1 Introduction

This article contains a formalization of the strong normalization theorem for the $\lambda_{ml}$-calculus. The formalization is based on a proof by Lindley and Stark [LS05]. An informal description of the formalization can be found in [DS09]. This formalization extends the example proof of strong normalization for the simply-typed $\lambda$-calculus, which is included in the Isabelle distribution [Nom].
The parts of the original proof which have been left unchanged are not displayed in this document.

The next section deals with the formalization of syntax, typing, and substitution. Section 3 contains the formalization of the reduction relation. Section 4 defines stacks which are used to define the reducibility relation in Section 5. The following sections contain proofs about the reducibility relation, ending with the normalization theorem in Section 9.

## 2 The Calculus

**atom-decl** name

**nominal-datatype** trm =

- `Var name`
- `App trm trm`
- `Lam ≪ name ≫ trm (Λ - . - [0,120] 120)`
- `To trm ≪ name ≫ trm (- to - in - [141,0,140] 140)`
- `Ret trm ([·])`

**declare** trm.inject[simp]

**lemmas** name-swap-bij = pt-swap-bij"[OF pt-name-inst at-name-inst]

**lemmas** ex-fresh = exists-fresh"[OF fin-supp]

**lemma** alpha" :

- fixes `x y :: name and t::trm`
- assumes `a: x ̸∈ t`
- shows `[y].t = [x].(((y,x)) · t)`

**proof** –

- from `a` have `aux: y ̸∈ [(y,x)] · t`
- by `(subst fresh-bij[THEN sym, of - - [(x,y)]]))`
- thus `?thesis`
  - by `auto simp add: alpha perm-swap name-swap-bij fresh-bij`

**qed**

Even though our types do not involve binders, we still need to formalize them as nominal datatypes to obtain a permutation action. This is required to establish equivariance of the typing relation.

**nominal-datatype** ty =

- `TBase`
- `TFun ty ty (infix → 200)`
- `T ty`

Since we cannot use typed variables, we have to formalize typing contexts. Typing contexts are formalized as lists. A context is `valid` if no name occurs twice.

**inductive**

- `valid :: (name×ty) list ⇒ bool`

**where**

- `v1[intro]: valid []`
- `v2[intro]: [valid Γ;x#Γ]⇒ valid ((x,σ)#Γ)`

**equivariance** valid
lemma fresh-ty:
  fixes x :: name and τ::ty
  shows x ∉ τ
by (induct τ rule: ty.induct) (auto)

lemma fresh-context:
  fixes Γ :: (name×ty)list
  assumes a: x ∉ Γ
  shows ¬(∃ τ . (x,τ)∈ set Γ)
using a
by (induct Γ) (auto simp add: fresh-prod fresh-list-cons fresh-atm)

inductive typing :: (name×ty) list⇒trm⇒ty⇒bool (- |- : - [60,60,60] 60)
where
  t1[intro]: [valid Γ; (x,τ)∈ set Γ] ⊢ Γ ⊢ Var x : τ
| t2[intro]: [Γ ⊢ s : τ⇒σ; Γ ⊢ t : τ] ⊢ Γ ⊢ App s t : σ
| t3[intro]: x ∉ Γ; ((x,τ)#Γ) ⊢ t : σ] ⊢ Γ ⊢ Λ x . t : τ⇒σ
| t4[intro]: [ Γ ⊢ s : σ ⇒ ] ⊢ Γ ⊢ [s] : T σ
| t5[intro]: [x ∉ Γ; (Γ,s); Γ ⊢ s : T σ ; ((x,σ)#Γ) ⊢ t : T τ ]
  ⊢ Γ ⊢ s to x in t : T τ

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

Except for the explicit requirement that contexts be valid in the variable case and the freshness requirements in t3 and t5, this typing relation is a direct translation of the original typing relation in [LS05] to Curry-style typing.

fun lookup :: (name×trm) list⇒name ⇒ trm
where
  lookup [] x = Var x
| lookup ((y,e)#ϑ) x = (if x=y then e else lookup ϑ x)

lemma lookup-eqv[egv]:
  fixes pi::name prm
  and ϑ:(name×trm) list
  and x::name
  shows pi · (lookup ϑ x) = lookup (pi · ϑ) (pi · x)
by (induct ϑ) (auto simp add: eqvts)

nominal-primrec
  psubst :: (name×trm) list⇒trm⇒trm · (<->) [95,95] 205)
where
  ϑ<Var x>= lookup ϑ x
| ϑ<App s t>= App (ϑ<s>) (ϑ<t>)
| x ∉ ϑ ⇒ ϑ<Λ x . s>= Λ x . (ϑ<s>)
| ϑ<λ t>= [ϑ<t>]
| [x ∉ ϑ ; x ∉ t] ⇒ ϑ<t> to x in s>= (ϑ<t>) to x in (ϑ<s>)
by(finite-guess+ , (simp add: abs-fresh)+ , fresh-guess+)

lemma psubst-eqv[egv]:
  fixes pi::name prm
  shows pi · (ϑ<t>) = (pi · ϑ)<(pi · t)>
abbreviation
  subst :: trm ⇒ name ⇒ trm ⇒ trm (-::=-) [200,100,100] 200
where
  t[x::=t'] ≡ ((x,t'))<t>

lemma subst[simp]:
  shows (Var x)[y::=v] = (if x = y then v else Var x)
  and (App s t)[y::=v] = App (s[y::=v]) (t[y::=v])
  and x♯(y,v) ===> (Λ x . t)[y::=v] = Λ x . t[y::=v]
  and [[s]][y::=v] = [s[y::=v]]
by(simp-all add: fresh-list-cons fresh-list-nil)

lemma subst-rename:
  assumes a: y♯t
  shows (((y,x):t)[y::=v] = t[x::=v]
using a
by(nominal-induct t avoiding: x y v rule: trm.strong-induct)
  (auto simp add: calc-atm fresh-atm abs-fresh fresh-prod fresh-aux)
lemmas subst-rename' = subst-rename[THEN sym]

lemma forget: x♯t ==> t[x::=v] = t
by(nominal-induct t avoiding: x v rule: trm.strong-induct)
  (auto simp add: abs-fresh-fresh-atm)

lemma fresh-fact:
  fixes x::name
  assumes x: x♯v x♯t
  shows x♯t[y::=v]
using x
by(nominal-induct t avoiding: x y v rule: trm.strong-induct)
  (auto simp add: fresh-fresh-atm)

lemma fresh-fact':
  fixes x::name
  assumes x: x♯v
  shows x♯t[x::=v]
using x
by(nominal-induct t avoiding: x v rule: trm.strong-induct)
  (auto simp add: fresh-fresh-atm)

lemma subst-lemma:
  assumes a: x≠y
  and b: x♯u
  shows s[x::=v][y::=u] = s[y::=u][x::=v][y::=u]
using a b
by(nominal-induct s avoiding: x y u v rule: trm.strong-induct)
  (auto simp add: fresh-fact forget)

lemma id-subs:
  shows t[x::=Var x] = t
by(nominal-induct t avoiding: x rule:trm.strong-induct) auto
In addition to the facts on simple substitution we also need some facts on parallel substitution. In particular we want to be able to extend a parallel substitution with a simple one.

**Lemma lookup-fresh:**
- **Fixes** $z :: \text{name}$
- **Assumes** $z \sharp \vartheta \quad z \sharp x$
- **Shows** $z \sharp \text{lookup} \vartheta x$
- **Using** `assms`
- **By** (induct rule: `lookup.induct`)
- (auto simp add: `fresh-list-cons`)

**Lemma lookup-fresh':**
- **Assumes** $a: z \sharp \vartheta$
- **Shows** $\text{lookup} \vartheta z = \text{Var} z$
- **Using** `a`
- **By** (induct rule: `lookup.induct`)
- (auto simp add: `fresh-list-cons fresh-prod fresh-atm`)

**Lemma psubst-fresh-fact:**
- **Fixes** $x :: \text{name}$
- **Assumes** $a: x \not\in t \quad b: x \not\in \vartheta$
- **Shows** $x \not\in \vartheta <t>$
- **Using** `a b`
- **By** (nominal-induct `t` avoiding: $\vartheta \quad x$ rule: `trm.strong-induct`)
- (auto simp add: `lookup-fresh abs-fresh`)

**Lemma psubst-subst:**
- **Assumes** $a: x \not\in \vartheta$
- **Shows** $\vartheta <t>[x::s] = ((x,s)\not\in \vartheta) <t>$
- **Using** `a`
- **By** (nominal-induct `t` avoiding: $\vartheta \quad x \quad s$ rule: `trm.strong-induct`)
- (auto simp add: `fresh-list-cons fresh-atm forget lookup-fresh lookup-fresh' fresh-prod psubst-fresh-fact`)

### 3 The Reduction Relation

With substitution in place, we can now define the reduction relation on $\lambda_{ml}$-terms. To derive strong induction and case rules, all the rules must be vc-compatible (cf. [Urb08]). This requires some additional freshness conditions. Note that in this particular case the additional freshness conditions only serve the technical purpose of automatically deriving strong reasoning principles. To show that the version with freshness conditions defines the same relation as the one without the freshness conditions, we also state this version and prove equality of the two relations.

This requires quite some effort and is something that is certainly undesirable in nominal reasoning. Unfortunately, handling the reduction rule `r10` which rearranges the binding structure, appeared to be impossible without going through this.

**Inductive std-reduction :: trm ⇒ trm ⇒ bool (- ⇒ - [80,80] 80)**

**Where**
- `std-r1[intro]::s ⇒ s'  ⇒⇒ App s t ⇒⇒ App s' t`
- `std-r2[intro]::t ⇒ t'  ⇒⇒ App s t ⇒⇒ App s t'`
In order to show adequacy, the extra freshness conditions in the rules r3, r6, r7, r8, r9, and r10 need to be discharged.

