x,T_1)\#\Gamma'\) using a b by auto}

\texttt{lemma fresh-context :}

\texttt{fixes \( \Gamma :: Ctxt \) and a :: name}

\texttt{assumes a\#\Gamma shows \neg(\exists \tau :: ty. (a,\tau) \in set \Gamma) using assms by (induct \Gamma) (auto simp add: fresh-prod fresh-list-cons fresh-atm) }

\texttt{lemma type-unicity-in-context :}

\texttt{assumes a :: valid \( \Gamma \) and b \((x,T_1) \in set \Gamma \) and c \((x,T_2) \in set \Gamma \) shows T_1 = T_2 using a b c by (induct \Gamma) (auto dest!: fresh-context)
4.2 Definition of the typing relation

Now, we can define the typing judgements for terms. The rules are given in figure 1.

Now, we generate the equivariance lemma and the strong induction principle and we derive the lemma about validity.

equivariance typing

nominal-inductive typing
by (simp-all add: abs-fresh)

lemma typing-implies-valid:
assumes a: Γ ⊢ t : T
shows valid Γ
using a by (induct) (auto)

4.3 Inversion lemmas for the typing relation

We generate some inversion lemmas for the typing judgment and add them as elimination rules for the automatic tactics. During the generation of these lemmas, we need the injectivity properties of the constructor of the nominal datatypes. These are not added by default in the set of simplification rules to prevent unwanted simplifications in the rest of the development. In the future, the inductive_cases will be reworked to allow to use its own set of rules instead of the whole 'simpset'.

declare trm.inject [simp add]
declare ty.inject [simp add]

inductive-cases2 t-Lam-elmin-auto[elim]: Γ ⊢ Lam [x] . t : T
inductive-cases2 t-Var-elmin-auto[elim]: Γ ⊢ Var x : T
inductive-cases2 t-App-elmin-auto[elim]: Γ ⊢ App x y : T
inductive-cases2 t-Const-elmin-auto[elim]: Γ ⊢ Const n : T
inductive-cases2 t-Unit-elmin-auto[elim]: Γ ⊢ Unit : TUnit
inductive-cases2 t-Unit-elmin-auto′[elim]: Γ ⊢ s : TUnit

declare trm.inject [simp del]
declare ty.inject [simp del]
5 Definitional Equivalence

\[
\begin{align*}
\Gamma \vdash t : T & \quad \Gamma \vdash t \equiv s : T \\
\Gamma \vdash t : T & \quad \Gamma \vdash t \equiv u : T \\
\Gamma \vdash s \equiv t : T & \quad \Gamma \vdash s \equiv u : T \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash s \equiv t_1 : T_1 \rightarrow T_2 & \quad \Gamma \vdash s \equiv t_2 : T_1 \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \text{App } s_1 \; s_2 \equiv \text{App } t_1 \; t_2 : T_2 \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \text{App } (\text{Lam } [x]. \; s_1) \; s_2 \equiv \text{Lam } [x]. \; t_1 : T_2 \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \text{App } (\text{Var } x) \equiv \text{App } t (\text{Var } x) : T_2 \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash s \equiv t : TUnit & \quad \Gamma \vdash t : TUnit \\
\Gamma \vdash s \equiv t : TUnit & \quad \Gamma \vdash t : TUnit \\
\end{align*}
\]

It is now a tradition, we derive the lemma about validity, and we generate the equivariance lemma and the strong induction principle.

equivalence def-equiv

nominal-inductive def-equiv
by (simp-all add: abs-fresh fresh-subst"")

lemma def-equiv-implies-valid:
assumes \( a : \Gamma \vdash t \equiv s : T \)
shows valid \( \Gamma \)
using \( a \) by (induct) (auto elim: typing-implies-valid)

6 Type-driven equivalence algorithm

We follow the original presentation. The algorithm is described using inference rules only.

6.1 Weak head reduction

6.1.1 Inversion lemma for weak head reduction

declare trm.inject [simp add]
declare ty.inject [simp add]

inductive-cases2 whr-Gen[elim]: \( t \rightsquigarrow t' \)
inductive-cases\textsubscript{2} whr-Lam[elim]: Lam \([x].t \rightsquigarrow t'\)
inductive-cases\textsubscript{2} whr-App-Lam[elim]: App (Lam \([x].t\).t\textsubscript{2}) \rightsquigarrow t
inductive-cases\textsubscript{2} whr-Var[elim]: Var \(x \rightsquigarrow t\)
inductive-cases\textsubscript{2} whr-Const[elim]: Const \(n \rightsquigarrow t\)
inductive-cases\textsubscript{2} whr-App[elim]: App \(p\) \(q \rightsquigarrow t\)
inductive-cases\textsubscript{2} whr-Const-Right[elim]: \(t \rightsquigarrow\) Const \(n\)
inductive-cases\textsubscript{2} whr-Var-Right[elim]: \(t \rightsquigarrow\) Var \(x\)
inductive-cases\textsubscript{2} whr-App-Right[elim]: \(t \rightsquigarrow\) App \(p\) \(q\)

declare \(\text{trm.inj}[\text{simp del}]\)

declare \(\text{ty.inj}[\text{simp del}]\)

equivariance whr-def

6.2 Weak head normalization

abbreviation

\text{nf} :: \text{trm} \Rightarrow \text{bool} (\rightsquigarrow| 100) \equiv \neg (\exists u. t \rightsquigarrow u)

\[
\begin{array}{c}
\frac{s \rightsquigarrow t \quad t \downarrow u}{s \downarrow u} \quad \text{QAN\_REDUCE} \\
\frac{t \rightsquigarrow t}{t \downarrow t} \quad \text{QAN\_NORMAL}
\end{array}
\]

declare \(\text{trm.inj}[\text{simp}]\)

inductive-cases\textsubscript{2} whn-inv-auto[elim]: \(t \Downarrow t'\)

declare \(\text{trm.inj}[\text{simp del}]\)

lemma whn-eqvt[eqvt]:

\begin{itemize}
  \item fixes \(\pi\) :: name \(\text{prm}\)
  \item assumes \(a: t \Downarrow t'\)
  \item shows \((\pi\cdot t) \Downarrow (\pi\cdot t')\)
\end{itemize}

using \(a\)

apply(induct)

apply(rule QAN-Reduce)

