

Strong Normalization of Moggi's Computational Metalanguage

Christian Doczkal
Saarland University

May 26, 2024

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	1
2	The Calculus	2
3	The Reduction Relation	5
4	Stacks	9
5	Reducibility for Terms and Stacks	12
6	Properties of the Reducibility Relation	13
7	Abstraction Preserves Reducibility	18
8	Sequencing Preserves Reducibility	19
	8.1 Central lemma	24
9	Fundamental Theorem	27
	9.1 Strong normalization theorem	28

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* # *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*)]])
(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* Γ; *x*#Γ] ⇒ *valid* ((*x*,*σ*)#Γ)

equivariance *valid*

lemma *fresh-ty*:
fixes $x :: \text{name}$ **and** $\tau :: \text{ty}$
shows $x \# \tau$
by (*induct* τ *rule: ty.induct*) (*auto*)

lemma *fresh-context*:
fixes $\Gamma :: (\text{name} \times \text{ty}) \text{list}$
assumes $a: x \# \Gamma$
shows $\neg(\exists \tau . (x, \tau) \in \text{set } \Gamma)$
using a
by (*induct* Γ) (*auto simp add: fresh-prod fresh-list-cons fresh-atm*)

inductive
typing $:: (\text{name} \times \text{ty}) \text{list} \Rightarrow \text{trm} \Rightarrow \text{ty} \Rightarrow \text{bool}$ ($- \vdash - : - [60, 60, 60] 60$)
where
 $t1[\text{intro}]: \llbracket \text{valid } \Gamma; (x, \tau) \in \text{set } \Gamma \rrbracket \Longrightarrow \Gamma \vdash \text{Var } x : \tau$
 $t2[\text{intro}]: \llbracket \Gamma \vdash s : \tau \rightarrow \sigma; \Gamma \vdash t : \tau \rrbracket \Longrightarrow \Gamma \vdash \text{App } s \ t : \sigma$
 $t3[\text{intro}]: \llbracket x \# \Gamma; ((x, \tau) \# \Gamma) \vdash t : \sigma \rrbracket \Longrightarrow \Gamma \vdash \Lambda \ x . t : \tau \rightarrow \sigma$
 $t4[\text{intro}]: \llbracket \Gamma \vdash s : \sigma \rrbracket \Longrightarrow \Gamma \vdash [s] : T \ \sigma$
 $t5[\text{intro}]: \llbracket x \# (\Gamma, s); \Gamma \vdash s : T \ \sigma; ((x, \sigma) \# \Gamma) \vdash t : T \ \tau \rrbracket$
 $\Longrightarrow \Gamma \vdash s \ \text{to } x \ \text{in } t : T \ \tau$

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 $:: (\text{name} \times \text{trm}) \text{list} \Rightarrow \text{name} \Rightarrow \text{trm}$
where
 $\text{lookup } [] \ x = \text{Var } x$
 $\text{lookup } ((y, e) \# \vartheta) \ x = (\text{if } x=y \ \text{then } e \ \text{else } \text{lookup } \vartheta \ x)$

lemma *lookup-eqvt*[*eqvt*]:
fixes $pi :: \text{name}$ *prm*
and $\vartheta :: (\text{name} \times \text{trm}) \text{list}$
and $x :: \text{name}$
shows $pi \cdot (\text{lookup } \vartheta \ x) = \text{lookup } (pi \cdot \vartheta) (pi \cdot x)$
by (*induct* ϑ) (*auto simp add: eqvts*)

nominal-primrec
psubst $:: (\text{name} \times \text{trm}) \text{list} \Rightarrow \text{trm} \Rightarrow \text{trm}$ ($- < - > [95, 95] 205$)
where
 $\vartheta < \text{Var } x > = \text{lookup } \vartheta \ x$
 $\vartheta < \text{App } s \ t > = \text{App } (\vartheta < s >) (\vartheta < t >)$
 $x \# \vartheta \Longrightarrow \vartheta < \Lambda \ x . s > = \Lambda \ x . (\vartheta < s >)$
 $\vartheta < [t] > = [\vartheta < t >]$
 $\llbracket x \# \vartheta ; x \# t \rrbracket \Longrightarrow \vartheta < t \ \text{to } x \ \text{in } s > = (\vartheta < t >) \ \text{to } x \ \text{in } (\vartheta < s >)$
by(*finite-guess+* , (*simp add: abs-fresh*) $+$, *fresh-guess+*)

lemma *psubst-eqvt*[*eqvt*]:
fixes $pi :: \text{name}$ *prm*
shows $pi \cdot (\vartheta < t >) = (pi \cdot \vartheta) < (pi \cdot t) >$

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

abbreviation

subst :: *trm* \Rightarrow *name* \Rightarrow *trm* \Rightarrow *trm* ($[-::=]$ [200,100,100] 200)

where

$t[x::=t'] \equiv ((x,t')) <t>$

lemma *subst[simp]*:

shows ($\text{Var } x[y::=v] = (\text{if } x = y \text{ then } v \text{ else } \text{Var } x)$)
and ($\text{App } s \ t[y::=v] = \text{App } (s[y::=v]) \ (t[y::=v])$)
and $x \# (y,v) \Longrightarrow (\Lambda \ x \ . \ t)[y::=v] = \Lambda \ x \ . \ t[y::=v]$
and $x \# (s,y,v) \Longrightarrow (s \ \text{to } x \ \text{in } t)[y::=v] = s[y::=v] \ \text{to } x \ \text{in } t[y::=v]$
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 $((y,x) \cdot t)[y::=v] = t[x::=v]$