**lemma r3'[intro]:** \( \text{App} (\Lambda x . t) s \rightsquigarrow t[x::=s] \)

**proof**

- **obtain** \( x':\text{name} \) **where** \( s: x' \not\in s\) and \( t: x' \not\in t\)

- **using** \( \text{ex-fresh[of } (s,t)\) by \( (\text{auto simp add: fresh-prod}) \)

- **from** \( t\) **have** \( \text{App} (\Lambda x . t) s = \text{App} (\Lambda x' . \cdot (x,x') \cdot t)) s \)

- **by** \( (\text{simp add: alpha'})\)

- **also from** \( s\) **have** \( \ldots \mapsto \cdot ((x, x') \cdot t)[x':=s] \ldots \)

- **also have** \( \ldots = t[x::=s] \) **using** \( t\)

- **by** \( (\text{auto simp add: subst-rename'})\) \( (\text{metis perm-swap})\)

- **finally show** \( \text{thesis} \).

**qed**

**declare r3'[rule del]**

**lemma r6'[intro]:**

**fixes** \( s:: \text{trm} \)

**assumes** \( r: s \rightsquigarrow s'\)

**shows** \( s \to x \in t \rightsquigarrow s' \to x \in t\)

**using** \( \text{assms} \)
proof –

obtain \( x'::\text{name where } s' :: (s', s) \text{ and } t' :: t \)
using ex-fresh[of \((s', s), t\)] by (auto simp add: fresh-prod)

from \( t \) have \( s \) to \( x \) in \( t = s \) to \( x' \) in \( \langle[x, x'] \cdot t \rangle \)
by (simp add: alpha')

also from \( s \) to \( x \) have \( \ldots \rightarrow s' \) to \( x' \) in \( \langle[x, x'] \cdot t \rangle \) ..
also from \( t \) have \( \ldots = s' \) to \( x \) in \( t \)
by (simp add: alpha')

finally show \(?thesis\).

qed declare r6[rule del]

lemma r7[intro]:

fixes \( t :: \text{trm} \)
assumes \( t \rightarrow t' \)
shows \( s \) to \( x \) in \( t \) \( \rightarrow \) \( s \) to \( x \) in \( t' \)
using assms

proof –

obtain \( x'::\text{name where } f' :: [f', f] \cdot t' \cdot x' :: x \cdot x \)
using ex-fresh[of \( \langle t', t, s \rangle \)] by (auto simp add: fresh-prod)

hence \( \exists \langle s, t \rangle \cdot \langle x, x' \rangle \in \langle[x, x'] \cdot t \rangle \)
by (simp add: eqvts)

from assms have \( \langle[x, x'] \cdot t \rangle \rightarrow \langle[x, x'] \cdot t' \rangle \)
by (simp add: eqvts)

hence \( f \) to \( x \) in \( t' \) \( \rightarrow \) \( s \) to \( x' \) in \( \langle[x, x'] \cdot t \rangle \)
using \( f \) by auto

from \( f \) have \( s \) to \( x \) in \( t' \) \( = \) \( s \) to \( x' \) in \( \langle[x, x'] \cdot t \rangle \)
by (auto simp add: alpha')

with \( a \) \( r \) show \(?thesis\) by (simp del: trm.inject)

qed declare r7[rule del]

lemma r8[intro]: \([s] \) to \( x \) in \( t \) \( \rightarrow \) \( t[x::=s] \)

proof –

obtain \( x'::\text{name where } s' :: [s', s] \cdot x' :: x \)
using ex-fresh[of \( s, t \)] by (auto simp add: fresh-prod)

from \( t \) have \([s] \) to \( x \) in \( t = [s] \) to \( x' \) in \( \langle[x, x'] \cdot t \rangle \)
by (simp add: alpha')

also from \( s \) have \( \ldots \rightarrow \langle[x, x'] \cdot t \rangle[x'::=s] \) ..
also have \( \ldots = t[x::=s] \) using \( t \)
by (auto simp add: subst-rename')(metis perm-swap)

finally show \(?thesis\).

qed declare r8[rule del]

lemma r9[intro]: \( s \) to \( x \) in \([\text{Var } x]\) \( \rightarrow s\)

proof –

obtain \( x'::\text{name where } f' :: [f', f] \cdot x' :: x \)
using ex-fresh[of \( s, x \)] by (auto simp add: fresh-prod)

hence \( s \) to \( x' \) in \([\text{Var } x]\) \( \rightarrow s \) by auto

moreover have \( s \) to \( x' \) in \( \langle[\text{Var } x'] \cdot t \rangle \) \( = \) \( s \) to \( x \) in \( \langle[\text{Var } x] \rangle \)
by (auto simp add: alpha fresh-atm swap-simps)

ultimately show \(?thesis\) by simp

qed declare r9[rule del]
While discharging these freshness conditions is easy for rules involving only one binder it unfortunately becomes quite tedious for the assoc rule $r10$. This is due to the complex binding structure of this rule which includes four binding occurrences of two different names. Furthermore, the binding structure changes from the left to the right: On the left hand side, $x$ is only bound in $t$, whereas on the right hand side the scope of $x$ extends over the whole term $t$ to $y$ in $u$.

**Lemma** $r10$[intro]:

*assumes*** $xf: x \notin y \cdot x \notin u$  
*shows*** $(s \to x \in t) \to y \in u \mapsto s \to x \in (t \to y \in u)$  

**Proof**

- **obtain** $y': \text{name} \;— \;\text{suitably fresh}$
  - **where** $y: y' \notin s \cdot y' \notin x \cdot y' \notin t \cdot y' \notin u$
  - **using** $\text{ex-fresh}[\text{of} \; (s,x,t,u,[[x,x']] \cdot t)]$  
  - **by** (auto simp add: fresh-prod)

- **obtain** $x': \text{name}$
  - **where** $x: x' \notin s \cdot x' \notin y' \cdot x' \notin y \cdot x' \notin t \cdot x' \notin u$  
  - $x' \notin (((y,y')) \cdot u)$
  - **using** $\text{ex-fresh}[\text{of} \; (s,y',y,t,u,[[y,y']] \cdot u)]$  
  - **by** (auto simp add: fresh-prod)

- from $x \cdot y$ have $yax': \text{yxax'} \cdot x' \in t$  
  - **by** (simp add: fresh-left perm-fresh-fresh fresh-atm)

- have $(s \to x \in t) \to y \in u = (s \to x \in t)$ to $y' \in ((y,y') \cdot u)$
  - **using** $(y' \notin w \;\text{by} \;\text{simp add: alpha''})$

- also have $(s \to x' \in (((x,x')] \cdot t)) \to y' \in ((y,y') \cdot u)$
  - **using** $(x' \notin u \;\text{by} \;\text{simp add: fresh-prod})$

- also have $(s \to x' \in (((x,x')] \cdot t) \to y \in u)$
  - **using** $(y' \notin w \;\text{by} \;\text{simp add: abs-fun-eq1 alpha''})$

- also have $(s \to x \in (t \to y \in u)$
  - **proof** (subst trm.inject)

- from $xf' x$ have $\text{swap}: \text{yxax'} \cdot \text{yxax'} \cdot u = u$
  - **by** (auto simp add: fresh-atm perm-fresh-fresh)

- with $x$ show $s = s \land \text{yxax'} \cdot \text{yxax'} \cdot t \to y \in u = [x].t \to y \in u$
  - **by** (auto simp add: alpha''[of $x' \cdot x$] abs-fresh abs-fun-eq1 swap)

**Qed**

**Finally show** $\text{?thesis}$.

**Qed**

**Declare** $r10$[rule del]

Since now all the introduction rules of the vc-compatible reduction relation exactly match their standard counterparts, both directions of the adequacy proof are trivial inductions.

**Theorem** adequacy: $s \mapsto t = s \rightarrow t$

**by** (auto elim:reduction.induct std-reduction.induct)

Next we show that the reduction relation preserves freshness and is in turn preserved under substitution.

**Lemma** reduction-fresh:

- **fixes** $x::\text{name}$
- **assumes** $r: t \mapsto t'$
- **shows** $x \notin t \mapsto x \notin t'$

**Using** $r$
by (nominal-induct t t’ avoiding: x rule: reduction.strong-induct)
   (auto simp add: abs-fresh fresh-fact fresh-atm)

lemma reduction-subst:
  assumes a: t \rightarrow t’
  shows t[x::v] \rightarrow t’[x::v]
using a
by (nominal-induct t t’ avoiding: x v rule: reduction.strong-induct)
   (auto simp add: fresh-atm fresh-fact subst-lemma fresh-prod abs-fresh)

Following [Nom], we use an inductive variant of strong normalization, as it allows for inductive proofs on terms being strongly normalizing, without establishing that the reduction relation is finitely branching.

inductive SN :: trm \Rightarrow bool
where
  SN-intro: (\forall t’. t \rightarrow t’ \Rightarrow SN t’) \Rightarrow SN t

lemma SN-preserved [intro]:
  assumes a: SN t t’ \rightarrow t’
  shows SN t’
using a by (cases) (auto)

definition NORMAL :: trm \Rightarrow bool
where
  NORMAL t \equiv \neg (\exists t’. t \rightarrow t’)

lemma normal-var: NORMAL (Var x)
unfolding NORMAL-def by (auto elim: reduction.cases)

lemma normal-implies-sn: NORMAL s \Rightarrow SN s
unfolding NORMAL-def by (auto intro: SN-intro)

4 Stacks

As explained in [LS05], the monadic type structure of the \(\lambda_{ml}\)-calculus does not lend itself to an easy definition of a logical relation along the type structure of the calculus. Therefore, we need to introduce stacks as an auxiliary notion to handle the monadic type constructor \(T\). Stacks can be thought of as lists of term abstractions \([x].t\). The notation for stacks is chosen with this resemblance in mind.