apply(rule whr-def.eqvt)

apply(assumption)+

apply(rule QAN-Normal)

apply(auto)

apply(drule-tac \(\pi\)=rev \(\pi\) in whr-def.eqvt)

apply(perm-simp)

done

lemma red-unicity :

\begin{itemize}
  \item assumes \(a: x \rightsquigarrow a\)
  \item and \(b: x \rightsquigarrow b\)
  \item shows \(a=b\)
\end{itemize}
using $a \ b$
apply (induct arbitrary: b)
apply (erule whr-App-Lam)
apply (clarify)
apply (rule subst-fun-eq)
apply (simp)
apply (force)
apply (erule whr-App)
apply (blast)+
done

lemma nf-unicity:
  assumes $x \Downarrow a$ and $x \Downarrow b$
  shows $a = b$
proof (induct arbitrary: b)
  case (QAN-Reduce $x \ t \ a \ b$)
  have $h$: $x \rightsquigarrow t \ t \Downarrow a$ by fact
  have $ih$: $\forall b. \ t \Downarrow b \Rightarrow a = b$ by fact
  have $x \Downarrow b$ by fact
  then obtain $t'$ where $x \rightsquigarrow t'$ and $h: t' \Downarrow b$ using $h$ by auto
  then have $t = t'$ using $h \ \text{red-unicity}$ by auto
  then show $a = b$ using $ih \ hl$ by auto
qed (auto)

6.3 Algorithmic term equivalence and algorithmic path equivalence

\[
\begin{align*}
  & s \Downarrow p \\
  & t \Downarrow q \\
  & \Gamma \vdash p \leftrightarrow q : TBase \\
  & \hline
  & \Gamma \vdash s \leftrightarrow t : TBase \\
\end{align*}
\]
QAT\_BASE

\[
\begin{align*}
  & x \not\in (\Gamma, s, t) \\
  & (x, T_1) \not\in \Gamma \vdash \text{App } s \ (\text{Var } x) \leftrightarrow \text{App } t \ (\text{Var } x) : T_2 \\
  & \hline
  & \Gamma \vdash s \leftrightarrow t : T_1 \rightarrow T_2 \\
  & \hline
  & \text{valid } \Gamma \\
  & \Gamma \vdash s \leftrightarrow t : TUnit \\
  & \hline
  & \text{valid } \Gamma \\
  & (x, T) \in \text{set } \Gamma \\
  & \Gamma \vdash \text{Var } x \leftrightarrow \text{Var } x : T \\
  & \hline
  & \Gamma \vdash p \leftrightarrow q : T_1 \rightarrow T_2 \\
  & \Gamma \vdash s \leftrightarrow t : T_1 \\
  & \hline
  & \Gamma \vdash \text{App } p \ s \leftrightarrow \text{App } q \ t : T_2 \\
  & \hline
  & \text{valid } \Gamma \\
  & \Gamma \vdash \text{Const } n \leftrightarrow \text{Const } n : TBase
\end{align*}
\]
QAP\_VAR
QAP\_APP
QAP\_CONST

Again we generate the equivariance lemma and the strong induction principle.

equivariance alg-equiv

nominal-inductive alg-equiv
  avoids QAT\_Arrow: $x$
  by simp-all
6.3.1 Inversion lemmas for algorithmic term and path equivalences

declare trm.inject [simp add]
declare ty.inject [simp add]

inductive-cases2 alg-equiv-Base-inv-auto[elim]: Γ ⊢ s ⇔ t : TBase
inductive-cases2 alg-equiv-Arrow-inv-auto[elim]: Γ ⊢ s ⇔ t : T₁ → T₂

inductive-cases2 alg-path-equiv-Base-inv-auto[elim]: Γ ⊢ s ↔ t : TBase
inductive-cases2 alg-path-equiv-Unit-inv-auto[elim]: Γ ⊢ s ↔ t : TUnit
inductive-cases2 alg-path-equiv-Arrow-inv-auto[elim]: Γ ⊢ s ↔ t : T₁ → T₂
inductive-cases2 alg-path-equiv-Var-left-inv-auto[elim]: Γ ⊢ Var x ↔ t : T
inductive-cases2 alg-path-equiv-Var-right-inv-auto[elim]: Γ ⊢ s ↔ Var x : T
inductive-cases2 alg-path-equiv-Const-left-inv-auto[elim]: Γ ⊢ Const n ↔ t : T
inductive-cases2 alg-path-equiv-Const-right-inv-auto[elim]: Γ ⊢ s ↔ Const n : T
inductive-cases2 alg-path-equiv-App-left-inv-auto[elim]: Γ ⊢ App p t ↔ t : T
inductive-cases2 alg-path-equiv-Lam-left-inv-auto[elim]: Γ ⊢ Lam[x].s ↔ t : T
inductive-cases2 alg-path-equiv-Lam-right-inv-auto[elim]: Γ ⊢ t ↔ Lam[x].s : T

lemma Q-Arrow-strong-inversion:
  assumes fs: x#Γ x#t x#u
  and h: Γ ⊢ t ↔ u : T₁→T₂
  shows (x,T₁)#Γ ⊢ App t (Var x) ↔ App u (Var x) : T₂
proof –
  obtain y where fs2: y#(Γ,t,u) and (y,T₁)#Γ ⊢ App t (Var y) ↔ App u (Var y) : T₂
  using h by auto
  then have [[[(x,y)],[y,T₁],[Γ]]] ⊢ [(x,y)]: App t (Var y) ↔ [(x,y)]: App u (Var y) : T₂
  using alg-eqv.eqvt[simplified] by blast
  then show ?thesis using fs fs2 by (perm-simp)
qed

For the algorithmic_transitivity lemma we need a unicity property. But one has to be cautious, because this unicity property is true only for algorithmic path. Indeed the following lemma is false:

\[ \Gamma ⊢ s ↔ t : T; \Gamma ⊢ s ↔ u : T' \implies T = T' \]