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*: $x \# t \Longrightarrow t[x::=v] = t$

by(*nominal-induct* t *avoiding*: $x \ v$ *rule*: *trm.strong-induct*)
(auto simp add: abs-fresh fresh-atm)

lemma *fresh-fact*:

fixes $x::\text{name}$
assumes $x: x \# v \quad x \# t$
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 $x::\text{name}$
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 $a: x \neq y$
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 $t[x::= \text{Var } x] = t$

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  $z \# \vartheta$   $z \# x$ 
  shows  $z \# \text{lookup } \vartheta \ x$ 
using assms
by (induct rule: lookup.induct)
    (auto simp add: fresh-list-cons)

```

```

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

```

```

lemma psubst-subst:
  assumes  $a: x \# \vartheta$ 
  shows  $\vartheta < t > [x ::= s] = ((x,s) \# \vartheta) < t >$ 
  using a
by (nominal-induct t avoiding: \vartheta 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_{mt} -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 ::  $\text{trm} \Rightarrow \text{trm} \Rightarrow \text{bool}$  ( $- \rightsquigarrow - [80,80] \ 80$ )
where
  std-r1[intro!]:  $s \rightsquigarrow s' \Longrightarrow \text{App } s \ t \rightsquigarrow \text{App } s' \ t$ 
  | std-r2[intro!]:  $t \rightsquigarrow t' \Longrightarrow \text{App } s \ t \rightsquigarrow \text{App } s \ t'$ 

```

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

| *std-r4*[*intro!*]: $t \rightsquigarrow t' \implies \Lambda x . t \rightsquigarrow \Lambda x . t'$

| *std-r5*[*intro!*]: $x \# t \implies \Lambda x . App t (Var x) \rightsquigarrow t$

| *std-r6*[*intro!*]: $\llbracket s \rightsquigarrow s' \rrbracket \implies s \text{ to } x \text{ in } t \rightsquigarrow s' \text{ to } x \text{ in } t$

| *std-r7*[*intro!*]: $\llbracket t \rightsquigarrow t' \rrbracket \implies s \text{ to } x \text{ in } t \rightsquigarrow s \text{ to } x \text{ in } t'$

| *std-r8*[*intro!*]: $[s] \text{ to } x \text{ in } t \rightsquigarrow t[x::=s]$

| *std-r9*[*intro!*]: $x \# s \implies s \text{ to } x \text{ in } [Var x] \rightsquigarrow s$

| *std-r10*[*intro!*]: $\llbracket x \# y; x \# u \rrbracket$
 $\implies (s \text{ to } x \text{ in } t) \text{ to } y \text{ in } u \rightsquigarrow s \text{ to } x \text{ in } (t \text{ to } y \text{ in } u)$

| *std-r11*[*intro!*]: $s \rightsquigarrow s' \implies [s] \rightsquigarrow [s']$

inductive

reduction :: *trm* \Rightarrow *trm* \Rightarrow *bool* (- \mapsto - [80,80] 80)

where

r1[*intro!*]: $s \mapsto s' \implies App s t \mapsto App s' t$

| *r2*[*intro!*]: $t \mapsto t' \implies App s t \mapsto App s t'$

| *r3*[*intro!*]: $x \# s \implies App (\Lambda x . t) s \mapsto t[x::=s]$

| *r4*[*intro!*]: $t \mapsto t' \implies \Lambda x . t \mapsto \Lambda x . t'$

| *r5*[*intro!*]: $x \# t \implies \Lambda x . App t (Var x) \mapsto t$

| *r6*[*intro!*]: $\llbracket x \# (s,s') ; s \mapsto s' \rrbracket \implies s \text{ to } x \text{ in } t \mapsto s' \text{ to } x \text{ in } t$

| *r7*[*intro!*]: $\llbracket x \# s ; t \mapsto t' \rrbracket \implies s \text{ to } x \text{ in } t \mapsto s \text{ to } x \text{ in } t'$

| *r8*[*intro!*]: $x \# s \implies [s] \text{ to } x \text{ in } t \mapsto t[x::=s]$

| *r9*[*intro!*]: $x \# s \implies s \text{ to } x \text{ in } [Var x] \mapsto s$

| *r10*[*intro!*]: $\llbracket x \# (y,s,u) ; y \# (s,t) \rrbracket$
 $\implies (s \text{ to } x \text{ in } t) \text{ to } y \text{ in } u \mapsto s \text{ to } x \text{ in } (t \text{ to } y \text{ in } u)$

| *r11*[*intro!*]: $s \mapsto s' \implies [s] \mapsto [s']$

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!*]: $App (\Lambda x . t) s \mapsto t[x::=s]$

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** $App (\Lambda x . t) s = App (\Lambda x' . ((x,x') \cdot t)) s$

by (*simp add: alpha''*)

also from *s* **have** $\dots \mapsto ((x, x') \cdot t)[x'::=s] ..$

also have $\dots = t[x::=s]$ **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 $s \text{ to } x \text{ in } t \mapsto s' \text{ to } x \text{ in } t$

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* $([(x, x')] \cdot t)$
by (*simp add: alpha''*)
also from s r **have** $\dots \mapsto s'$ *to* x' *in* $([(x, x')] \cdot t)$..
also from t **have** $\dots = s'$ *to* x *in* t
by (*simp add: alpha''*)
finally show *?thesis* .

qed

declare $r6$ [*rule del*]

lemma $r7'$ [*intro*]:

fixes $t :: trm$

assumes $t \mapsto t'$

shows s *to* x *in* $t \mapsto s$ *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* $([(x, x')] \cdot t)$

by (*auto simp add: alpha''*)

from *assms* **have** $([(x, x')] \cdot t) \mapsto [(x, x')] \cdot t'$

by (*simp add: eqvts*)

hence $r: s$ *to* x' *in* $([(x, x')] \cdot t) \mapsto s$ *to* x' *in* $([(x, x')] \cdot t')$

using f **by** *auto*

from f **have** s *to* x *in* $t' = s$ *to* x' *in* $([(x, x')] \cdot t')$

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* $t \mapsto t[x::=s]$

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* $([(x, x')] \cdot t)$

by (*simp add: alpha''*)

