Strong Normalization of Moggis's Computational Metalanguage

Christian Doczkal
Saarland University

December 14, 2021

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* ♯ *t*
  shows [*y*].*t* = [*x*].([(*y*,*x*)] ∙ *t*)
⟨*proof*⟩

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

nominal-datatype *ty* =
  TBase
| *TFun ty ty* (**infix** → *200*)
| *T ty*

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

inductive
  valid :: (*name*×*ty*) *list* ⇒ *bool*
where
  v1[*intro*]: *valid* []
| *v2*[*intro*]: ⟦*valid* Γ;*x*♯Γ⟧⟹ *valid* ((*x*,σ)#Γ)
equivariance *valid*

lemma *fresh-ty*:
  fixes *x* :: *name* **and** τ::*ty*
  shows *x* ♯ τ
⟨*proof*⟩

lemma *fresh-context*:
  fixes $\Gamma :: (name \times ty)list$
  assumes *a*: $x \sharp \Gamma$
  shows $\neg(\exists\ \tau\ .\ (x,\tau) \in set\ \Gamma)$
⟨*proof*⟩

inductive
  typing :: $(name \times ty)\ list \Rightarrow trm \Rightarrow ty \Rightarrow bool$ (- ⊢ - : - [60,60,60] 60)
where
  t1[*intro*]: $[\![valid\ \Gamma;\ (x,\tau) \in set\ \Gamma]\!] \Longrightarrow \Gamma \vdash Var\ x : \tau$
| *t2*[*intro*]: $[\![\Gamma \vdash s : \tau \to \sigma;\ \Gamma \vdash t : \tau]\!] \Longrightarrow \Gamma \vdash App\ s\ t : \sigma$
| *t3*[*intro*]: $[\![x \sharp \Gamma;\ ((x,\tau)\#\Gamma) \vdash t : \sigma]\!] \Longrightarrow \Gamma \vdash \Lambda\ x\ .\ t : \tau \to \sigma$
| *t4*[*intro*]: $[\![\ \Gamma \vdash s : \sigma\ ]\!] \Longrightarrow \Gamma \vdash [s] : T\ \sigma$
| *t5*[*intro*]: $[\![x \sharp (\Gamma,s);\ \Gamma \vdash s : T\ \sigma\ ;\ ((x,\sigma)\#\Gamma) \vdash t : T\ \tau\ ]\!]$
  $\Longrightarrow \Gamma \vdash s\ to\ x\ in\ t : T\ \tau$
equivariance *typing*
nominal-inductive *typing*
  ⟨*proof*⟩

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

lemma *lookup-eqvt*[*eqvt*]:
  fixes *pi*::*name prm*
  and ϑ::$(name \times trm)\ list$
  and x::*name*
  shows $pi \cdot (lookup\ \vartheta\ x) = lookup\ (pi \cdot \vartheta)\ (pi \cdot x)$
⟨*proof*⟩

nominal-primrec
  psubst :: $(name \times trm)\ list \Rightarrow trm \Rightarrow trm$ (-<-> [95,95] 205)
where
    $\vartheta{<}Var\ x{>} = lookup\ \vartheta\ x$
  | $\vartheta{<}App\ s\ t{>} = App\ (\vartheta{<}s{>})\ (\vartheta{<}t{>})$
  | $x \sharp \vartheta \Longrightarrow \vartheta{<}\Lambda\ x\ .s{>} = \Lambda\ x\ .\ (\vartheta{<}s{>})$
  | $\vartheta{<}[t]{>} = [\vartheta{<}t{>}]$
  | $[\![\ x \sharp \vartheta\ ;\ x \sharp\ t\ ]\!] \Longrightarrow \vartheta{<}t\ to\ x\ in\ s{>} = (\vartheta{<}t{>})\ to\ x\ in\ (\vartheta{<}s{>})$
⟨*proof*⟩

