Normalization by Evaluation

Klaus Aehlig and Tobias Nipkow

March 17, 2025

Abstract

This article formalizes normalization by evaluation as implemented in Isabelle. Lambda calculus plus term rewriting is compiled into a functional program with pattern matching. It is proved that the result of a successful evaluation is a) correct, i.e. equivalent to the input, and b) in normal form.

An earlier version of this theory is described in a paper by Aehlig *et al.* [1]. The normal form proof is not in that paper.

1 Terms

type-synonym *vname* = *nat*
type-synonym *ml-vname* = *nat*

type-synonym *cname* = *int*

ML terms:

datatype *ml* =
— ML
C-ML cname (‹C_{ML}›)
| *V-ML ml-vname* (‹V_{ML}›)
| *A-ML ml* (*ml list*) (‹A_{ML}›)
| *Lam-ML ml* (‹Lam_{ML}›)
— the universal datatype
| C_U *cname* (*ml list*)
| V_U *vname* (*ml list*)
| *Clo ml* (*ml list*) *nat*
— ML function *apply*
| *apply ml ml*

Lambda-terms:

datatype *tm* = *C cname* | *V vname* | Λ *tm* | *At tm tm* (**infix** ‹·› *100*)
| *term ml* — ML function `term`

The following locale captures type conventions for variables. It is not actually used, merely a formal comment.

locale *Vars* =
fixes *r s t*:: *tm*
and *rs ss ts* :: *tm list*
and *u v w* :: *ml*
and *us vs ws* :: *ml list*
and *nm* :: *cname*
and *x* :: *vname*
and *X* :: *ml-vname*

The subset of pure terms:

inductive *pure* :: *tm* ⇒ *bool* **where**
pure(*C nm*) |
pure(*V x*) |
Lam: *pure t* ⟹ *pure*(Λ *t*) |
pure s ⟹ *pure t* ⟹ *pure*(*s*•*t*)

declare *pure.intros*[*simp*]
declare *Lam*[*simp del*]

lemma *pure-Lam*[*simp*]: *pure*(Λ *t*) = *pure t*
proof
assume *pure*(Λ *t*) **thus** *pure t*
proof *cases* **qed** *auto*
next
assume *pure t* **thus** *pure*(Λ *t*) **by**(*rule Lam*)
qed

Closed terms w.r.t. ML variables:

fun *closed-ML* :: *nat* ⇒ *ml* ⇒ *bool* (‹*closed*$_{ML}$›) **where**
closed$_{ML}$ *i* (*C*$_{ML}$ *nm*) = *True* |
closed$_{ML}$ *i* (*V*$_{ML}$ *X*) = (*X*<*i*) |
closed$_{ML}$ *i* (*A*$_{ML}$ *v vs*) = (*closed*$_{ML}$ *i v* ∧ (∀ *v* ∈ *set vs*. *closed*$_{ML}$ *i v*)) |
closed$_{ML}$ *i* (*Lam*$_{ML}$ *v*) = *closed*$_{ML}$ (*i*+*1*) *v* |
closed$_{ML}$ *i* (*C*$_{U}$ *nm vs*) = (∀ *v* ∈ *set vs*. *closed*$_{ML}$ *i v*) |
closed$_{ML}$ *i* (*V*$_{U}$ *nm vs*) = (∀ *v* ∈ *set vs*. *closed*$_{ML}$ *i v*) |
closed$_{ML}$ *i* (*Clo f vs n*) = (*closed*$_{ML}$ *i f* ∧ (∀ *v* ∈ *set vs*. *closed*$_{ML}$ *i v*)) |
closed$_{ML}$ *i* (*apply v w*) = (*closed*$_{ML}$ *i v* ∧ *closed*$_{ML}$ *i w*)

fun *closed-tm-ML* :: *nat* ⇒ *tm* ⇒ *bool* (‹*closed*$_{ML}$›) **where**
closed-tm-ML i (*r*•*s*) = (*closed-tm-ML i r* ∧ *closed-tm-ML i s*) |
closed-tm-ML i (Λ *t*) = (*closed-tm-ML i t*) |
closed-tm-ML i (*term v*) = *closed-ML i v* |
closed-tm-ML i v = *True*

Free variables:

fun *fv-ML* :: *ml* ⇒ *ml-vname set* (‹*fv*$_{ML}$›) **where**
fv$_{ML}$ (*C*$_{ML}$ *nm*) = {} |
fv$_{ML}$ (*V*$_{ML}$ *X*) = {*X*} |
fv$_{ML}$ (*A*$_{ML}$ *v vs*) = *fv*$_{ML}$ *v* ∪ (⋃ *v* ∈ *set vs*. *fv*$_{ML}$ *v*) |

$fv_{ML}\ (Lam_{ML}\ v) = \{X.\ Suc\ X : fv_{ML}\ v\}$ |
$fv_{ML}\ (C_U\ nm\ vs) = (\bigcup v \in set\ vs.\ fv_{ML}\ v)$ |
$fv_{ML}\ (V_U\ nm\ vs) = (\bigcup v \in set\ vs.\ fv_{ML}\ v)$ |
$fv_{ML}\ (Clo\ f\ vs\ n) = fv_{ML}\ f \cup (\bigcup v \in set\ vs.\ fv_{ML}\ v)$ |
$fv_{ML}\ (apply\ v\ w) = fv_{ML}\ v \cup fv_{ML}\ w$

primrec $fv :: tm \Rightarrow vname\ set$ **where**
$fv\ (C\ nm) = \{\}$ |
$fv\ (V\ X) = \{X\}$ |
$fv\ (s \cdot t) = fv\ s \cup fv\ t$ |
$fv\ (\Lambda\ t) = \{X.\ Suc\ X : fv\ t\}$

1.1 Iterated Term Application

abbreviation *foldl-At* (**infix** $\langle\cdot\cdot\rangle$ *90*) **where**
$t \cdot\cdot ts \equiv foldl\ (\cdot)\ t\ ts$

Auxiliary measure function:

primrec *depth-At* :: $tm \Rightarrow nat$
where
depth-At$(C\ nm) = 0$
| *depth-At*$(V\ x) = 0$
| *depth-At*$(s \cdot t) =$ *depth-At* $s + 1$
| *depth-At*$(\Lambda\ t) = 0$
| *depth-At*$(term\ v) = 0$

lemma *depth-At-foldl*:
depth-At$(s \cdot\cdot ts) =$ *depth-At* $s +$ *size* ts
by (*induct ts arbitrary*: s) *simp-all*

lemma *foldl-At-eq-lemma*: *size* $ts =$ *size* $ts' \Longrightarrow$
$s \cdot\cdot ts = s' \cdot\cdot ts' \longleftrightarrow s = s' \wedge ts = ts'$
by (*induct arbitrary*: $s\ s'$ *rule*:*list-induct2*) *simp-all*

lemma *foldl-At-eq-length*:
$s \cdot\cdot ts = s \cdot\cdot ts' \Longrightarrow$ *length* $ts =$ *length* ts'
apply(*subgoal-tac depth-At*$(s \cdot\cdot ts) =$ *depth-At*$(s \cdot\cdot ts')$)
apply(*erule thin-rl*)
apply (*simp add*:*depth-At-foldl*)
apply *simp*
done

lemma *foldl-At-eq*[*simp*]: $s \cdot\cdot ts = s \cdot\cdot ts' \longleftrightarrow ts = ts'$
apply(*rule*)
prefer *2* **apply** *simp*
apply(*blast dest*:*foldl-At-eq-lemma foldl-At-eq-length*)
done

lemma *term-eq-foldl-At*[*simp*]:
term $v = t \cdot\cdot ts \longleftrightarrow t =$ *term* $v \wedge ts = []$

by (*induct ts arbitrary*:*t*) *auto*

lemma *At-eq-foldl-At*[*simp*]:
  $r \cdot s = t \bullet\bullet ts \longleftrightarrow$
  (*if ts*=[] *then* $t = r \cdot s$ *else* $s = last\ ts \wedge r = t \bullet\bullet butlast\ ts$)
apply (*induct ts arbitrary*:*t*)
 apply *fastforce*
apply *rule*
 apply *clarsimp*
 apply *rule*
  apply *clarsimp*
 apply *clarsimp*
 apply(*subgoal-tac* $\exists ts'\ t'.\ ts = ts' @ [t']$)
  apply *clarsimp*
 defer
 apply (*clarsimp split*:*list.split*)
apply (*metis append-butlast-last-id*)
done

lemma *foldl-At-eq-At*[*simp*]:
  $t \bullet\bullet ts = r \cdot s \longleftrightarrow$
  (*if ts*=[] *then* $t = r \cdot s$ *else* $s = last\ ts \wedge r = t \bullet\bullet butlast\ ts$)
by(*metis At-eq-foldl-At*)

lemma *Lam-eq-foldl-At*[*simp*]:
  $\Lambda\ s = t \bullet\bullet ts \longleftrightarrow t = \Lambda\ s \wedge ts = []$
by (*induct ts arbitrary*:*t*) *auto*

lemma *foldl-At-eq-Lam*[*simp*]:
  $t \bullet\bullet ts = \Lambda\ s \longleftrightarrow t = \Lambda\ s \wedge ts = []$
by (*induct ts arbitrary*:*t*) *auto*

lemma [*simp*]: $s \cdot t \neq s$
apply(*subgoal-tac size*$(s \cdot t) \neq$ *size* s)
apply *metis*
apply *simp*
done

fun *atomic-tm* :: *tm* ⇒ *bool* **where**
atomic-tm($s \cdot t$) = *False* |
atomic-tm(-) = *True*

fun *head-tm* **where**
head-tm($s \cdot t$) = *head-tm* s |
head-tm(s) = s

fun *args-tm* **where**

args-tm($s \cdot t$) = *args-tm* s @ [t] |
args-tm(-) = []

lemma *head-tm-foldl-At*[*simp*]: *head-tm*(s •• ts) = *head-tm* s
by(*induct ts arbitrary*: s) *auto*

lemma *args-tm-foldl-At*[*simp*]: *args-tm*(s •• ts) = *args-tm* s @ ts
by(*induct ts arbitrary*: s) *auto*

lemma *tm-eq-iff*:
  atomic-tm(*head-tm* s) $\Longrightarrow$ *atomic-tm*(*head-tm* t)
  $\Longrightarrow$ $s = t \longleftrightarrow$ *head-tm* s = *head-tm* $t \wedge$ *args-tm* s = *args-tm* t
apply(*induct s arbitrary*: t)
apply(*case-tac* t, *simp*+)+
done

declare
  tm-eq-iff[*of h* •• *ts*, *simp*]
  tm-eq-iff[*of* - *h* •• *ts*, *simp*]
  for *h ts*

lemma *atomic-tm-head-tm*: *atomic-tm*(*head-tm* t)
by(*induct* t) *auto*

lemma *head-tm-idem*: *head-tm*(*head-tm* t) = *head-tm* t
by(*induct* t) *auto*

lemma *args-tm-head-tm*: *args-tm*(*head-tm* t) = []
by(*induct* t) *auto*

lemma *eta-head-args*: t = *head-tm* t •• *args-tm* t
by (*subst tm-eq-iff*) (*auto simp*: *atomic-tm-head-tm head-tm-idem args-tm-head-tm*)

lemma *tm-vector-cases*:
  ($\exists$ n ts. t = V n •• ts) $\vee$
  ($\exists$ nm ts. t = C nm •• ts) $\vee$
  ($\exists$ t' ts. t = Λ t' •• ts) $\vee$
  ($\exists$ v ts. t = *term* v •• ts)
apply(*induct* t)
apply *simp-all*
by (*metis snoc-eq-iff-butlast*)

lemma *fv-head-C*[*simp*]: *fv* (t •• ts) = *fv* $t \cup$ ($\bigcup t \in set\ ts$. *fv* t)
by(*induct ts arbitrary*:t) *auto*