also from s **have** $\dots \mapsto (([x, x'] \cdot t)[x'::=s])$..

also have $\dots = t[x'::=s]$ **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* $[Var\ x] \mapsto s$

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* $[Var\ x'] \mapsto s$ **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 $xf: x \# y \quad x \# u$

shows $(s \text{ to } x \text{ in } t) \text{ to } y \text{ in } u \mapsto s \text{ to } x \text{ in } (t \text{ to } y \text{ in } u)$

proof –

obtain $y':name$ — suitably fresh

where $y: y' \# s \quad y' \# x \quad y' \# t \quad y' \# u$

using $ex\text{-}fresh[of (s,x,t,u,[(x, x')] \cdot t)]$

by ($auto \text{ simp add: fresh\text{-}prod$)

obtain $x':name$

where $x: x' \# s \quad x' \# y' \quad x' \# y \quad x' \# t \quad x' \# u$

$x' \# [(y,y')] \cdot u$

using $ex\text{-}fresh[of (s,y',y,t,u,([(y,y')] \cdot u))]$

by ($auto \text{ simp add: fresh\text{-}prod$)

from $x \ y \ \text{have} \ yaux: y' \# [(x, x')] \cdot t$

by ($simp \text{ add: fresh\text{-}left \text{ perm\text{-}fresh\text{-}fresh \text{ fresh\text{-}atm}$)

have $(s \text{ to } x \text{ in } t) \text{ to } y \text{ in } u = (s \text{ to } x \text{ in } t) \text{ to } y' \text{ in } [(y,y')] \cdot u$

using $\langle y' \# u \rangle$ **by** ($simp \text{ add: alpha''}$)

also have $\dots = (s \text{ to } x' \text{ in } [(x,x')] \cdot t) \text{ to } y' \text{ in } [(y,y')] \cdot u$

using $\langle x' \# t \rangle$ **by** ($simp \text{ add: alpha''}$)

also have $\dots \mapsto s \text{ to } x' \text{ in } [(x,x')] \cdot t \text{ to } y' \text{ in } [(y,y')] \cdot u$

using $x \ y \ yaux$ **by** ($auto \text{ simp add: fresh\text{-}prod$)

also have $\dots = s \text{ to } x' \text{ in } [(x,x')] \cdot t \text{ to } y \text{ in } u$

using $\langle y' \# u \rangle$ **by** ($simp \text{ add: abs\text{-}fun\text{-}eq1 \text{ alpha''}$)

also have $\dots = s \text{ to } x \text{ in } (t \text{ to } y \text{ in } u)$

proof ($subst \text{ trm.inject}$)

from $xf \ x \ \text{have} \ swap: [(x,x')] \cdot y = y \quad [(x,x')] \cdot u = u$

by ($auto \text{ simp add: fresh\text{-}atm \text{ perm\text{-}fresh\text{-}fresh}$)

with $x \ \text{show} \ s = s \wedge [x].([(x, x')] \cdot t) \text{ to } y \text{ in } u = [x].t \text{ to } y \text{ in } u$

by ($auto \text{ simp add: alpha''[of } x' - x] \text{ abs\text{-}fresh \text{ abs\text{-}fun\text{-}eq1 \text{ swap}$)

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 ($auto \text{ elim:reduction.induct \text{ std\text{-}reduction.induct}$)

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 $x \# t \implies x \# t'$

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

$SN\text{-intro}: (\bigwedge t'. t \mapsto t' \implies SN\ t') \implies SN\ t$

lemma *SN-preserved[intro]*:

assumes $a: SN\ t\ t \mapsto t'$

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* : $NORMAL\ s \implies SN\ s$

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* = $Id \mid St\ \langle name \rangle trm\ stack\ ([-]\gg-)$

lemma *stack-exhaust* :

fixes $c :: 'a::fs-name$

shows $k = Id \vee (\exists y\ n\ l . y \# l \wedge y \# c \wedge k = [y]n\gg l)$

by(*nominal-induct* k *avoiding: c* *rule: stack.strong-induct*) (*auto*)

nominal-primrec

$length :: stack \Rightarrow nat\ (\ | - |)$

where

$|Id| = 0$
 $|y \# L \implies \text{length}([y]n \gg L) = 1 + |L|$
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* \Rightarrow *stack* \Rightarrow *trm* (- * - [160,160] 160)

where

$t \star Id = t$ |
 $x \# (K, t) \implies t \star ([x]s \gg K) = (t \text{ to } x \text{ in } s) \star K$

proof — — pattern completeness

fix *P* :: *bool* **and** *arg*::*trm* \times *stack*

assume *id*: $\bigwedge t. \text{arg} = (t, \text{stack.Id}) \implies P$

and *st*: $\bigwedge x K t s. \llbracket x \# (K, t); \text{arg} = (t, [x]s \gg K) \rrbracket \implies P$

{ **assume** *snd arg* = *Id*

hence *P* **by** (*metis id*[**where** *t=fst arg*] *surjective-pairing*) }

moreover

{ **fix** *y n L* **assume** *snd arg* = $[y]n \gg L$ $y \# (L, \text{fst arg})$

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 $(t, [x]s \gg K) = (t', [x']s' \gg K')$

thus $?g (t \text{ to } x \text{ in } s, K) = ?g (t' \text{ to } x' \text{ in } s', K')$

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* ($\lambda(t, K). |K|$))(*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-eqv*[*eqvt*]:

fixes *pi* :: (*name* \times *name*) *list*

shows $pi \cdot (t \star K) = (pi \cdot t) \star (pi \cdot K)$

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 $(x \# (t \star k)) = (x \# t \wedge x \# k)$

by(*nominal-induct k avoiding: t x rule: stack.strong-induct*)

(*simp-all*)

lemma *dismantle-simp*[*simp*]: $s \star [y]n \gg L = (s \text{ to } y \text{ in } n) \star L$
proof –
obtain $x :: \text{name}$ **where** $f: x \# s \quad x \# L \quad x \# n$
using *ex-fresh*[*of* (s, L, n)] **by** (*auto simp add: fresh-prod*)
hence $t: s \text{ to } y \text{ in } n = s \text{ to } x \text{ in } ([y, x] \cdot n)$