nominal-datatype stack = Id | St <name> trm stack ([]-\Rightarrow-)

lemma stack-exhaust:
  fixes c :: 'a:fs-name
  shows k = Id \lor (\exists y n l . y \# l \land y \# c \land k = [y]n\Rightarrow l)
by (nominal-induct k avoiding: c rule: stack.strong-induct) (auto)

nominal-primrec
  length :: stack \Rightarrow nat ( |-)
where
Together with the stack datatype, we introduce the notion of dismantling a term onto a stack. Unfortunately, the dismantling operation has no easy primitive recursive formulation. The Nominal package, however, only provides a recursion combinator for primitive recursion. This means that for dismantling one has to prove pattern completeness, right uniqueness, and termination explicitly.

function dismantle :: trm ⇒ stack ⇒ trm (- * - [160,160] 160)
where
t * Id = t |
  x :: (K,t) ⇒ t * ([x]s⇒K) = (t to x in s) * K
proof — — pattern completeness
  fix P :: bool and arg:trm × stack
  assume id:  \( \forall t. \; arg = (t, stack.Id) ⇒ P \)
  and st:  \( \forall K t s. \; [x :: (K, t); \; arg = (t, [x]s⇒K)] ⇒ P \)
  {  assume snd arg = Id
      hence P by (metis id[where t=fst arg] surjective-pairing) }
  moreover
  {  fix y n L assume snd arg = [y]n⇒L  y :: (L, fst arg)
      hence P by (metis st[where t=fst arg] surjective-pairing) }
ultimately show P using stack-exhaust[af  snd arg  fst arg] by auto
next
— right uniqueness
— only the case of the second equation matching both args needs to be shown.
  fix t t' :: trm and x x' :: name and s s' :: trm and K K' :: stack
  let ?g = dismantle-sumC — graph of dismantle
  assume x :: (K, t)  x' :: (K', t')
  and (t, [x]s⇒K) = (t', [x']s⇒K')
  thus ?g (t to x in s, K) = ?g (t' to x' in s', K')
  by (auto intro!: arg-cong[where f=?g] simp add: stack.inject)
qed (simp-all add: stack.inject) — all other cases are trivial

termination dismantle
by(relation measure (λ(t,K).  |K| ))(auto)

Like all our constructions, dismantling is equivariant. Also, freshness can be pushed over dismantling, and the freshness requirement in the second defining equation is not needed

lemma dismantle-eqvt[eqvt]:
  fixes pi :: (name × name) list
  shows pi · (t * K) = (pi · t) * (pi · K)
by(nominal-induct K avoiding: pi t rule:stack.strong-induct)
  (auto simp add: eqvts fresh-bij)

lemma dismantle-fresh[iff]:
  fixes x :: name
  shows (x :: (t * k)) = (x :: t ∧ x :: k)
by(nominal-induct k avoiding: t x rule: stack.strong-induct)
  (simp-all)
lemma dismantle-simp[simp]: \( s \star [y]\mapsto n \mapsto L = (s \text{ to } y \text{ in } n) \star L \)

proof

obtain \( x :: \text{name} \) where \( f : x \not\in s \quad x \not\in L \quad x \not\in n \)
using ex-fresh[of \((s, L, n)\)] by (auto simp add: fresh-prod)

hence \( t : s \text{ to } y \text{ in } n = s \text{ to } x \text{ in } ([\langle y, x \rangle] \cdot n) \)
by (auto simp add: alpha"")

from \( f \) have \( [y]\mapsto n \mapsto L = [x][\langle (y, x) \rangle \cdot n] \mapsto L \) by simp

also have \( \ldots = (s \text{ to } y \text{ in } n) \star L \) using \( f \) by (simp del: trm.inject)

thus \( s \star [y]\mapsto n \mapsto L = (s \text{ to } x \text{ in } ([\langle y, x \rangle] \cdot n) \mapsto L) \)
by (rule reduction-fresh)

finally show \( \text{thesis} \).

qed

We also need a notion of reduction on stacks. This reduction relation allows us to define strong normalization not only for terms but also for stacks and is needed to prove the properties of the logical relation later on.

definition stack-reduction :: \( \text{stack} \Rightarrow \text{stack} \Rightarrow \text{bool} \)

where \( k \mapsto \mapsto k' \equiv \forall (t :: \text{trm}). (t \star k) \mapsto \mapsto (t \star k') \)

lemma stack-reduction-fresh:

fixes \( k :: \text{stack} \) and \( x :: \text{name} \)

assumes \( r : k \mapsto \mapsto k'\) and \( f : x \not\in k \)

shows \( x \not\in k' \)

proof

from \( ex\text{-fresh[of } x \) obtain \( z :: \text{name} \) where \( f' : z \not\in x \).

from \( r \) have \( \text{Var } z \star k \mapsto \mapsto \text{Var } z \star k'\) unfolding stack-reduction-def..

moreover from \( f \) have \( x \not\in \text{Var } z \star k \) by (auto simp add: fresh-atm)

ultimately have \( x \not\in \text{Var } z \star k' \) by (rule reduction-fresh)

thus \( x \not\in k' \) by simp

qed

lemma dismantle-red[intro]:

fixes \( m :: \text{trm} \)

assumes \( r : m \mapsto m' \)

shows \( m \star k \mapsto \mapsto m' \star k \)

using \( r \)

by (nominal-induct \( k \) avoiding: \( m \) \( m' \) rule: stack.strong-induct) auto

Next we define a substitution operation for stacks. The main purpose of this is to distribute substitution over dismantling.

nominal-primrec

\( \text{sssubst} :: \text{name} \Rightarrow \text{trm} \Rightarrow \text{stack} \Rightarrow \text{stack} \)

where

\( \text{sssubst } x \ v \ \text{Id} = \text{Id} \)

| \( y \not\in (k, x, v) \) \( \implies \) \( \text{sssubst } x \ v \ ([y]\mapsto n \mapsto k) = [y][n[x := v]] \mapsto (\text{sssubst } x \ v) k \)

by (finite-guess+, (simp add: abs-fresh)+, fresh-guess+)

lemma sssubst-fresh:

fixes \( y :: \text{name} \)

assumes \( y \not\in (x, v, k) \)

shows \( y \not\in \text{sssubst } x \ v \)

using \( \text{assms} \)

by (nominal-induct \( k \) avoiding: \( y \ x \ v \) rule: stack.strong-induct)
(auto simp add: fresh-prod fresh-atm abs-fresh fresh-fact)

**lemma** ssusb-forgot:
fixes x :: name
assumes x δ k
shows ssusb x v k = k
using assms
by(nominal-induct k avoiding: x v rule: stack.strong-induct) (auto simp add: abs-fresh fresh-atm forget)

**lemma** subst-dismantle[simp]: (t ⋆ k)[x ::= v] = (t[x ::= v]) ⋆ ssusb x v k
by(nominal-induct k avoiding: t x v rule: stack.strong-induct) (auto simp add: ssusb-fresh fresh-prod fresh-fact)

### 5 Reducibility for Terms and Stacks

Following [Nom], we formalize the logical relation as a function RED of type \( ty \Rightarrow trm \ set \) for the term part and accordingly SRED of type \( ty \Rightarrow stack \ set \) for the stack part of the logical relation.

**lemma** ty-exhaust: ty = TBase \( \lor (\exists \sigma \tau \ . \ ty = \sigma \rightarrow \tau) \lor (\exists \sigma \ . \ ty = T\sigma) \)
by(induct ty rule: ty.induct) (auto)

**function** RED :: ty \( \Rightarrow \) trm set
and SRED :: ty \( \Rightarrow \) stack set
where

\[
\begin{align*}
RED (TBase) &= \{t. SN(t)\} \\
RED (\sigma \rightarrow \tau) &= \{t. \forall u \in RED \tau \ . \ (App t u) \in RED \sigma\} \\
RED (T \tau) &= \{t. \forall k \in SRED \sigma \ . \ SN(t \star k)\} \\
SRED \tau &= \{k. \forall t \in RED \tau \ . \ SN ([t] \star k)\}
\end{align*}
\]

by(auto simp add: ty.inject, case-tac x rule: sum.exhaust,insert ty-exhaust) (blast)+

This is the second non-primitive function in the formalization. Since types do not involve binders, pattern completeness and right uniqueness are mostly trivial. The termination argument is not as simple as for the dismantling function, because the definition of SRED \( \tau \) involves a recursive call to RED \( \tau \) without reducing the size of \( \tau \).

**nominal-primrec**

tsize :: ty \( \Rightarrow \) nat
where
\[
\begin{align*}
tsize TBase &= 1 \\
tsize (\sigma \rightarrow \tau) &= 1 + tsize \sigma + tsize \tau \\
tsize (T \tau) &= 1 + tsize \tau
\end{align*}
\]
by(rule TrueI)+

In the termination argument below, Inl \( \tau \) corresponds to the call RED \( \tau \), whereas Inr \( \tau \) corresponds to SRED \( \tau \)

**termination** RED
by(relation measure
\[
(\lambda x . \ case x of Inl \tau \Rightarrow 2 \ast tsize \tau \\
| Inr \tau \Rightarrow 2 \ast tsize \tau + 1)) \ (auto)
\]
6 Properties of the Reducibility Relation

After defining the logical relations we need to prove that the relation implies strong normalization, is preserved under reduction, and satisfies the head expansion property.