1.2 Lifting and Substitution

fun *lift-ml* :: *nat* $\Rightarrow$ *ml* $\Rightarrow$ *ml* (‹*lift*›) **where**

$lift\ i\ (C_{ML}\ nm) = C_{ML}\ nm$ |
$lift\ i\ (V_{ML}\ X) = V_{ML}\ X$ |
$lift\ i\ (A_{ML}\ v\ vs) = A_{ML}\ (lift\ i\ v)\ (map\ (lift\ i)\ vs)$ |
$lift\ i\ (Lam_{ML}\ v) = Lam_{ML}\ (lift\ i\ v)$ |
$lift\ i\ (C_U\ nm\ vs) = C_U\ nm\ (map\ (lift\ i)\ vs)$ |
$lift\ i\ (V_U\ x\ vs) = V_U\ (if\ x < i\ then\ x\ else\ x{+}1)\ (map\ (lift\ i)\ vs)$ |
$lift\ i\ (Clo\ v\ vs\ n) = Clo\ (lift\ i\ v)\ (map\ (lift\ i)\ vs)\ n$ |
$lift\ i\ (apply\ u\ v) = apply\ (lift\ i\ u)\ (lift\ i\ v)$

lemmas *ml-induct* = *lift-ml.induct*[*of* $\lambda i\ v.\ P\ v$] **for** P

fun *lift-tm* :: $nat \Rightarrow tm \Rightarrow tm$ (‹*lift*›) **where**
$lift\ i\ (C\ nm) = C\ nm$ |
$lift\ i\ (V\ x) = V(if\ x < i\ then\ x\ else\ x{+}1)$ |
$lift\ i\ (s{\cdot}t) = (lift\ i\ s){\cdot}(lift\ i\ t)$ |
$lift\ i\ (\Lambda\ t) = \Lambda(lift\ (i{+}1)\ t)$ |
$lift\ i\ (term\ v) = term\ (lift\ i\ v)$

fun *lift-ML* :: $nat \Rightarrow ml \Rightarrow ml$ (‹$lift_{ML}$›) **where**
$lift_{ML}\ i\ (C_{ML}\ nm) = C_{ML}\ nm$ |
$lift_{ML}\ i\ (V_{ML}\ X) = V_{ML}\ (if\ X < i\ then\ X\ else\ X{+}1)$ |
$lift_{ML}\ i\ (A_{ML}\ v\ vs) = A_{ML}\ (lift_{ML}\ i\ v)\ (map\ (lift_{ML}\ i)\ vs)$ |
$lift_{ML}\ i\ (Lam_{ML}\ v) = Lam_{ML}\ (lift_{ML}\ (i{+}1)\ v)$ |
$lift_{ML}\ i\ (C_U\ nm\ vs) = C_U\ nm\ (map\ (lift_{ML}\ i)\ vs)$ |
$lift_{ML}\ i\ (V_U\ x\ vs) = V_U\ x\ (map\ (lift_{ML}\ i)\ vs)$ |
$lift_{ML}\ i\ (Clo\ v\ vs\ n) = Clo\ (lift_{ML}\ i\ v)\ (map\ (lift_{ML}\ i)\ vs)\ n$ |
$lift_{ML}\ i\ (apply\ u\ v) = apply\ (lift_{ML}\ i\ u)\ (lift_{ML}\ i\ v)$

definition
cons :: $tm \Rightarrow (nat \Rightarrow tm) \Rightarrow (nat \Rightarrow tm)$ (**infix** ‹##› 65) **where**
$t\#\#\sigma \equiv \lambda i.\ case\ i\ of\ 0 \Rightarrow t \mid Suc\ j \Rightarrow lift\ 0\ (\sigma\ j)$

definition
cons-ML :: $ml \Rightarrow (nat \Rightarrow ml) \Rightarrow (nat \Rightarrow ml)$ (**infix** ‹##› 65) **where**
$v\#\#\sigma \equiv \lambda i.\ case\ i\ of\ 0 \Rightarrow v{::}ml \mid Suc\ j \Rightarrow lift_{ML}\ 0\ (\sigma\ j)$

Only for pure terms!

primrec *subst* :: $(nat \Rightarrow tm) \Rightarrow tm \Rightarrow tm$
where
$subst\ \sigma\ (C\ nm) = C\ nm$
| $subst\ \sigma\ (V\ x) = \sigma\ x$
| $subst\ \sigma\ (\Lambda\ t) = \Lambda(subst\ (V\ 0\ \#\#\ \sigma)\ t)$
| $subst\ \sigma\ (s{\cdot}t) = (subst\ \sigma\ s) \cdot (subst\ \sigma\ t)$

fun *subst-ML* :: $(nat \Rightarrow ml) \Rightarrow ml \Rightarrow ml$ (‹$subst_{ML}$›) **where**
$subst_{ML}\ \sigma\ (C_{ML}\ nm) = C_{ML}\ nm$ |
$subst_{ML}\ \sigma\ (V_{ML}\ X) = \sigma\ X$ |
$subst_{ML}\ \sigma\ (A_{ML}\ v\ vs) = A_{ML}\ (subst_{ML}\ \sigma\ v)\ (map\ (subst_{ML}\ \sigma)\ vs)$ |
$subst_{ML}\ \sigma\ (Lam_{ML}\ v) = Lam_{ML}\ (subst_{ML}\ (V_{ML}\ 0\ \#\#\ \sigma)\ v)$ |

$subst_{ML}\ \sigma\ (C_U\ nm\ vs) = C_U\ nm\ (map\ (subst_{ML}\ \sigma)\ vs)\ |$
$subst_{ML}\ \sigma\ (V_U\ x\ vs) = V_U\ x\ (map\ (subst_{ML}\ \sigma)\ vs)\ |$
$subst_{ML}\ \sigma\ (Clo\ v\ vs\ n) = Clo\ (subst_{ML}\ \sigma\ v)\ (map\ (subst_{ML}\ \sigma)\ vs)\ n\ |$
$subst_{ML}\ \sigma\ (apply\ u\ v) = apply\ (subst_{ML}\ \sigma\ u)\ (subst_{ML}\ \sigma\ v)$

abbreviation
subst-decr :: *nat* ⇒ *tm* ⇒ *nat* ⇒ *tm* **where**
subst-decr k t ≡ λ*n*. *if n*<*k then V n else if n*=*k then t else V*(*n* − *1*)
abbreviation
subst-decr-ML :: *nat* ⇒ *ml* ⇒ *nat* ⇒ *ml* **where**
subst-decr-ML k v ≡ λ*n*. *if n*<*k then* V_{ML} *n else if n*=*k then v else* V_{ML}(*n* − *1*)
abbreviation
subst1 :: *tm* ⇒ *tm* ⇒ *nat* ⇒ *tm* (‹(-/[-′/-])› [*300*, *0*, *0*] *300*) **where**
s[*t*/*k*] ≡ *subst* (*subst-decr k t*) *s*
abbreviation
subst1-ML :: *ml* ⇒ *ml* ⇒ *nat* ⇒ *ml* (‹(-/[-′/-])› [*300*, *0*, *0*] *300*) **where**
u[*v*/*k*] ≡ $subst_{ML}$ (*subst-decr-ML k v*) *u*

lemma *apply-cons*[*simp*]:
(*t*##σ) *i* = (*if i*=*0 then t*::*tm else lift 0* (σ(*i* − *1*)))
by(*simp add*: *cons-def split*:*nat.split*)

lemma *apply-cons-ML*[*simp*]:
(*v*##σ) *i* = (*if i*=*0 then v*::*ml else* $lift_{ML}$ *0* (σ(*i* − *1*)))
by(*simp add*: *cons-ML-def split*:*nat.split*)

lemma *lift-foldl-At*[*simp*]:
lift k (*s* •• *ts*) = (*lift k s*) •• (*map* (*lift k*) *ts*)
by(*induct ts arbitrary*:*s*) *simp-all*

lemma *lift-lift-ml*: **fixes** *v* :: *ml* **shows**
i < *k*+*1* ⟹ *lift* (*Suc k*) (*lift i v*) = *lift i* (*lift k v*)
by(*induct i v rule*:*lift-ml.induct*)
simp-all
lemma *lift-lift-tm*: **fixes** *t* :: *tm* **shows**
i < *k*+*1* ⟹ *lift* (*Suc k*) (*lift i t*) = *lift i* (*lift k t*)
by(*induct t arbitrary*: *i rule*:*lift-tm.induct*)(*simp-all add*:*lift-lift-ml*)

lemma *lift-lift-ML*:
i < *k*+*1* ⟹ $lift_{ML}$ (*Suc k*) ($lift_{ML}$ *i v*) = $lift_{ML}$ *i* ($lift_{ML}$ *k v*)
by(*induct v arbitrary*: *i rule*:*lift-ML.induct*)
simp-all

lemma *lift-lift-ML-comm*:
lift j ($lift_{ML}$ *i v*) = $lift_{ML}$ *i* (*lift j v*)
by(*induct v arbitrary*: *i j rule*:*lift-ML.induct*)
simp-all

lemma *V-ML-cons-ML-subst-decr*[*simp*]:
$V_{ML}\ 0\ \#\#\ subst\text{-}decr\text{-}ML\ k\ v = subst\text{-}decr\text{-}ML\ (Suc\ k)\ (lift_{ML}\ 0\ v)$
by(*rule ext*)(*simp add*:*cons-ML-def split*:*nat.split*)

lemma *shift-subst-decr*[*simp*]:
$V\ 0\ \#\#\ subst\text{-}decr\ k\ t = subst\text{-}decr\ (Suc\ k)\ (lift\ 0\ t)$
by(*rule ext*)(*simp add*:*cons-def split*:*nat.split*)

lemma *lift-comp-subst-decr*[*simp*]:
$lift\ 0\ o\ subst\text{-}decr\text{-}ML\ k\ v = subst\text{-}decr\text{-}ML\ k\ (lift\ 0\ v)$
by(*rule ext*) *simp*

lemma *subst-ML-ext*: $\forall\, i.\ \sigma\ i = \sigma'\ i \Longrightarrow subst_{ML}\ \sigma\ v = subst_{ML}\ \sigma'\ v$
by(*metis ext*)

lemma *subst-ext*: $\forall\, i.\ \sigma\ i = \sigma'\ i \Longrightarrow subst\ \sigma\ v = subst\ \sigma'\ v$
by(*metis ext*)

lemma *lift-Pure-tms*[*simp*]: $pure\ t \Longrightarrow pure(lift\ k\ t)$
by(*induct arbitrary*:*k pred*:*pure*) *simp-all*

lemma *cons-ML-V-ML*[*simp*]: $(V_{ML}\ 0\ \#\#\ V_{ML}) = V_{ML}$
by(*rule ext*) *simp*

lemma *cons-V*[*simp*]: $(V\ 0\ \#\#\ V) = V$
by(*rule ext*) *simp*

lemma *lift-o-shift*: $lift\ k \circ (V_{ML}\ 0\ \#\#\ \sigma) = (V_{ML}\ 0\ \#\#\ (lift\ k \circ \sigma))$
by(*rule ext*)(*simp add*: *lift-lift-ML-comm*)

lemma *lift-subst-ML*:
$lift\ k\ (subst_{ML}\ \sigma\ v) = subst_{ML}\ (lift\ k \circ \sigma)\ (lift\ k\ v)$
apply(*induct* σ *v rule*:*subst-ML.induct*)
apply(*simp-all add*: *o-assoc lift-o-shift del*:*apply-cons-ML*)
apply(*simp add*:*o-def*)
done