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 $\dots = (s \text{ to } y \text{ in } n) \star L$ **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* :: $\text{stack} \Rightarrow \text{stack} \Rightarrow \text{bool}$ ($- \mapsto -$)
where
 $k \mapsto k' \equiv \forall (t :: \text{trm}) . (t \star k) \mapsto (t \star k')$

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

lemma *dismantle-red*[*intro*]:
fixes $m :: \text{trm}$
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
 $\text{ssubst} :: \text{name} \Rightarrow \text{trm} \Rightarrow \text{stack} \Rightarrow \text{stack}$
where
 $\text{ssubst } x \ v \ \text{Id} = \text{Id}$
 $| \ y \ \# \ (k, x, v) \implies \text{ssubst } x \ v \ ([y]n \gg k) = [y](n[x ::= v]) \gg (\text{ssubst } x \ v \ k)$
by (*finite-guess+* , (*simp add: abs-fresh*)⁺ , *fresh-guess+*)

lemma *ssubst-fresh*:
fixes $y :: \text{name}$
assumes $y \# (x, v, k)$
shows $y \# \text{ssubst } x \ v \ k$
using *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 *ssubst-forget*:

fixes $x :: \text{name}$

assumes $x \# k$

shows $\text{ssubst } x \ v \ k = k$

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]*: $(t \star k)[x ::= v] = (t[x ::= v]) \star \text{ssubst } x \ v \ k$

by(*nominal-induct k avoiding: t x v rule: stack.strong-induct*)

(*auto simp add: ssubst-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*: $ty = TBase \vee (\exists \sigma \tau . ty = \sigma \rightarrow \tau) \vee (\exists \sigma . ty = T \sigma)$

by(*induct ty rule:ty.induct*) (*auto*)

function *RED* :: $ty \Rightarrow trm \ set$

and *SRED* :: $ty \Rightarrow stack \ set$

where

$RED \ TBase = \{t. SN(t)\}$

| $RED \ (\sigma \rightarrow \tau) = \{t. \forall u \in RED \ \tau . (App \ t \ u) \in RED \ \sigma \}$

| $RED \ (T \ \sigma) = \{t. \forall k \in SRED \ \sigma . SN(t \star k) \}$

| $SRED \ \tau = \{k. \forall t \in RED \ \tau . SN([t] \star k) \}$

by(*auto simp add: ty.inject, case-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* τ involves a recursive call to *RED* τ without reducing the size of τ .

nominal-primrec

tsize :: $ty \Rightarrow nat$

where

$tsize \ TBase = 1$

| $tsize \ (\sigma \rightarrow \tau) = 1 + tsize \ \sigma + tsize \ \tau$

| $tsize \ (T \ \tau) = 1 + tsize \ \tau$

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*

$(\lambda x . \text{case } x \text{ of } Inl \ \tau \Rightarrow 2 * tsize \ \tau$

| $Inr \ \tau \Rightarrow 2 * tsize \ \tau + 1))$ (*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 $NEUT :: trm \Rightarrow bool$

where

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

definition $CR1 :: ty \Rightarrow bool$

where

$$CR1\ \tau \equiv \forall t. (t \in RED\ \tau \longrightarrow SN\ t)$$

definition $CR2 :: ty \Rightarrow bool$

where

$$CR2\ \tau \equiv \forall t\ t'. (t \in RED\ \tau \wedge t \mapsto t') \longrightarrow t' \in RED\ \tau$$

definition $CR3-RED :: trm \Rightarrow ty \Rightarrow bool$

where

$$CR3-RED\ t\ \tau \equiv \forall t'. t \mapsto t' \longrightarrow t' \in RED\ \tau$$

definition $CR3 :: ty \Rightarrow bool$

where

$$CR3\ \tau \equiv \forall t. (NEUT\ t \wedge CR3-RED\ t\ \tau) \longrightarrow t \in RED\ \tau$$

definition $CR4 :: ty \Rightarrow bool$

where

$$CR4\ \tau \equiv \forall t. (NEUT\ t \wedge NORMAL\ t) \longrightarrow t \in RED\ \tau$$

lemma $CR3\text{-}implies\text{-}CR4$ [intro]: $CR3\ \tau \Longrightarrow CR4\ \tau$

by (*auto simp add: CR3-def CR3-RED-def CR4-def NORMAL-def*)

inductive

$$FST :: trm \Rightarrow trm \Rightarrow bool\ (-\ \gg\ -\ [80,80]\ 80)$$

where

$$fst[intr0!]: (App\ t\ s) \gg t$$

lemma $SN\text{-}of\text{-}FST\text{-}of\text{-}App$:

assumes $a: SN\ (App\ t\ s)$

shows $SN\ t$

proof –

from a **have** $\forall z. (App\ t\ s \gg z) \longrightarrow SN\ z$

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

$$SSN :: stack \Rightarrow bool$$

where

$SSN\text{-intro}: (\bigwedge k' . k \mapsto k' \implies SSN k') \implies SSN k$

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

$SND\text{-DIS} :: trm \Rightarrow stack \Rightarrow bool (- \triangleright -)$

where

$snd\text{-dis}[\text{intro!}]: t \star k \triangleright k$

lemma $SN\text{-SSN}$:

assumes $a: SN (t \star k)$

shows $SSN k$

proof –

from a **have** $\forall z. (t \star k \triangleright z) \longrightarrow SSN z$ **by** (*induct rule: SN.induct*)

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

thus $SSN k$ **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 .

$$\frac{t \star k \mapsto r \quad \bigwedge t'. \llbracket t \mapsto t'; r = t' \star k \rrbracket \implies P \quad NEUT t \quad \bigwedge k'. \llbracket k \mapsto k'; r = t \star k \rrbracket \implies P}{P}$$

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 $t :: trm$

assumes $r: t \star k \mapsto r$

and $T: \bigwedge t'. \llbracket t \mapsto t'; r = t' \star k \rrbracket \implies P$

and $K: \bigwedge k'. \llbracket k \mapsto k'; r = t \star k' \rrbracket \implies P$

and $B: \bigwedge s y n l. \llbracket t = [s]; k = [y]n \gg l; r = (n[y::=s]) \star l \rrbracket \implies P$

and $A: \bigwedge u x v y n l. \llbracket x \# y; x \# n; t = u \text{ to } x \text{ in } v;$