**definition** NEUT :: trm ⇒ bool
where
NEUT t ≡ (∃ a. t = Var a) ∨ (∃ t1 t2. t = App t1 t2)

**definition** CR1 :: ty ⇒ bool
where
CR1 τ ≡ ∀ t. (t ∈ RED τ → SN t)

**definition** CR2 :: ty ⇒ bool
where
CR2 τ ≡ ∀ t t'. (t ∈ RED τ ∧ t → t') → t' ∈ RED τ

**definition** CR3-RED :: trm ⇒ ty ⇒ bool
where
CR3-RED t τ ≡ ∀ t'. t ↦→ t' → t' ∈ RED τ

**definition** CR3 :: ty ⇒ bool
where
CR3 τ ≡ ∀ t. (NEUT t ∧ CR3-RED t τ) → t ∈ RED τ

**definition** CR4 :: ty ⇒ bool
where
CR4 τ ≡ ∀ t. (NEUT t ∧ NORMAL t) → t ∈ RED τ

**lemma** CR3-implies-CR4 [intro]: CR3 τ =⇒ CR4 τ
by (auto simp add: CR3-def CR3-RED-def CR4-def NORMAL-def)

**inductive** FST :: trm ⇒ trm ⇒ bool
where
fst[intro!]: (App t s) ⇒ t

**lemma** SN-of-FST-of-App:
assumes a: SN (App t s)
shows SN t
proof –
from a have ∀ z. (App t s ⇒ z) → SN z
by (induct rule: SN.induct)
(blast elim: FST.cases intro: SN-intro)
then show SN t by blast
qed

The lemma above is a simplified version of the one used in [Nom]. Since we have generalized our notion of reduction from terms to stacks, we can also generalize the notion of strong normalization. The new induction principle will be used to prove the T case of the properties of the reducibility relation.

**inductive** SSN :: stack ⇒ bool
where

\[ \text{SSN-intro: } (\forall k'. k \mapsto k' \Rightarrow \text{SSN } k') \Rightarrow \text{SSN } k \]

Furthermore, the approach for deriving strong normalization of subterms from above can be generalized to terms of the form \( t \ast k \). In contrast to the case of applications, \( t \ast k \) does not uniquely determine \( t \) and \( k \). Thus, the extraction is a proper relation in this case.

\[ \text{inductive} \]

\[ \text{SND-DIS :: } \text{trm} \Rightarrow \text{stack} \Rightarrow \text{bool} \ (\cdot \triangleright \cdot) \]

where

\[ \text{snd-dis[intro]}: \ t \ast k \triangleright k \]

**lemma** **SN-SSN:**

**assumes** \( a: \text{SN } (t \ast k) \)

**shows** \( \text{SSN } k \)

**proof**

- from \( a \) have \( \forall z. (t \ast k \triangleright z) \Rightarrow \text{SSN } z \) by (induct rule: SN.induct)

  (metis SND-DIS.cases SSN-intro snd-dis stack-reduction-def)

  thus \( \text{SSN } k \) by blast

**qed**

To prove CR1-3, the authors of [LS05] use a case distinction on the reducts of \( t \ast k \), where \( t \) is a neutral term and therefore no interaction occurs between \( t \) and \( k \).

\[
\begin{array}{c}
t \ast k \mapsto r \\
\land t' \cdot [t \mapsto t'; r = t' \ast k] \Rightarrow P \\
\land k' \cdot [k \mapsto k'; r = t \ast k'] \Rightarrow P \\
\end{array}
\]

\[
\text{NEUT } t \\
\text{P}
\]

We strive for a proof of this rule by structural induction on \( k \). The general idea of the case where \( k = [y]n \gg l \) is to move the first stack frame into the term \( t \) and then apply the induction hypothesis as a case rule. Unfortunately, this term is no longer neutral, so, for the induction to go through, we need to generalize the claim to also include the possible interactions of non-neutral terms and stacks.

**lemma** **dismantle-cases:**

**fixes** \( t :: \text{trm} \)

**assumes** \( r: t \ast k \mapsto r \)

and \( T: \land t' \cdot [t \mapsto t'; r = t' \ast k] \Rightarrow P \)

and \( K: \land k' \cdot [k \mapsto k'; r = t \ast k'] \Rightarrow P \)

and \( B: \land s y n l \cdot [t = [s]; k = [y]n \gg l; r = (n[y::=s]) \ast l] \Rightarrow P \)

and \( A: \land u x v y n l \cdot [x \not\equiv y; x \not\equiv n; t = u \to x \in v; k = [y]n \gg l; r = (u \to x \in (v \to y \in n)) \ast l] \Rightarrow P \)

**shows** \( P \)

**using** \( \text{assms} \)

**proof** (nominal-induct \( k \) avoiding: \( t \ast k \mapsto r \) rule:stack.strong-induct)

- case \( (\text{St } y n l) \) **note** \( \text{yfresh } = \langle y \not\equiv t \rangle \langle y \not\equiv r \rangle \langle y \not\equiv l \rangle \)

**note** \( IH = \text{St}(\{\}) \)

- and \( T = \text{St}(6) \) and \( K = \text{St}(7) \) and \( B = \text{St}(8) \) and \( A = \text{St}(9) \)

**thus** \( P \) **proof** (cases rule:IH[where \( b=t \) to \( y \in n \) and \( ba=r \)])

- case \( (\not\equiv r') \) **have** \( r \). \( t \to y \in n \mapsto r' \) **and** \( r': r \ast L \) by fact+
If $m$ to $y$ in $n$ makes a step we reason by case distinction on the successors of $m$ to $y$ in $n$. We want to use the strong inversion principle for the reduction relation. For this we need that $y$ is fresh for $t$ to $y$ in $n$ and $r'$.

from yfresh $r$ have $y$ $\not\in$ $t$ to $y$ in $n$ $y$ $\not\in$ $r'$
  by (auto simp add: abs-fresh)

obtain $z$ where $z$: $z$ $\not\in$ $y$ $z$ $\not\in$ $r'$ $z$ $\not\in$ $t$ to $y$ in $n$
  using ex-fresh[of $(y,r',t$ to $y$ in $n)$]
  by (auto simp add: fresh-prod fresh-atm)

from red $r$ show $P$
proof (cases rule: reduction.strong-cases)
  [ where $x$: $x$ $\not\in$ $y$ $x y$ $\not\in$ $z$ $x$ $\not\in$ $r'$ $x y$ $\not\in$ $z$ $\not\in$ $t$ to $y$ in $n$]

  case $(r$ $t$ $u)$ — if $t$ makes a step we use assumption $T$
   with $y$ have $m$: $t$ $\mapsto$ $t'$ $r' = t'$ to $y$ in $n$ by auto
   thus $P$ using $T$[of $t'$] $r$ by auto

next

  case $(r$ $t$ $u)$ with $y$ have $n$: $n$ $\mapsto$ $n'$ and $r'$: $r' = t$ to $y$ in $n'$
   by (auto simp add: alpha)

Since $k = [y]n\gg L$, the reduction $n \mapsto n'$ occurs within the stack $k$. Hence, we need to establish this stack reduction.

have $[y]n\gg L \mapsto [y]n'\gg L$
  unfolding stack-reduction-def

proof
  fix $u$ have $u$ to $y$ in $n$ $\mapsto$ $u$ to $y$ in $n'$ using $n$ ..
  hence $(u$ to $y$ in $n$) $\star$ $L$ $\mapsto$ $(u$ to $y$ in $n')$ $\star$ $L$ ..
  thus $u$ $\star$ $[y]n\gg L$ $\mapsto$ $u$ $\star$ $[y]n'\gg L$
    by simp

qed

moreover have $r = t$ $\star$ $[y]n'\gg L$ using $r$ $r'$ by simp
ultimately show $P$ by (rule $K$)

next

  case $(r$ $t$ $u)$ — the case of a $\beta$-reduction is exactly $B$
  with $y$ have $t$: $[s]$ $r' = n[y := s]$ by (auto simp add: alpha)
  thus $P$ using $B$[of $s$ $y$ $n$ $L$] $r$ by auto

next

  case $(r$ $t$ $u)$ — The case of an $\eta$-reduction is a stack reduction as well.
  with $y$ have $n$: $n = [\Var y]$ and $r'$: $r' = t$
    by (auto simp add: alpha)
  { fix $u$ have $u$ to $y$ in $n$ $\mapsto$ $u$ unfolding $n$ ..
    hence $(u$ to $y$ in $n$) $\star$ $L$ $\mapsto$ $u$ $\star$ $L$ ..
    hence $u$ $\star$ $[y]n\gg L$ $\mapsto$ $u$ $\star$ $L$ by simp
  }

  hence $[y]n\gg L \mapsto L$
  unfolding stack-reduction-def ..
moreover have $r = t$ $\star$ $L$ using $r$ $r'$ by simp
ultimately show $P$ by (rule $K$)

next

  case $(r$ $t$ $u$ $v)$ — The assoc case holds by $A$.
  with $y$ $z$ have
    $l = (u$ to $z$ in $v)$
    $r' = u$ to $z$ in $(v$ to $y$ in $n)$
    $z$ $\not\in$ $(y,n)$ by (auto simp add: fresh-prod alpha)
    thus $P$ using $A$[of $z$ $y$ $n$] $r$ by auto

  qed (insert $y$, auto) — No other reductions are possible.

next

Next we have to solve the case where a reduction occurs deep within $L$. We get a reduction of the stack $k$ by moving the first stack frame “[y]n” back to the right hand
side of the dismantling operator.