corollary *lift-subst-ML1*:
$\forall\, v\ k.\ lift\text{-}ml\ 0\ (u[v/k]) = (lift\text{-}ml\ 0\ u)[lift\ 0\ v/k]$
apply(*induct u rule*:*ml-induct*)
apply(*simp-all add*:*lift-lift-ml lift-subst-ML*)
apply(*subst lift-lift-ML-comm*)**apply** *simp*
done

lemma *lift-ML-subst-ML*:
$lift_{ML}\ k\ (subst_{ML}\ \sigma\ v) =$
$subst_{ML}\ (\lambda i.\ if\ i<k\ then\ lift_{ML}\ k\ (\sigma\ i)\ else\ if\ i=k\ then\ V_{ML}\ k\ else\ lift_{ML}\ k\ (\sigma(i - 1)))\ (lift_{ML}\ k\ v)$
(**is** - = $subst_{ML}\ (?insrt\ k\ \sigma)\ (lift_{ML}\ k\ v)$)

apply (*induct k v arbitrary*: σ *k rule*: *lift-ML.induct*)
apply (*simp-all add*: *o-assoc lift-o-shift*)
apply(*subgoal-tac* V_{ML} *0 ## ?insrt k* σ = *?insrt* (*Suc k*) (V_{ML} *0 ##* σ))
apply *simp*
apply (*simp add*:*fun-eq-iff lift-lift-ML cons-ML-def split*:*nat.split*)
done

corollary *subst-cons-lift*:
$subst_{ML}$ (V_{ML} *0 ##* σ) *o* ($lift_{ML}$ *0*) = $lift_{ML}$ *0 o* ($subst_{ML}$ σ)
apply(*rule ext*)
apply(*simp add*: *lift-ML-subst-ML*)
apply(*subgoal-tac* (V_{ML} *0 ##* σ) = (λ*i. if i = 0 then* V_{ML} *0 else* $lift_{ML}$ *0* (σ (*i − 1*))))
apply *simp*
apply(*rule ext*, *simp*)
done

lemma *lift-ML-id*[*simp*]: $closed_{ML}$ *k v* $\Longrightarrow$ $lift_{ML}$ *k v* = *v*
by(*induct k v rule*: *lift-ML.induct*)(*simp-all add*:*list-eq-iff-nth-eq*)

lemma *subst-ML-id*:
$closed_{ML}$ *k v* $\Longrightarrow$ $\forall$ *i*<*k*. σ *i* = V_{ML} *i* $\Longrightarrow$ $subst_{ML}$ σ *v* = *v*
apply (*induct* σ *v arbitrary*: *k rule*: *subst-ML.induct*)
apply (*auto simp add*: *list-eq-iff-nth-eq*)
apply(*simp add*:*Ball-def*)
apply(*erule-tac x*=*vs*!*i* **in** *meta-allE*)
apply(*erule-tac x*=*k* **in** *meta-allE*)
apply(*erule-tac x*=*k* **in** *meta-allE*)
apply *simp*
apply(*erule-tac x*=*vs*!*i* **in** *meta-allE*)
apply(*erule-tac x*=*k* **in** *meta-allE*)
apply *simp*
apply(*erule-tac x*=*vs*!*i* **in** *meta-allE*)
apply(*erule-tac x*=*k* **in** *meta-allE*)
apply *simp*
apply(*erule-tac x*=*vs*!*i* **in** *meta-allE*)
apply(*erule-tac x*=*k* **in** *meta-allE*)
apply(*erule-tac x*=*k* **in** *meta-allE*)
apply *simp*
done

corollary *subst-ML-id2*[*simp*]: $closed_{ML}$ *0 v* $\Longrightarrow$ $subst_{ML}$ σ *v* = *v*
using *subst-ML-id*[**where** *k*=*0*] **by** *simp*

lemma *subst-ML-coincidence*:
$closed_{ML}$ *k v* $\Longrightarrow$ $\forall$ *i*<*k*. σ *i* = σ' *i* $\Longrightarrow$ $subst_{ML}$ σ *v* = $subst_{ML}$ σ' *v*
by (*induct* σ *v arbitrary*: *k* σ' *rule*: *subst-ML.induct*) *auto*

lemma *subst-ML-comp*:

$subst_{ML}\ \sigma\ (subst_{ML}\ \sigma'\ v) = subst_{ML}\ (subst_{ML}\ \sigma\ \circ \sigma')\ v$
apply (*induct* σ' *v arbitrary*: σ *rule*: *subst-ML.induct*)
apply (*simp-all add*: *list-eq-iff-nth-eq*)
apply(*rule subst-ML-ext*)
apply *simp*
apply (*metis o-apply subst-cons-lift*)
done

lemma *subst-ML-comp2*:
$\forall i.\ \sigma''\ i = subst_{ML}\ \sigma\ (\sigma'\ i) \Longrightarrow subst_{ML}\ \sigma\ (subst_{ML}\ \sigma'\ v) = subst_{ML}\ \sigma''\ v$
by(*simp add*:*subst-ML-comp subst-ML-ext*)

lemma *closed-tm-ML-foldl-At*:
$closed_{ML}\ k\ (t \bullet\bullet ts) \longleftrightarrow closed_{ML}\ k\ t \wedge (\forall t \in set\ ts.\ closed_{ML}\ k\ t)$
by(*induct ts arbitrary*: *t*) *simp-all*

lemma *closed-ML-lift*[*simp*]:
fixes *v* :: *ml* **shows** $closed_{ML}\ k\ v \Longrightarrow closed_{ML}\ k\ (lift\ m\ v)$
by(*induct k v arbitrary*: *m rule*: *lift-ML.induct*)
(*simp-all add*:*list-eq-iff-nth-eq*)

lemma *closed-ML-Suc*: $closed_{ML}\ n\ v \Longrightarrow closed_{ML}\ (Suc\ n)\ (lift_{ML}\ k\ v)$
by (*induct k v arbitrary*: *n rule*: *lift-ML.induct*) *simp-all*

lemma *closed-ML-subst-ML*:
$\forall i.\ closed_{ML}\ k\ (\sigma\ i) \Longrightarrow closed_{ML}\ k\ (subst_{ML}\ \sigma\ v)$
by(*induct* σ *v arbitrary*: *k rule*: *subst-ML.induct*) (*auto simp*: *closed-ML-Suc*)

lemma *closed-ML-subst-ML2*:
$closed_{ML}\ k\ v \Longrightarrow \forall i{<}k.\ closed_{ML}\ l\ (\sigma\ i) \Longrightarrow closed_{ML}\ l\ (subst_{ML}\ \sigma\ v)$
by(*induct* σ *v arbitrary*: *k l rule*: *subst-ML.induct*)(*auto simp*: *closed-ML-Suc*)

lemma *subst-foldl*[*simp*]:
$subst\ \sigma\ (s \bullet\bullet ts) = (subst\ \sigma\ s) \bullet\bullet (map\ (subst\ \sigma)\ ts)$
by (*induct ts arbitrary*: *s*) *auto*

lemma *subst-V*: *pure t* $\Longrightarrow$ *subst V t* = *t*
by(*induct pred*:*pure*) *simp-all*

lemma *lift-subst-aux*:
pure t $\Longrightarrow \forall i{<}k.\ \sigma'\ i = lift\ k\ (\sigma\ i) \Longrightarrow$
$\forall i{\geq}k.\ \sigma'(Suc\ i) = lift\ k\ (\sigma\ i) \Longrightarrow$
$\sigma'\ k = V\ k \Longrightarrow lift\ k\ (subst\ \sigma\ t) = subst\ \sigma'\ (lift\ k\ t)$
apply(*induct arbitrary*:σ σ' *k pred*:*pure*)
apply (*simp-all add*: *split*:*nat.split*)
apply(*erule meta-allE*)+
apply(*erule meta-impE*)
defer

apply(*erule meta-impE*)
defer
apply(*erule meta-mp*)
apply (*simp-all add*: *cons-def lift-lift-ml lift-lift-tm split*:*nat.split*)
done

corollary *lift-subst*:
  pure t $\Longrightarrow$ *lift 0* (*subst* σ *t*) = *subst* (*V 0* ## σ) (*lift 0 t*)
by (*simp add*: *lift-subst-aux lift-lift-ml*)

lemma *subst-comp*:
  pure t $\Longrightarrow$ $\forall$ *i*. *pure*(σ' *i*) $\Longrightarrow$
   σ'' = (λ*i*. *subst* σ (σ' *i*)) $\Longrightarrow$ *subst* σ (*subst* σ' *t*) = *subst* σ'' *t*
apply(*induct arbitrary*:σ σ' σ'' *pred*:*pure*)
apply *simp*
apply *simp*
defer
apply *simp*
apply (*simp* (*no-asm*))
apply(*erule meta-allE*)+
apply(*erule meta-impE*)
defer
apply(*erule meta-mp*)
prefer *2* **apply** *simp*
apply(*rule ext*)
apply(*simp add*:*lift-subst*)
done

2 Reduction

2.1 Patterns

inductive *pattern* :: *tm* $\Rightarrow$ *bool*
      and *patterns* :: *tm list* $\Rightarrow$ *bool* **where**
       patterns ts $\equiv$ $\forall$ *t*$\in$*set ts*. *pattern t* |
pat-V: *pattern*(*V X*) |
pat-C: *patterns ts* $\Longrightarrow$ *pattern*(*C nm* $\bullet\bullet$ *ts*)

lemma *pattern-Lam*[*simp*]: $\neg$ *pattern*(Λ *t*)
by(*auto elim*!: *pattern.cases*)

lemma *pattern-At'D12*: *pattern r* $\Longrightarrow$ *r* = (*s* $\cdot$ *t*) $\Longrightarrow$ *pattern s* $\wedge$ *pattern t*
proof(*induct arbitrary*: *s t pred*:*pattern*)
  case *pat-V* **thus** *?case* **by** *simp*
next
  case *pat-C* **thus** *?case*
    by (*simp add*: *atomic-tm-head-tm split*:*if-split-asm*)
       (*metis eta-head-args in-set-butlastD pattern.pat-C*)
qed

lemma *pattern-AtD12*: *pattern*($s \cdot t$) $\Longrightarrow$ *pattern* $s \wedge$ *pattern* t
by(*metis pattern-At'D12*)

lemma *pattern-At-vecD*: *pattern*($s \bullet\bullet ts$) $\Longrightarrow$ *patterns ts*
apply(*induct ts rule*:*rev-induct*)
apply *simp*
apply (*fastforce dest*!:*pattern-AtD12*)
done

lemma *pattern-At-decomp*: *pattern*($s \cdot t$) $\Longrightarrow \exists\, nm\ ss.\ s = C\ nm \bullet\bullet ss$
proof(*induct s arbitrary*: *t*)
case (*At s1 s2*) **show** *?case*
using *At* **by** (*metis foldl-Cons foldl-Nil foldl-append pattern-AtD12*)
qed (*auto elim*!: *pattern.cases split*:*if-split-asm*)

