Strong Normalization of Moggis's Computational Metalanguage

Christian Doczkal
Saarland University

March 17, 2025

Abstract

Handling variable binding is one of the main difficulties in formal proofs. In this context, Moggi's computational metalanguage serves as an interesting case study. It features monadic types and a commuting conversion rule that rearranges the binding structure. Lindley and Stark have given an elegant proof of strong normalization for this calculus. The key construction in their proof is a notion of relational $\top\top$-lifting, using stacks of elimination contexts to obtain a Girard-Tait style logical relation.

I give a formalization of their proof in Isabelle/HOL-Nominal with a particular emphasis on the treatment of bound variables.

Contents

1 Introduction

This article contains a formalization of the strong normalization theorem for the λ_{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 λ-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: $x \sharp t$
  shows $[y].t = [x].([(y,x)] \cdot t)$
proof −
  from a **have** *aux*: $y \sharp [(y,\ x)] \cdot t$
    by (*subst fresh-bij*[*THEN sym, of* - - [(*x*,*y*)]])
      (*auto simp add*: *perm-swap calc-atm*)
  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* $\Gamma;x\sharp\Gamma]\!] \Longrightarrow$ *valid* $((x,\sigma)\#\Gamma)$
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,s); Γ ⊢ s : T σ ; ((x,σ)#Γ) ⊢ t : T τ ⟧
                ⟹ Γ ⊢ s to x in t : T τ
equivariance 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-eqvt[eqvt]:
  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<[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-eqvt[eqvt]:
  fixes pi::name prm
  shows pi • (ϑ<t>) = (pi • ϑ)<(pi • t)>
```

```
by(nominal-induct t avoiding: ϑ rule:trm.strong-induct)
  (auto simp add: eqvts fresh-bij)

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 x ♯ (s,y,v) ⟹ (s to x in t)[y::=v] = s[y::=v] to x in 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: abs-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: abs-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::*name*
  assumes $z\sharp\vartheta$ $z\sharp x$
  shows $z\sharp$ *lookup* ϑ x
using *assms*
by(*induct rule*: *lookup.induct*)
  (*auto simp add*: *fresh-list-cons*)

lemma *lookup-fresh'*:
  assumes a: $z\sharp\vartheta$
  shows *lookup* ϑ z = *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 :: *name*
  assumes a: $x \sharp t$ **and** b: $x \sharp \vartheta$
  shows $x \sharp \vartheta<t>$
using a b
by(*nominal-induct t avoiding*: ϑ x *rule*:*trm.strong-induct*)
  (*auto simp add*: *lookup-fresh abs-fresh*)

lemma *psubst-subst*:
  assumes a: $x \sharp \vartheta$
  shows $\vartheta<t>[x::=s] = ((x,s)\#\vartheta)<t>$
  using a
by(*nominal-induct t avoiding*: ϑ x 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 λ_{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*!]:$s \leadsto s' \Longrightarrow$ *App* s $t \leadsto$ *App* s' t
| *std-r2*[*intro*!]:$t \leadsto t' \Longrightarrow$ *App* s $t \leadsto$ *App* s t'

| *std-r3*[*intro*!]:*App* (Λ *x* . *t*) *s* $\rightsquigarrow$ *t*[*x*::=*s*]

| *std-r4*[*intro*!]:*t* $\rightsquigarrow$ *t'* $\Longrightarrow$ Λ *x* . *t* $\rightsquigarrow$ Λ *x* . *t'*
| *std-r5*[*intro*!]:*x* $\sharp$ *t* $\Longrightarrow$ Λ *x* . *App* *t* (*Var* *x*) $\rightsquigarrow$ *t*

| *std-r6*[*intro*!]:⟦ *s* $\rightsquigarrow$ *s'* ⟧ $\Longrightarrow$ *s to x in t* $\rightsquigarrow$ *s' to x in t*
| *std-r7*[*intro*!]:⟦ *t* $\rightsquigarrow$ *t'* ⟧ $\Longrightarrow$ *s to x in t* $\rightsquigarrow$ *s to x in t'*
| *std-r8*[*intro*!]:[*s*] *to x in t* $\rightsquigarrow$ *t*[*x*::=*s*]
| *std-r9*[*intro*!]:*x* $\sharp$ *s* $\Longrightarrow$ *s to x in* [*Var x*] $\rightsquigarrow$ *s*
| *std-r10*[*intro*!]: ⟦ *x* $\sharp$ *y*; *x* $\sharp$ *u*⟧
$\Longrightarrow$ (*s to x in t*) *to y in u* $\rightsquigarrow$ *s to x in* (*t to y in u*)
| *std-r11*[*intro*!]: *s* $\rightsquigarrow$ *s'* $\Longrightarrow$ [*s*] $\rightsquigarrow$ [*s'*]

inductive
reduction :: *trm* $\Rightarrow$ *trm* $\Rightarrow$ *bool* (‹- $\mapsto$ -› [*80*,*80*] *80*)
where
r1[*intro*!]:*s* $\mapsto$ *s'* $\Longrightarrow$ *App s t* $\mapsto$ *App s' t*
| *r2*[*intro*!]:*t* $\mapsto$ *t'* $\Longrightarrow$ *App s t* $\mapsto$ *App s t'*
| *r3*[*intro*!]:*x* $\sharp$ *s* $\Longrightarrow$ *App* (Λ *x* . *t*) *s* $\mapsto$ *t*[*x*::=*s*]

| *r4*[*intro*!]:*t* $\mapsto$ *t'* $\Longrightarrow$ Λ *x* . *t* $\mapsto$ Λ *x* . *t'*
| *r5*[*intro*!]:*x* $\sharp$ *t* $\Longrightarrow$ Λ *x* . *App t* (*Var x*) $\mapsto$ *t*

| *r6*[*intro*!]:⟦ *x* $\sharp$ (*s*,*s'*) ; *s* $\mapsto$ *s'* ⟧ $\Longrightarrow$ *s to x in t* $\mapsto$ *s' to x in t*
| *r7*[*intro*!]:⟦ *x* $\sharp$ *s* ; *t* $\mapsto$ *t'* ⟧ $\Longrightarrow$ *s to x in t* $\mapsto$ *s to x in t'*
| *r8*[*intro*!]:*x* $\sharp$ *s* $\Longrightarrow$ [*s*] *to x in t* $\mapsto$ *t*[*x*::=*s*]
| *r9*[*intro*!]:*x* $\sharp$ *s* $\Longrightarrow$ *s to x in* [*Var x*] $\mapsto$ *s*
| *r10*[*intro*!]: ⟦ *x* $\sharp$ (*y*,*s*,*u*) ; *y* $\sharp$ (*s*,*t*) ⟧
$\Longrightarrow$ (*s to x in t*) *to y in u* $\mapsto$ *s to x in* (*t to y in u*)
| *r11*[*intro*!]: *s* $\mapsto$ *s'* $\Longrightarrow$ [*s*] $\mapsto$ [*s'*]
equivariance *reduction*
nominal-inductive *reduction*
by(*auto simp add*: *abs-fresh fresh-fact' fresh-prod fresh-atm*)