case (\$L^r\$)
  hence \(L \Rightarrow L'\) and \(r: r = (t \times y \in n) \ast L'\) by auto
  \{ fix \; s \; from \; L \; have \; (s \times y \in n) \ast L \Rightarrow (s \times y \in n) \ast L' \}
  unfolding stack-reduction-def  ..
  hence \(s \ast \{y\} n \gg \Rightarrow L \Rightarrow s \ast \{y\} n \gg \Rightarrow L'\) by simp
  \} hence \(\{y\} n \gg \Rightarrow L \Rightarrow \{y\} n \gg \Rightarrow L'\) unfolding stack-reduction-def by auto
next
  case (\$z \; x \; n' \; s \; v \; K\$) — The “assoc” case is again a stack reduction
  have \(x f: x \notin z \times x \notin n'\)
  — We get the following equalities
  and red: \(t \times y \in n = s \times x \in v\)
  \(L = \{z\} n \gg \Rightarrow K\)
  \(r = (s \times t \in v \times z \in n') \ast K\) by fact+
  \{ fix \; u \; from \; red \; have \; u \ast \{y\} n \gg \Rightarrow L = ((u \times t \in v) \times z \in n') \ast K \\
  by(auto intro: arg-cong[where \(f = \lambda x . x \ast K\)])
  moreover
  \{ from \; x f \; have \; (u \times t \in v) \times z \in n' \Rightarrow u \times t \in v \times v \times z \in n') ..
  hence \(((u \times t \in v) \times z \in n') \ast K \Rightarrow (u \times t \in v \times v \times z \in n') \ast K \)
  by rule
  \} ultimately have \(u \ast \{y\} n \gg \Rightarrow L \Rightarrow (u \times t \in v \times v \times z \in n') \ast K\)
  by (simp (no-asm-simp) del:dismantle-simp)
  hence \(u \ast \{y\} n \gg \Rightarrow u \ast \{x\} (v \times z \times n') \gg \Rightarrow K\) by simp
  \} hence \(\{y\} n \gg \Rightarrow \{x\} (v \times z \times n') \gg \Rightarrow K\)
  unfolding stack-reduction-def by simp
  moreover have \(r = t \ast \{x\} (v \times z \times n') \gg \Rightarrow K\) using red
  by (auto)
  ultimately show \(P\) by (rule K)
  qed (insert St, auto)
qed auto

Now that we have established the general claim, we can restrict \(t\) to neutral terms only and drop the cases dealing with possible interactions.

lemma dismantle-cases:"[consumes 2, case-names T K]:
fixes \(m :: \text{trm}\)
assumes \(r: t \times k \mapsto r\)
and \(\text{NEUT} \; t\)
and \(\{ \; t' \; . \; \{ t \mapsto t' ; r = t' \times k \} \mapsto P \; \}
and \(\{ k' \; . \; \{ k \mapsto k' ; r = t \times k' \} \mapsto P \; \)
shows \(P\)
using assms unfolding NEUT-def
by (cases rule: dismantle-cases[of \(t \; k \; r\)]) (auto)

lemma red-Ret:
fixes \(t :: \text{trm}\)
assumes \([s] \mapsto t\)
shows \(\exists \; s', \; t = [s'] \land s \mapsto s'\)
using assms by cases (auto)

lemma SN-Ret: SN \(u \Rightarrow SN \; [u]\)
by(induct rule:SN.induct) (metis SN.intros red-Ret)

All the properties of reducibility are shown simultaneously by induction on
the type. Lindley and Stark [LS05] only spell out the cases dealing with the monadic type constructor $T$. We do the same by reusing the proofs from [Nom] for the other cases. To shorten the presentation, these proofs are omitted.

**Lemma RED-props:**
- shows $CR1 \tau$ and $CR2 \tau$ and $CR3 \tau$

**Proof**

- **case** $T$Base
  - **case** $(TFun \tau 1 \tau 2)$next
  - **case** $(T \sigma)$
    - **case 1** — follows from the fact that $stack.Id \in SRED \sigma$
      - have $ih-\text{CR1-}^{-}\sigma$: $CR1 \sigma$ by fact
      - fix $t$ assume $t-red$: $t \in RED (T \sigma)$
        - fix $s$ assume $s \in RED \sigma$
          - hence $SN s$ using $ih-\text{CR1-}^{-}\sigma$ by (auto simp add: CR1-def)
            - hence $SN ([s])$ by (rule $SN-Ret$)
          - hence $SN ([s] * Id)$ by simp
          - hence $Id \in SRED \sigma$ by simp
        - with $t-red$ have $SN (t)$ by (auto simp del: SRED.simps)
    - thus $CR1 (T \sigma)$ unfolding $CR1-def$ by blast
  - **next**
    - **case 2** — follows since $SN$ is preserved under reduction
      - fix $t$ assume $t'-red$: $t' \in RED (T \sigma)$ and $t-t': t \mapsto t'$
      - fix $k$ assume $k$: $k \in SRED \sigma$
        - with $t-red$ have $SN(t * k)$ by simp
        - moreover from $t-t'$ have $t * k \mapsto t' * k$ .
          - ultimately have $SN(t' * k)$ by (rule $SN-preserved$)
      - hence $t' \in RED (T \sigma)$ by (simp del: SRED.simps)
        - thus $CR2 (T \sigma)$ unfolding $CR2-def$ by blast
  - **next**
    - **case 3** from $(CR3 \sigma)$ have $ih-\text{CR4-}^{-}\sigma$: $CR4 \sigma$ ..
      - fix $t$ assume $t'-red$: $\bigwedge t', t \mapsto t' \implies t' \in RED (T \sigma)$
        - and $neut-t$: $\text{NEUT } t$
      - fix $r$ have $\text{NEUT } (\forall x) $ unfolding $\text{NEUT-def}$ by simp
        - hence $\forall x \in RED \sigma$ using $\text{normal-var } ih-\text{CR4-}\sigma$
          - by (simp add: CR4-def)
        - hence $SN ([\forall x] * k)$ using $k-red$ by simp
        - hence $SSN k$ by (rule $SN-SSN$)
      - then have $SN (t * k)$ using $k-red$
        - **proof** (induct $k$ rule:$SSN.induct$)
          - case $(SSN-intro k)$
            - have $ih : [k' ; [k \mapsto k'; k' \in SRED \sigma ] \implies SN (t * k')$
              - and $k-red$: $k \in SRED \sigma$ by fact+
            - fix $r$ assume $r: t * k \mapsto r$
              - hence $SN r$ using $\text{neut-t}$
                - **proof** (cases rule:$\text{dismantle-cases}$)
                  - case $(T t')$ hence $t-t': t \mapsto t'$ and $r-def$: $r = t' * k$ .
                    - from $t-t'$ have $t' \in RED (T \sigma)$ by (rule $t'-\text{red}$)
                      - thus $SN r$ using $k-red$ $r-def$ by simp
                - **next**
                  - case $(K k')$ hence $k-k': k \mapsto k'$ and $r-def$: $r = t * k'$ .
                    - fix $s$ assume $s \in RED \sigma$
                      - hence $SN ([s] * k)$ using $k-red$
                        - by simp
                      - moreover have $[s] * k \mapsto [s] * k'$
Let $t$ be neutral such that $t' \in RED_{T\sigma}$ whenever $t \mapsto t'$. We have to show that $(t \star k)$ is $SN$ for each $k \in SRED_{\sigma}$. First, we have that $[x] \star k$ is $SN$, as $x \in RED_{\sigma}$ by the induction hypothesis. Hence $k$ itself is $SN$, and we can work by induction on $\max(k)$. Application $t \star k$ may reduce as follows:

- $t' \star k$, where $t \mapsto t'$, which is $SN$ as $k \in SRED_{\sigma}$ and $t' \in RED_{T\sigma}$.
- $t \star k'$, where $k \mapsto k'$. For any $s \in RED_{\sigma}$, $[s] \star k$ is $SN$ as $k \in SRED_{\sigma}$; and $[s] \star k \mapsto [s] \star k'$, so $[s] \star k'$ is also $SN$. From this we have $k' \in SRED_{\sigma}$ with $\max(k') < \max(k)$, so by induction hypothesis $t \star k'$ is $SN$.

There are no other possibilities as $t$ is neutral. Hence $t \star k$ is strongly normalizing for every $k \in SRED_{\sigma}$, and so $t \in RED_{T\sigma}$ as required.

Figure 1: Proof of the case $T\sigma$ subcase CR3 as in [LS05]

```
using k-k' unfolding stack-reduction-def .. 
ultimately have SN ([s] \star k') .. 
} hence k' \in SRED \sigma by simp  
with k-k' show SN r unfolding r-def by (rule ih) 
qed } thus SN (t \star k) .. 
qed } hence t \in RED (T \sigma) by simp 
} thus CR3 (T \sigma) unfolding CR3-def CR3-RED-def by blast 
}
qed
```

The last case above shows that, once all the reasoning principles have been established, some proofs have a formalization which is amazingly close to the informal version. For a direct comparison, the informal proof is presented in Figure 1.

Now that we have established the properties of the reducibility relation, we need to show that reducibility is preserved by the various term constructors. The only nontrivial cases are abstraction and sequencing.

7 Abstraction Preserves Reducibility

Once again we could reuse the proofs from [Nom]. The proof uses the double-$SN$ rule and the lemma $red-Lam$ below. Unfortunately, this time the proofs are not fully identical to the proofs in [Nom] because we consider $\beta\eta$-reduction rather than $\beta$-reduction only. However, the differences are only minor.

```
lemma double-SN[concludes 2]:
  assumes a: SN a
  and b: SN b
  and c: \forall(x::trm) (z::trm).
            [\forall y. x \mapsto y \implies P y z; \forall u. z \mapsto u \implies P x u] \implies P x z
  shows P a b c
using a b c
```
lemma red-Lam:
assumes a: Λ x . t ↦→ r
shows (∃t'. r = Λ x . t' ↦→ t' ) ∨ (t = App r (Var x) ∧ x ∉ r)
proof
obtain z::name where z :: z ∉ x z ∉ t z ∉ r
using ex-fresh[of (x,t,r)] by (auto simp add: fresh-prod)
have x ∉ Λ x . t by (simp add: abs-fresh)
with a have x ∉ r by (simp add: reduction-fresh)
with a show ?thesis using z
by (cases rule: reduction-strong-cases
[where x =x and xa=x and xb=x and xc=x and xd=x and xe=x and xf=x and xg=x and y=z])
(auto simp add: abs-fresh alpha fresh-atm)
qed

lemma abs-RED:
assumes asm: ∀ s ∈ RED τ. t[x::=s] ∈ RED σ
shows Λ x . t ∈ RED (τ→σ)

8 Sequencing Preserves Reducibility

This section corresponds to the main part of the paper being formalized and as such deserves special attention. In the lambda case one has to formalize doing induction on max(s) + max(t) for two strongly normalizing terms s and t (cf. [GTL89, Section 6.3]). Above, this was done through a double-SN rule. The central Lemma 7 of Lindley and Stark’s paper uses an even more complicated induction scheme. They assume terms p and n as well as a stack K such that SN p and SN (n[x::=p] ⋆ K). The induction is then done on |K| + max(n ⋆ K) + max(p). See Figure 2 in for details.