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

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 (*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*]]
⟨*proof*⟩

lemma *subst-rename*:
  assumes *a*: *y* ♯ *t*
  shows ([(*y*,*x*)]•*t*)[*y*::=*v*] = *t*[*x*::=*v*]
⟨*proof*⟩
lemmas *subst-rename′* = *subst-rename*[*THEN sym*]

lemma *forget*: *x* ♯ *t* ⟹ *t*[*x*::=*v*] = *t*
⟨*proof*⟩

lemma *fresh-fact*:
  fixes *x*::*name*
  assumes *x*: *x* ♯ *v* *x* ♯ *t*
  shows *x* ♯ *t*[*y*::=*v*]
⟨*proof*⟩

lemma *fresh-fact′*:
  fixes *x*::*name*
  assumes *x*: *x* ♯ *v*
  shows *x* ♯ *t*[*x*::=*v*]
⟨*proof*⟩

lemma *subst-lemma*:
  assumes *a*: *x*≠*y*
  and *b*: *x* ♯ *u*
  shows *s*[*x*::=*v*][*y*::=*u*] = *s*[*y*::=*u*][*x*::=*v*[*y*::=*u*]]
⟨*proof*⟩

lemma *id-subs*:
  shows *t*[*x*::=*Var x*] = *t*
⟨*proof*⟩

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*♯ϑ *z*♯*x*
  shows *z*♯ *lookup* ϑ *x*
⟨*proof*⟩

lemma *lookup-fresh′*:
  assumes *a*: *z*♯ϑ
  shows *lookup* ϑ *z* = *Var z*
⟨*proof*⟩

lemma *psubst-fresh-fact*:
  fixes *x* :: *name*

assumes *a*: $x \sharp t$ **and** *b*: $x \sharp \vartheta$
shows $x \sharp \vartheta\langle t\rangle$
⟨*proof*⟩

lemma *psubst-subst*:
assumes *a*: $x \sharp \vartheta$
shows $\vartheta\langle t\rangle[x::=s] = ((x,s)\#\vartheta)\langle t\rangle$
⟨*proof*⟩

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*!]:$s \rightsquigarrow s' \Longrightarrow App\ s\ t \rightsquigarrow App\ s'\ t$
| *std-r2*[*intro*!]:$t \rightsquigarrow t' \Longrightarrow App\ s\ t \rightsquigarrow App\ s\ t'$
| *std-r3*[*intro*!]:$App\ (\Lambda\ x\ .\ t)\ s \rightsquigarrow t[x::=s]$

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

| *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* ⇒ *trm* ⇒ *bool* (- ↦ - [*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*!]:$x \sharp s \Longrightarrow App\ (\Lambda\ x\ .\ t)\ s \mapsto t[x::=s]$

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

| *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*
⟨*proof*⟩

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*⟩
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
⟨*proof*⟩
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'
⟨*proof*⟩
declare *r7*[*rule del*]

lemma *r8'*[*intro*!]: $[s]$ *to* x *in* $t \mapsto t[x::=s]$
⟨*proof*⟩
declare *r8*[*rule del*]

lemma *r9'*[*intro*!]: s *to* x *in* $[Var\ x] \mapsto s$
⟨*proof*⟩
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*⟩
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*: $s \mapsto t = s \rightsquigarrow t$

⟨*proof*⟩

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 $x \sharp t \Longrightarrow x \sharp t'$
⟨*proof*⟩

lemma *reduction-subst*:
  assumes a: $t \mapsto t'$
  shows $t[x::=v] \mapsto t'[x::=v]$
⟨*proof*⟩

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* ⇒ *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 $SN\ t'$
⟨*proof*⟩

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

lemma *normal-var*: $NORMAL\ (Var\ x)$
⟨*proof*⟩

lemma *normal-implies-sn* : $NORMAL\ s \Longrightarrow SN\ s$
⟨*proof*⟩

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* | *St* «*name*»*trm stack* ([-]-≫-)

lemma *stack-exhaust* :
  fixes c :: $'a$::*fs-name*
  shows $k = Id \vee (\exists\ y\ n\ l\ .\ y \sharp l \wedge y \sharp c \wedge k = [y]n \gg l)$

⟨*proof*⟩

nominal-primrec
  length :: *stack* ⇒ *nat* (|-|)
where
  |*Id*| = *0*
| *y* ♯ *L* ⟹ *length* ([*y*]*n*≫*L*) = *1* + |*L*|
⟨*proof*⟩