$k = [y]n \gg l; r = (u \text{ to } x \text{ in } (v \text{ to } y \text{ in } n)) \star l \rrbracket \implies P$

shows P

using *assms*

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

case $(St y n L)$ **note** $y\text{fresh} = \langle y \# t \rangle \langle y \# r \rangle \langle y \# L \rangle$

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** $red: t \text{ to } y \text{ in } n \mapsto r'$ **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  $y: y \# t$  to  $y$  in  $n$   $y \# r'$ 
  by (auto simp add: abs-fresh)
obtain  $z$  where  $z: z \neq y$   $z \# r'$   $z \# t$  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  $\beta$ -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 (r9 -) — 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 (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  $n)$ 
     $z \# (y, n)$  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 ( $\beta$   $L'$ )
hence  $L: L \mapsto L'$  and  $r: r = (t \text{ to } y \text{ in } n) \star L'$  by auto
{ fix  $s$  from  $L$  have  $(s \text{ to } y \text{ in } n) \star L \mapsto (s \text{ to } y \text{ in } n) \star L'$ 
  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 ( $\delta$   $x z n' s v K$ ) — The “assoc” case is again a stack reduction
have  $xf: x \# z \quad x \# n'$ 
  — We get the following equalities
and  $red: t \text{ to } y \text{ in } n = s \text{ to } x \text{ in } v$ 
   $L = [z]n' \gg K$ 
   $r = (s \text{ to } x \text{ in } v \text{ to } z \text{ in } n') \star K$  by fact+
{ fix  $u$  from  $red$  have  $u \star [y]n \gg L = ((u \text{ to } x \text{ in } v) \text{ to } z \text{ in } n') \star K$ 
  by(auto intro: arg-cong[where f= $\lambda x . x \star K$ ])
  moreover
  { from  $xf$  have  $(u \text{ to } x \text{ in } v) \text{ to } z \text{ in } n' \mapsto u \text{ to } x \text{ in } (v \text{ to } z \text{ in } n')$  ..
    hence  $((u \text{ to } x \text{ in } v) \text{ to } z \text{ in } n') \star K \mapsto (u \text{ to } x \text{ in } (v \text{ to } z \text{ in } n')) \star K$ 
    by rule
  } ultimately have  $u \star [y]n \gg L \mapsto (u \text{ to } x \text{ in } (v \text{ to } z \text{ in } n')) \star K$ 
    by (simp (no-asm-simp) del:dismantle-simp)
  hence  $u \star [y]n \gg L \mapsto u \star [x](v \text{ to } z \text{ in } n') \gg K$  by simp
} hence  $[y]n \gg L \mapsto [x](v \text{ to } z \text{ in } n') \gg K$ 
  unfolding stack-reduction-def by simp
moreover have  $r = t \star ([x](v \text{ to } z \text{ in } n') \gg K)$  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  $\bigwedge t' . \llbracket t \mapsto t' ; r = t' \star k \rrbracket \implies P$ 
and  $\bigwedge k' . \llbracket k \mapsto k' ; r = t \star k' \rrbracket \implies P$ 
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*: $SN \ u \implies SN \ [u]$

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* $\tau 1$ $\tau 2$)**next**

case (*T* σ)

{ **case** 1 — follows from the fact that *stack.Id* \in *SRED* σ

have *ih-CR1- σ* : *CR1* σ **by** *fact*

{ **fix** t **assume** *t-red*: $t \in$ *RED* (*T* σ)

{ **fix** s **assume** $s \in$ *RED* σ

hence *SN* s **using** *ih-CR1- σ* **by** (*auto simp add: CR1-def*)

hence *SN* ($[s]$) **by** (*rule SN-Ret*)

hence *SN* ($[s] \star Id$) **by** *simp*

} **hence** *Id* \in *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 redcuton

{ **fix** $t t'$:*trm* **assume** *t-red*: $t \in$ *RED* (*T* σ) **and** $t \mapsto t'$

{ **fix** k **assume** $k \in$ *SRED* σ

with *t-red* **have** *SN*($t \star k$) **by** *simp*

moreover from $t \mapsto t'$ **have** $t \star k \mapsto t' \star k$..

ultimately have *SN*($t' \star k$) **by** (*rule SN-preserved*)

} **hence** $t' \in$ *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*: $\bigwedge t' . t \mapsto t' \implies t' \in$ *RED* (*T* σ)

and *neut-t*: *NEUT* t

{ **fix** k **assume** *k-red*: $k \in$ *SRED* σ

fix x **have** *NEUT* (*Var* x) **unfolding** *NEUT-def* **by** *simp*

hence *Var* $x \in$ *RED* σ **using** *normal-var ih-CR4- σ*

by (*simp add: CR4-def*)

hence *SN* ($[Var x] \star k$) **using** *k-red* **by** *simp*

hence *SSN* k **by** (*rule SN-SSN*)

then have *SN* ($t \star k$) **using** *k-red*

proof (*induct k rule:SSN.induct*)

case (*SSN-intro* k)

have *ih* : $\bigwedge k' . \llbracket k \mapsto k' ; k' \in$ *SRED* $\sigma \rrbracket \implies$ *SN* ($t \star k'$)

and *k-red*: $k \in$ *SRED* σ **by** *fact+*

{ **fix** r **assume** r : $t \star k \mapsto r$

hence *SN* r **using** *neut-t*

proof (*cases rule:dismantle-cases'*)

case (*T* t') **hence** $t \mapsto t'$ **and** *r-def*: $r = t' \star k$.

from $t \mapsto t'$ **have** $t' \in$ *RED* (*T* σ) **by** (*rule t'-red*)

thus *SN* r **using** *k-red r-def* **by** *simp*

next

case (*K* k') **hence** $k \mapsto k'$ **and** *r-def*: $r = t \star k'$.

{ **fix** s **assume** $s \in$ *RED* σ

hence *SN* ($[s] \star k$) **using** *k-red*

by *simp*

moreover have $[s] \star k \mapsto [s] \star k'$

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 β -reduction only. However, the differences are only minor.