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*!]: *App* (Λ *x* . *t*) *s* $\mapsto$ *t*[*x*::=*s*]
proof −
obtain *x'*::*name* **where** *s*: *x'* $\sharp$ *s* **and** *t*: *x'* $\sharp$ *t*
using *ex-fresh*[*of* (*s*,*t*)] **by** (*auto simp add*: *fresh-prod*)
from *t* **have** *App* (Λ *x* . *t*) *s* = *App* (Λ *x'* . ([(*x*,*x'*)] • *t*)) *s*
by (*simp add*: *alpha''*)
also from *s* **have** ... $\mapsto$ ([(*x*, *x'*)] • *t*)[*x'*::=*s*] **..**
also have ... = *t*[*x*::=*s*] **using** *t*
by (*auto simp add*: *subst-rename'*) (*metis perm-swap*)
finally show *?thesis* **.**
qed
declare *r3*[*rule del*]

lemma *r6'*[*intro*]:
fixes *s* :: *trm*
assumes *r*: *s* $\mapsto$ *s'*
shows *s to x in t* $\mapsto$ *s' to x in t*
using *assms*

proof −
  obtain *x'::name* **where** *s*: *x'* ♯ (*s*, *s'*) **and** *t*: *x'* ♯ *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* ([(*x*,*x'*)] • *t*)
    by (*simp add*: *alpha''*)
  also from *s r* **have** ... ↦ *s' to x' in* ([(*x*, *x'*)] • *t*) **..**
  also from *t* **have** ... = *s' to x in t*
    by (*simp add*: *alpha''*)
  finally show *?thesis* **.**
qed
declare *r6*[*rule del*]

lemma *r7'*[*intro*]:
  fixes *t* :: *trm*
  assumes *t* ↦ *t'*
  shows *s to x in t* ↦ *s to x in t'*
using *assms*
proof −
  obtain *x'::name* **where** *f*: *x'* ♯ *t*   *x'* ♯ *t'*   *x'* ♯ *s*   *x'* ♯ *x*
    using *ex-fresh*[*of* (*t*,*t'*,*s*,*x*)] **by**(*auto simp add*:*fresh-prod*)
  hence *a*: *s to x in t* = *s to x' in* ([(*x*,*x'*)] • *t*)
    by (*auto simp add*: *alpha''*)
  from *assms* **have** ([(*x*,*x'*)] • *t*) ↦ [(*x*,*x'*)] • *t'*
    by (*simp add*: *eqvts*)
  hence *r*: *s to x' in* ([(*x*,*x'*)] • *t*) ↦ *s to x' in* ([(*x*,*x'*)] • *t'*)
    using *f* **by** *auto*
  from *f* **have** *s to x in t'* = *s to x' in* ([(*x*,*x'*)] • *t'*)
    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* ↦ *t*[*x*::=*s*]
proof −
  obtain *x'::name* **where** *s*: *x'* ♯ *s* **and** *t*: *x'* ♯ *t*
    using *ex-fresh*[*of* (*s*,*t*)] **by** (*auto simp add*: *fresh-prod*)
  from *t* **have** [*s*] *to x in t* = [*s*] *to x' in* ([(*x*,*x'*)] • *t*)
    by (*simp add*: *alpha''*)
  also from *s* **have** ... ↦ ([(*x*, *x'*)] • *t*)[*x'*::=*s*] **..**
  also have ... = *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* [*Var x*] ↦ *s*
proof −
  obtain *x'::name* **where** *f*: *x'* ♯ *s*   *x'* ♯ *x*
    using *ex-fresh*[*of* (*s*,*x*)] **by**(*auto simp add*:*fresh-prod*)
  hence *s to x' in* [*Var x'*] ↦ *s* **by** *auto*
  moreover have *s to x' in* ([*Var x'*]) = *s to x in* ([*Var x*])
    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*: $x \sharp y \quad x \sharp u$
  shows $(s\ to\ x\ in\ t)\ to\ y\ in\ u \mapsto s\ to\ x\ in\ (t\ to\ y\ in\ u)$
proof −
  obtain y'::*name* — suitably fresh
    where *y*: $y' \sharp s \quad y' \sharp x \quad y' \sharp t \quad y' \sharp u$
    using *ex-fresh*[*of* $(s,x,t,u,[(x, x')] \bullet t)$]
    by (*auto simp add*: *fresh-prod*)
  obtain x'::*name*
    where *x*: $x' \sharp s \quad x' \sharp y' \quad x' \sharp y \quad x' \sharp t \quad x' \sharp u$
      $x' \sharp ([(y,y')] \bullet u)$
    using *ex-fresh*[*of* $(s,y',y,t,u,([(y,y')] \bullet u))$]
    by (*auto simp add*: *fresh-prod*)
  from *x y* **have** *yaux*: $y' \sharp [(x, x')] \bullet t$
    by(*simp add*: *fresh-left perm-fresh-fresh fresh-atm*)
  have $(s\ to\ x\ in\ t)\ to\ y\ in\ u = (s\ to\ x\ in\ t)\ to\ y'\ in\ ([(y,y')] \bullet u)$
    using ‹$y' \sharp u$› **by** (*simp add*: *alpha''*)
  also have ... $= (s\ to\ x'\ in\ ([(x,x')] \bullet t))\ to\ y'\ in\ ([(y,y')] \bullet u)$
    using ‹$x' \sharp t$› **by** (*simp add*: *alpha''*)
  also have ... $\mapsto s\ to\ x'\ in\ (([(x,x')] \bullet t)\ to\ y'\ in\ ([(y,y')] \bullet u))$
    using *x y yaux* **by** (*auto simp add*: *fresh-prod*)
  also have ... $= s\ to\ x'\ in\ (([(x,x')] \bullet t)\ to\ y\ in\ u)$
    using ‹$y' \sharp u$› **by** (*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** *swap*: $[(x,x')] \bullet y = y \quad [(x,x')] \bullet u = u$
      by(*auto simp add*: *fresh-atm perm-fresh-fresh*)
    with *x* **show** $s = s \wedge [x'].([(x, x')] \bullet t)\ to\ y\ in\ u = [x].t\ to\ y\ in\ u$
      by (*auto simp add*: *alpha''*[*of* x' - x] *abs-fresh abs-fun-eq1 swap*)
  qed
  finally show *?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*: $s \mapsto t = s \rightsquigarrow t$
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*::*name*
  assumes *r*: $t \mapsto t'$
  shows $x \sharp t \Longrightarrow x \sharp t'$
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*: $t \mapsto t'$
  shows $t[x{::=}v] \mapsto t'[x{::=}v]$
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: $(\bigwedge t' .\ t \mapsto t' \Longrightarrow SN\ t') \Longrightarrow SN\ t$

lemma *SN-preserved*[*intro*]:
  assumes *a*: $SN\ t \quad t \mapsto t'$
  shows $SN\ t'$
using *a* **by** (*cases*) (*auto*)

definition $NORMAL$ :: $trm \Rightarrow bool$
where
  $NORMAL\ t \equiv \neg(\exists\, t'.\ t \mapsto t')$

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

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

4 Stacks

As explained in [LS05], the monadic type structure of the λ_{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 $stack = Id \mid St$ «*name*»*trm stack* (‹[-]-≫-›)

lemma *stack-exhaust* :
  fixes c :: $'a$::*fs-name*
  shows $k = Id \vee (\exists\ y\ n\ l\ .\ y \sharp l \wedge y \sharp c \wedge k = [y]n{\gg}l)$
by(*nominal-induct* k *avoiding*: c *rule*: *stack.strong-induct*) (*auto*)

nominal-primrec
  length :: $stack \Rightarrow nat$ (‹|-|›)
where

|*Id*| = *0*
| *y* ♯ *L* ⟹ *length* ([*y*]*n*≫*L*) = *1* + |*L*|
by(*finite-guess*+,*auto simp add*: *fresh-nat*,*fresh-guess*)