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*⟩

termination *dismantle*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *dismantle-fresh*[*iff*]:
  fixes *x* :: *name*
  shows (*x* ♯ (*t* ⋆ *k*)) = (*x* ♯ *t* ∧ *x* ♯ *k*)
⟨*proof*⟩

lemma *dismantle-simp*[*simp*]: *s* ⋆ [*y*]*n*≫*L* = (*s to y in n*) ⋆ *L*
⟨*proof*⟩

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 ↦ *k*′ ≡ ∀ (*t*::*trm*) . (*t* ⋆ *k*) ↦ (*t* ⋆ *k*′)

lemma *stack-reduction-fresh*:
  fixes *k* :: *stack* **and** *x* :: *name*
  assumes *r* : *k* ↦ *k*′ **and** *f* :*x* ♯ *k*
  shows *x* ♯ *k*′
⟨*proof*⟩

lemma *dismantle-red*[*intro*]:
fixes $m :: trm$
assumes r: $m \mapsto m'$
shows $m \star k \mapsto m' \star k$
⟨*proof*⟩

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

nominal-primrec
$ssubst :: name \Rightarrow trm \Rightarrow stack \Rightarrow stack$
where
$ssubst\ x\ v\ Id = Id$
| $y \sharp (k,x,v) \Longrightarrow ssubst\ x\ v\ ([y]n \gg k) = [y](n[x::=v]) \gg (ssubst\ x\ v\ k)$
⟨*proof*⟩

lemma *ssubst-fresh*:
fixes $y :: name$
assumes $y \sharp (x,v,k)$
shows $y \sharp ssubst\ x\ v\ k$
⟨*proof*⟩

lemma *ssubst-forget*:
fixes $x :: name$
assumes $x \sharp k$
shows $ssubst\ x\ v\ k = k$
⟨*proof*⟩

lemma *subst-dismantle*[*simp*]: $(t \star k)[x ::= v] = (t[x::=v]) \star ssubst\ x\ v\ k$
⟨*proof*⟩

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*: $ty = TBase \vee (\exists\ \sigma\ \tau\ .\ ty = \sigma \to \tau) \vee (\exists\ \sigma\ .\ ty = T\ \sigma)$
⟨*proof*⟩

function $RED :: ty \Rightarrow trm\ set$
and $SRED :: ty \Rightarrow stack\ set$
where
$RED\ (TBase) = \{t.\ SN(t)\}$
| $RED\ (\tau \to \sigma) = \{t.\ \forall\ u \in RED\ \tau\ .\ (App\ t\ u) \in RED\ \sigma\ \}$
| $RED\ (T\ \sigma) = \{t.\ \forall\ k \in SRED\ \sigma\ .\ SN(t \star k)\ \}$
| $SRED\ \tau = \{k.\ \forall\ t \in RED\ \tau\ .\ SN\ ([t] \star k)\ \}$
⟨*proof*⟩

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 $SRED\ \tau$ involves a recursive call to $RED\ \tau$ without reducing the size of τ.

nominal-primrec
  tsize :: *ty* ⇒ *nat*
where
    tsize TBase = *1*
  | *tsize* (*σ*→*τ*) = *1* + *tsize σ* + *tsize τ*
  | *tsize* (*T τ*) = *1* + *tsize τ*
⟨*proof*⟩

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

termination *RED*
⟨*proof*⟩

6 Properties of the Reducibility Relation

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

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

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

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

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

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

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

lemma *CR3-implies-CR4*[*intro*]: *CR3 τ* ⟹ *CR4 τ*
⟨*proof*⟩

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*⟩

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: $(\bigwedge k' . k \mapsto k' \Longrightarrow SSN\ k') \Longrightarrow SSN\ k$

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-DIS :: *trm* ⇒ *stack* ⇒ *bool* (- ▷ -)
where
  snd-dis[*intro*!]: $t \star k \triangleright k$

lemma *SN-SSN*:
  assumes *a*: $SN\ (t \star k)$
  shows $SSN\ k$
⟨*proof*⟩

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.