Here is the counter example:

\[ \Gamma ⊢ Const n ↔ Const n : TBase \text{ and } \Gamma ⊢ Const n ↔ Const n : TUnit \]
lemma algorithmic-path-type-unicty:

shows $\Gamma \vdash s \leftrightarrow t : T \Rightarrow \Gamma \vdash s \leftrightarrow a : T' \Rightarrow T = T'$

proof (induct arbitrary: $u T'$

  rule: alg-equiv-alg-path-equiv.inducts(2) [of $a b c d$. True ])

case (QAP-Var $\Gamma x T u T'$)

  have $\Gamma \vdash Var x \leftrightarrow u : T' \text{ by fact}$

  then have $u = \text{Var x and } (x, T') \in \text{set } \Gamma \text{ by auto}$

  moreover have valid $\Gamma (x, T) \in \text{set } \Gamma \text{ by fact}$

  ultimately show $T = T'$ using type-unicity-in-context by auto

next

case (QAP-App $\Gamma p q T_1 T_2 s t u T'_2$)

  have $\text{ih} : \forall u T. \Gamma \vdash p \leftrightarrow u : T \Rightarrow T_1 \Rightarrow T_2 = T \text{ by fact}$

  have $\Gamma \vdash \text{App p s} \leftrightarrow u : T'_2 \text{ by fact}$

  then obtain $r t T'_1 \text{ where } u = \text{App r t } \Gamma \vdash p \leftrightarrow r : T'_1 \Rightarrow T'_2 \text{ by auto}$

  then have $T_1 \Rightarrow T_2 = T'_1 \Rightarrow T'_2 \text{ by auto}$

  then show $T_2 = T'_2 \text{ using ty.inject by auto}$

qed (auto)

lemma alg-path-equiv-implies-valid:

shows $\Gamma \vdash s \leftrightarrow t : T \Rightarrow \text{valid } \Gamma$

and $\Gamma \vdash s \leftrightarrow t : T \Rightarrow \text{valid } \Gamma$

by (induct rule : alg-equiv-alg-path-equiv.inducts, auto)

lemma algorithmic-symmetry:

shows $\Gamma \vdash s \leftrightarrow t : T \Rightarrow \Gamma \vdash t \leftrightarrow s : T$

and $\Gamma \vdash s \leftrightarrow t : T \Rightarrow \Gamma \vdash t \leftrightarrow s : T$

by (induct rule: alg-equiv-alg-path-equiv.inducts)

(auto simp add: fresh-prod)

lemma algorithmic-transitivity:

shows $\Gamma \vdash s \leftrightarrow t : T \Rightarrow \Gamma \vdash t \leftrightarrow u : T \Rightarrow \Gamma \vdash s \leftrightarrow u : T$

and $\Gamma \vdash s \leftrightarrow t : T \Rightarrow \Gamma \vdash t \leftrightarrow u : T \Rightarrow \Gamma \vdash s \leftrightarrow u : T$

proof (nominal-induct $\Gamma s t T \text{ and } \Gamma s t T_1 \text{ avoiding: u rule: alg-equiv-alg-path-equiv.strong-inducts}$)

case (QAT-Base $s t q \Gamma u$)

  have $\Gamma \vdash t \leftrightarrow u : \text{TBase by fact}$

  then obtain $r' q' \text{ where } b1: t \nleq q' \text{ and } b2: u \nleq r' \text{ and } b3: \Gamma \vdash q' \leftrightarrow r' : \text{TBase by auto}$

  have $\text{ih} : \Gamma \vdash q \leftrightarrow r' : \text{TBase} \Rightarrow \Gamma \vdash p \leftrightarrow r' : \text{TBase by fact}$

  have $t \nleq q \text{ by fact}$

  with $b1$ have $\text{eq: q=q' by simp add: nf-unicity}$

  with $\text{ih b3}$ have $\Gamma \vdash p \leftrightarrow r' : \text{TBase by simp}$

  moreover have $s \nleq p \text{ by fact}$

  ultimately show $\Gamma \vdash s \leftrightarrow u : \text{TBase using b2 by auto}$

next

case (QAT-Arrow $x \Gamma s t T_1 T_2 u$)

  have $\text{ih} (x, T_1) \# \Gamma \vdash \text{App t (Var x)} \leftrightarrow \text{App u (Var x)} : T_2$

  $\Rightarrow (x, T_1) \# \Gamma \vdash \text{App s (Var x)} \leftrightarrow \text{App u (Var x)} : T_2 \text{ by fact}$

  have $fs : x \# \Gamma x\#s x\#t x\#u \text{ by fact}$

  have $\Gamma \vdash t \leftrightarrow u : T_1 \Rightarrow T_2 \text{ by fact}$

  then have $(x, T_1) \# \Gamma \vdash \text{App t (Var x)} \leftrightarrow \text{App u (Var x)} : T_2 \text{ using fs}$

  by (simp add: Q-Arrow-strong-inversion)

  with $\text{ih}$ have $(x, T_1) \# \Gamma \vdash \text{App s (Var x)} \leftrightarrow \text{App u (Var x)} : T_2 \text{ by simp}$