Since we have settled for a different characterization of strong normalization, we have to derive an induction principle similar in spirit to the double-SN rule. Furthermore, it turns out that it is not necessary to formalize the fact that stack reductions do not increase the length of the stack. Doing induction on the sum above, this is necessary to handle the case of a reduction occurring in K. We differ from [LS05] and establish an induction principle which to some extent resembles the lexicographic order on

\((SN, \mapsto) \times (SN, \mapsto) \times (N, >)\).

lemma triple-induct[consumes 2]:
assumes a: SN (p)
and b: SN (q)
and hyp: ⋀ (p::trm) (q::trm) (k::stack) .
\[ ⋀ p' . p ↦→ p' \Longrightarrow P p' q k ; \]
\[ ⋀ q' . q ↦→ q' \Longrightarrow P p q' k ; \]
\[ ⋀ k' . |k'| < |k| \Longrightarrow P p q k' \] \Longrightarrow P p q k
shows P p q k
proof

\footnote{This possibility was only discovered after having formalized K → K' ⇒ |K| ≥ |K'|. The proof of this seemingly simple fact was about 90 lines of Isar code.}
Lemma 8.1. (Lemma 7) Let \( p, n \) be terms and \( K \) a stack such that \( SN(p) \) and \( SN(n[x := p] * K) \). Then \( SN(([p] \text{ to } x \text{ in } n) * K) \)

Proof. We show by induction on \( |K| + \max(n * K) + \max(p) \) that the reducts of \(([p] \text{ to } x \text{ in } n) * K \) are all strongly normalizing. The interesting reductions are as follows:

- \( T.\beta \) giving \( n[x := p] * K \) which is strongly normalizing by hypothesis.
- \( T.\eta \) when \( n = [x] \) giving \( [p] * K \). But \( [p] * K = n[x := p] * K \) which is again strongly normalizing by hypothesis.
- \( T.assoc \) in the case where \( K = [y]m \gg K' \) with \( x \notin fv(m) \); giving the reduct \(([p] \text{ to } x \text{ in } (n \text{ to } y \text{ in } m)) * K \). We aim to apply the induction hypothesis with \( K' \) and \( (n \text{ to } y \text{ in } m) \) for \( K \) and \( n \) respectively. Now

\[
(n \text{ to } y \text{ in } m)[x := p] * K' = (n[x := p] \text{ to } y \text{ in } m) * K'
\]

\[
= n[x := p] * K
\]

which is strongly normalizing by induction hypothesis. Also

\[
|K'| + \max((n \text{ to } y \text{ in } m) * K') + \max(p) < |K| + \max(n * K) + \max(p)
\]

as \( |K'| < |K| \) and \( (n \text{ to } y \text{ in } m) * K' = n * K \). This last equation explains the use of \( \max(n * K) \); it remains fixed under \( T.assoc \) unlike \( \max(K) \) and \( \max(n) \). Applying the induction hypothesis gives \( SN(([p] \text{ to } x \text{ in } (n \text{ to } y \text{ in } m)) * K) \) as required.

Other reductions are confined to \( K, n \) or \( p \) and can be treated by the induction hypothesis, decreasing either \( \max(n * K) \) or \( \max(p) \).
from a have \( \land q \) \( . \) \( SN q \iff P \ p q K \)  

proof (induct \( p \))  

\( \text{case (SN-intro \( p \))} \)  

have sn1: \( \land q' \) \( . \) \( [p \mapsto p'; SN q] \iff P \ p' q K \) by fact  

have sn-q: \( SN q \) \( \iff \) \( SN q \) by fact+  

thus \( P \ p q K \)  

proof (induct \( q \) arbitrary: \( K \))  

\( \text{case (SN-intro \( q K \))} \)  

have sn2: \( \land q' K \) \( . \) \( [q \mapsto q'; SN q'] \iff P \ p q' K \) by fact  

show \( P \ p q K \)  

proof (induct \( K \) rule: measure-induct-rule[where \( f=\text{length} \)])  

\( \text{case (less \( k \))} \)  

have le: \( \land k' \) \( . \) \( |k'| < |k| \implies P \ p q k \) by fact  

\{ fix \( p' \) assume \( p \mapsto p' \) moreover have \( SN q \) by fact  

ultimately have \( P \ p' q k \) using sn1 by auto \}  

moreover  

\{ fix \( q' K \) assume \( r: q \mapsto q' \)  

have \( SN q \) by fact  

hence \( SN q' \) using \( r \) by (rule \( SN\)-preserved)  

with \( r \) have \( P \ p q' K \) using sn2 by auto \}  

ultimately show \( ?\text{case using le} \)  

by (auto intro: hyp)  

qed  

qed  

qed  

with \( b \) show \( ?\text{thesis by blast} \)  

qed  

Here we strengthen the case rule for terms of the form \( t \times k \mapsto r \). The freshness requirements on \( x,y, \) and \( z \) correspond to those for the rule \( \text{reduction.strong-cases}, \) the strong inversion principle for the reduction relation.  

lemma dismantle-strong-cases:  

\( \text{fixes } t :: \text{trm} \)  

\( \text{assumes } r: t \times k \mapsto r \)  

\( \text{and } f: y \notin \{t,k,r\} \quad x \notin \{z,t,k,r\} \quad z \notin \{t,k,r\} \)  

\( \text{and } T: \land t', \[ t \mapsto t'; r = t' \times k \] \implies P \)  

\( \text{and } K: \land k', \[ k \mapsto k'; r = t \times k' \] \implies P \)  

\( \text{and } B: \land s n l . \[ t = [s] ; k = [y]_n \gg l ; r = (n[y:=s]) \times l \] \implies P \)  

\( \text{and } A: \land u v n l . \quad \)  

\[ x \notin \{z,n\} ; t = u \to x \in v ; k = [z]_n \gg l ; r = (u \to x \in (v \to z \in n) \times l) \] \implies P \)  

shows \( P \)  

proof (cases rule:dismantle-cases[of \( t k r P \)])  

\( \text{case (\( 4 \ s y' n L \) have ch:} \)  

\( t = [s] \)  

\( k = [y']_n \gg L \)  

\( r = n[y':=s] \times L \) by fact+  

The equations we get look almost like those we need to instantiate the hypothesis \( B \). The only difference is that \( B \) only applies to \( y \), and since we want \( y \) to become an instantiation variable of the strengthened rule, we only know that \( y \) satisfies \( f \) and nothing else. But the condition \( f \) is just strong enough to rename \( y' \) to \( y \) and apply \( B \).
with \( f \) have \( y = y' \lor y \notin n \)
by (auto simp add: fresh-prod abs-fresh)

hence \( n[y::=s] = \langle(y,y') \cdot n\rangle[y::=s] \)
and \( [y]n \gg L = [y][((y,y')] \cdot n) \gg L \)
by (auto simp add: name-swap-bij subst-renam subst-stack inject alpha)

with \( ch \) have \( t = [s] \)
\( k = [y][((y,y')] \cdot n) \gg L \)
\( r = \langle((y,y')] \cdot n)[y::=s] \gg L \)
by (auto)

thus \( P \) by (rule B)

next

\begin{enumerate}
\item case \( (u \cdot v \cdot z' \cdot n) \)

\item with \( f \) have \( x \cdot x = x' \lor x \notin v \cdot z' \cdot n \)
and \( z = z' \lor z \notin n \)
by (auto simp add: fresh-prod abs-fresh)

\item from \( f \) \( ch \) have \( x' \cdot x \notin n \cdot x' \notin z' \cdot n \)
and \( xx' \cdot x = z' \lor x \notin n \)
by (auto simp add: name-swap-bij alpha fresh-prod fresh-atm abs-fresh)

\item from \( f \) \( ch \) have \( x' \cdot z \cdot x \notin z' \cdot n \)
by (auto simp add: fresh-prod)

\item with \( zz' \cdot z \)

\item by (auto simp add: fresh-atm fresh-bij name-swap-bij fresh-prod fresh-atm fresh-aux fresh-left)

\item moreover from \( x \) \( ch \) have \( t = u \cdot x \cdot v \cdot z' \cdot n \cdot v \in n \)
by (auto simp add: name-swap-bij alpha)

\item moreover from \( z \) \( ch \) have \( k = [z][((z,z')] \cdot n) \gg L \)
by (auto simp add: name-swap-bij stack inject alpha)
\end{enumerate}

The first two \( \alpha \)-renamings are simple, but here we have to handle the nested binding structure of the assoc rule. Since \( x \) scopes over the whole term \( v \cdot z' \cdot n \), we have to push the swapping over \( z' \)

moreover \{ from \( x \) have \( u \cdot x' \cdot v \cdot z' \cdot n \) \}
\begin{enumerate}
\item by (auto simp add: name-swap-bij alpha' simp del: trm_perm)
\item also from \( xx' \cdot x \)