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*: ⋀*t*. *arg* = (*t*, *stack.Id*) ⟹ *P*
and *st*: ⋀*x 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*[*of 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*]: $s \star [y]n \gg L = (s\ to\ y\ in\ n) \star L$
proof −
obtain x::*name* **where** f: $x \sharp s \quad x \sharp L \quad x \sharp n$
using *ex-fresh*[*of* (s,L,n)] **by**(*auto simp add*:*fresh-prod*)
hence t: $s\ to\ y\ in\ n = s\ to\ x\ in\ ([(y,x)] \cdot n)$
by(*auto simp add*: *alpha''*)
from f **have** $[y]n \gg L = [x]([(y,x)] \cdot n) \gg L$
by (*auto simp add*: *stack.inject alpha''*)
hence $s \star [y]n \gg L = s \star [x]([(y,x)] \cdot n) \gg L$ **by** *simp*
also have $\ldots = (s\ to\ y\ in\ n) \star L$ **using** $f\ t$ **by**(*simp del*:*trm.inject*)
finally show *?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* :: *stack* ⇒ *stack* ⇒ *bool* (‹ - ↦ - ›)
where
$k \mapsto k' \equiv \forall\ (t{::}trm)\ .\ (t \star k) \ \mapsto (t \star k')$

lemma *stack-reduction-fresh*:
fixes k :: *stack* **and** x :: *name*
assumes r : $k \mapsto k'$ **and** f :$x \sharp k$
shows $x \sharp k'$
proof −
from *ex-fresh*[*of* x] **obtain** z::*name* **where** f': $z \sharp x$ **..**
from r **have** $Var\ z \star k \mapsto Var\ z \star k'$ **unfolding** *stack-reduction-def* **..**
moreover from $f\ f'$ **have** $x \sharp Var\ z \star k$ **by**(*auto simp add*: *fresh-atm*)
ultimately have $x \sharp Var\ z \star k'$ **by**(*rule reduction-fresh*)
thus $x \sharp k'$ **by** *simp*
qed

lemma *dismantle-red*[*intro*]:
fixes m :: *trm*
assumes r: $m \mapsto m'$
shows $m \star k \mapsto m' \star k$
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
ssubst :: *name* ⇒ *trm* ⇒ *stack* ⇒ *stack*
where
$ssubst\ x\ v\ Id = Id$
$|\ \ y \sharp (k,x,v) \implies ssubst\ x\ v\ ([y]n \gg k) = [y](n[x{::=}v]) \gg (ssubst\ x\ v\ k)$
by(*finite-guess*+ , (*simp add*: *abs-fresh*)+ , *fresh-guess*+)

lemma *ssubst-fresh*:
fixes y :: *name*
assumes $y \sharp (x,v,k)$
shows $y \sharp ssubst\ x\ v\ k$
using *assms*
by(*nominal-induct* k *avoiding*: $y\ x\ v$ *rule*: *stack.strong-induct*)

($auto\ simp\ add$: $fresh\text{-}prod\ fresh\text{-}atm\ abs\text{-}fresh\ fresh\text{-}fact$)

lemma $ssubst\text{-}forget$:
fixes $x :: name$
assumes $x \sharp k$
shows $ssubst\ x\ v\ k = k$
using $assms$
by($nominal\text{-}induct\ k\ avoiding$: $x\ v\ rule$: $stack.strong\text{-}induct$)
($auto\ simp\ add$: $abs\text{-}fresh\ fresh\text{-}atm\ forget$)

lemma $subst\text{-}dismantle[simp]$: $(t \star k)[x ::= v] = (t[x::=v]) \star ssubst\ x\ v\ k$
by($nominal\text{-}induct\ k\ avoiding$: $t\ x\ v\ \ rule$: $stack.strong\text{-}induct$)
($auto\ simp\ add$: $ssubst\text{-}fresh\ fresh\text{-}prod\ fresh\text{-}fact$)

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 $ty\text{-}exhaust$: $ty = TBase \vee (\exists\ \sigma\ \tau\ .\ ty = \sigma \to \tau) \vee (\exists\ \sigma\ .\ ty = T\ \sigma)$
by($induct\ ty\ rule{:}ty.induct$) ($auto$)

function $RED :: ty \Rightarrow trm\ set$
and $SRED :: ty \Rightarrow stack\ set$
where
$RED\ (TBase) = \{t.\ SN(t)\}$
$|\ RED\ (\tau{\to}\sigma) = \{t.\ \forall\ \ u \in RED\ \tau\ .\ (App\ t\ u) \in RED\ \sigma\ \}$
$|\ RED\ (T\ \sigma) = \{t.\ \forall\ \ k \in SRED\ \sigma\ .\ SN(t \star k)\ \}$
$|\ SRED\ \tau = \{k.\ \forall\ \ t \in RED\ \tau\ .\ SN\ ([t] \star k)\ \}$
by($auto\ simp\ add$: $ty.inject$, $case\text{-}tac\ x\ rule$: $sum.exhaust, insert\ ty\text{-}exhaust$)
($blast$)+

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 definiton of *SRED* τ involves a recursive call to *RED* τ without reducing the size of τ.

nominal-primrec
$tsize :: ty \Rightarrow nat$
where
$tsize\ TBase = 1$
$|\ tsize\ (\sigma{\to}\tau) = 1 + tsize\ \sigma + tsize\ \tau$
$|\ tsize\ (T\ \tau) = 1 + tsize\ \tau$
by ($rule\ TrueI$)+

In the termination argument below, *Inl* τ corresponds to the call *RED* τ, whereas *Inr* τ corresponds to *SRED* τ

termination *RED*
by($relation\ measure$
$(\lambda\ x\ .\ case\ x\ of\ Inl\ \tau \Rightarrow 2 * tsize\ \tau$
$|\ Inr\ \tau \Rightarrow 2 * tsize\ \tau + 1))$ ($auto$)

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
$NEUT\ t \equiv (\exists\, a.\ t = Var\ a) \lor (\exists\, t1\ t2.\ t = App\ t1\ t2)$

definition *CR1* :: *ty* ⇒ *bool*
where
$CR1\ \tau \equiv \forall\, t.\ (t{\in}RED\ \tau \longrightarrow SN\ t)$

definition *CR2* :: *ty* ⇒ *bool*
where
$CR2\ \tau \equiv \forall\, t\ t'.\ (t{\in}RED\ \tau \land t \mapsto t') \longrightarrow t'{\in}RED\ \tau$

definition *CR3-RED* :: *trm* ⇒ *ty* ⇒ *bool*
where
$CR3\text{-}RED\ t\ \tau \equiv \forall\, t'.\ t \mapsto t' \longrightarrow\ t'{\in}RED\ \tau$

definition *CR3* :: *ty* ⇒ *bool*
where
$CR3\ \tau \equiv \forall\, t.\ (NEUT\ t \land CR3\text{-}RED\ t\ \tau) \longrightarrow t{\in}RED\ \tau$

definition *CR4* :: *ty* ⇒ *bool*
where
$CR4\ \tau \equiv \forall\, t.\ (NEUT\ t \land NORMAL\ t) \longrightarrow t{\in}RED\ \tau$

lemma *CR3-implies-CR4*[*intro*]: $CR3\ \tau \Longrightarrow CR4\ \tau$
by (*auto simp add*: *CR3-def CR3-RED-def CR4-def NORMAL-def*)

inductive
FST :: *trm*⇒*trm*⇒*bool* (‹ - » -› [*80*,*80*] *80*)
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
  SSN-intro: $(\bigwedge k' .\ k \mapsto k' \Longrightarrow \mathit{SSN}\ k') \Longrightarrow \mathit{SSN}\ k$

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

inductive
  $\mathit{SND\text{-}DIS} :: \mathit{trm} \Rightarrow \mathit{stack} \Rightarrow \mathit{bool}$ (‹- ▷ -›)
where
  snd-dis[*intro*!]: $t \star k \rhd k$

lemma *SN-SSN*:
  assumes *a*: $\mathit{SN}\ (t \star k)$
  shows $\mathit{SSN}\ k$
proof −
  from *a* **have** $\forall z.\ (t \star k \rhd z) \longrightarrow \mathit{SSN}\ z$ **by** (*induct rule*: *SN.induct*)
    (*metis SND-DIS.cases SSN-intro snd-dis stack-reduction-def*)
  thus $\mathit{SSN}\ k$ **by** *blast*
qed