2.2 Reduction of λ-terms

The source program:

axiomatization R :: (*cname* $*$ *tm list* $*$ *tm*)*set* **where**
pure-R: (*nm*,*ts*,*t*) : $R \Longrightarrow (\forall\, t \in set\ ts.\ pure\ t) \wedge pure\ t$ **and**
fv-R: (*nm*,*ts*,*t*) : $R \Longrightarrow X : fv\ t \Longrightarrow \exists\, t' \in set\ ts.\ X : fv\ t'$ **and**
pattern-R: (*nm*,*ts*,t') : $R \Longrightarrow$ *patterns ts*

inductive-set
Red-tm :: (*tm* $*$ *tm*)*set*
and *red-tm* :: [*tm*, *tm*] => *bool* (**infixl** ‹$\rightarrow$› *50*)
where
$s \rightarrow t \equiv (s, t) \in$ *Red-tm*
— β-reduction
| $(\Lambda\ t) \cdot s \rightarrow t[s/0]$
— η-expansion
| $t \rightarrow \Lambda\ ((lift\ 0\ t) \cdot (V\ 0))$
— Rewriting
| (*nm*,*ts*,*t*) : $R \Longrightarrow (C\ nm) \bullet\bullet (map\ (subst\ \sigma)\ ts) \rightarrow subst\ \sigma\ t$
| $t \rightarrow t' \Longrightarrow \Lambda\ t \rightarrow \Lambda\ t'$
| $s \rightarrow s' \Longrightarrow s \cdot t \rightarrow s' \cdot t$
| $t \rightarrow t' \Longrightarrow s \cdot t \rightarrow s \cdot t'$

abbreviation
reds-tm :: [*tm*, *tm*] => *bool* (**infixl** ‹$\rightarrow*$› *50*) **where**
$s \rightarrow* t \equiv (s, t) \in$ *Red-tm*$\hat{}*$

inductive-set
Reds-tm-list :: (*tm list* $*$ *tm list*) *set*
and *reds-tm-list* :: [*tm list*, *tm list*] $\Rightarrow$ *bool* (**infixl** ‹$\rightarrow*$› *50*)
where
$ss \rightarrow* ts \equiv (ss, ts) \in$ *Reds-tm-list*
| [] $\rightarrow*$ []

$| \; ts \to* ts' \Longrightarrow t \to* t' \Longrightarrow t\#ts \to* t'\#ts'$

declare *Reds-tm-list.intros*[*simp*]

lemma *Reds-tm-list-refl*[*simp*]: **fixes** *ts* :: *tm list* **shows** *ts* $\to*$ *ts*
by(*induct ts*) *auto*

lemma *Red-tm-append*: *rs* $\to*$ *rs'* $\Longrightarrow$ *ts* $\to*$ *ts'* $\Longrightarrow$ *rs* @ *ts* $\to*$ *rs'* @ *ts'*
by(*induct set*: *Reds-tm-list*) *auto*

lemma *Red-tm-rev*: *ts* $\to*$ *ts'* $\Longrightarrow$ *rev ts* $\to*$ *rev ts'*
by(*induct set*: *Reds-tm-list*) (*auto simp*:*Red-tm-append*)

lemma *red-Lam*[*simp*]: *t* $\to*$ *t'* $\Longrightarrow$ Λ *t* $\to*$ Λ *t'*
apply(*induct rule*:*rtrancl-induct*)
apply(*simp-all*)
apply(*blast intro*: *rtrancl-into-rtrancl Red-tm.intros*)
done

lemma *red-At1*[*simp*]: *t* $\to*$ *t'* $\Longrightarrow$ *t* $\cdot$ *s* $\to*$ *t'* $\cdot$ *s*
apply(*induct rule*:*rtrancl-induct*)
apply(*simp-all*)
apply(*blast intro*: *rtrancl-into-rtrancl Red-tm.intros*)
done

lemma *red-At2*[*simp*]: *t* $\to*$ *t'* $\Longrightarrow$ *s* $\cdot$ *t* $\to*$ *s* $\cdot$ *t'*
apply(*induct rule*:*rtrancl-induct*)
apply(*simp-all*)
apply(*blast intro*:*rtrancl-into-rtrancl Red-tm.intros*)
done

lemma *Reds-tm-list-foldl-At*:
ts $\to*$ *ts'* $\Longrightarrow$ *s* $\to*$ *s'* $\Longrightarrow$ *s* $\bullet\bullet$ *ts* $\to*$ *s'* $\bullet\bullet$ *ts'*
apply(*induct arbitrary*:*s s' rule*:*Reds-tm-list.induct*)
apply *simp*
apply *simp*
apply(*blast dest*: *red-At1 red-At2 intro*:*rtrancl-trans*)
done

2.3 Reduction of ML-terms

The compiled rule set:

consts *compR* :: (*cname* $*$ *ml list* $*$ *ml*)*set*

The actual definition is given in §4 below.

Now we characterize ML values that cannot possibly be rewritten by a rule in *compR*.

lemma *termination-no-match-ML*:
  $i < length\ ps \Longrightarrow rev\ ps\ !\ i = C_U\ nm\ vs$
  $\Longrightarrow sum\text{-}list\ (map\ size\ vs) < sum\text{-}list\ (map\ size\ ps)$
apply(*subgoal-tac* C_U *nm vs* : *set ps*)
apply(*drule sum-list-map-remove1*[*of - - size*])
apply (*simp add*:*size-list-conv-sum-list*)
apply (*metis in-set-conv-nth length-rev set-rev*)
done

declare *conj-cong*[*fundef-cong*]

function *no-match-ML* (‹*no′-match*$_{ML}$›) **where**
no-match$_{ML}$ *ps os* =
  (∃ *i* < *min* (*size os*) (*size ps*).
   ∃ *nm nm′ vs vs′*. (*rev ps*)!*i* = C_U *nm vs* ∧ (*rev os*)!*i* = C_U *nm′ vs′* ∧
     (*nm*=*nm′* ⟶ *no-match*$_{ML}$ *vs vs′*))
by *pat-completeness auto*
termination
apply(*relation measure*(%(*vs*::*ml list*,-). $\sum$ *v*←*vs*. *size v*))
apply (*auto simp*:*termination-no-match-ML*)
done

abbreviation
no-match-compR nm os ≡
  ∀ (*nm′*,*ps*,*v*)∈ *compR*. *nm*=*nm′* ⟶ *no-match*$_{ML}$ *ps os*

declare *no-match-ML.simps*[*simp del*]

inductive-set
  Red-ml :: (*ml* ∗ *ml*)*set*
  and *Red-ml-list* :: (*ml list* ∗ *ml list*)*set*
  and *red-ml* :: [*ml*, *ml*] => *bool* (**infixl** ‹⇒› *50*)
  and *red-ml-list* :: [*ml list*, *ml list*] => *bool* (**infixl** ‹⇒› *50*)
  and *reds-ml* :: [*ml*, *ml*] => *bool* (**infixl** ‹⇒∗› *50*)
where
  s ⇒ *t* ≡ (*s*, *t*) ∈ *Red-ml*
| *ss* ⇒ *ts* ≡ (*ss*, *ts*) ∈ *Red-ml-list*
| *s* ⇒∗ *t* ≡ (*s*, *t*) ∈ *Red-ml*^∗
— ML β-reduction
| A_{ML} (Lam_{ML} *u*) [*v*] ⇒ *u*[*v*/*0*]
— Execution of a compiled rewrite rule
| (*nm*,*vs*,*v*) : *compR* ⟹ ∀ *i*. *closed*$_{ML}$ *0* (*σ i*) ⟹
   A_{ML} (C_{ML} *nm*) (*map* (*subst*$_{ML}$ *σ*) *vs*) ⇒ *subst*$_{ML}$ *σ v*
— default rule:
| ∀ *i*. *closed*$_{ML}$ *0* (*σ i*)
   ⟹ *vs* = *map* V_{ML} [*0*..<*arity nm*] ⟹ *vs′* = *map* (*subst*$_{ML}$ *σ*) *vs*
   ⟹ *no-match-compR nm vs′*
   ⟹ A_{ML} (C_{ML} *nm*) *vs′* ⇒ *subst*$_{ML}$ *σ* (C_U *nm vs*)

— Equations for function `apply`
| *apply-Clo1*: $apply\ (Clo\ f\ vs\ (Suc\ 0))\ v \Rightarrow A_{ML}\ f\ (v\ \#\ vs)$
| *apply-Clo2*: $n > 0 \Longrightarrow$
$apply\ (Clo\ f\ vs\ (Suc\ n))\ v \Rightarrow Clo\ f\ (v\ \#\ vs)\ n$
| *apply-C*: $apply\ (C_U\ nm\ vs)\ v \Rightarrow C_U\ nm\ (v\ \#\ vs)$
| *apply-V*: $apply\ (V_U\ x\ vs)\ v \Rightarrow V_U\ x\ (v\ \#\ vs)$
— Context rules
| *ctxt-C*: $vs \Rightarrow vs' \Longrightarrow C_U\ nm\ vs \Rightarrow C_U\ nm\ vs'$
| *ctxt-V*: $vs \Rightarrow vs' \Longrightarrow V_U\ x\ vs \Rightarrow V_U\ x\ vs'$
| *ctxt-Clo1*: $f \Rightarrow f' \Longrightarrow Clo\ f\ vs\ n \Rightarrow Clo\ f'\ vs\ n$
| *ctxt-Clo3*: $vs \Rightarrow vs' \Longrightarrow Clo\ f\ vs\ n \Rightarrow Clo\ f\ vs'\ n$
| *ctxt-apply1*: $s \Rightarrow s' \Longrightarrow apply\ s\ t \Rightarrow apply\ s'\ t$
| *ctxt-apply2*: $t \Rightarrow t' \Longrightarrow apply\ s\ t \Rightarrow apply\ s\ t'$
| *ctxt-A-ML1*: $f \Rightarrow f' \Longrightarrow A_{ML}\ f\ vs \Rightarrow A_{ML}\ f'\ vs$
| *ctxt-A-ML2*: $vs \Rightarrow vs' \Longrightarrow A_{ML}\ f\ vs \Rightarrow A_{ML}\ f\ vs'$
| *ctxt-list1*: $v \Rightarrow v' \Longrightarrow v\#vs \Rightarrow v'\#vs$
| *ctxt-list2*: $vs \Rightarrow vs' \Longrightarrow v\#vs \Rightarrow v\#vs'$

inductive-set
Red-term :: $(tm * tm)set$
and *red-term* :: $[tm, tm] => bool$ (**infixl** ‹⇒› *50*)
and *reds-term* :: $[tm, tm] => bool$ (**infixl** ‹⇒*› *50*)
where
$s \Rightarrow t \equiv (s, t) \in Red\text{-}term$
| $s \Rightarrow* t \equiv (s, t) \in Red\text{-}term\hat{}*$
— function `term`
| *term-C*: $term\ (C_U\ nm\ vs) \Rightarrow (C\ nm) \bullet\bullet (map\ term\ (rev\ vs))$
| *term-V*: $term\ (V_U\ x\ vs) \Rightarrow (V\ x) \bullet\bullet (map\ term\ (rev\ vs))$
| *term-Clo*: $term(Clo\ vf\ vs\ n) \Rightarrow \Lambda\ (term\ (apply\ (lift\ 0\ (Clo\ vf\ vs\ n))\ (V_U\ 0\ [])))$
— context rules
| *ctxt-Lam*: $t \Rightarrow t' \Longrightarrow \Lambda\ t \Rightarrow \Lambda\ t'$
| *ctxt-At1*: $s \Rightarrow s' \Longrightarrow s \cdot t \Rightarrow s' \cdot t$
| *ctxt-At2*: $t \Rightarrow t' \Longrightarrow s \cdot t \Rightarrow s \cdot t'$
| *ctxt-term*: $v \Rightarrow v' \Longrightarrow term\ v \Rightarrow term\ v'$

3 Kernel

First a special size function and some lemmas for the termination proof of the kernel function.