$$\frac{\begin{array}{ll} t \star k \mapsto r & \bigwedge t'.\ [\![t \mapsto t';\ r = t' \star k]\!] \Longrightarrow P \\ NEUT\ t & \bigwedge k'.\ [\![\ k \mapsto k'\ ;\ r = t \star k']\!] \Longrightarrow P \end{array}}{P}$$

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*: $\bigwedge t' . [\![\ t \mapsto t'\ ;\ r = t' \star k\ ]\!] \Longrightarrow P$
  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*: $\bigwedge u\ x\ v\ y\ n\ l.[\![\ x \sharp y;\ x \sharp n\ ;\ t = u\ to\ x\ in\ v\ ;$
    $k = [y]n \gg l\ ;\ r = (u\ to\ x\ in\ (v\ to\ y\ in\ n)) \star l\ ]\!] \Longrightarrow P$
  shows P
⟨*proof*⟩

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

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]

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
⟨*proof*⟩

lemma *red-Ret*:
fixes t :: *trm*
assumes $[s] \mapsto t$
shows $\exists\ s' . t = [s'] \wedge s \mapsto s'$
⟨*proof*⟩

lemma *SN-Ret*: $SN\ u \Longrightarrow SN\ [u]$
⟨*proof*⟩

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*⟩⟨*proof*⟩⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *abs-RED*:
assumes *asm*: $\forall\, s{\in}RED\ \tau.\ t[x::=s]{\in}RED\ \sigma$
shows $\Lambda\ x\ .\ t\ {\in}RED\ (\tau{\rightarrow}\sigma)$

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 $SN\ p$ 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*)

[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]

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*⟩

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*⟩

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*⟩

lemma *sn-forget'*:
assumes sn: $SN\ (t[x::=p] \star k)$
and x: $x \sharp k$
shows $SN\ (t \star k)$
⟨*proof*⟩

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

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*: $s \mapsto^* s' \Longrightarrow [s] \mapsto^* ([s'])$
⟨*proof*⟩

lemma *red-subst*: $p \mapsto p' \Longrightarrow (m[x::=p]) \mapsto^* (m[x::=p'])$
⟨*proof*⟩

lemma *SN-trans* : $[\![\ \ p \mapsto^* p'\ ;\ SN\ p\ ]\!] \Longrightarrow SN\ p'$
⟨*proof*⟩

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*⟩

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*⟩

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
$mapsto :: (name \times trm)\ list \Rightarrow name \Rightarrow trm \Rightarrow bool\ (\text{-}\ maps\ \text{-}\ to\ \text{-}\ [55,55,55]\ 55)$
where
$\vartheta\ maps\ x\ to\ e \equiv (lookup\ \vartheta\ x) = e$

abbreviation
$closes :: (name \times trm)\ list \Rightarrow (name \times ty)\ list \Rightarrow bool\ (\text{-}\ closes\ \text{-}\ [55,55]\ 55)$
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 $\vartheta\langle t\rangle \in RED\ \tau$
⟨*proof*⟩
⟨*proof*⟩

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*,τ)#Γ) = (*x*, *Var x*)#(*id* Γ)

lemma *id-maps*:
  shows (*id* Γ) *maps a to* (*Var a*)
⟨*proof*⟩

lemma *id-fresh*:
  fixes *x*::*name*
  assumes *x*: *x* ♯ Γ
  shows *x* ♯ (*id* Γ)
⟨*proof*⟩

lemma *id-apply*:
  shows (*id* Γ)<*t*> = *t*
⟨*proof*⟩

lemma *id-closes*:
  shows (*id* Γ) *closes* Γ
⟨*proof*⟩

9.1 Strong normalization theorem

lemma *typing-implies-RED*:
  assumes *a*: Γ ⊢ *t* : τ
  shows *t* ∈ *RED* τ
⟨*proof*⟩

theorem *strong-normalization*:
  assumes *a*: Γ ⊢ *t* : τ
  shows *SN*(*t*)
⟨*proof*⟩

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 $\top\top$-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.