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

$$\frac{t \star k \mapsto r \qquad \bigwedge t'.\ [\![t \mapsto t';\ r = t' \star k]\!] \Longrightarrow P \qquad \mathit{NEUT}\ t \qquad \bigwedge k'.\ [\![\ k \mapsto k'\ ;\ r = t \star k']\!] \Longrightarrow P}{P}$$

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 $t :: \mathit{trm}$
  assumes *r*: $t \star k \mapsto r$
  and *T*: $\bigwedge t' .\ [\![\ t \mapsto t'\ ;\ r = t' \star k\ ]\!] \Longrightarrow P$
  and *K*: $\bigwedge k' .\ [\![\ k \mapsto k'\ ;\ r = t \star k'\ ]\!] \Longrightarrow P$
  and *B*: $\bigwedge s\ y\ n\ l\ .[\![\ t = [s]\ ;\ k = [y]n{\gg}l\ ;\ r = (n[y{::=}s]) \star l\ ]\!] \Longrightarrow P$
  and *A*: $\bigwedge u\ x\ v\ y\ n\ l.[\![\ x \sharp y;\ x \sharp n\ ;\ t = u\ \mathit{to}\ x\ \mathit{in}\ v\ ;$
      $k = [y]n{\gg}l\ ;\ r = (u\ \mathit{to}\ x\ \mathit{in}\ (v\ \mathit{to}\ y\ \mathit{in}\ n)) \star l\ ]\!] \Longrightarrow P$
  shows P
using *assms*
proof (*nominal-induct k avoiding*: *t r rule*:*stack.strong-induct*)
  case (*St y n L*) **note** *yfresh* = ‹$y \sharp t$› ‹$y \sharp r$› ‹$y \sharp L$›
  note *IH* = *St*(*4*)
    and *T* = *St*(*6*) **and** *K* = *St*(*7*) **and** *B* = *St*(*8*) **and** *A* = *St*(*9*)
  thus *P* **proof** (*cases rule*:*IH*[**where** *b*=*t to y in n* **and** *ba*=*r*])
    case (*2 r'*) **have** *red*: $t\ \mathit{to}\ y\ \mathit{in}\ n \mapsto r'$ **and** *r*: $r = r' \star L$ **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*: $y \sharp$ *t to y in n* $y \sharp r'$
  by (*auto simp add*: *abs-fresh*)
obtain *z* **where** *z*: $z \neq y$ $z \sharp r'$ $z \sharp$ *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*=*y***and** *xa*=*y* **and** *xb*=*y* **and** *xc*=*y* **and** *xd*=*y*
   and *xe*=*y* **and** *xf*=*y* **and** *xg*=*z* **and** *y*=*y*])
  case (*r6 s 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 (*r7 - - n′*) **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 (*r8 s -*) — the case of a β-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 (*r9 -*) — The case of an η-reduction is a stack reduction as well.
  with *y* **have** *n*: $n = [\mathit{Var}\ y]$ **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 (*r10 u - v*) — The assoc case holds by A.
  with *y z* **have**
   $t =$ (*u to z in v*)
   $r' =$ *u to z in* (*v to y in n*)
   $z \sharp (y,n)$ **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 (*3 L′*)
hence *L*: $L \mapsto L'$ **and** *r*: $r = (t\ to\ y\ in\ n) \star L'$ **by** *auto*
{ fix *s* **from** *L* **have** $(s\ to\ y\ in\ n) \star L \mapsto (s\ to\ y\ in\ n) \star L'$
unfolding *stack-reduction-def* **..**
hence $s \star [y]n{\gg}L \mapsto s \star [y]n{\gg}L'$ **by** *simp*
} hence $[y]n{\gg}L \mapsto [y]n{\gg}L'$ **unfolding** *stack-reduction-def* **by** *auto*
moreover from *r* **have** $r = t \star [y]n{\gg}L'$ **by** *simp*
ultimately show *P* **by** (*rule K*)
next
case (*5 x z n′ s v K*) — The "assoc" case is again a stack reduction
have *xf*: $x \sharp z \quad x \sharp n'$
— We get the following equalities
and *red*: $t\ to\ y\ in\ n = s\ to\ x\ in\ v$
$L = [z]n'{\gg}K$
$r = (s\ to\ x\ in\ v\ to\ z\ in\ n') \star K$ **by** *fact+*
{ fix *u* **from** *red* **have** $u \star [y]n{\gg}L = ((u\ to\ x\ in\ v)\ to\ z\ in\ n') \star K$
by(*auto intro*: *arg-cong*[**where** $f{=}\lambda\ x\ .\ x \star K$])
moreover
{ from *xf* **have** $(u\ to\ x\ in\ v)\ to\ z\ in\ n' \mapsto u\ to\ x\ in\ (v\ to\ z\ in\ n')$ **..**
hence $((u\ to\ x\ in\ v)\ to\ z\ in\ n') \star K \mapsto (u\ to\ x\ in\ (v\ to\ z\ in\ n')) \star K$
by *rule*
} ultimately have $u \star [y]n{\gg}L \mapsto (u\ to\ x\ in\ (v\ to\ z\ in\ n')) \star K$
by (*simp* (*no-asm-simp*) *del*:*dismantle-simp*)
hence $u \star [y]n{\gg}L \mapsto u \star [x](v\ to\ z\ in\ n'){\gg}K$ **by** *simp*
} hence $[y]n{\gg}L \mapsto [x](v\ to\ z\ in\ n'){\gg}K$
unfolding *stack-reduction-def* **by** *simp*
moreover have $r = t \star ([x](v\ to\ z\ in\ n'){\gg}K)$ **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* :: *trm*
assumes *r*: $t \star k \mapsto r$
and *NEUT t*
and $\bigwedge t'\ .\ [\![\ t \mapsto t'\ ;\ r = t' \star k\ ]\!] \Longrightarrow P$
and $\bigwedge k'\ .\ [\![\ k \mapsto k'\ ;\ r = t \star k'\ ]\!] \Longrightarrow P$
shows *P*
using *assms* **unfolding** *NEUT-def*
by (*cases rule*: *dismantle-cases*[*of t k r*]) (*auto*)

lemma *red-Ret*:
fixes *t* :: *trm*
assumes $[s] \mapsto t$
shows $\exists\ s'\ .\ t = [s'] \wedge s \mapsto s'$
using *assms* **by** *cases* (*auto*)

lemma *SN-Ret*: $SN\ u \Longrightarrow SN\ [u]$
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 τ and CR2 τ and CR3 τ
proof (nominal-induct τ rule: ty.strong-induct)
  case TBasenext
  case (TFun τ1 τ2)next
  case (T σ)
  { case 1 — follows from the fact that stack.Id ∈ SRED σ
    have ih-CR1-σ: CR1 σ by fact
    { fix t assume t-red: t ∈ RED (T σ)
      { fix s assume s ∈ RED σ
        hence SN s using ih-CR1-σ by (auto simp add: CR1-def)
        hence SN ([s]) by (rule SN-Ret)
        hence SN ([s] ⋆ Id) by simp
      } hence Id ∈ SRED σ by simp
      with t-red have SN (t) by (auto simp del: SRED.simps)
    } thus CR1 (T σ) unfolding CR1-def by blast
  next
    case 2 — follows since SN is preserved under redcution
    { fix t t'::trm  assume t-red: t ∈ RED (T σ) and t-t': t ↦ t'
      { fix k assume k: k ∈ SRED σ
        with t-red have SN(t ⋆ k) by simp
        moreover from t-t' have t ⋆ k ↦ t' ⋆ k ..
        ultimately have SN(t' ⋆ k) by (rule SN-preserved)
      } hence t' ∈ RED (T σ)  by (simp del: SRED.simps)
    } thus CR2 (T σ)unfolding CR2-def by blast
  next
    case 3 from ‹CR3 σ› have ih-CR4-σ : CR4 σ ..
    { fix t assume t'-red: ⋀ t' . t ↦ t' ⟹ t' ∈ RED (T σ)
      and neut-t: NEUT t
      { fix k assume k-red: k ∈ SRED σ
        fix x have NEUT (Var x) unfolding NEUT-def by simp
        hence Var x ∈ RED σ using normal-var ih-CR4-σ
          by (simp add: CR4-def)
        hence SN ([Var 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 ↦ k' ; k' ∈ SRED σ ⟧  ⟹ SN (t ⋆ k')
            and k-red: k ∈ SRED σ by fact+
          { fix r assume r: t ⋆ k ↦ r
            hence SN r using neut-t
            proof (cases rule:dismantle-cases')
              case (T t') hence t-t': t ↦ t' and r-def: r = t' ⋆ k .
              from t-t' have t' ∈ RED (T σ) by (rule t'-red)
              thus SN r using k-red r-def by simp
            next
              case (K k') hence k-k': k ↦ k' and r-def: r = t ⋆ k' .
              { fix s assume s ∈ RED σ
                hence SN ([s] ⋆ k) using k-red
                  by simp
                moreover have [s] ⋆ k ↦ [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* σ **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 β-reduction only. However, the differences are only minor.