fun $size'$ :: $ml \Rightarrow nat$ **where**
$size'\ (C_{ML}\ nm) = 1$ |
$size'\ (V_{ML}\ X) = 1$ |
$size'\ (A_{ML}\ v\ vs) = (size'\ v + (\sum v{\leftarrow}vs.\ size'\ v))+1$ |
$size'\ (Lam_{ML}\ v) = size'\ v + 1$ |
$size'\ (C_U\ nm\ vs) = (\sum v{\leftarrow}vs.\ size'\ v)+1$ |
$size'\ (V_U\ nm\ vs) = (\sum v{\leftarrow}vs.\ size'\ v)+1$ |
$size'\ (Clo\ f\ vs\ n) = (size'\ f + (\sum v{\leftarrow}vs.\ size'\ v))+1$ |
$size'\ (apply\ v\ w) = (size'\ v + size'\ w)+1$

lemma *sum-list-size′*[*simp*]:
v ∈ *set vs* ⟹ *size′ v* < *Suc*(*sum-list* (*map size′ vs*))
by(*induct vs*)(*auto*)

corollary *cor-sum-list-size′*[*simp*]:
v ∈ *set vs* ⟹ *size′ v* < *Suc*(*m* + *sum-list* (*map size′ vs*))
using *sum-list-size′*[*of v vs*] **by** *arith*

lemma *size′-lift-ML*: *size′* (*lift*$_{ML}$ *k v*) = *size′ v*
apply(*induct v arbitrary*:*k rule*:*size′.induct*)
apply *simp-all*
apply(*rule arg-cong*[**where** *f* = *sum-list*])
apply(*rule map-ext*)
apply *simp*
apply(*rule arg-cong*[**where** *f* = *sum-list*])
apply(*rule map-ext*)
apply *simp*
apply(*rule arg-cong*[**where** *f* = *sum-list*])
apply(*rule map-ext*)
apply *simp*
apply(*rule arg-cong*[**where** *f* = *sum-list*])
apply(*rule map-ext*)
apply *simp*
done

lemma *size′-subst-ML*[*simp*]:
∀ *i j*. *size′*(*σ i*) = *1* ⟹ *size′* (*subst*$_{ML}$ *σ v*) = *size′ v*
apply(*induct v arbitrary*:*σ rule*:*size′.induct*)
apply *simp-all*
apply(*rule arg-cong*[**where** *f* = *sum-list*])
apply(*rule map-ext*)
apply *simp*
apply(*erule meta-allE*)
apply(*erule meta-mp*)
apply(*simp add*: *size′-lift-ML split*:*nat.split*)
apply(*rule arg-cong*[**where** *f* = *sum-list*])
apply(*rule map-ext*)
apply *simp*
apply(*rule arg-cong*[**where** *f* = *sum-list*])
apply(*rule map-ext*)
apply *simp*
apply(*rule arg-cong*[**where** *f* = *sum-list*])
apply(*rule map-ext*)
apply *simp*
done

lemma *size′-lift*[*simp*]: *size′* (*lift i v*) = *size′ v*
apply(*induct v arbitrary*:*i rule*:*size′.induct*)

apply *simp-all*
  apply(*rule arg-cong*[**where** *f* = *sum-list*])
  apply(*rule map-ext*)
  apply *simp*
 apply(*rule arg-cong*[**where** *f* = *sum-list*])
 apply(*rule map-ext*)
 apply *simp*
apply(*rule arg-cong*[**where** *f* = *sum-list*])
apply(*rule map-ext*)
apply *simp*
done

function *kernel* :: *ml* ⇒ *tm* (‹-!› *300*) **where**
(C_{ML} *nm*)! = *C nm* |
(A_{ML} *v vs*)! = *v*! •• (*map kernel* (*rev vs*)) |
(Lam_{ML} *v*)! = Λ (((*lift 0 v*)[V_U *0* []/*0*])!) |
(C_U *nm vs*)! = (*C nm*) •• (*map kernel* (*rev vs*)) |
(V_U *x vs*)! = (*V x*) •• (*map kernel* (*rev vs*)) |
(*Clo f vs n*)! = *f*! •• (*map kernel* (*rev vs*)) |
(*apply v w*)! = *v*! • (*w*!) |
(V_{ML} *X*)! = *undefined*
by *pat-completeness auto*
termination by(*relation measure size*′) *auto*

primrec *kernelt* :: *tm* ⇒ *tm* (‹-!› *300*)
where
  (*C nm*)! = *C nm*
| (*V x*)! = *V x*
| (*s* • *t*)! = (*s*!) • (*t*!)
| (Λ *t*)! = Λ(*t*!)
| (*term v*)! = *v*!

abbreviation
  kernels :: *ml list* ⇒ *tm list* (‹-!› *300*) **where**
  vs! ≡ *map kernel vs*

lemma *kernel-pure*: **assumes** *pure t* **shows** *t*! = *t*
using *assms* **by** (*induct*) *simp-all*

lemma *kernel-foldl-At*[*simp*]: (*s* •• *ts*)! = (*s*!) •• (*map kernelt ts*)
by (*induct ts arbitrary*: *s*) *simp-all*

lemma *kernelt-o-term*[*simp*]: (*kernelt* ∘ *term*) = *kernel*
by(*rule ext*) *simp*

lemma *pure-foldl*:

pure t $\Longrightarrow$ $\forall$ *t*$\in$*set ts. pure t* $\Longrightarrow$
(!!*s t. pure s* $\Longrightarrow$ *pure t* $\Longrightarrow$ *pure*(*f s t*)) $\Longrightarrow$
pure(*foldl f t ts*)
by(*induct ts arbitrary*: *t*) *simp-all*

lemma *pure-kernel*: **fixes** *v* :: *ml* **shows** $closed_{ML}$ *0 v* $\Longrightarrow$ *pure*(*v*!)
proof(*induct v rule*:*kernel.induct*)
case (*3 v*)
hence $closed_{ML}$ (*Suc 0*) (*lift 0 v*) **by** *simp*
then have $subst_{ML}$ (λ*n*. V_U *0* []) (*lift 0 v*) = *lift 0 v*[V_U *0* []/*0*]
by(*rule subst-ML-coincidence*) *simp*
moreover have $closed_{ML}$ *0* ($subst_{ML}$ (λ*n*. V_U *0* []) (*lift 0 v*))
by(*simp add*: *closed-ML-subst-ML*)
ultimately have $closed_{ML}$ *0* (*lift 0 v*[V_U *0* []/*0*]) **by** *simp*
thus *?case* **using** *3*(*1*) **by** (*simp add*:*pure-foldl*)
qed (*simp-all add*:*pure-foldl*)

corollary *subst-V-kernel*: **fixes** *v* :: *ml* **shows**
$closed_{ML}$ *0 v* $\Longrightarrow$ *subst V* (*v*!) = *v*!
by (*metis pure-kernel subst-V*)

lemma *kernel-lift-tm*: **fixes** *v* :: *ml* **shows**
$closed_{ML}$ *0 v* $\Longrightarrow$ (*lift i v*)! = *lift i* (*v*!)
apply(*induct v arbitrary*: *i rule*: *kernel.induct*)
apply (*simp-all add*:*list-eq-iff-nth-eq*)
apply(*simp add*: *rev-nth*)
defer
apply(*simp add*: *rev-nth*)
apply(*simp add*: *rev-nth*)
apply(*simp add*: *rev-nth*)
apply(*erule-tac x*=*Suc i* **in** *meta-allE*)
apply(*erule meta-impE*)
defer
apply (*simp add*:*lift-subst-ML*)
apply(*subgoal-tac lift* (*Suc i*) $\circ$ (λ*n*. *if n* = *0 then* V_U *0* [] *else* V_{ML} (*n* $-$ *1*)) =
(λ*n*. *if n* = *0 then* V_U *0* [] *else* V_{ML} (*n* $-$ *1*)))
apply (*simp add*:*lift-lift-ml*)
apply(*rule ext*)
apply(*simp*)
apply(*subst closed-ML-subst-ML2*[*of 1*])
apply(*simp*)
apply(*simp*)
apply(*simp*)
done

3.1 An auxiliary substitution

This function is only introduced to prove the involved susbtitution lemma *kernel-subst1* below.

fun *subst-ml* :: $(nat \Rightarrow nat) \Rightarrow ml \Rightarrow ml$ **where**
$subst\text{-}ml\ \sigma\ (C_{ML}\ nm) = C_{ML}\ nm$ |
$subst\text{-}ml\ \sigma\ (V_{ML}\ X) = V_{ML}\ X$ |
$subst\text{-}ml\ \sigma\ (A_{ML}\ v\ vs) = A_{ML}\ (subst\text{-}ml\ \sigma\ v)\ (map\ (subst\text{-}ml\ \sigma)\ vs)$ |
$subst\text{-}ml\ \sigma\ (Lam_{ML}\ v) = Lam_{ML}\ (subst\text{-}ml\ \sigma\ v)$ |
$subst\text{-}ml\ \sigma\ (C_U\ nm\ vs) = C_U\ nm\ (map\ (subst\text{-}ml\ \sigma)\ vs)$ |
$subst\text{-}ml\ \sigma\ (V_U\ x\ vs) = V_U\ (\sigma\ x)\ (map\ (subst\text{-}ml\ \sigma)\ vs)$ |
$subst\text{-}ml\ \sigma\ (Clo\ v\ vs\ n) = Clo\ (subst\text{-}ml\ \sigma\ v)\ (map\ (subst\text{-}ml\ \sigma)\ vs)\ n$ |
$subst\text{-}ml\ \sigma\ (apply\ u\ v) = apply\ (subst\text{-}ml\ \sigma\ u)\ (subst\text{-}ml\ \sigma\ v)$

lemma *lift-ML-subst-ml*:
$lift_{ML}\ k\ (subst\text{-}ml\ \sigma\ v) = subst\text{-}ml\ \sigma\ (lift_{ML}\ k\ v)$
apply (*induct* σ *v arbitrary*: *k rule*:*subst-ml.induct*)
apply (*simp-all add*:*list-eq-iff-nth-eq*)
done

lemma *subst-ml-subst-ML*:
$subst\text{-}ml\ \sigma\ (subst_{ML}\ \sigma'\ v) = subst_{ML}\ (subst\text{-}ml\ \sigma\ o\ \sigma')\ (subst\text{-}ml\ \sigma\ v)$
apply (*induct* σ' *v arbitrary*: σ *rule*: *subst-ML.induct*)
apply(*simp-all add*:*list-eq-iff-nth-eq*)
apply(*subgoal-tac* $(subst\text{-}ml\ \sigma' \circ V_{ML}\ 0\ \#\#\ \sigma) = V_{ML}\ 0\ \#\#\ (subst\text{-}ml\ \sigma' \circ \sigma)$)
apply *simp*
apply(*rule ext*)
apply(*simp add*: *lift-ML-subst-ml*)
done