\item by (auto simp add: abs-fun_eq1 swap-simps alpha''

\item (metis alpha'' fresh-atm perm fresh-fresh swap-simps(1) x'))

\item also from \( z \) have \( u \cdot x \cdot v \cdot z \cdot n \)

\item by (auto simp add: abs-fun_eq1 alpha' name-swap-bij)
\end{enumerate}

\begin{enumerate}
\item finally

\item have \( r = (u \cdot x \cdot v \cdot z \cdot n) \cdot v \in n \)

\item using \( ch \) by (simp del: trm.inject)
\end{enumerate}

ultimately show \( P \)
by (rule A[where \( n=[(z,x')] \cdot n \) and \( v=[(z,x')] \cdot v \)])

\textbf{qed} (insert \( \tau\) \( T\) \( K\), auto)

The lemma in Figure 2 assumes \( SN (n[x::=p] \gg K) \) but the actual induction is done on \( SN (n \gg K) \). The stronger assumption \( SN (n[x::=p] \gg K) \) is needed to handle the \( \beta \) and \( \eta \) cases.
To be able to handle the case where $p \Rightarrow \cdots$, where $\text{redrtrans}$ lemma explicitly.

those cases where substitution is pushed to two subterms needs to be proven automatically. Similarly, in the $\text{red-subst}$ lemma, only

and $x \not\in k$
shows $SN (t \ast k)$

\begin{verbatim}
abbreviation
redrtrans :: trm ⇒ trm ⇒ bool ( · ⇒ · )
where redrtrans ≡ reduction"**
\end{verbatim}

To be able to handle the case where $p$ makes a step, we need to establish $p \Rightarrow p' \Rightarrow m[x:=p] \Rightarrow m[x:=p']$ as well as the fact that strong normalization is preserved for an arbitrary number of reduction steps. The first claim involves a number of simple transitivity lemmas. Here we can benefit from having removed the freshness conditions from the reduction relation as this allows all the cases to be proven automatically. Similarly, in the $\text{red-subst}$ lemma, only those cases where substitution is pushed to two subterms needs to be proven explicitly.

\begin{verbatim}
lemma red-trans:
  shows r1-trans: $s \Rightarrow s' \Rightarrow App s t \Rightarrow App s' t$
  and r2-trans: $t \Rightarrow t' \Rightarrow App s t \Rightarrow App s t'$
  and r4-trans: $t \Rightarrow t' \Rightarrow \Lambda x \cdot t \Rightarrow \Lambda x \cdot t'$
  and r6-trans: $s \Rightarrow s' \Rightarrow s \to x in t \Rightarrow s' \to x in t$
  and r7-trans: $[ t \Rightarrow t' ] \Rightarrow s \to x in t \Rightarrow s \to x in t'$
  and r11-trans: $s \Rightarrow s' \Rightarrow [s] \Rightarrow [s']$
by (induct rule: rtranclp-induct, (auto intro: transitive-closure-trans)(2))
\end{verbatim}

\begin{verbatim}
lemma red-subst: $p \Rightarrow p' \Rightarrow m[x:=p] \Rightarrow (m[x:=p'])$
proof(nominal-induct m avoiding: x p' rule: trm.strong-induct)
  case (App s t)
  hence $App (s[x:=p]) (t[x:=p]) \Rightarrow App (s[x:=p']) (t[x:=p])$
  by (auto intro: r1-trans)
\end{verbatim}

23
also from App have \( \ldots \mapsto^{*} \text{App} (s[x\mathbin{:=}p']) (t[x\mathbin{:=}p']) \)
by (auto intro: r2-trans)
finally show \(?case by auto 
next
\begin{description}
\item [case (To s y n) hence] \((s[x\mathbin{:=}p]) \text{ to } y \text{ in } (n[x\mathbin{:=}p]) \mapsto^{*} (s[x\mathbin{:=}p']) \text{ to } y \text{ in } (n[x\mathbin{:=}p])\)
by (auto intro: r6-trans)
\item [also from To have \( \ldots \mapsto^{*} (s[x\mathbin{:=}p']) \text{ to } y \text{ in } (n[x\mathbin{:=}p])\)]
by (auto intro: r7-trans)
\end{description}
finally show \(?case using To by auto 
qed (auto intro:red-trans)

\textbf{Lemma SN-trans : \[ p \mapsto^{*} p' ; \text{SN} \ p \\] \implies \text{SN} \ p'}
by (induct rule: rtranclp-induct) (auto intro: SN-preserved)

\section{Central lemma}

Now we have everything in place we need to tackle the central “Lemma 7” of [LS05] (cf. Figure 2). The proof is quite long, but for the most part, the reasoning is that of [LS05].

\textbf{Lemma to-RED-aux:}
assumes \( p\) : SN \( p\)
and \( \mathbf{x} : x \not\in p \quad x \not\in k \)
and \( \mathbf{npk} : \text{SN} \ (n[x\mathbin{:=}p] \ast k) \)
shows \( \text{SN} \ (([p] \to x \mathbin{in} n) \ast k) \)

\textbf{Proof} –
\begin{itemize}
\item \{ fix \( q \) assume \( \text{SN} \ q \mathbin{with} p \)
\item have \( \bigwedge m . \ ([q = m \ast k] \land \text{SN}(m[x\mathbin{:=}p] \ast k)) \)
\item \( \implies \text{SN} \ (([p] \to x \mathbin{in} m) \ast k) \)
\end{itemize}
using \( x \)
\textbf{Proof} (induct \( p \mathbin{p q} \mathbin{rule: triple-induct[where k=k]} \))
\begin{enumerate}
\item [case (1 \( p \mathbin{q k} \)] We obtain an induction hypothesis for \( p, q, \) and \( k. \)
\item have \( \mathbf{ih-p} : \bigwedge p' \mathbin{m} . \ ([p \mapsto p'] ; q = m \ast k ; \text{SN} \ (m[x\mathbin{:=}p'] \ast k) ; x \not\in p' ; x \not\in k] \)
\item \( \implies \text{SN} \ (([p'] \to x \mathbin{in} m) \ast k) \) by fact
\item have \( \mathbf{ih-q} : \bigwedge q' \mathbin{m k} . \ ([q \mapsto q'] ; q' = m \ast k ; \text{SN} \ (m[x\mathbin{:=}p] \ast k) ; x \not\in p ; x \not\in k] \)
\item \( \implies \text{SN} \ (([p] \to x \mathbin{in} m) \ast k) \) by fact
\item have \( \mathbf{ih-k} : \bigwedge k' \mathbin{m} . \ ([k'] < [k] ; q = m \ast k' ; \text{SN} \ (m[x\mathbin{:=}p] \ast k') ; x \not\in p ; x \not\in k'] \)
\item \( \implies \text{SN} \ (([p] \to x \mathbin{in} m) \ast k') \) by fact
\item have \( q : q = m \ast k \) and \( \mathbf{sn} : \text{SN} \ (m[x\mathbin{:=}p] \ast k) \) by fact+
\item have \( x : x \not\in p \) and \( z : x \not\in k \) by fact+
\end{enumerate}

Once again we want to reason via case distinction on the successors of a term including a dismantling operator. Since this time we also need to handle the cases where interactions occur, we want to use the strengthened case rule. We already require \( x \) to be suitably fresh. To instantiate the rule, we need another fresh name.
\begin{itemize}
\item \{ fix \( r \) assume red : \([p] \to x \mathbin{in} m) \ast k \mapsto r \)
\item from \( x z \mathbin{sk} \) have \( x l : x \not\in ([p] \to x \mathbin{in} m) \ast k \)
\item by (simp add: abs-fresh)
\item with \( \mathbf{red} \) have \( x 2 : x \not\in r \) by (rule reduction-fresh)
\item obtain \( z : \mathbf{name where} x : x \not\in (x,p,m,k,r) \)
\item using \( \mathbf{ex-fresh}[of \ (x,p,m,k,r)] \) by (auto simp add: fresh-prod)
\item have \( \text{SN} \ r \)
\end{itemize}
The case where $p$ in the reduction rule for the reduction relation. Contract the freshness conditions to this subterm. This allows the use of the strong transitive closure of the reduction relation.

Proof (cases rule:dismantle-strong-cases

[of \([p] \to x \in m \quad k \quad x \quad x \quad z\])

Case (5 $r'$) have $r = r' * k$ and $r'$: $[p] \to x \in m \mapsto r'$ by fact+

To handle the case of a reduction occurring somewhere in $[p] \to x \in m$, we need to contract the freshness conditions to this subterm. This allows the use of the strong inversion rule for the reduction relation.