lemma *double-SN*[*consumes 2*]:
assumes *a*: *SN a*
and *b*: *SN b*
and *c*: $\bigwedge(x{::}trm)\ (z{::}trm).$
$[\![\bigwedge y.\ x \mapsto y \Longrightarrow P\ y\ z;\ \bigwedge u.\ z \mapsto u \Longrightarrow P\ x\ u]\!] \Longrightarrow P\ x\ z$
shows *P a b*
using *a b c*

lemma *red-Lam*:
  assumes *a*: $\Lambda\ x\ .\ t \mapsto r$
  shows $(\exists\, t'.\ r = \Lambda\ x\ .\ t' \wedge t \mapsto t') \vee (t = App\ r\ (Var\ x) \wedge x \sharp\ r\ )$
proof –
  obtain $z{::}name$ **where** z: $z \sharp\ x \quad z \sharp\ t \quad z \sharp\ r$
    using *ex-fresh*[*of* (*x*,*t*,*r*)] **by** (*auto simp add*: *fresh-prod*)
  have $x \sharp\ \Lambda\ x\ .\ t$ **by** (*simp add*: *abs-fresh*)
  with *a* **have** $x \sharp\ r$ **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*: $\forall\, s{\in}RED\ \tau.\ t[x{::=}s]{\in}RED\ \sigma$
  shows $\Lambda\ x\ .\ t\ {\in}RED\ (\tau{\to}\sigma)$

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 $SN\ (n[x{::=}p] \star K)$. The induction is then done on $|K| + max(n \star K) + max(p)$. 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.[1] 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

$$(SN, \mapsto) \times (SN, \mapsto) \times (\mathbb{N}, >)\,.$$

lemma *triple-induct*[*consumes 2*]:
  assumes *a*: $SN\ (p)$
  and *b*: $SN\ (q)$
  and *hyp*: $\bigwedge\ (p{::}trm)\ (q{::}trm)\ (k{::}stack)\ .$
  $[\![\bigwedge\ p'\ .\ p \mapsto p' \Longrightarrow P\ p'\ q\ k\ ;$
    $\bigwedge\ q'\ k\ .\ q \mapsto q' \Longrightarrow P\ p\ q'\ k;$
    $\bigwedge\ k'\ .\ \ |k'| < |k| \Longrightarrow P\ p\ q\ k'\ ]\!] \Longrightarrow P\ p\ q\ k$
  shows $P\ p\ q\ k$
proof –

[1] This possibility was only discovered *after* having formalized $K \mapsto K' \Rightarrow |K| \geq |K'|$. 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 $SN(n[x ::= p] \star K)$. Then $SN(([p] \ \mathsf{to} \ x \ \mathsf{in} \ n) \star K)$*

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

- $T.\beta$ giving $n[x ::= p] \star K$ which is strongly normalizing by hypothesis.
- $T.\eta$ when $n = [x]$ giving $[p] \star K$. But $[p] \star K = n[x ::= p] \star K$ 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 $([p] \ \mathsf{to} \ x \ \mathsf{in} \ (n \ \mathsf{to} \ y \ \mathsf{in} \ m)) \star K$. We aim to apply the induction hypothesis with K' and $(n \ \mathsf{to} \ y \ \mathsf{in} \ m)$ for K and n respectively. Now

$$\begin{aligned}(n \ \mathsf{to} \ y \ \mathsf{in} \ m)[x ::= p] \star K' &= (n[x ::= p] \ \mathsf{to} \ y \ \mathsf{in} \ m) \star K' \\ &= n[x ::= p] \star K\end{aligned}$$

  which is strongly normalizing by induction hypothesis. Also

$$|K'| + max((n \ \mathsf{to} \ y \ \mathsf{in} \ m) \star K') + max(p) < |K| + max(n \star K) + max(p)$$

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

Other reductions are confined to K, n or p and can be treated by the induction hypothesis, decreasing either $max(n \star K)$ or $max(p)$.

Figure 2: Proof of Lemma 7 as in [LS05]

from a **have** $\bigwedge q\ K\ .\ SN\ q \Longrightarrow P\ p\ q\ K$
proof (*induct p*)
case (*SN-intro p*)
have *sn1*: $\bigwedge\ p'\ q\ K\ .\ [\![p \mapsto p';\ SN\ q]\!] \Longrightarrow P\ p'\ q\ K$ **by** *fact*
have *sn-q*: $SN\ q \quad SN\ q$ **by** *fact+*
thus $P\ p\ q\ K$
proof (*induct q arbitrary*: K)
case (*SN-intro q K*)
have *sn2*: $\bigwedge\ q'\ K\ .\ [\![q \mapsto q';\ SN\ q']\!] \Longrightarrow P\ p\ q'\ K$ **by** *fact*
show $P\ p\ q\ K$
proof (*induct K rule*: *measure-induct-rule*[**where** *f=length*])
case (*less k*)
have *le*: $\bigwedge\ k'\ .\ |k'| < |k| \Longrightarrow P\ p\ q\ k'$ **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 *?case* **using** *le*
by (*auto intro*:*hyp*)
qed
qed
qed
with b **show** *?thesis* **by** *blast*
qed

Here we strengthen the case rule for terms of the form $t \star k \mapsto r$. The freshness requirements on x,y, and z correspond to those for the rule *reduction.strong-cases*, the strong inversion principle for the reduction relation.

lemma *dismantle-strong-cases*:
fixes t :: *trm*
assumes r: $t \star k \mapsto r$
and f: $y \sharp (t,k,r) \quad x \sharp (z,t,k,r) \quad z \sharp (t,k,r)$
and T: $\bigwedge\ t'\ .\ [\![\ t \mapsto t'\ ;\ r = t' \star k\ ]\!] \Longrightarrow P$
and K: $\bigwedge\ k'\ .\ [\![\ k \mapsto k'\ ;\ r = t \star k'\ ]\!] \Longrightarrow P$
and B: $\bigwedge\ s\ n\ l\ .\ [\![\ t = [s]\ ;$
$k = [y]n \gg l\ ;\ r = (n[y{::=}s]) \star l\ ]\!] \Longrightarrow P$
and A: $\bigwedge\ u\ v\ n\ l\ .$
$[\![\ x \sharp (z,n);\ t = u\ to\ x\ in\ v\ ;\ k = [z]n \gg l\ ;$
$r = (u\ to\ x\ in\ (v\ to\ z\ in\ n)) \star l\ ]\!] \Longrightarrow P$
shows P
proof (*cases rule*:*dismantle-cases*[*of t k r P*])
case (*4 s y' n L*) **have** *ch*:
$t = [s]$
$k = [y']n \gg L$
$r = n[y'{::=}s] \star L$ **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** $y = y' \vee y \sharp n$
  by (*auto simp add*: *fresh-prod abs-fresh*)
hence $n[y'{::=}s] = ([(y,y')] \bullet n)[y{::=}s]$
  and $[y']n{\gg}L = [y]([(y,y')] \bullet n){\gg}L$
  by(*auto simp add*: *name-swap-bij subst-rename′ stack.inject alpha′*)
with *ch* **have** $t = [s]$
  $k = [y]([(y,y')] \bullet n){\gg}L$
  $r = ([(y,y')] \bullet n)[y{::=}s] \star L$
  by (*auto*)
thus P **by** (*rule B*)
next
case (*5 u x′ v z′ n L*) **have** *ch*:
  $x' \sharp z' \quad x' \sharp n$
  $t = u \text{ to } x' \text{ in } v$
  $k = [z']n{\gg}L$
  $r = (u \text{ to } x' \text{ in } v \text{ to } z' \text{ in } n) \star L$ **by** *fact+*