```

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

```

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

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 $\text{SN } p$ and $\text{SN } (n[x::=p] \star K)$. 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

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

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

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

Proof. We show by induction on $|K| + \max(n \star K) + \max(p)$ that the reducts of $[p] \text{ to } x \text{ in } n \star K$ 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
- $T.\text{assoc}$ in the case where $K = [y]m \gg K'$ with $x \notin \text{fv}(m)$; giving the reduct $[p] \text{ to } x \text{ in } (n \text{ to } y \text{ in } m) \star K$. We aim to apply the induction hypothesis with K' and $(n \text{ to } y \text{ in } m)$ for K and n respectively. Now

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

which is strongly normalizing by induction hypothesis. Also

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

as $|K'| < |K|$ and $(n \text{ to } y \text{ in } m) \star K' = n \star K$. This last equation explains the use of $\max(n \star K)$; it remains fixed under $T.\text{assoc}$ unlike $\max(K)$ and $\max(n)$. Applying the induction hypothesis gives $SN([p] \text{ to } x \text{ in } (n \text{ to } y \text{ in } m) \star K)$ 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  $\bigwedge q K . SN\ q \implies P\ p\ q\ K$ 
proof (induct p)
  case (SN-intro p)
  have  $sn1: \bigwedge p' q K . \llbracket p \mapsto p'; SN\ q \rrbracket \implies P\ p' q K$  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  $sn2: \bigwedge q' K . \llbracket q \mapsto q'; SN\ q' \rrbracket \implies P\ p\ q' K$  by fact
    show  $P\ p\ q\ K$ 
    proof (induct K rule: measure-induct-rule[where f=length])
      case (less k)
      have  $le: \bigwedge k' . |k'| < |k| \implies P\ p\ q\ k'$  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  $?case$  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  $f: y \# (t, k, r) \quad x \# (z, t, k, r) \quad z \# (t, k, r)$ 
and  $T: \bigwedge t' . \llbracket t \mapsto t'; r = t' \star k \rrbracket \implies P$ 
and  $K: \bigwedge k' . \llbracket k \mapsto k'; r = t \star k' \rrbracket \implies P$ 
and  $B: \bigwedge s n l . \llbracket t = [s];$ 
       $k = [y]n \gg l; r = (n[y::=s]) \star l \rrbracket \implies P$ 
and  $A: \bigwedge u v n l .$ 
       $\llbracket x \# (z, n); t = u\ to\ x\ in\ v; k = [z]n \gg l;$ 
       $r = (u\ to\ x\ in\ (v\ to\ z\ in\ n)) \star l \rrbracket \implies P$ 
shows  $P$ 
proof (cases rule:dismantle-cases[of t k r P])
  case ( $\not\llcorner s\ y'\ n\ L$ ) 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** $y = y' \vee y \# n$
by (*auto simp add: fresh-prod abs-fresh*)
hence $n[y'::=s] = [(y, y')] \cdot n[y::=s]$
and $[y']n \gg L = [y]([(y, y')] \cdot n) \gg L$
by (*auto simp add: name-swap-bij subst-rename' stack.inject alpha'*)
with ch **have** $t = [s]$
 $k = [y]([(y, y')] \cdot n) \gg L$
 $r = [(y, y')] \cdot n[y::=s] \star L$
by (*auto*)
thus P **by** (*rule B*)
next
case ($5 u x' v z' n L$) **have** ch :
 $x' \# z' \quad x' \# n$
 $t = u \text{ to } x' \text{ in } v$
 $k = [z']n \gg L$
 $r = (u \text{ to } x' \text{ in } v \text{ to } z' \text{ in } n) \star L$ **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** $x: x = x' \vee x \# v \text{ to } z' \text{ in } n$
and $z: z = z' \vee z \# n$
by (*auto simp add: fresh-prod abs-fresh*)
from f ch **have** $x': x' \# n \quad x' \# z'$
and $xz': x = z' \vee x \# n$
by (*auto simp add: name-swap-bij alpha fresh-prod fresh-atm abs-fresh*)
from f ch **have** $x \# z \quad x \# [z'].n$ **by** (*auto simp add: fresh-prod*)
with $xz' z$ **have** $x \# (z, [(z, z')] \cdot n)$
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** $t = u \text{ to } x \text{ in } [(x, x')] \cdot v$
by (*auto simp add: name-swap-bij alpha'*)
moreover from z ch **have** $k = [z]([(z, z')] \cdot n) \gg L$
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**
 $u \text{ to } x' \text{ in } (v \text{ to } z' \text{ in } n) = u \text{ to } x \text{ in } [(x, x')] \cdot (v \text{ to } z' \text{ in } n)$
by (*auto simp add: name-swap-bij alpha' simp del: trm.perm*)
also from $xz' x'$ **have** $\dots = u \text{ to } x \text{ in } ([x, x'] \cdot v) \text{ to } z' \text{ in } n$
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** $\dots = u \text{ to } x \text{ in } ([x, x'] \cdot v) \text{ to } z \text{ in } [(z, z')] \cdot n$
by (*auto simp add: abs-fun-eq1 alpha' name-swap-bij*)
finally
have $r = (u \text{ to } x \text{ in } ([x, x'] \cdot v) \text{ to } z \text{ in } [(z, z')] \cdot n) \star L$
using ch **by** (*simp del: trm.inject*) }
ultimately show P