Maybe this should be the def of lift:

lemma *lift-is-subst-ml*: *lift k v* = *subst-ml* (λn. *if n<k then n else n+1*) *v*
by(*induct k v rule*:*lift-ml.induct*)(*simp-all add*:*list-eq-iff-nth-eq*)

lemma *subst-ml-comp*: *subst-ml* σ (*subst-ml* σ' *v*) = *subst-ml* (σ *o* σ') *v*
by(*induct* σ' *v rule*:*subst-ml.induct*)(*simp-all add*:*list-eq-iff-nth-eq*)

lemma *subst-kernel*:
$closed_{ML}\ 0\ v \Longrightarrow\ subst\ (\lambda n.\ V(\sigma\ n))\ (v!) = (subst\text{-}ml\ \sigma\ v)!$
apply (*induct v arbitrary*: σ *rule*:*kernel.induct*)
apply (*simp-all add*:*list-eq-iff-nth-eq*)
apply(*simp add*: *rev-nth*)
defer
apply(*simp add*: *rev-nth*)
apply(*simp add*: *rev-nth*)
apply(*simp add*: *rev-nth*)
apply(*erule-tac x*=$\lambda n.$ *case n of 0* $\Rightarrow$ *0* | *Suc k* $\Rightarrow$ *Suc*(σ *k*) **in** *meta-allE*)
apply(*erule-tac meta-impE*)
apply(*rule closed-ML-subst-ML2*[**where** *k*=*Suc 0*])
apply (*metis closed-ML-lift*)
apply *simp*
apply(*subgoal-tac* ($\lambda n.$ *V*(*case n of 0* $\Rightarrow$ *0* | *Suc k* $\Rightarrow$ *Suc* (σ *k*))) = (*V 0* ## ($\lambda n.$ *V*(σ *n*))))

apply (*simp add*:*subst-ml-subst-ML*)
defer
apply(*simp add*:*fun-eq-iff split*:*nat.split*)
apply(*simp add*:*lift-is-subst-ml subst-ml-comp*)
apply(*rule arg-cong*[**where** *f* = *kernel*])
apply(*subgoal-tac* (*case-nat 0* (λ*k*. *Suc* (σ *k*)) ∘ *Suc*) = *Suc o* σ)
prefer *2* **apply**(*simp add*:*fun-eq-iff split*:*nat.split*)
apply(*subgoal-tac* (*subst-ml* (*case-nat 0* (λ*k*. *Suc* (σ *k*))) ∘
(λ*n*. *if n* = *0 then* V_U *0* [] *else* V_{ML} (*n* − *1*)))
= (λ*n*. *if n* = *0 then* V_U *0* [] *else* V_{ML} (*n* − *1*)))
apply *simp*
apply(*simp add*: *fun-eq-iff*)
done

lemma *if-cong0*: *If x y z* = *If x y z*
by *simp*

lemma *kernel-subst1*:
$closed_{ML}$ *0 v* ⟹ $closed_{ML}$ (*Suc 0*) *u* ⟹
kernel(*u*[*v*/*0*]) = (*kernel*((*lift 0 u*)[V_U *0* []/*0*]))[*v*!/*0*]
proof(*induct u arbitrary*:*v rule*:*kernel.induct*)
case (*3 w*)
show *?case* (**is** *?L* = *?R*)
proof −
have *?L* = Λ(*lift 0* (*w*[$lift_{ML}$ *0 v*/*Suc 0*])[V_U *0* []/*0*] !)
by (*simp cong*:*if-cong0*)
also have ... = Λ((*lift 0 w*)[$lift_{ML}$ *0* (*lift 0 v*)/*Suc 0*][V_U *0* []/*0*]!)
by(*simp only*: *lift-subst-ML1 lift-lift-ML-comm*)
also have ... = Λ($subst_{ML}$ (λ*n*. *if n*=*0 then* V_U *0* [] *else*
if n=*Suc 0 then lift 0 v else* V_{ML} (*n* − *2*)) (*lift 0 w*) !)
apply *simp*
apply(*rule arg-cong*[**where** *f* = *kernel*])
apply(*rule subst-ML-comp2*)
using *3*
apply *auto*
done
also have ... = Λ((*lift 0 w*)[V_U *0* []/*0*][*lift 0 v*/*0*]!)
apply *simp*
apply(*rule arg-cong*[**where** *f* = *kernel*])
apply(*rule subst-ML-comp2*[*symmetric*])
using *3*
apply *auto*
done
also have ... = Λ((*lift-ml 0* ((*lift-ml 0 w*)[V_U *0* []/*0*]))[V_U *0* []/*0*]![(*lift 0 v*)!/*0*])
apply(*rule arg-cong*[**where** *f* = Λ])
apply(*rule 3*(*1*))
apply (*metis closed-ML-lift 3*(*2*))
apply(*subgoal-tac* $closed_{ML}$ (*Suc*(*Suc 0*)) *w*)

defer
using *3*
apply *force*
apply(*subgoal-tac* $closed_{ML}$ (*Suc* (*Suc 0*)) (*lift 0 w*))
defer
apply(*erule closed-ML-lift*)
apply(*erule closed-ML-subst-ML2*)
apply *simp*
done
also have ... = Λ((*lift-ml 0* (*lift-ml 0 w*)[V_U *1* []/*0*])[V_U *0* []/*0*]![(*lift 0 v*)!/*0*])
(**is** - = *?M*)
apply(*subgoal-tac lift-ml 0* (*lift-ml 0 w*[V_U *0* []/*0*])[V_U *0* []/*0*] =
lift-ml 0 (*lift-ml 0 w*)[V_U *1* []/*0*][V_U *0* []/*0*])
apply *simp*
apply(*subst lift-subst-ML*)
apply(*simp add*:*comp-def if-distrib*[**where** *f*=*lift-ml 0*] *cong*:*if-cong*)
done
finally have *?L* = *?M* **.**
have *?R* = Λ (*subst* (*V 0 ## subst-decr 0* (*v*!))
(*lift 0* (*lift-ml 0 w*[V_U *0* []/*Suc 0*])[V_U *0* []/*0*]!))
apply(*subgoal-tac* (V_{ML} *0 ##* (λ*n. if n* = *0 then* V_U *0* [] *else* V_{ML} (*n* − *Suc 0*))) = *subst-decr-ML* (*Suc 0*) (V_U *0* []))
apply(*simp cong*:*if-cong*)
apply(*simp add*:*fun-eq-iff cons-ML-def split*:*nat.splits*)
done
also have ... = Λ (*subst* (*V 0 ## subst-decr 0* (*v*!))
((*lift 0* (*lift-ml 0 w*))[V_U *1* []/*Suc 0*][V_U *0* []/*0*]!))
apply(*subgoal-tac lift 0* (*lift 0 w*[V_U *0* []/*Suc 0*]) = *lift 0* (*lift 0 w*)[V_U *1* []/*Suc 0*])
apply *simp*
apply(*subst lift-subst-ML*)
apply(*simp add*:*comp-def if-distrib*[**where** *f*=*lift-ml 0*] *cong*:*if-cong*)
done
also have (*lift-ml 0* (*lift-ml 0 w*))[V_U *1* []/*Suc 0*][V_U *0* []/*0*] =
(*lift 0* (*lift-ml 0 w*))[V_U *0* []/*0*][V_U *1* []/ *0*] (**is** *?l* = *?r*)
proof −
have *?l* = $subst_{ML}$ (λ*n. if n*= *0 then* V_U *0* [] *else if n* = *1 then* V_U *1* [] *else*
V_{ML} (*n* − *2*))
(*lift-ml 0* (*lift-ml 0 w*))
by(*auto intro*!:*subst-ML-comp2*)
also have ... = *?r* **by**(*auto intro*!:*subst-ML-comp2*[*symmetric*])
finally show *?thesis* **.**
qed
also have Λ (*subst* (*V 0 ## subst-decr 0* (*v*!)) (*?r* !)) = *?M*
proof−
have *subst* (*subst-decr* (*Suc 0*) (*lift-tm 0* (*kernel v*))) (*lift-ml 0* (*lift-ml 0 w*)[V_U *0* []/*0*][V_U *1* []/*0*]!) =
subst (*subst-decr 0* (*kernel*(*lift-ml 0 v*))) (*lift-ml 0* (*lift-ml 0 w*)[V_U *1* []/*0*][V_U *0* []/*0*]!) (**is** *?a* = *?b*)

```
  proof−
    define pi where pi n = (if n = 0 then 1 else if n = 1 then 0 else n) for n ::
nat
    have (λi. V (pi i)[lift 0 (v!)/0]) = subst-decr (Suc 0) (lift 0 (v!))
      by(rule ext)(simp add:pi-def)
    hence ?a =
 subst (subst-decr 0 (lift-tm 0 (kernel v))) (subst (λ n. V(pi n)) (lift-ml 0 (lift-ml
0 w)[V_U 0 []/0][V_U 1 []/0]!))
      apply(subst subst-comp[OF - - refl])
      prefer 3 apply simp
      using 3(3)
      apply simp
      apply(rule pure-kernel)
      apply(rule closed-ML-subst-ML2[where k=Suc 0])
      apply(rule closed-ML-subst-ML2[where k=Suc(Suc 0)])
      apply simp
      apply simp
      apply simp
      apply simp
      done
    also have ... =
(subst-ml pi (lift-ml 0 (lift-ml 0 w)[V_U 0 []/0][V_U 1 []/0]))![lift-tm 0 (v!)/0]
      apply(subst subst-kernel)
      using 3 apply auto
      apply(rule closed-ML-subst-ML2[where k=Suc 0])
      apply(rule closed-ML-subst-ML2[where k=Suc(Suc 0)])
      apply simp
      apply simp
      apply simp
      done
  also have ... = (subst-ml pi (lift-ml 0 (lift-ml 0 w)[V_U 0 []/0][V_U 1 []/0]))![lift
0 v!/0]
    proof −
      have lift 0 (v!) = lift 0 v! by (metis 3(2) kernel-lift-tm)
      thus ?thesis by (simp cong:if-cong)
    qed
    also have ... = ?b
    proof−
      have 1: subst-ml pi (lift 0 (lift 0 w)) = lift 0 (lift 0 w)
        apply(simp add:lift-is-subst-ml subst-ml-comp)
        apply(subgoal-tac pi ∘ (Suc ∘ Suc) = (Suc ∘ Suc))
        apply(simp)
        apply(simp add:pi-def fun-eq-iff)
        done
      have subst-ml pi (lift-ml 0 (lift-ml 0 w)[V_U 0 []/0][V_U 1 []/0]) =
          lift-ml 0 (lift-ml 0 w)[V_U 1 []/0][V_U 0 []/0]
        apply(subst subst-ml-subst-ML)
        apply(subst subst-ml-subst-ML)
        apply(subst 1)
```

apply(*subst subst-ML-comp*)
apply(*rule subst-ML-comp2*[*symmetric*])
apply(*auto simp*:*pi-def*)
done
thus *?thesis* **by** *simp*
qed
finally show *?thesis* **.**
qed
thus *?thesis* **by**(*simp cong*:*if-cong0 add*:*shift-subst-decr*)
qed
finally have *?R* = *?M* **.**
then show *?L* = *?R* **using** ‹*?L* = *?M*› **by** *metis*
qed
qed (*simp-all add*:*list-eq-iff-nth-eq*, (*simp-all add*:*rev-nth*) *?*)

4 Compiler

axiomatization *arity* :: *cname* ⇒ *nat*

primrec *compile* :: *tm* ⇒ (*nat* ⇒ *ml*) ⇒ *ml*
where
compile (*V x*) *σ* = *σ x*
| *compile* (*C nm*) *σ* =
(*if arity nm* > *0 then Clo* (C_{ML} *nm*) [] (*arity nm*) *else* A_{ML} (C_{ML} *nm*) [])
| *compile* (*s* ⋅ *t*) *σ* = *apply* (*compile s σ*) (*compile t σ*)
| *compile* (Λ *t*) *σ* = *Clo* (Lam_{ML} (*compile t* (V_{ML} *0* ## *σ*))) [] *1*

Compiler for open terms and for terms with fixed free variables:

definition *comp-open t* = *compile t* V_{ML}
abbreviation *comp-fixed t* ≡ *compile t* (λ*i*. V_U *i* [])