  then show $\Gamma \vdash s \leftrightarrow u : T_1 \Rightarrow T_2 \text{ using fs by (auto simp add: fresh-prod)}$
next
  case (QAP-App Γ p q T₁ T₂ s t u)
  have Γ ⊢ App q t ↔ u : T₂ by fact
  then obtain r T₁' v where ha: Γ ⊢ q ↔ r : T₁' → T₂ and hb: Γ ⊢ t ↔ v : T₁' and eq: u = App r v
  by auto
  have ih₁: Γ ⊢ q ↔ r : T₁ → T₂ ⇒ Γ ⊢ p ↔ r : T₁ → T₂ by fact
  have ih₂: Γ ⊢ t ↔ v : T₁ ⇒ Γ ⊢ s ↔ v : T₁ by fact
  have Γ ⊢ p ↔ q : T₁ → T₂ by fact
  then have Γ ⊢ q ⊢ p : T₁ → T₂ by (simp add: algorithmic-symmetry)
  with ha have T₁' → T₂ = T₁ → T₂ using algorithmic-path-type-unicity by simp
  then have T₁' = T₁ by (simp add: ty.inject)
  then have Γ ⊢ s ⊢ v : T₁ Γ ⊢ p ⊢ r : T₁ → T₂ using ih₁ ih₂ ha hb by auto
  then show Γ ⊢ App p s ↔ u : T₂ using eq by auto
  qed (auto)
lemma algorithmic-weak-head-closure:
  shows Γ ⊢ s ⊢ t : T ⇒ s' ⊢ t' ⇒ t' ⇒ Γ ⊢ s' ⊢ t' : T
  apply (nominal-induct Γ s t T avoiding: s' t')
  rule: alg-equiv-alg-path-eqeqv.strong-inducts(1) [of - - - - %a b c d e. True]
  apply(auto intro!: QAT-Arrow)
  done
lemma algorithmic-monotonicity:
  shows Γ ⊢ s ⊢ t : T ⇒ Γ ⊆ Γ' ⇒ valid Γ' ⇒ Γ' ⊢ s ⊢ t : T
  and Γ ⊢ s ' ⊢ t : T ⇒ Γ ⊆ Γ ' ⇒ valid Γ ' ⇒ Γ' ⊢ s ' ⊢ t : T
proof (nominal-induct Γ s t T and Γ s t T avoiding: Γ rule: alg-equiv-alg-path-eqeqv.strong-inducts)
  case (QAT-Arrow x Γ s t T₁ T₂ Γ')
  have fs:x#Γ x#s x#t x#Γ by fact
  have h2:Γ ⊆ Γ' by fact
  have ih:∀Γ', [(x,T₁)#Γ ⊆ Γ'; valid Γ'] ⇒ Γ' ⊢ App s (Var x) ⇔ App t (Var x) : T₂ by fact
  have valid Γ' by fact
  then have valid ((x,T₁)#Γ') using fs by auto
  moreover
  have sub: (x,T₁)#Γ ⊆ (x,T₁)#Γ' using h2 by auto
  ultimately have (x,T₁)#Γ' ⊢ App s (Var x) ⇔ App t (Var x) : T₂ using ih by simp
  then show Γ' ⊢ s ⊢ t : T₁ → T₂ using fs by (auto simp add: fresh-prod)
  qed (auto)
lemma path-equiv-implies-nf:
  assumes Γ ⊢ s ⊢ t : T
  shows s ⊢| and t ⊢| using assms
  by (induct rule: alg-equiv-alg-path-eqeqv.inducts(2)) (simp, auto)

6.4 Definition of the logical relation

We define the logical equivalence as a function. Note that here we can not use an inductive definition because of the negative occurrence in the arrow case.

function log-equiv :: (Ctx ⇒ trm ⇒ trm ⇒ ty ⇒ bool) (- ⊢ - is - - [60,60,60,60] 60)
where
\[ \Gamma \vdash s \text{ is } t : T \text{Unit } = \text{True} \]
| \( \Gamma \vdash s \text{ is } t : T \text{Base} \) = \( \Gamma \vdash s \leftrightarrow t : T \text{Base} \)
| \( \Gamma \vdash s \text{ is } t : (T_1 \rightarrow T_2) = (\forall \Gamma' s' t'. \Gamma \subseteq \Gamma' \Rightarrow \Gamma' \vdash s' \text{ is } t' : T_1 \rightarrow (\Gamma' \vdash (\text{App } s s') \text{ is } (\text{App } t t') : T_2)) \)
apply (auto simp add: ty.inject)
apply (subgoal_tac (\( \exists T_1 T_2. b=T_1 \rightarrow T_2 \) \& b=TUnit \& b=TBase) )
apply (force)
apply (rule ty-cases)
done

termination
apply (relation measure (\( \lambda(\cdot,\cdot,\cdot). \text{size } T) \))
apply (auto)
done

Monotonicity of the logical equivalence relation.

lemma logical-monotonicity :
assumes a1: \( \Gamma \vdash s \text{ is } t : T \)
and a2: \( \Gamma \subseteq \Gamma' \)
and a3: \( \text{valid } \Gamma' \)
shows \( \Gamma' \vdash s \text{ is } t : T \)
using a1 a2 a3
proof (induct arbitrary: \( \Gamma' \) rule: log-eq.e.induct)
case \( (\exists T \Gamma s t \Gamma') \)
then show \( \Gamma' \vdash s \text{ is } t : T \)Base using algorithmic-monotonicity by auto
next
case \( (\exists \Gamma s t T_1 T_2 \Gamma') \)
have \( \Gamma \vdash s \text{ is } t : T_1 \rightarrow T_2 \)
and \( \Gamma \subseteq \Gamma' \)
and \( \text{valid } \Gamma ' \) by fact
then show \( \Gamma' \vdash s \text{ is } t : T_1 \rightarrow T_2 \) by simp
qed (auto)