From $x1 x2 r$

have $x1(x \notin [p] \to x \in m)$ and $x2 x : r'$ by auto

from $z$ have $zl \notin ([p] \to x \in m) \quad x \notin z$

by (auto simp add: abs-fresh fresh-prod fresh-atm)

Case (r6 $s'$ t) hence $ch: [p] \mapsto s'$ $r' = s'$ to $x \in m$

using $x1 x2$ by (auto)

From this obtain $p'$ where $s: s' = [p']$ and $p : p \mapsto p'$

by (blast dest:red-Red)

From $p$ have $(m * k)(\alpha := p) \mapsto ((m * k)(\alpha := p'))$

by (rule red-subst)

With $zk$ have $(m(\alpha := p) * k) \mapsto ((m(\alpha := p) * k)$

by (simp add: ss subst-forget)

Hence $sn: SN ((m(\alpha := p') * k)$ using $sn$ by (rule SN-trans)

From $p$ $xp$ have $xp': x \notin p'$ by (rule reduction-fresh)

From $ch$ $s$ have $rr: r' = [p']$ to $x \in m$ by simp

From $p$ $q$ $sn$ $xp'$ $zk$

Show $SN r$ unfolding $r$ $rr$ by (rule ih-p)

Next

Case (r7 $s$ t $m'$) hence $r' = [p] \to x \in m'$ and $m \mapsto m'$

Using $x1 x2$ by (auto simp add: alpha)

Hence $rr: r' = [p] \to x \in m'$ by simp

From $q$ $m \mapsto m'$ have $q \mapsto m' * k$ by (simp add: dismantle-red)

Moreover have $m' * k = m' * k$ .. — a triviality

Moreover \{ From $m \mapsto m'$ have $(m(\alpha := p)) * k \mapsto (m(\alpha := p)) * k$

by (simp add: dismantle-red reduction-subst)

With $sn$ have $SN (m(\alpha := p) * k)$ .. \}

Ultimately show $SN r$ using $xp$ $zk$ unfolding $r$ $rr$ by (rule ih-q)

Next

Case (r8 $s$ t) — the \beta-case is handled by assumption

Hence $r' = m(\alpha := p)$ using $x1 x2$ by (auto simp add: alpha)

Thus $SN r$ unfolding $r$ using $sn$ by simp

Next

Case (r9 $s$) — the \eta-case is handled by assumption as well

Hence $m = [\text{Var} x]$ and $r' = [p]$ using $x1 x2$

by (auto simp add: alpha)

Hence $r' = m(\alpha := p)$ by simp

Thus $SN r$ unfolding $r$ using $sn$ by simp

QED (simp-all only: $xr x1 z l x r$ abs-fresh , auto)
There are no other possible reductions of \([p]\) to \(x\) in \(m\).

The case of an assoc interaction between \([p]\) to \(x\) in \(m\) and \(k\) is easily handled by the induction hypothesis, since \(m[x:=p] * k\) remains fixed under assoc.

```plaintext

next

case (6 \(k'\))
  have \(k: k \mapsto k'\) and \(r: r = ([p] to x in m) * k'\) by fact+
  from \(q k\) have \(q \mapsto m * k'\) unfolding stack-reduction-def by blast
  moreover have \(m * k' = m * k'\) ..
  moreover \{ have \(SN (m[x:=p] * k)\) by fact
    moreover have \((m[x:=p]) * k \mapsto (m[x:=p]) * k'\)
    using \(k\) unfolding stack-reduction-def ..
    ultimately have \(SN (m[x:=p] * k')\) .. \}
  moreover note \(xp\)
  moreover from \(k\) \(xk\) have \(x \not\in k'\)
    by (rule stack-reduction-fresh)
  ultimately show \(SN r\) unfolding \(r\) by (rule ih-q)

next

\}

thus \(SN ([p] to x in m) * k\) ..

qed \}

moreover have \(SN ((n[x:=p]) * k)\) by fact
moreover hence \(SN (n * k)\) using \((x \not\in k)\) by (rule sn-forget')
ultimately show \(?thesis\) by blast

qed

Having established the claim above, we use it show that to-bindings preserve reducibility.

Lemma to-RED:
  assumes \(s: s \in RED (T \tau)\)
  and \(t: \forall p \in RED \sigma . t[x:=p] \in RED (T \tau)\)
  shows \(s\) to \(x\) in \(t\) \(\in RED (T \tau)\)

proof –

\{ fix \(K\) assume \(k: K \in SRED \tau\)
  \{ fix \(p\) assume \(p: p \in RED \sigma\)
    hence \(sn\) : \(SN p\) using \(RED\)-props by(simp add: CR1-def)
    obtain \(x\)'s name where \(x: x' \not\in (t, p, K)\)
      using \(ex\)-fresh[of \((t, p, K)\)] by (auto)
    from \(p t k\) have \(SN((t[x:=p]) * K)\) by auto
    with \(x\) have \(SN ((((x',x) \cdot t)[x'::=p]) * K)\)
  \}

26
by (simp add: fresh-prod subst-rename)
with snp x have snx': SN (([p] to x' in ([x',x] * t )) * K)
by (auto intro: to-RED-aux)
from x have [p] to x' in ([x',x] * t ) = [p] to x in t
by simp (metis alpha' fresh-prod name-swap-bij x)
moreover have ([p] to x in t) * K = [p] * [x] t\gg K by simp
ultimately have snx: SN([p] * [x] t\gg K) using snx'
by (simp del: trm.inject)
} hence [x] t\gg K \in SRED σ by simp
with s have SN((s to x in t) * K) by(auto simp del: SRED.simps)
} thus s to x in t \in RED (T τ) by simp
qed

9 Fundamental Theorem

The remainder of this section follows [Nom] very closely. We first establish
that all well typed terms are reducible if we substitute reducible terms for the
free variables.

abbreviation
mapsto :: (name×trm) list \Rightarrow name \Rightarrow trm \Rightarrow bool (- maps - to - [55,55,55] 55)
where
\vartheta maps x to e \equiv (lookup \vartheta x) = e

abbreviation
closes :: (name×trm) list \Rightarrow (name×ty) list \Rightarrow bool (- closes - [55,55,55] 55)
where
\vartheta closes \Gamma \equiv \forall x. \tau. ((x,\tau) \in set \Gamma \rightarrow (\exists t. \vartheta maps x to t \land t \in RED \tau))

theorem fundamental-theorem:
assumes a: \Gamma \vdash t : \tau and b: \vartheta closes \Gamma
shows \vartheta\langle t\rangle \in RED \tau
using a b
proof(nominal-induct avoiding: \vartheta rule: typing.strong-induct)
case (t\langle a Γ σ t τ \vartheta \rangle) — lambda case
next
case \langle t\langle x Γ s σ t τ \vartheta \rangle \rangle — to case
have ihs : \bigwedge \vartheta . \vartheta closes \Gamma \implies \vartheta\langle s\rangle \in RED (T σ) by fact
have iht : \bigwedge \vartheta . \vartheta closes ((x, σ) \# Γ) \implies \vartheta\langle t\rangle \in RED (T τ) by fact
have \vartheta-cond: \vartheta closes Γ by fact
have fresh: x \notin \vartheta \land x \notin Γ \land x \notin s by fact+
from ihs have \vartheta\langle s\rangle \in RED (T σ) using \vartheta-cond by simp
moreover
{} from iht have \forall s\in RED σ \cdot ((x, \sigma)\#\emptyset)\langle t\rangle \in RED (T τ)
using fresh \vartheta-cond fresh-context by simp
hence \forall s\in RED σ \cdot \vartheta\langle t\rangle[x::=s] \in RED (T τ)
using fresh by (simp add: psubst-subst) 
ultimately have (\vartheta\langle s\rangle) to x in (\vartheta\langle t\rangle) \in RED (T τ) by (simp only: to-RED)
thus \vartheta\langle s\rangle to x in t \in RED (T τ) using fresh by simp
qed auto — all other cases are trivial

The final result then follows using the identity substitution, which is Γ-closing
since all variables are reducible at any type.

fun
id :: (name×ty) list \Rightarrow (name×trm) list

27
where
\[ \text{id}[] = [] \]
\[ \text{id}((x,\tau)\#\Gamma) = (x, \text{Var } x)\#(\text{id }\Gamma) \]

**lemma id-maps:**
shows \((\text{id }\Gamma)\) maps \(a\) to \((\text{Var } a)\)
by \((\text{induct }\Gamma)\) (auto)

**lemma id-fresh:**
fixes \(x::\text{name}\)
assumes \(x: x \notin \Gamma\)
shows \(x \notin (\text{id }\Gamma)\)
using \(x\)
by \((\text{induct }\Gamma)\) (auto simp add: fresh-list-nil fresh-list-cons)

**lemma id-apply:**
shows \((\text{id }\Gamma)<t> = t\)
by \((\text{nominal-induct } t\ \text{avoiding: } \Gamma\ \text{rule: } \text{trn\_strong\_induct})\)
(auto simp add: id-maps id-fresh)

**lemma id-closes:**
shows \((\text{id }\Gamma)\) closes \(\Gamma\)
proof –
\{ fix \(x\ \tau\) assume \((x,\tau) \in \text{set }\Gamma\)
  have \(\text{CR4 }\tau\) by (simp add: RED-props CR3.implies-CR4)
  hence \(\text{Var } x \in \text{RED }\tau\)
    by (auto simp add: NEUT-def normal-var CR4-def)
  hence \((\text{id }\Gamma)\) maps \(x\) to \(\text{Var } x \wedge \text{Var } x \in \text{RED }\tau\)
    by (simp add: id-maps)
\}
thus ?thesis by blast
qed

9.1 Strong normalization theorem

**lemma typing-implies-RED:**
assumes \(a: \Gamma \vdash t: \tau\)
shows \(t \in \text{RED }\tau\)
proof –
\{ have \((\text{id }\Gamma)<t> \in \text{RED }\tau\)
  proof –
  \{ have \((\text{id }\Gamma)\) closes \(\Gamma\) by \((\text{rule id\_closes})\)
    with a show ?thesis by \((\text{rule fundamental\_theorem})\)
    qed
  \}
  thus \(t \in \text{RED }\tau\) by \((\text{simp add: id\_apply})\)
  qed

**theorem strong-normalization:**
assumes \(a: \Gamma \vdash t: \tau\)
shows \(SN(t)\)
proof –
from \(a\) have \(t \in \text{RED }\tau\) by \((\text{rule typing-implies-RED})\)
moreover have \(\text{CR1 }\tau\) by \((\text{rule RED-props})\)
ultimately show \(SN(t)\) by \((\text{simp add: CR1-def})\)
qed

This finishes our formalization effort. This article is generated from the Is-
abelle theory file, which consists of roughly 1500 lines of proof code. The reader is invited to replay some of the more technical proofs using the theory file provided.

Acknowledgments

I thank Christian Urban, the Nominal Methods group, and the members of the Isabelle mailing list for their helpful answers to my questions.

References