We want to do the same trick as above but at this point we have to take care of the possibility that x might coincide with x' or z'. Similarly, z might coincide with z'.

with f **have** x: $x = x' \vee x \sharp v \text{ to } z' \text{ in } n$
  and z: $z = z' \vee z \sharp n$
  by (*auto simp add*: *fresh-prod abs-fresh*)
from *f ch* **have** x': $x' \sharp n \quad x' \sharp z'$
  and xz': $x = z' \vee x \sharp n$
  by (*auto simp add*:*name-swap-bij alpha fresh-prod fresh-atm abs-fresh*)
from *f ch* **have** $x \sharp z$ $x \sharp [z'].n$ **by** (*auto simp add*: *fresh-prod*)
with xz' z **have** $x \sharp (z , ([(z, z')] \bullet n))$
  by (*auto simp add*: *fresh-atm fresh-bij name-swap-bij*
    fresh-prod abs-fresh calc-atm fresh-aux fresh-left)
moreover from x *ch* **have** $t = u \text{ to } x \text{ in } ([(x,x')] \bullet v)$
  by (*auto simp add*:*name-swap-bij alpha′*)
moreover from z *ch* **have** $k = [z]([(z,z')] \bullet n){\gg}L$
  by (*auto simp add*:*name-swap-bij stack.inject alpha′*)

The first two α-renamings are simple, but here we have to handle the nested binding structure of the assoc rule. Since x scopes over the whole term $v \text{ to } z' \text{ in } n$, we have to push the swapping over z'

moreover { **from** x **have**
  $u \text{ to } x' \text{ in } (v \text{ to } z' \text{ in } n) = u \text{ to } x \text{ in } ([(x,x')] \bullet (v \text{ to } z' \text{ in } n))$
    by (*auto simp add*:*name-swap-bij alpha′ simp del*: *trm.perm*)
  also from xz' x' **have** $\ldots = u \text{ to } x \text{ in } (([(x,x')] \bullet v) \text{ to } z' \text{ in } n)$
    by (*auto simp add*: *abs-fun-eq1 swap-simps alpha″*)
      (*metis alpha″ fresh-atm perm-fresh-fresh swap-simps*(*1*) *x′*)
  also from z **have** $\ldots = u \text{ to } x \text{ in } (([(x,x')] \bullet v) \text{ to } z \text{ in } ([(z,z')] \bullet n))$
    by (*auto simp add*: *abs-fun-eq1 alpha′ name-swap-bij*)
  finally
    have $r = (u \text{ to } x \text{ in } (([(x, x')] \bullet v) \text{ to } z \text{ in } ([(z, z')] \bullet n))) \star L$
      using *ch* **by** (*simp del*: *trm.inject*) }
ultimately show P
  by (*rule A*[**where** $n{=}[(z, z')] \bullet n$ **and** $v{=}([(x, x')] \bullet v)$])
qed (*insert r T K*, *auto*)

The lemma in Figure 2 assumes $SN\ (n[x{::=}p] \star K)$ but the actual induction in done on $SN\ (n \star K)$. The stronger assumption $SN\ (n[x{::=}p] \star K)$ is needed to handle the β and η cases.

lemma *sn-forget*:
  assumes *a*: $SN(t[x::=v])$
  shows $SN\ t$
proof −
  define q **where** $q = t[x::=v]$
  from *a* **have** $SN\ q$ **unfolding** *q-def* **.**
  thus $SN\ t$ **using** *q-def*
  proof (*induct q arbitrary*: *t*)
    case (*SN-intro t*)
    hence *ih*: $\bigwedge\ t'.\ [\![t[x::=v] \mapsto t'[x::=v]]\!] \Longrightarrow SN\ t'$ **by** *auto*
    { fix t' **assume** $t \mapsto t'$
      hence $t[x::=v] \mapsto t'[x::=v]$ **by** (*rule reduction-subst*)
      hence $SN\ t'$ **by** (*rule ih*) **}**
    thus $SN\ t$ **..**
  qed
qed

lemma *sn-forget'*:
  assumes *sn*: $SN\ (t[x::=p] \star k)$
  and *x*: $x \sharp k$
  shows $SN\ (t \star k)$
proof −
  from *x* **have** $t[x::=p] \star k = (t \star k)[x::=p]$ **by** (*simp add*: *ssubst-forget*)
  with *sn* **have** $SN(\ (t \star k)[x::=p]\ )$ **by** *simp*
  thus *?thesis* **by** (*rule sn-forget*)
qed

abbreviation
redrtrans :: *trm* ⇒ *trm* ⇒ *bool* (‹ - $\mapsto^*$ - ›)
where *redrtrans* ≡ *reduction^***

To be able to handle the case where p makes a step, we need to establish $p \mapsto p' \Longrightarrow\ m[x::=p] \mapsto^* m[x::=p']$ 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 *red-subst* lemma, only those cases where substitution is pushed to two subterms needs to be proven explicitly.

lemma *red-trans*:
  shows *r1-trans*: $s \mapsto^* s' \Longrightarrow App\ s\ t \mapsto^* App\ s'\ t$
  and *r2-trans*: $t \mapsto^* t' \Longrightarrow App\ s\ t \mapsto^* App\ s\ t'$
  and *r4-trans*: $t \mapsto^* t' \Longrightarrow \Lambda\ x\ .\ t \mapsto^* \Lambda\ x\ .\ t'$
  and *r6-trans*: $s \mapsto^* s' \Longrightarrow s\ to\ x\ in\ t \mapsto^* s'\ to\ x\ in\ t$
  and *r7-trans*: $[\![\ t \mapsto^* t'\ ]\!] \Longrightarrow s\ to\ x\ in\ t \mapsto^* s\ \ to\ x\ in\ t'$
  and *r11-trans*: $s \mapsto^* s' \Longrightarrow [s] \mapsto^* ([s'])$
by − (*induct rule*: *rtranclp-induct* , (*auto intro*:
transitive-closurep-trans')[2])+

lemma *red-subst*: $p \mapsto p' \Longrightarrow (m[x::=p]) \mapsto^* (m[x::=p'])$
proof(*nominal-induct m avoiding*: *x p p' rule*:*trm.strong-induct*)
  case (*App s t*)
  hence $App\ (s[x::=p])\ (t[x::=p]) \mapsto^* App\ (s[x::=p'])\ (t[x::=p])$
    by (*auto intro*: *r1-trans*)

also from *App* **have** ... $\mapsto^*$ *App* $(s[x::=p'])$ $(\ t[x::=p'])$
  by (*auto intro*: *r2-trans*)
  finally show *?case* **by** *auto*
next
  case (*To s y n*) **hence**
    $(s[x::=p])$ *to y in* $(n[x::=p]) \mapsto^* (s[x::=p'])$ *to y in* $(n[x::=p])$
    by (*auto intro*: *r6-trans*)
  also from *To* **have** ... $\mapsto^* (s[x::=p'])$ *to y in* $(n[x::=p'])$
    by (*auto intro*: *r7-trans*)
  finally show *?case* **using** *To* **by** *auto*
qed (*auto intro*:*red-trans*)

lemma *SN-trans* : $[\![\ p \mapsto^* p'\ ;\ SN\ p\ ]\!] \Longrightarrow SN\ p'$
by (*induct rule*: *rtranclp-induct*) (*auto intro*: *SN-preserved*)