lemma main-lemma:
shows \( \Gamma \vdash s \text{ is } t : T \Rightarrow \text{valid } \Gamma \Rightarrow \Gamma \vdash s \leftrightarrow t : T \)
and \( \Gamma \vdash p \leftrightarrow q : T \Rightarrow \Gamma \vdash p \text{ is } q : T \)
proof (nominal-induct \( T \) arbitrary: \( \Gamma j s t p q \) rule: ty.induct)
case (Arrow \( T_1 T_2 \))
{
case \( (t \Gamma s t) \)
have \( \text{ih1:} \bigwedge T \Gamma s t. \big[ \Gamma \vdash s \text{ is } t : T_2; \text{ valid } \Gamma \big] \Rightarrow \Gamma \vdash s \leftrightarrow t : T_2 \text{ by fact} \)
have \( \text{ih2:} \bigwedge T \Gamma s t. \Gamma \vdash s \leftrightarrow t : T_1 \Rightarrow \Gamma \vdash s \text{ is } t : T_1 \text{ by fact} \)
have \( h: \Gamma \vdash s \text{ is } t : T_1 \rightarrow T_2 \text{ by fact} \)
obtain \( x::\text{name where } fs\#: (\Gamma, s, t) \) by (erule exists-fresh[of fs-name1])
have \( \text{valid } \Gamma \text{ by fact} \)
then have \( v: \text{ valid } ((x, T_1)\#\Gamma) \text{ using } fs \text{ by auto} \)
then have \( (x, T_1)\#\Gamma \vdash \text{Var } x \leftrightarrow \text{Var } x : T_1 \text{ by auto} \)
then have \( (x, T_1)\#\Gamma \vdash \text{Var } x \text{ is } \text{Var } x : T_1 \text{ using } \text{ih2 by auto} \)
then have \( (x, T_1)\#\Gamma \vdash \text{App } s (\text{Var } x) \text{ is } \text{App } t (\text{Var } x) : T_2 \text{ using } \text{h by auto} \)
then have \( (x, T_1)\#\Gamma \vdash \text{App } s (\text{Var } x) \leftrightarrow \text{App } t (\text{Var } x) : T_2 \text{ using } \text{ih1 v by auto} \)
then show \( \Gamma \vdash s \leftrightarrow t : T_1 \rightarrow T_2 \text{ using } fs \text{ by } (auto \text{ simp add: fresh-prod}) \)
next
case (2 Γ p q)
  have h: Γ ⊢ p ↔ q : T₁ → T₂ by fact
  have ih1: ∀ s t. Γ ⊢ s ↔ t : T₂ ⇒ Γ ⊢ s is t : T₂ by fact
  have ih2: ∀ s t. [Γ ⊢ s is t : T₁; valid Γ] ⇒ Γ ⊢ s ⇔ t : T₁ by fact
  { fix Γ′ s t
    assume Γ ⊆ Γ′ and hl: Γ′ ⊢ s is t : T₁ and hk: valid Γ′
    then have Γ′ ⊢ p ↔ q : T₁ → T₂ using h algorithmic-monotonicity by auto
    moreover have Γ′ ⊢ s ⇔ t : T₁ using ih2 hl hk by auto
    ultimately have Γ′ ⊢ App p s is App q t : T₂ by auto
    then have Γ′ ⊢ App p s is s t : T₂ using ih1 by auto
    } then show Γ ⊢ p is q : T₁ → T₂ by simp
next
  case TBase
  { case 2
    have h: Γ ⊢ s t : TBase by fact
    then have s t : TBase using path-equiv-implies-nf by auto
    then have Γ ⊢ s ⇔ t : TBase using h by auto
    then show Γ ⊢ s is t : TBase by auto
  }
qed (auto elim: alg-path-equiv-implies-valid)
corollary corollary-main:
  assumes a: Γ ⊢ s ↔ t : T
  shows Γ ⊢ s t : T
using a main-lemma alg-path-equiv-implies-valid by blast
lemma logical-symmetry:
  assumes a: Γ ⊢ s t : T
  shows Γ ⊢ t s : T
using a by (nominal-induct arbitrary: Γ s t rule: ty.induct)
  (auto simp add: algorithmic-symmetry)
lemma logical-transitivity:
  assumes Γ ⊢ s t : T Γ ⊢ t u : T
  shows Γ ⊢ s u : T
using assms
proof (nominal-induct arbitrary: Γ s u rule: ty.induct)
  case TBase
  then show Γ ⊢ s is u : TBase by (auto elim: algorithmic-transitivity)
next
  case (Arrow T₁ T₂ Γ s t u)
  have h1: Γ ⊢ s t : T₁ → T₂ by fact
  have h2: Γ ⊢ t u : T₁ → T₂ by fact
  have ih1: ∀ s t u. [Γ ⊢ s t : T₁; Γ ⊢ t u : T₂] ⇒ Γ ⊢ s u : T₁ by fact
  have ih2: ∀ s t u. [Γ ⊢ s t : T₂; Γ ⊢ t u : T₁] ⇒ Γ ⊢ s u : T₂ by fact
  { fix Γ′ s′ u′
    assume hsub: Γ ⊆ Γ′ and hl: Γ′ ⊢ s′ u′ : T₁ and hk: valid Γ′
then have $\Gamma' \vdash u'$ is $s'$ : $T_1$ using logical-symmetry by blast
then have $\Gamma' \vdash u'$ is $u'$ : $T_1$ using $ih\, hl$ by blast
then have $\Gamma' \vdash App\ t\ u'$ is $App\ u\ u'$ : $T_2$ using $h2\ hsub\ hl\ hk$ by auto
moreover have $\Gamma' \vdash App\ s\ s'$ is $App\ t\ u'$ : $T_2$ using $h1\ hsub\ hl\ hk$ by auto
ultimately have $\Gamma' \vdash App\ s\ s'$ is $App\ u\ u'$ : $T_2$ using $ih2$ by blast

\text{then show} $\Gamma \vdash s$ is $u$ : $T_1 \rightarrow T_2$ by auto
\text{qed (auto)}

To simplify the formal proof, here we derive two lemmas which are weaker than the lemma in the paper version. We omit the reflexive and transitive closure of the relation $s' \leadsto s$ in the assumptions.

\textbf{lemma logical-weak-head-closure:}
\begin{itemize}
  \item \textbf{assumes} $a$: $\Gamma \vdash s$ is $t$ : $T$
  \item \text{and} $b$: $s' \leadsto s$
  \item \text{and} $c$: $t' \leadsto t$
  \item \text{shows} $\Gamma \vdash s'$ is $t'$ : $T$
\end{itemize}
\text{using} $a\ b\ c$ algorithmic-weak-head-closure
\text{by} (nominal-induct arbitrary: $\Gamma \ s\ s'\ t'\$ rule: ty.induct)
(auto, blast)

\textbf{lemma logical-weak-head-closure':}
\begin{itemize}
  \item \textbf{assumes} $\Gamma \vdash s$ is $t$ : $T$
  \item \text{and} $s' \leadsto s$
  \item \text{shows} $\Gamma \vdash s'$ is $t$ : $T$
\end{itemize}
\text{using} assms
\text{proof (nominal-induct arbitrary: $\Gamma \ s\ s'\ t'$ rule: ty.induct)}
\begin{itemize}
  \item \text{case (}TBase\ $\Gamma \ s\ t\ s'\$)
  \item \text{then show} ?case by force
\end{itemize}
\text{next}
\begin{itemize}
  \item \text{case (}TUnit\ $\Gamma \ s\ t\ s'\$)
  \item \text{then show} ?case by auto
\end{itemize}
\text{next}
\begin{itemize}
  \item \text{case (}Arrow\ T_1\ T_2\ \Gamma \ s\ t\ s'\$)
  \item \text{have $h1:s' \leadsto s$ by fact}
  \item \text{have $ih:\forall\ s\ t\ s'.\ [\Gamma \vdash s$ is $t$ : $T_2;\ s' \leadsto s]\ \implies \Gamma \vdash s'$ is $t$ : $T_2$ by fact}
  \item \text{have $h2:\forall\ s\ t\ :\ T_1\rightarrow T_2$ by fact}
  \item \text{then}
  \begin{itemize}
    \item \text{have $hb:\forall\ s'\ t'.\ \Gamma' \subseteq \Gamma' \implies valid\ \Gamma' \implies \Gamma' \vdash s'$ is $t'$ : $T_1 \rightarrow (\Gamma' \vdash (App\ s\ s')$ is $\ (App\ t\ t')) : T_2)$ by auto
  \end{itemize}
\end{itemize}
\text{by auto}
\begin{itemize}
  \item \text{fix $\Gamma' s_2\ t_2$}
  \item \text{assume $\Gamma' \subseteq \Gamma'$ and $\Gamma' \vdash s_2$ is $t_2$ : $T_1$ and valid $\Gamma'$}
  \item \text{then have $\Gamma' \vdash (App\ s\ s_2)$ is $\ (App\ t\ t_2)$ : $T_2$ using $hb$ by auto}
  \item \text{moreover have $(App\ s'\ s_2)$ $\leadsto$ $(App\ s\ s_2)$ using $h1$ by auto}
  \item \text{ultimately have $\Gamma' \vdash App\ s'\ s_2$ is $App\ t\ t_2$ : $T_2$ using $ih$ by auto}
\end{itemize}
\text{then show} $\Gamma \vdash s'$ is $t$ : $T_1 \rightarrow T_2$ by auto
\text{qed}