by (*rule A[where n=[(z, z')] \cdot n and v=[(x, x')] \cdot v]*)
qed (*insert r T K, auto*)

The lemma in Figure 2 assumes $SN (n[x::=p] \star K)$ but the actual induction is 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** $q\text{-def}$.
thus $SN\ t$ **using** $q\text{-def}$
proof (*induct* q *arbitrary*: t)
case ($SN\text{-intro}$ t)
hence $ih: \bigwedge t'. \llbracket t[x::=v] \mapsto t'[x::=v] \rrbracket \implies SN\ t'$ **by** *auto*
{ fix t' **assume** $t \mapsto t'$
hence $t[x::=v] \mapsto t'[x::=v]$ **by** (*rule* $reduction\text{-subst}$)
hence $SN\ t'$ **by** (*rule* ih) **}**
thus $SN\ t$..
qed
qed

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

abbreviation
 $redrtrans :: trm \Rightarrow trm \Rightarrow bool\ (- \mapsto^* -)$
where $redrtrans \equiv reduction\widehat{**}$

To be able to handle the case where p makes a step, we need to establish $p \mapsto p' \implies m[x::=p] \mapsto^* m[x::=p']$ 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 $r1\text{-trans}: s \mapsto^* s' \implies App\ s\ t \mapsto^* App\ s'\ t$
and $r2\text{-trans}: t \mapsto^* t' \implies App\ s\ t \mapsto^* App\ s\ t'$
and $r4\text{-trans}: t \mapsto^* t' \implies \Lambda\ x.\ t \mapsto^* \Lambda\ x.\ t'$
and $r6\text{-trans}: s \mapsto^* s' \implies s\ to\ x\ in\ t \mapsto^* s'\ to\ x\ in\ t$
and $r7\text{-trans}: \llbracket t \mapsto^* t' \rrbracket \implies s\ to\ x\ in\ t \mapsto^* s\ to\ x\ in\ t'$
and $r11\text{-trans}: s \mapsto^* s' \implies [s] \mapsto^* ([s'])$
by – (*induct* $rule: rtranclp\text{-induct}$, (*auto* $intro: transitive\text{-closurep-trans}'[2]$))+

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

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

lemma $SN-trans : \llbracket p \mapsto^* p' ; SN\ p \rrbracket \implies SN\ p'$
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 $x: x \# p\ x \# k$
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 $\bigwedge m . \llbracket q = m \star k ; SN(m[x::=p] \star k) \rrbracket$
 $\implies SN\ (([p]\ to\ x\ in\ m) \star k)$
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 $ih-p$:
 $\bigwedge p' m . \llbracket p \mapsto p' ; q = m \star k ; SN\ (m[x::=p'] \star k) ; x \# p' ; x \# k \rrbracket$
 $\implies SN\ (([p']\ to\ x\ in\ m) \star k)$ **by** $fact$
have $ih-q$:
 $\bigwedge q' m k . \llbracket q \mapsto q' ; q' = m \star k ; SN\ (m[x::=p] \star k) ; x \# p ; x \# k \rrbracket$
 $\implies SN\ (([p]\ to\ x\ in\ m) \star k)$ **by** $fact$
have $ih-k$:
 $\bigwedge k' m . \llbracket |k'| < |k| ; q = m \star k' ; SN\ (m[x::=p] \star k') ; x \# p ; x \# k' \rrbracket$
 $\implies SN\ (([p]\ to\ x\ in\ m) \star k')$ **by** $fact$
have $q: q = m \star k$ **and** $sn: SN\ (m[x::=p] \star k)$ **by** $fact+$
have $xp: x \# p$ **and** $xk: x \# k$ **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** $x1 : x \# ([p]\ to\ x\ in\ m) \star k$
by $(simp\ add: abs-fresh)$
with red **have** $x2: x \# r$ **by** $(rule\ reduction-fresh)$
obtain $z::name$ **where** $z: z \# (x,p,m,k,r)$
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 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 $xl:(x \# [p]$ *to* x *in* $m)$ **and** $xr:x \# r'$ **by** *auto*

from z **have** $zl: z \# ([p]$ *to* x *in* $m)$ $x \neq z$

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

with r' **have** $zr: z \# r'$ **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 reasoning 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** $xp': x \# p'$ **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 $\langle m \mapsto m' \rangle$ **have** $q \mapsto m' \star k$ **by** (*simp add: dismantle-red*)

moreover **have** $m' \star k = m' \star k$.. — a triviality

moreover { **from** $\langle m \mapsto m' \rangle$ **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** $x \# k'$
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 $u: x \# u$
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** $x \# p$ **and** $x \# ?k$ **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** $\langle x \# k \rangle$ **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 σ)
and $t: \forall p \in RED$ $\sigma . t[x::=p] \in RED$ (T τ)
shows s to x in $t \in RED$ (T τ)

proof —

{ **fix** K **assume** $k: K \in SRED$ τ
{ **fix** p **assume** $p: p \in RED$ σ
hence $snp: SN$ p **using** *RED-props* **by**(*simp add: CR1-def*)
obtain $x':name$ **where** $x: x' \# (t, p, K)$
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]) · 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≫K by simp
  ultimately have snx: SN([p] ★ [x]t≫K) using snx'
    by (simp del: trm.inject)
} hence [x]t≫K ∈ 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 ∈ 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 ⇒ name ⇒ trm ⇒ bool (- maps - to - [55,55,55] 55)
where
  ∅ maps x to e ≡ (lookup ∅ x) = e