Compiled rules:

lemma *size-args-less-size-tm*[*simp*]: *s* ∈ *set* (*args-tm t*) ⟹ *size s* < *size t*
by(*induct t*) *auto*

fun *comp-pat* **where**
comp-pat t =
(*case head-tm t of*
C nm ⇒ C_U *nm* (*map comp-pat* (*rev* (*args-tm t*)))
| *V X* ⇒ V_{ML} *X*)

declare *comp-pat.simps*[*simp del*] *size-args-less-size-tm*[*simp del*]

lemma *comp-pat-V*[*simp*]: *comp-pat*(*V X*) = V_{ML} *X*
by(*simp add*:*comp-pat.simps*)

lemma *comp-pat-C*[*simp*]:
comp-pat(*C nm* •• *ts*) = C_U *nm* (*map comp-pat* (*rev ts*))
by(*simp add*:*comp-pat.simps*)

lemma *comp-pat-C-Nil*[*simp*]: *comp-pat*(*C nm*) = C_U *nm* []
by(*simp add*:*comp-pat.simps*)

overloading *compR* ≡ *compR*
begin
definition *compR* ≡ (λ(*nm*,*ts*,*t*). (*nm*, *map comp-pat* (*rev ts*), *comp-open t*)) ‘
R
end

lemma *fv-ML-comp-open*: *pure t* ⟹ fv_{ML}(*comp-open t*) = *fv t*
by(*induct t pred*:*pure*) (*simp-all add*:*comp-open-def*)

lemma *fv-ML-comp-pat*: *pattern t* ⟹ fv_{ML}(*comp-pat t*) = *fv t*
by(*induct t pred*:*pattern*)(*simp-all add*:*comp-open-def*)

lemma *fv-compR-aux*:
(*nm*,*ts*,*t*′) : *R* ⟹ *x* ∈ fv_{ML} (*comp-open t*′)
⟹ ∃ *t*∈*set ts*. *x* ∈ fv_{ML}(*comp-pat t*)
apply(*frule pure-R*)
apply(*simp add*:*fv-ML-comp-open*)
apply(*frule* (*1*) *fv-R*)
apply *clarsimp*
apply(*rule bexI*) **prefer** *2* **apply** *assumption*
apply(*drule pattern-R*)
apply(*simp add*:*fv-ML-comp-pat*)
done

lemma *fv-compR*:
(*nm*,*vs*,*v*) : *compR* ⟹ *x* ∈ fv_{ML} *v* ⟹ ∃ *u*∈*set vs*. *x* ∈ fv_{ML} *u*
by(*fastforce simp add*:*compR-def image-def dest*: *fv-compR-aux*)

lemma *lift-compile*:
pure t ⟹ ∀ σ *k*. *lift k* (*compile t* σ) = *compile t* (*lift k* ∘ σ)
apply(*induct pred*:*pure*)
apply *simp-all*
apply *clarsimp*
apply(*rule-tac f* = *compile t* **in** *arg-cong*)
apply(*rule ext*)
apply (*clarsimp simp*: *lift-lift-ML-comm*)
done

lemma *subst-ML-compile*:
pure t ⟹ $subst_{ML}$ σ′ (*compile t* σ) = *compile t* ($subst_{ML}$ σ′ *o* σ)
apply(*induct arbitrary*: σ σ′ *pred*:*pure*)
apply *simp-all*
apply(*erule-tac x*=V_{ML} *0* ## σ′ **in** *meta-allE*)

apply(*erule-tac x*= $V_{ML}\ 0\ \#\#\ (lift_{ML}\ 0 \circ \sigma)$ **in** *meta-allE*)
apply(*rule-tac f* = *compile t* **in** *arg-cong*)
apply(*rule ext*)
apply (*auto simp add*:*subst-ML-ext lift-ML-subst-ML*)
done

theorem *kernel-compile*:
  pure t $\Longrightarrow \forall\, i.\ \sigma\ i = V_U\ i\ [] \Longrightarrow$ (*compile t* σ)! = *t*
apply(*induct arbitrary*: σ *pred*:*pure*)
apply *simp-all*
apply(*subst lift-compile*) **apply** *simp*
apply(*subst subst-ML-compile*) **apply** *simp*
apply(*subgoal-tac* ($subst_{ML}\ (\lambda n.\ if\ n = 0\ then\ V_U\ 0\ []\ else\ V_{ML}\ (n - 1)) \circ$
  (*lift 0* $\circ\ V_{ML}\ 0\ \#\#\ \sigma$)) = ($\lambda a.\ V_U\ a\ []$))
apply(*simp*)
apply(*rule ext*)
apply(*simp*)
done

lemma *kernel-subst-ML-pat*:
  pure t $\Longrightarrow$ *pattern t* $\Longrightarrow \forall\, i.\ closed_{ML}\ 0\ (\sigma\ i) \Longrightarrow$
  ($subst_{ML}\ \sigma$ (*comp-pat t*))! = *subst* (*kernel* $\circ\ \sigma$) *t*
apply(*induct arbitrary*: σ *pred*:*pure*)
apply *simp-all*
apply(*frule pattern-At-decomp*)
apply(*frule pattern-AtD12*)
apply *clarsimp*
apply(*subst comp-pat.simps*)
apply(*simp add*: *rev-map*)
done

lemma *kernel-subst-ML*:
  pure t $\Longrightarrow \forall\, i.\ closed_{ML}\ 0\ (\sigma\ i) \Longrightarrow$
  ($subst_{ML}\ \sigma$ (*comp-open t*))! = *subst* (*kernel* $\circ\ \sigma$) *t*
proof(*induct arbitrary*: σ *pred*:*pure*)
  case (*Lam t*)
  have *lift 0 o* V_{ML} = V_{ML} **by** (*simp add*:*fun-eq-iff*)
  hence ($subst_{ML}\ \sigma$ (*comp-open* (Λ *t*)))! =
    Λ ($subst_{ML}$ (*lift 0* $\circ\ V_{ML}\ 0\ \#\#\ \sigma$) (*comp-open t*)[$V_U\ 0\ []/0$]!)
    using *Lam* **by**(*simp add*: *lift-subst-ML comp-open-def lift-compile*)
  also have ... = Λ (*subst* (*V 0* $\#\#$ (*kernel* $\circ\ \sigma$)) *t*) **using** *Lam*
    by(*simp add*: *subst-ML-comp subst-ext kernel-lift-tm*)
  also have ... = *subst* (*kernel o* σ) (Λ *t*) **by** *simp*
  finally show *?case* **.**
qed (*simp-all add*:*comp-open-def*)

lemma *kernel-subst-ML-pat-map*:
  $\forall\, t \in$ *set ts. pure t* $\Longrightarrow$ *patterns ts* $\Longrightarrow \forall\, i.\ closed_{ML}\ 0\ (\sigma\ i) \Longrightarrow$
   map kernel (*map* ($subst_{ML}\ \sigma$) (*map comp-pat ts*)) =

map (*subst* (*kernel* ∘ *σ*)) *ts*
by(*simp add*:*list-eq-iff-nth-eq kernel-subst-ML-pat*)

lemma *compR-Red-tm*: (*nm*, *vs*, *v*) : *compR* ⟹ ∀ *i*. $closed_{ML}$ *0* (*σ i*)
⟹ *C nm* •• (*map* ($subst_{ML}$ *σ*) (*rev vs*))! →∗ ($subst_{ML}$ *σ v*)!
apply(*auto simp add*:*compR-def rev-map simp del*: *map-map*)
apply(*frule pure-R*)
apply(*subst kernel-subst-ML*) **apply** *fast*+
apply(*subst kernel-subst-ML-pat-map*)
apply *fast*
apply(*fast dest*:*pattern-R*)
apply *assumption*
apply(*rule r-into-rtrancl*)
apply(*erule Red-tm.intros*)
done

5 Correctness

lemma *eq-Red-tm-trans*: *s* = *t* ⟹ *t* → *t′* ⟹ *s* → *t′*
by *simp*

Soundness of reduction:

theorem fixes *v* :: *ml* **shows** *Red-ml-sound*:
v ⇒ *v′* ⟹ $closed_{ML}$ *0 v* ⟹ *v*! →∗ *v′*! ∧ $closed_{ML}$ *0 v′* **and**
vs ⇒ *vs′* ⟹ ∀ *v*∈*set vs*. $closed_{ML}$ *0 v* ⟹
vs! →∗ *vs′*! ∧ (∀ *v′*∈*set vs′*. $closed_{ML}$ *0 v′*)
proof(*induct rule*:*Red-ml-Red-ml-list.inducts*)
fix *u v*
let *?v* = A_{ML} (Lam_{ML} *u*) [*v*]
assume *cl*: $closed_{ML}$ *0* (A_{ML} (Lam_{ML} *u*) [*v*])
let *?u′* = (*lift-ml 0 u*)[V_U *0* []/*0*]
have *?v*! = (Λ((*?u′*)!)) ∙ (*v* !) **by** *simp*
also have ... → (*?u′* !)[*v*!/*0*] (**is** - → *?R*) **by**(*rule Red-tm.intros*)
also(*eq-Red-tm-trans*) **have** *?R* = *u*[*v*/*0*]! **using** *cl*
apply(*cut-tac u* = *u* **and** *v* = *v* **in** *kernel-subst1*)
apply(*simp-all*)
done
finally have *kernel*(A_{ML} (Lam_{ML} *u*) [*v*]) →∗ *kernel*(*u*[*v*/*0*]) (**is** *?A*)
by(*rule r-into-rtrancl*)
moreover have $closed_{ML}$ *0* (*u*[*v*/*0*]) (**is** *?C*)
proof −
let *?σ* = λ*n*. *if n* = *0 then v else* V_{ML} (*n* − *1*)
let *?σ′* = λ*n*. *v*
have *clu*: $closed_{ML}$ (*Suc 0*) *u* **and** *clv*: $closed_{ML}$ *0 v* **using** *cl* **by** *simp*+
have $closed_{ML}$ *0* ($subst_{ML}$ *?σ′ u*)
by (*metis closed-ML-subst-ML clv*)
hence $closed_{ML}$ *0* ($subst_{ML}$ *?σ u*)
using *subst-ML-coincidence*[*OF clu*, *of ?σ ?σ′*] **by** *auto*
thus *?thesis* **by** *simp*

qed
ultimately show *?A* ∧ *?C* **..**
next
fix σ :: *nat* ⇒ *ml* **and** *nm vs v*
assume σ: ∀ *i*. $closed_{ML}$ *0* (σ *i*) **and** *compR*: (*nm*, *vs*, *v*) ∈ *compR*
have *map* (*subst V*) (*map* ($subst_{ML}$ σ) (*rev vs*)!) = *map* ($subst_{ML}$ σ) (*rev vs*)!
by(*simp add*:*list-eq-iff-nth-eq subst-V-kernel closed-ML-subst-ML*[*OF* σ])
with *compR-Red-tm*[*OF compR* σ]
have (*C nm*) ·· ((*map* ($subst_{ML}$ σ) (*rev vs*)) !) →∗ ($subst_{ML}$ σ *v*)!
by(*simp add*:*subst-V-kernel closed-ML-subst-ML*[*OF* σ])
hence A_{ML} (C_{ML} *nm*) (*map* ($subst_{ML}$ σ) *vs*)! →∗ $subst_{ML}$ σ *v*! (**is** *?A*)
by(*simp add*:*rev-map*)
moreover
have $closed_{ML}$ *0* ($subst_{ML}$ σ *v*) (**is** *?C*) **by**(*metis closed-ML-subst-ML* σ)
ultimately show *?A* ∧ *?C* **..**
qed (*auto simp*:*Reds-tm-list-foldl-At Red-tm-rev rev-map*[*symmetric*])

theorem *Red-term-sound*:
t ⇒ *t*′ ⟹ $closed_{ML}$ *0 t* ⟹ *kernelt t* →∗ *kernelt t*′ ∧ $closed_{ML}$ *0 t*′
proof(*induct rule*:*Red-term.inducts*)
case *term-C* **thus** *?case*
by (*auto simp*:*closed-tm-ML-foldl-At*)
next
case *term-V* **thus** *?case*
by (*auto simp*:*closed-tm-ML-foldl-At*)
next
case (*term-Clo vf vs n*)
hence (*lift 0 vf*!) ·· *map kernel* (*rev* (*map* (*lift 0*) *vs*))
= *lift 0* (*vf*! ·· (*rev vs*)!)
apply (*simp add*:*kernel-lift-tm list-eq-iff-nth-eq*)
apply(*simp add*:*rev-nth rev-map kernel-lift-tm*)
done
hence *term* (*Clo vf vs n*)! →∗
Λ (*term* (*apply* (*lift 0* (*Clo vf vs n*)) (V_U *0* []))!
using *term-Clo*
by(*simp del*:*lift-foldl-At add*: *r-into-rtrancl Red-tm.intros*(*2*))
moreover
have $closed_{ML}$ *0* (Λ (*term* (*apply* (*lift 0* (*Clo vf vs n*)) (V_U *0* []))))
using *term-Clo* **by** *simp*
ultimately show *?case* **..**
next
case *ctxt-term* **thus** *?case* **by** *simp* (*metis Red-ml-sound*)
qed *auto*