\textbf{abbreviation}
\begin{itemize}
  \item log-equiv-for-psubsts :: Ctxt $\Rightarrow$ Subst $\Rightarrow$ Subst $\Rightarrow$ Ctxt $\Rightarrow$ bool ($\vdash -$ is $-$ over $-$ $[60,60] - 60$)
\end{itemize}
\text{where}
Γ' ⊢ θ is θ' over Γ ≡ ∀ x T. (x, T) ∈ set Γ → Γ' ⊢ θ<Var x> is θ'<Var x> : T

Now, we can derive that the logical equivalence is almost reflexive.

**Lemma logical-pseudo-reflexivity:**

*Assumes* Γ' ⊢ t is s over Γ

*Shows* Γ' ⊢ s is s over Γ

*Proof*

- *Have* Γ' ⊢ t is s over Γ by *fact*
- *Moreover then have* Γ' ⊢ s is t over Γ using *logical-symmetry* by *blast*
- *Ultimately show* Γ' ⊢ s is s over Γ using *logical-transitivity* by *blast*

qed

**Lemma logical-subst-monotonicity:**

*Assumes* a: Γ' ⊢ s is t over Γ

and

b: Γ' ⊆ Γ''

and

c: valid Γ''

*Shows* Γ'' ⊢ s is t over Γ

*Using* a b c *logical-monotonicity* by *blast*

**Lemma equiv-subst-ext:**

*Assumes* h1: Γ' ⊢ θ is θ' over Γ

and

h2: Γ' ⊢ s is t : T

and

fs: x # Γ

*Shows* Γ' ⊢ (x, s)#θ is (x, t)#θ' over (x, T)#Γ

*Using* *assms*

*Proof*

- {Fix y U
  
  Assume (y, U) ∈ set ((x, T)#Γ)

  Moreover

  - Assume (y, U) ∈ set [(x, T)]
    
    Then have Γ' ⊢ (x, s)#θ<Var y> is (x, t)#θ'<Var y> : U by *auto*
  }

  Moreover

  - Assume hl: (y, U) ∈ set Γ
    
    Then have ¬ y # Γ by (induct Γ) (auto simp add: fresh-list-cons fresh-atm fresh-prod)

    Then have hlf: x # Var y using fs by (auto simp add: fresh-atm)

    Then have (x, s)#θ<Var y> = θ<Var y> (x, t)#θ'<Var y> = θ'<Var y> using fresh-psubst-simp

    by *blast +*

    Moreover have Γ' ⊢ θ<Var y> is θ'<Var y> : U using h1 hl by *auto*

    Ultimately have Γ' ⊢ (x, s)#θ<Var y> is (x, t)#θ'<Var y> : U by *auto*

    Ultimately have Γ' ⊢ (x, s)#θ<Var y> is (x, t)#θ'<Var y> : U by *auto*

  }

  Then show Γ' ⊢ (x, s)#θ is (x, t)#θ' over (x, T)#Γ by *auto*

qed
6.5 Fundamental theorems

\textbf{theorem fundamental-theorem-1:}
\begin{align*}
& \text{assumes } h1: \Gamma \vdash t : T \\
& \text{and } h2: \Gamma' \vdash \theta \text{ is } \theta' \text{ over } \Gamma \\
& \text{and } h3: \text{valid } \Gamma' \\
& \text{shows } \Gamma' \vdash \theta < t > \text{ is } \theta' < t > : T
\end{align*}
\begin{proof} (nominal-induct $\Gamma \; t \; T$ avoiding; $\Gamma' \; \theta \; \theta'$ rule: typing.strong-induct)
\begin{enumerate}
\item case $(\text{t-Lam} \; x \; \Gamma \; T_1 \; t_2 \; \Gamma' \; \theta \; \theta')$
\item have $fs : x \# \theta \; x \# \theta' \; x \# \Gamma$ by fact
\item have $h: \Gamma' \vdash \theta \text{ is } \theta' \text{ over } \Gamma$ by fact
\item have $ih : \bigwedge \Gamma' \; \theta \; \theta' \; [\bigwedge \Gamma' \vdash \theta < t > \text{ is } \theta' < t > : T_2 \text{ by fact}
\{\text{fix } \Gamma'' \; s' \; t'\}
\item assume $\Gamma'' \subseteq \Gamma''$ and $h: \Gamma'' \vdash s' \text{ is } t' : T_1$ and $v : \text{valid } \Gamma''$
\item then have $\Gamma'' \vdash \theta \text{ is } \theta' \text{ over } \Gamma$ using logical-subst-monotonicity $h$ by blast
\item then have $\Gamma'' \vdash (x, s') \# \theta \text{ is } (x, t') \# \theta' \text{ over } (x, T_1) \# \Gamma$ using equiv-subst-ext $h l f s$ by blast
\item then have $\Gamma'' \vdash (x, s') \# \theta < t_2 > : T_2$ using $ih \; v$ by auto
\item then have $\Gamma'' \vdash \theta < t_2 > [x :: s'] \text{ is } \theta' < t_2 > [x :: t'] : T_2$ using psb-subst-psb-subst $f s$ by simp
\item moreover have $\text{App } (\text{Lam } [x], \theta < t_2 >) \; s' \rightarrow \theta < t_2 > [x :: s'] : \text{by auto}
\item moreover have $\text{App } (\text{Lam } [x], \theta' < t_2 >) \; t' \rightarrow \theta' < t_2 > [x :: t'] : \text{by auto}
\item ultimately have $\Gamma'' \vdash \text{App } (\text{Lam } [x], \theta < t_2 >) \; s' \text{ is } \text{App } (\text{Lam } [x], \theta' < t_2 >) \; t' : T_2$
\item using logical-weak-head-closure by auto\}
\item then show $\Gamma' \vdash \theta < \text{Lam } [x], t_2 > \text{ is } \theta' < \text{Lam } [x], t_2 > : T_1 \rightarrow T_2$ using $f s$ by simp
\end{enumerate}
\end{proof} (auto)