```

abbreviation

```

closes :: (name × trm) list ⇒ (name × ty) list ⇒ bool (- closes - [55,55] 55)
where
  ∅ closes Γ ≡ ∀ x τ. ((x, τ) ∈ set Γ ⟶ (∃ t. ∅ maps x to t ∧ t ∈ RED τ))

```

theorem fundamental-theorem:

```

assumes a: Γ ⊢ t : τ and b: ∅ closes Γ
shows ∅<t> ∈ RED τ
using a b
proof (nominal-induct avoiding: ∅ rule: typing.strong-induct)
  case (t3 a Γ σ t τ ∅) — lambda case
next
  case (t5 x Γ s σ t τ ∅) — to case
  have ihs : ∧ ∅ . ∅ closes Γ ⟹ ∅<s> ∈ RED (T σ) by fact
  have iht : ∧ ∅ . ∅ closes ((x, σ) # Γ) ⟹ ∅<t> ∈ RED (T τ) by fact
  have ∅-cond: ∅ closes Γ by fact
  have fresh: x # ∅ x # Γ x # s by fact+
  from ihs have ∅<s> ∈ RED (T σ) using ∅-cond by simp
  moreover
  { from iht have ∀ s ∈ RED σ. ((x, s) # ∅)<t> ∈ RED (T τ)
    using fresh ∅-cond fresh-context by simp
    hence ∀ s ∈ RED σ. ∅<t>[x::=s] ∈ RED (T τ)
      using fresh by (simp add: psubst-subst) }
  ultimately have (∅<s>) to x in (∅<t>) ∈ RED (T τ) by (simp only: to-RED)
  thus ∅<s to x in t> ∈ 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 ⇒ (name × trm) list

```

where
 $id \ [] = []$
 $| id ((x,\tau)\#\Gamma) = (x, Var x)\#(id \Gamma)$

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

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

lemma *id-apply*:
shows $(id \ \Gamma)\langle t \rangle = t$
by $(nominal-induct \ t \ avoiding: \ \Gamma \ rule: \ trm.strong-induct)$
 $(auto \ simp \ add: \ id-maps \ id-fresh)$

lemma *id-closes*:
shows $(id \ \Gamma)$ closes Γ
proof –
 $\{$ **fix** $x \ \tau$ **assume** $(x,\tau) \in set \ \Gamma$
have $CR4 \ \tau$ **by** $(simp \ add: \ RED-props \ CR3-implies-CR4)$
hence $Var \ x \in RED \ \tau$
by $(auto \ simp \ add: \ NEUT-def \ normal-var \ CR4-def)$
hence $(id \ \Gamma)$ maps x to $Var \ x \wedge Var \ x \in RED \ \tau$
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 $t \in RED \ \tau$
proof –
have $(id \ \Gamma)\langle t \rangle \in RED \ \tau$
proof –
have $(id \ \Gamma)$ closes Γ **by** $(rule \ id-closes)$
with a **show** *?thesis* **by** $(rule \ fundamental-theorem)$
qed
thus $t \in RED \ \tau$ **by** $(simp \ add: \ id-apply)$
qed

theorem *strong-normalization*:
assumes $a: \Gamma \vdash t : \tau$
shows $SN(t)$
proof –
from a **have** $t \in RED \ \tau$ **by** $(rule \ typing-implies-RED)$
moreover **have** $CR1 \ \tau$ **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 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

- [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 \$\top\top\$ -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.