8.1 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].

lemma *to-RED-aux*:
  assumes *p*: *SN p*
  and *x*: $x \sharp p \quad x \sharp k$
  and *npk*: $SN\ (n[x::=p] \star k)$
  shows $SN\ (([p]$ *to x in n*$) \star k)$
proof −
  { fix *q* **assume** *SN q* **with** *p*
    have $\bigwedge m\ .\ [\![\ q = m \star k\ ;\ SN(m[x::=p] \star k)\ ]\!]$
          $\Longrightarrow SN\ (([p]$ *to x in m*$) \star k)$
      using *x*
    proof (*induct p q rule*:*triple-induct*[**where** *k*=*k*])
      case (*1 p q k*) — We obtain an induction hypothesis for p, q, and k.
      have *ih-p*:
        $\bigwedge p'\ m\ .\ [\![p \mapsto p';\ q = m \star k;\ SN\ (m[x::=p'] \star k);\ x \sharp p';\ x \sharp k]\!]$
          $\Longrightarrow SN\ (([p']$ *to x in m*$) \star k)$ **by** *fact*
      have *ih-q*:
        $\bigwedge q'\ m\ k\ .\ [\![q \mapsto q';\ q' = m \star k;\ SN\ (m[x::=p] \star k);\ x \sharp p;\ x \sharp k]\!]$
          $\Longrightarrow SN\ (([p]$ *to x in m*$) \star k)$ **by** *fact*
      have *ih-k*:
        $\bigwedge k'\ m\ .\ [\![\ |k'| < |k|;\ q = m \star k';\ SN\ (m[x::=p] \star k');\ x \sharp p;\ x \sharp k']\!]$
          $\Longrightarrow SN\ (([p]$ *to x in m*$) \star k')$ **by** *fact*
      have *q*: $q = m \star k$ **and** *sn*: $SN\ (m[x::=p] \star k)$ **by** *fact+*
      have *xp*: $x \sharp p$ **and** *xk*: $x \sharp k$ **by** *fact+*

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.

      { fix *r* **assume** *red*: $([p]$ *to x in m*$) \star k \mapsto r$
        from *xp xk* **have** *x1* : $x \sharp ([p]$ *to x in m*$) \star k$
          by (*simp add*: *abs-fresh*)
        with *red* **have** *x2*: $x \sharp r$ **by** (*rule reduction-fresh*)
        obtain *z*::*name* **where** *z*: $z \sharp (x,p,m,k,r)$
          using *ex-fresh*[*of* (x,p,m,k,r)] **by** (*auto simp add*: *fresh-prod*)
        have *SN r*

proof (*cases rule*:*dismantle-strong-cases*
[*of* [*p*] *to x in m* *k r x x z*])
case (*5 r′*) **have** *r*: $r = r' \star k$ **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 *xl*:($x \sharp$ [*p*] *to x in m*) **and** *xr*:$x \sharp r'$ **by** *auto*
from *z* **have** *zl*: $z \sharp$ ([*p*] *to x in m*) $x \neq z$
by (*auto simp add*: *abs-fresh fresh-prod fresh-atm*)
with *r′* **have** *zr*: $z \sharp r'$ **by** (*blast intro*:*reduction-fresh*)
— handle all reductions of [*p*] *to x in m*
from *r′* **show** *SN r* **proof** (*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*])

The case where $p \mapsto p'$ is interesting, because it requires reasioning about the reflexive transitive closure of the reduction relation.

case (*r6 s s′ t*) **hence** *ch*: [*p*] $\mapsto s'$ $r' = s'$ *to x in m*
using *xl xr* **by** (*auto*)
from *this* **obtain** *p′* **where** *s*: $s' = [p']$ **and** *p* : $p \mapsto p'$
by (*blast dest*:*red-Ret*)
from *p* **have** $((m{\star}k)[x{::=}p]) \mapsto^* ((m{\star}k)[x{::=}p'])$
by (*rule red-subst*)
with *xk* **have** $((m[x{::=}p]) \star k) \mapsto^* ((m[x{::=}p']) \star k)$
by (*simp add*: *ssubst-forget*)
hence *sn*: *SN* $((m[x{::=}p']) \star k)$ **using** *sn* **by** (*rule SN-trans*)
from *p xp* **have** *xp′* : $x \sharp p'$ **by** (*rule reduction-fresh*)
from *ch s* **have** *rr*: $r' = [p']$ *to x in m* **by** *simp*
from *p q sn xp′ xk*
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 *xl xr* **by** (*auto simp add*: *alpha*)
hence *rr*: $r' = [p]$ *to x in m′* **by** *simp*
from *q* ‹$m \mapsto m'$› **have** $q \mapsto m' \star k$ **by**(*simp add*: *dismantle-red*)
moreover have $m' \star k = m' \star k$ **..** — a triviality
moreover **{ from** ‹$m \mapsto m'$› **have** $(m[x{::=}p]) \star k \mapsto (m'[x{::=}p]) \star k$
by (*simp add*: *dismantle-red reduction-subst*)
with *sn* **have** *SN*$(m'[x{::=}p] \star k)$ **.. }**
ultimately show *SN r* **using** *xp xk* **unfolding** *r rr* **by** (*rule ih-q*)
next

case (*r8 s t*) — the β-case is handled by assumption
hence $r' = m[x{::=}p]$ **using** *xl xr* **by**(*auto simp add*: *alpha*)
thus *SN r* **unfolding** *r* **using** *sn* **by** *simp*
next

case (*r9 s*) — the η-case is handled by assumption as well
hence $m = [Var\ x]$ **and** $r' = [p]$ **using** *xl xr*
by(*auto simp add*: *alpha*)
hence $r' = m[x{::=}p]$ **by** *simp*
thus *SN r* **unfolding** *r* **using** *sn* **by** *simp*
qed (*simp-all only*: *xr xl zl zr abs-fresh* , *auto*)

— There are no other possible reductions of $[p]$ *to x in m*.
next

case $(6\ k')$
have k: $k \mapsto k'$ **and** r: $r = ([p]\ to\ x\ in\ m) \star k'$ **by** *fact+*
from $q\ k$ **have** $q \mapsto m \star k'$ **unfolding** *stack-reduction-def* **by** *blast*
moreover have $m \star k' = m \star k'$ **..**
moreover { **have** $SN\ (m[x::=p] \star k)$ **by** *fact*
moreover have $(m[x::=p]) \star k \mapsto (m[x::=p]) \star k'$
using k **unfolding** *stack-reduction-def* **..**
ultimately have $SN\ (m[x::=p] \star k')$ **.. }**
moreover note *xp*
moreover from $k\ xk$ **have** $x \sharp k'$
by (*rule stack-reduction-fresh*)
ultimately show $SN\ r$ **unfolding** r **by** (*rule ih-q*)
next