\textbf{theorem fundamental-theorem-2:}
\begin{align*}
& \text{assumes } h1: \Gamma \vdash s \equiv t : T \\
& \text{and } h2: \Gamma' \vdash \theta \text{ is } \theta' \text{ over } \Gamma \\
& \text{and } h3: \text{valid } \Gamma' \\
& \text{shows } \Gamma' \vdash \theta < s > \text{ is } \theta' < t > : T
\end{align*}
\begin{proof} (nominal-induct $\Gamma \; s \; t \; T$ avoiding; $\Gamma' \; \theta \; \theta'$ rule: def-equiv.strong-induct)
\begin{enumerate}
\item case $(\text{Q-Refl} \; \Gamma \; t \; T \; \Gamma' \; \theta \; \theta')$
\item have $\Gamma \vdash t : T$
\item and valid $\Gamma'$ by fact
\item moreover have $\Gamma' \vdash \theta \text{ is } \theta' \text{ over } \Gamma$ by fact
\item ultimately show $\Gamma' \vdash \theta < t > \text{ is } \theta' < t > : T$ using fundamental-theorem-1 by blast
\end{enumerate}
\begin{proof} (auto)
\begin{enumerate}
\item case $(\text{Q-Symm} \; \Gamma \; t \; s \; T \; \Gamma' \; \theta \; \theta')$
\item have $\Gamma' \vdash \theta \text{ is } \theta' \text{ over } \Gamma$
\item and valid $\Gamma'$ by fact
\item moreover have $ih : \bigwedge \Gamma' \; \theta \; \theta' \; [\bigwedge \Gamma' \vdash \theta < s > \text{ is } \theta' < t > : T \text{ by fact}
\item ultimately show $\Gamma' \vdash \theta < s > \text{ is } \theta' < t > : T$ using logical-symmetry by blast
\item case $(\text{Q-Trans} \; \Gamma \; s \; t \; u \; \Gamma' \; \theta \; \theta')$
\item have $ih : \bigwedge \Gamma' \; \theta \; \theta' \; [\bigwedge \Gamma' \vdash \theta < s > \text{ is } \theta' < t > : T \text{ by fact}
\item have $ih2 : \bigwedge \Gamma' \; \theta \; \theta' \; [\bigwedge \Gamma' \vdash \theta < u > : T \text{ by fact}
\item have $k : \Gamma' \vdash \theta \text{ is } \theta' \text{ over } \Gamma$
\end{enumerate}
\end{proof} (auto)
and v: valid Γ' by fact
then have Γ' ⊢ θ' is θ' over Γ using logical-pseudo-reflexivity by auto
then have Γ' ⊢ θ'<t> is θ'<u> : T using sh₂ v by auto
moreover have Γ' ⊢ θ<s> is θ'<t> : T using ih₁ h v by auto
ultimately show Γ' ⊢ θ<s> is θ'<t> : T using logical-transitivity by blast
next
case (Q-Abs x Γ T₁ s₂ t₂ T₂ Γ' θ θ')
have fs: x≠Γ by fact
have h₂: Γ' ⊢ θ is θ' over Γ
and h₃: valid Γ' by fact
have ih: [Γ' ⊢ θ, Γ' ⊢ θ' over (x, T₁) ∈ Γ; valid Γ'] ⊢ Γ' ⊢ θ<s₂> is θ'<t₂> : T₂ by fact
{
fix Γ'' s' t'
assume Γ'' ⊑ Γ'' and h₃: Γ'' ⊢ s' is t': T₁ and h₃: valid Γ''
then have Γ'' ⊢ θ is θ' over Γ using h₂ logical-subst-monotonicity by blast
then have Γ'' ⊢ (x, s')#θ is (x, t')#θ' over (x, T₁) # Γ using equiv-subst-ext h₁ l₁ fs by blast
then have Γ'' ⊢ (x, s')#θ<s₂> is (x, t')#θ'<t₂> : T₂ using ih₁ h₆ by blast
then have Γ'' ⊢ θ<s₂>[x::s'] is θ'<t₂>[x::t'] : T₂ using fs₂ psubst-subst-psubst by auto
moreover have App (Lam [x]. θ<s₂>) s' ~ θ<s₂>[x::s']
and App (Lam [x], θ'<t₂>) t' ~ θ'<t₂>[x::t'] by auto
ultimately have Γ'' ⊢ App (Lam [x], θ<s₂>) s' is App (Lam [x], θ'<t₂>) t' : T₂
using logical-weak-head-closure by auto
}
moreover have valid Γ' using h₂ by auto
ultimately have Γ' ⊢ Lam [x]. θ<s₂> is Lam [x] θ'<t₂> : T₁ → T₂ by auto
then show Γ' ⊢ θ<App s₁ s₂> is θ'<App t₁ t₂> : T₂ by auto
next
case (Q-App Γ s₁ t₁ T₁ T₂ s₂ t₂ Γ' θ θ')
then show Γ' ⊢ θ<App s₁ s₂> is θ'<App t₁ t₂> : T₂ by auto
next
case (Q-Beta x Γ s₂ t₂ T₁ s₁₂ t₁₂ T₂ Γ' θ θ')
have k: Γ' ⊢ θ is θ' over Γ
and h*: valid Γ' by fact
have fs: x≠Γ by fact
have fs₂: x≠θ x≠θ' by fact
have ih₁: L¹[Γ' ⊢ θ, Γ' ⊢ θ' over Γ; valid Γ'] ⊢ Γ' ⊢ θ<s₂> is θ'<t₂> : T₁ by fact
have ih₂: L¹[Γ' ⊢ θ, Γ' ⊢ θ' over (x, T₁) # Γ; valid Γ'] ⊢ Γ' ⊢ θ<s₁₂> is θ'<t₁₂> : T₂ by fact
have Γ' ⊢ θ<s₂> is θ'<t₂> : T₁ using ih₁ h₁ h by auto
then have Γ' ⊢ (x, θ<s₂>)#θ is (x, θ'<t₂>)#θ' over (x, T₁) # Γ using equiv-subst-ext h₁ fs by blast
then have Γ' ⊢ (x, θ<s₂>)#θ<s₁₂> is (x, θ'<t₂>)#θ'<t₁₂> : T₂ using ih₂ h₂ by auto
then have Γ' ⊢ θ<s₁₂>[x::θ<s₂>] is θ'<t₁₂>[x::θ'<t₂>] : T₂ using fs₂ psubst-subst-psubst by auto
then have Γ' ⊢ θ<s₁₂>[x::θ<s₂>] is θ'<t₁₂>[x::θ'<t₂>] : T₂ using fs₂ psubst-subst-propagate by auto
moreover have App (Lam [x], θ<s₁₂>) (θ<s₂>) ~ θ<s₁₂>[x::θ<s₂>] by auto
ultimately have Γ' ⊢ App (Lam [x]. θ<s₁₂>) (θ<s₂>) is θ'<t₁₂>[x::t₂] : T₂
using logical-weak-head-closure by auto
then show Γ' ⊢ θ<App (Lam [x], s₁₂) s₂> is θ'<t₁₂>[x::t₂] : T₂ using fs₂ by simp
next
case (Q-Ext x Γ s t T₁ T₂ Γ' θ θ')
have h₂: Γ' ⊢ θ is θ' over Γ

and \(h2": \text{valid } \Gamma' \text{ by fact}\)

have \(fs : x \# \Gamma \ x \# s \ x \# t \) by fact

have \(ih \forall [\Gamma' \ \theta \ \theta'. \ [\Gamma' \vdash \theta \text{ over } (x,T_1) \# \Gamma; \text{ valid } \Gamma']] \implies \Gamma' \vdash \theta < \text{App } s \ (\text{Var } x) > \text{ is } \theta'<\text{App } t \ (\text{Var } x) > : T_2 \) by fact

\{
  \text{fix } \Gamma'' \ s' \ t' \n  \text{assume } \text{hsub: } \Gamma' \subseteq \Gamma'' \text{ and } hl: \Gamma'' \vdash \ s' \text{ is } \ t' : T_1 \text{ and } hk: \text{valid } \Gamma'' \\
  \text{then have } \Gamma'' \vdash \theta \text{ is } \theta' \text{ over } \Gamma \text{ using } \text{h2 logical-subst-monotonicity by blast} \\
  \text{then have } \Gamma'' \vdash (x,s')\#\theta \text{ is } (x,t')\#\theta' \text{ over } (x,T_1)\#\Gamma \text{ using } \text{equiv-subst-ext } hl \text{ fs by blast} \\
  \text{then have } \Gamma'' \vdash (x,s')\#\theta < \text{App } s \ (\text{Var } x) > \text{ is } (x,t')\#\theta'<\text{App } t \ (\text{Var } x) > : T_2 \text{ using } ih \ hk \text{ by blast} \\
  \text{then have } \Gamma'' \vdash \text{App } ((x,s')\#\theta<s>) \ ( ((x,s')\#\theta<(\text{Var } x)>) \text{ is } \text{App } ((x,t')\#\theta'<t>) \ ( (x,t')\#\theta'<(\text{Var } x)>) : T_2 \\
  \text{ by auto} \\
  \text{then have } \Gamma'' \vdash \text{App } ((x,s')\#\theta<s>) \ s' \text{ is } \text{App } ((x,t')\#\theta'<t>) \ t' : T_2 \text{ by auto} \\
  \text{then have } \Gamma'' \vdash \theta < s > \text{ is } \theta' < t > : T_1 \rightarrow T_2 \text{ by auto} \\
  \text{next} \\
  \text{case } (Q-Unit } \Gamma \ s t \Gamma' \theta \theta' \\
  \text{then show } \Gamma'' \vdash \theta < s > \text{ is } \theta' < t > : T\text{Unit by auto} \\
\}

6.6 Completeness

\textbf{theorem} \ \textit{completeness:}

\textit{assumes} \ \textit{asm: } \Gamma \vdash s \equiv t : T \\
\textit{shows} \ \Gamma \vdash s \leftrightarrow t : T \\
\textit{proof} –

\textit{have } \text{val: } \text{valid } \Gamma \text{ using } \text{def-equiv-implies-valid asm by simp} \\
\textit{moreover} \\
\textit{fix } x \ T \\
\textit{assume } (x,T) \in \text{set } \Gamma \text{ valid } \Gamma \\
\textit{then have } \Gamma \vdash \text{Var } x \text{ is } \text{Var } x : T \text{ using } \text{main-lemma(2) by blast} \\
\}

\textit{ultimately have } \Gamma \vdash [] \text{ is } [] \text{ over } \Gamma \text{ by auto} \\
\textit{then have } \Gamma \vdash []<s> \text{ is } []<t> : T \text{ using } \text{fundamental-theorem-2 val asm by blast} \\
\textit{then have } \Gamma \vdash s \leftrightarrow t : T \text{ by simp} \\
\textit{then show } \Gamma \vdash s \leftrightarrow t : T \text{ using } \text{main-lemma(1) val by simp} \\
\}

7 About soundness

We leave soundness as an exercise - like in the book :-)

If \(\Gamma \vdash s \leftrightarrow t : T\) and \(\Gamma \vdash t : T\) and \(\Gamma \vdash s : T\) then \(\Gamma \vdash s \equiv t : T.\)

\([\Gamma \vdash s \leftrightarrow t : T; \ \Gamma \vdash t : T; \ \Gamma \vdash s : T ] \implies \Gamma \vdash s \equiv t : T\)
end
References