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

case $(8\ s\ t\ u\ L)$
hence k: $k = [z]u{\gg}L$
and r: $r = ([p]\ to\ x\ in\ (m\ to\ z\ in\ u)) \star L$
and u: $x \sharp u$
by(*auto simp add*: *alpha fresh-prod*)
let $?k = L$ **and** $?m = m\ to\ z\ in\ u$
from $k\ z$ **have** $|?k| < |k|$ **by** (*simp add*: *fresh-prod*)
moreover have $q = \ ?m \star ?k$ **using** $k\ q$ **by** *simp*
moreover { **from** $k\ u\ z\ xp$ **have** $(?m[x::=p] \star ?k) = (m[x::=p]) \star k$
by(*simp add*: *fresh-prod forget*)
hence $SN\ (?m[x::=p] \star ?k)$ **using** *sn* **by** *simp* **}**
moreover from $xp\ xk\ k$ **have** $x \sharp p$ **and** $x \sharp ?k$ **by** *auto*
ultimately show $SN\ r$ **unfolding** r **by** (*rule ih-k*)
qed (*insert red z x1 x2 xp xk* ,
auto simp add: *fresh-prod fresh-atm abs-fresh*)
} **thus** $SN\ (([p]\ to\ x\ in\ m) \star k)$ **..**
qed }
moreover have $SN\ ((n[x::=p]) \star k)$ **by** *fact*
moreover hence $SN\ (n \star k)$ **using** $\langle x \sharp k \rangle$ **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: $s \in RED\ (T\ \sigma)$
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 *snp*: $SN\ p$ **using** *RED-props* **by**(*simp add*: *CR1-def*)
obtain x'::*name* **where** x: $x' \sharp (t,\ p,\ K)$
using *ex-fresh*[*of* (t,p,K)] **by** (*auto*)
from $p\ t\ k$ **have** $SN((t[x::=p]) \star K)$ **by** *auto*
with x **have** $SN\ ((([(x',x)] \cdot t\ )[x'::=p]) \star K)$

by (*simp add*: *fresh-prod subst-rename*)
with *snp x* **have** *snx'*: $SN\ (([p]\ to\ x'\ in\ ([(x',x)] \bullet t\ )) \star K)$
by (*auto intro*: *to-RED-aux*)
from *x* **have** $[p]\ to\ x'\ in\ ([(x',x)] \bullet t\ ) = [p]\ to\ x\ in\ t$
by *simp* (*metis alpha' fresh-prod name-swap-bij x*)
moreover have $([p]\ to\ x\ in\ t) \star K\ = [p] \star [x]t \gg K$ **by** *simp*
ultimately have *snx*: $SN([p] \star [x]t \gg K)$ **using** *snx'*
by (*simp del*: *trm.inject*)
} **hence** $[x]t \gg K \in SRED\ \sigma$ **by** *simp*
with *s* **have** $SN((s\ to\ x\ in\ t) \star K)$ **by**(*auto simp del*: *SRED.simps*)
} **thus** $s\ to\ x\ in\ t \in RED\ (T\ \tau)$ **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 :: $(name \times trm)\ list \Rightarrow name \Rightarrow trm \Rightarrow bool$ (‹- *maps* - *to* -› [*55*,*55*,*55*] *55*)
where
$\vartheta\ maps\ x\ to\ e \equiv (lookup\ \vartheta\ x) = e$

abbreviation
closes :: $(name \times trm)\ list \Rightarrow (name \times ty)\ list \Rightarrow bool$ (‹- *closes* -› [*55*,*55*] *55*)
where
$\vartheta\ closes\ \Gamma \equiv \forall x\ \tau.\ ((x,\tau) \in set\ \Gamma \longrightarrow (\exists t.\ \vartheta\ maps\ x\ to\ t \wedge t \in RED\ \tau))$

theorem *fundamental-theorem*:
assumes *a*: $\Gamma \vdash t : \tau$ **and** *b*: $\vartheta\ closes\ \Gamma$
shows $\vartheta<t> \in RED\ \tau$
using *a b*
proof(*nominal-induct avoiding*: ϑ *rule*: *typing.strong-induct*)
case ($t3\ a\ \Gamma\ \sigma\ t\ \tau\ \vartheta$) — lambda case
next
case ($t5\ x\ \Gamma\ s\ \sigma\ t\ \tau\ \vartheta$) — to case
have *ihs* : $\bigwedge \vartheta\ .\ \vartheta\ closes\ \Gamma \Longrightarrow \vartheta<s> \in RED\ (T\ \sigma)$ **by** *fact*
have *iht* : $\bigwedge \vartheta\ .\ \vartheta\ closes\ ((x,\ \sigma)\ \#\ \Gamma) \Longrightarrow \vartheta<t> \in RED\ (T\ \tau)$ **by** *fact*
have ϑ*-cond*: $\vartheta\ closes\ \Gamma$ **by** *fact*
have *fresh*: $x \sharp \vartheta\quad x \sharp \Gamma\quad x \sharp s$ **by** *fact+*
from *ihs* **have** $\vartheta<s> \in RED\ (T\ \sigma)$ **using** ϑ*-cond* **by** *simp*
moreover
{ from *iht* **have** $\forall s \in RED\ \sigma.\ ((x,s)\#\vartheta)<t> \in RED\ (T\ \tau)$
using *fresh* ϑ*-cond fresh-context* **by** *simp*
hence $\forall s \in RED\ \sigma.\ \vartheta<t>[x::=s] \in RED\ (T\ \tau)$
using *fresh* **by** (*simp add*: *psubst-subst*) **}**
ultimately have $(\vartheta<s>)\ to\ x\ in\ (\vartheta<t>) \in RED\ (T\ \tau)$ **by** (*simp only*: *to-RED*)
thus $\vartheta<s\ to\ x\ in\ t> \in RED\ (T\ \tau)$ **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 :: $(name \times ty)\ list \Rightarrow (name \times trm)\ list$

```
where
  id []      = []
| id ((x,τ)#Γ) = (x, Var x)#(id Γ)

lemma id-maps:
  shows (id Γ) maps a to (Var a)
by (induct Γ) (auto)

lemma id-fresh:
  fixes x::name
  assumes x: x ♯ Γ
  shows x ♯ (id Γ)
using x
by (induct Γ) (auto simp add: fresh-list-nil fresh-list-cons)

lemma id-apply:
  shows (id Γ)<t> = t
by (nominal-induct t avoiding: Γ rule: trm.strong-induct)
   (auto simp add: id-maps id-fresh)

lemma id-closes:
  shows (id Γ) closes Γ
proof −
  { fix x τ assume (x,τ) ∈ set Γ
    have CR4 τ by(simp add: RED-props CR3-implies-CR4)
    hence Var x ∈ RED τ
      by(auto simp add: NEUT-def normal-var CR4-def)
    hence (id Γ) maps x to Var x ∧ Var x ∈ RED τ
      by (simp add: id-maps)
  } thus ?thesis by blast
qed
```

9.1 Strong normalization theorem

```
lemma typing-implies-RED:
  assumes a: Γ ⊢ t : τ
  shows t ∈ RED τ
proof −
  have (id Γ)<t>∈RED τ
  proof −
    have (id Γ) closes Γ by (rule id-closes)
    with a show ?thesis by (rule fundamental-theorem)
  qed
  thust ∈ RED τ by (simp add: id-apply)
qed

theorem strong-normalization:
  assumes a: Γ ⊢ t : τ
  shows SN(t)
proof −
  from a have t ∈ RED τ by (rule typing-implies-RED)
  moreover have CR1 τ by (rule RED-props)
  ultimately show SN(t) by (simp add: CR1-def)
qed
```

This finishes our formalization effort. This article is generated from the Isabelle 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

[DS09] Christian Doczkal and Jan Schwinghammer. Formalizing a Strong Normalization proof for Moggi's Computational Metalanguage: A Case Study in Isabelle/HOL-Nominal. In *LFMTP '09: Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages*, pages 57–63, New York, NY, USA, 2009. ACM.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989.

[LS05] Samuel Lindley and Ian Stark. Reducibility and ⊤⊤-lifting for Computation Types. In *Proceedings of Typed Lambda Calculi and Applications (TLCA '05)*, volume 3461 of *Lecture Notes in Computer Science*, pages 262–277. Springer, Apr 2005.

[Nom] Nominal Methods Group. Strong normalisation proof from the proofs and types book. http://isabelle.in.tum.de/dist/library/ HOL/HOL-Nominal/Examples/SN.html.

[Urb08] Christian Urban. Nominal Techniques in Isabelle/HOL. *J. Autom. Reason.*, 40(4):327–356, 2008.