corollary *kernel-inv*:
(*t* :: *tm*) ⇒∗ *t*′ ⟹ $closed_{ML}$ *0 t* ⟹ *t*! →∗ *t*′! ∧ $closed_{ML}$ *0 t*′
apply(*induct rule*: *rtrancl.induct*)
apply (*metis rtrancl-eq-or-trancl*)
apply (*metis Red-term-sound rtrancl-trans*)

done

lemma *closed-ML-compile*:
pure t $\Longrightarrow$ $\forall$ *i. closed*$_{ML}$ *n* (*σ i*) $\Longrightarrow$ *closed*$_{ML}$ *n* (*compile t σ*)
proof(*induct arbitrary*:*n σ pred*:*pure*)
case (*Lam t*)
have *1*: $\forall$ *i. closed*$_{ML}$ (*Suc n*) ((*V*$_{ML}$ *0* ## *σ*) *i*) **using** *Lam*(*3*−)
by (*auto simp*: *closed-ML-Suc*)
show *?case* **using** *Lam*(*2*)[*OF 1*] **by** (*simp del*:*apply-cons-ML*)
qed *simp-all*

theorem *nbe-correct*: **fixes** *t* :: *tm*
assumes *pure t* **and** *term* (*comp-fixed t*) $\Rightarrow$* *t′* **and** *pure t′* **shows** *t* $\rightarrow$* *t′*
proof −
have *ML-cl*: *closed*$_{ML}$ *0* (*term* (*comp-fixed t*))
by (*simp add*: *closed-ML-compile*[*OF* ‹*pure t*›])
have (*term* (*comp-fixed t*))! = *t*
using *kernel-compile*[*OF* ‹*pure t*›] **by** *simp*
moreover have *term* (*comp-fixed t*)! $\rightarrow$* *t′*!
using *kernel-inv*[*OF assms*(*2*) *ML-cl*] **by** *auto*
ultimately have *t* $\rightarrow$* *t′*! **by** *simp*
thus *?thesis* **using** *kernel-pure*[*OF* ‹*pure t′*›] **by** *simp*
qed

6 Normal Forms

inductive *normal* :: *tm* $\Rightarrow$ *bool* **where**
$\forall$ *t*∈*set ts. normal t* $\Longrightarrow$ *normal*(*V x* •• *ts*) |
normal t $\Longrightarrow$ *normal*(Λ *t*) |
$\forall$ *t*∈*set ts. normal t* $\Longrightarrow$
$\forall$ *σ*. $\forall$ (*nm′*,*ls*,*r*)∈*R*. ¬(*nm* = *nm′* ∧ *take* (*size ls*) *ts* = *map* (*subst σ*) *ls*)
$\Longrightarrow$ *normal*(*C nm* •• *ts*)

fun *C-normal-ML* :: *ml* $\Rightarrow$ *bool* (‹*C′-normal*$_{ML}$›) **where**
C-normal$_{ML}$(*C*$_U$ *nm vs*) =
(($\forall$ *v*∈*set vs. C-normal*$_{ML}$ *v*) ∧ *no-match-compR nm vs*) |
C-normal$_{ML}$ (*C*$_{ML}$ -) = *True* |
C-normal$_{ML}$ (*V*$_{ML}$ -) = *True* |
C-normal$_{ML}$ (*A*$_{ML}$ *v vs*) = (*C-normal*$_{ML}$ *v* ∧ ($\forall$ *v* ∈ *set vs. C-normal*$_{ML}$ *v*)) |
C-normal$_{ML}$ (*Lam*$_{ML}$ *v*) = *C-normal*$_{ML}$ *v* |
C-normal$_{ML}$ (*V*$_U$ *x vs*) = ($\forall$ *v* ∈ *set vs. C-normal*$_{ML}$ *v*) |
C-normal$_{ML}$ (*Clo v vs* -) = (*C-normal*$_{ML}$ *v* ∧ ($\forall$ *v* ∈ *set vs. C-normal*$_{ML}$ *v*)) |
C-normal$_{ML}$ (*apply u v*) = (*C-normal*$_{ML}$ *u* ∧ *C-normal*$_{ML}$ *v*)

fun *size-tm* :: *tm* $\Rightarrow$ *nat* **where**
size-tm (*C* -) = *1* |
size-tm (*At s t*) = *size-tm s* + *size-tm t* + *1* |
size-tm - = *0*

lemma *size-tm-foldl-At*: *size-tm*(t ·· *ts*) = *size-tm t* + *size-list size-tm ts*
by (*induct ts arbitrary*:*t*) *auto*

lemma *termination-no-match*:
  i < *length ss* ⟹ *ss* ! i = C *nm* ·· *ts*
   ⟹ *sum-list* (*map size-tm ts*) < *sum-list* (*map size-tm ss*)
apply(*subgoal-tac* C *nm* ·· *ts* : *set ss*)
 apply(*drule sum-list-map-remove1*[*of* - - *size-tm*])
apply(*simp add*:*size-tm-foldl-At size-list-conv-sum-list*)
apply (*metis in-set-conv-nth*)
done

declare *conj-cong* [*fundef-cong*]

function *no-match* :: *tm list* ⇒ *tm list* ⇒ *bool* **where**
no-match ps ts =
  (∃ i < *min* (*size ts*) (*size ps*).
   ∃ *nm nm′ rs rs′*. *ps*!i = (C *nm*) ·· *rs* ∧ *ts*!i = (C *nm′*) ·· *rs′* ∧
      (*nm*=*nm′* ⟶ *no-match rs rs′*))
by *pat-completeness auto*
termination
apply(*relation measure*(%(*ts*::*tm list*,-). ∑ *t*←*ts*. *size-tm t*))
apply (*auto simp*:*termination-no-match*)
done

declare *no-match.simps*[*simp del*]

abbreviation
no-match-R nm ts ≡ ∀ (*nm′*,*ps*,*t*)∈ *R*. *nm*=*nm′* ⟶ *no-match ps ts*

lemma *no-match*: *no-match ps ts* ⟹ ¬(∃ σ. *map* (*subst* σ) *ps* = *ts*)
proof(*induct ps ts rule*:*no-match.induct*)
  case (*1 ps ts*)
  thus *?case*
    apply *auto*
    apply(*subst* (*asm*) *no-match.simps*[*of ps*])
    apply *fastforce*
    done
qed

lemma *no-match-take*: *no-match ps ts* ⟹ *no-match ps* (*take* (*size ps*) *ts*)
apply(*subst* (*asm*) *no-match.simps*)
apply(*subst no-match.simps*)
apply *fastforce*
done

fun *dterm-ML* :: *ml* ⇒ *tm* (‹*dterm*$_{ML}$›) **where**
dterm$_{ML}$ (C_U *nm vs*) = C *nm* ·· *map dterm*$_{ML}$ (*rev vs*) |

$dterm_{ML}$ - = V *0*

fun *dterm* :: *tm* ⇒ *tm* **where**
dterm (*V n*) = *V n* |
dterm (*C nm*) = *C nm* |
dterm (*s* ⋅ *t*) = *dterm s* ⋅ *dterm t* |
dterm (Λ *t*) = Λ (*dterm t*) |
dterm (*term v*) = $dterm_{ML}$ *v*

lemma *dterm-pure*[*simp*]: *pure t* ⟹ *dterm t* = *t*
by (*induct pred*:*pure*) *auto*

lemma *map-dterm-pure*[*simp*]: ∀ *t*∈*set ts*. *pure t* ⟹ *map dterm ts* = *ts*
by (*induct ts*) *auto*

lemma *map-dterm-term*[*simp*]: *map dterm* (*map term vs*) = *map* $dterm_{ML}$ *vs*
by (*induct vs*) *auto*

lemma *dterm-foldl-At*[*simp*]: *dterm*(*t* ⋅⋅ *ts*) = *dterm t* ⋅⋅ *map dterm ts*
by(*induct ts arbitrary*: *t*) *auto*

lemma *no-match-coincide*:
  $no\text{-}match_{ML}$ *ps vs* ⟹
  no-match (*map* $dterm_{ML}$ (*rev ps*)) (*map* $dterm_{ML}$ (*rev vs*))
apply(*induct ps vs rule*:*no-match-ML.induct*)
apply(*rotate-tac 1*)
apply(*subst* (*asm*) *no-match-ML.simps*)
apply (*elim exE conjE*)
apply(*case-tac nm*=*nm*′)
prefer *2*
apply(*subst no-match.simps*)
apply(*rule-tac x*=*i* **in** *exI*)
apply *rule*
apply (*simp* (*no-asm*))
apply (*metis min-less-iff-conj*)
apply(*simp add*:*min-less-iff-conj nth-map*)
apply *safe*
apply(*erule-tac x*=*i* **in** *meta-allE*)
apply(*erule-tac x*=*nm*′ **in** *meta-allE*)
apply(*erule-tac x*=*nm*′ **in** *meta-allE*)
apply(*erule-tac x*=*vs* **in** *meta-allE*)
apply(*erule-tac x*=*vs*′ **in** *meta-allE*)
apply(*subst no-match.simps*)
apply(*rule-tac x*=*i* **in** *exI*)
apply *rule*
apply (*simp* (*no-asm*))
apply (*metis min-less-iff-conj*)
apply(*rule-tac x*=*nm*′ **in** *exI*)
apply(*rule-tac x*=*nm*′ **in** *exI*)

apply(*rule-tac x=map dterm$_{ML}$ (rev vs)* **in** *exI*)
apply(*rule-tac x=map dterm$_{ML}$ (rev vs')* **in** *exI*)
apply(*simp*)
done

lemma *dterm-ML-comp-patD*:
pattern t $\Longrightarrow$ *dterm$_{ML}$ (comp-pat t) = C nm ·· rs* $\Longrightarrow$ $\exists$ *ts. t = C nm ·· ts*
by(*induct pred:pattern*) *simp-all*

lemma *no-match-R-coincide-aux*[*rule-format*]: *patterns ts* $\Longrightarrow$
no-match (map (dterm$_{ML}$ $\circ$ *comp-pat) ts) rs* $\longrightarrow$ *no-match ts rs*
apply(*induct ts rs rule:no-match.induct*)
apply(*subst (1 2) no-match.simps*)
apply *clarsimp*
apply(*rule-tac x=i* **in** *exI*)
apply *simp*
apply(*rule-tac x=nm* **in** *exI*)
apply(*cut-tac t = ps!i* **in** *dterm-ML-comp-patD*, *simp*, *assumption*)
apply(*clarsimp*)
apply(*erule-tac x = i* **in** *meta-allE*)
apply(*erule-tac x = nm'* **in** *meta-allE*)
apply(*erule-tac x = nm'* **in** *meta-allE*)
apply(*erule-tac x = tsa* **in** *meta-allE*)
apply(*erule-tac x = rs'* **in** *meta-allE*)
apply (*simp add:rev-map*)
apply (*metis in-set-conv-nth pattern-At-vecD*)
done

lemma *no-match-R-coincide*:
no-match-compR nm (rev vs) $\Longrightarrow$ *no-match-R nm (map dterm$_{ML}$ vs)*
apply *auto*
apply(*drule-tac x=(nm, map comp-pat (rev aa), comp-open b)* **in** *bspec*)
unfolding *compR-def*
apply (*simp add:image-def*)
apply (*force*)
apply (*simp*)
apply(*drule no-match-coincide*)
apply(*frule pure-R*)
apply(*drule pattern-R*)
apply(*clarsimp simp add: rev-map no-match.simps*[*of - map dterm$_{ML}$ vs*])
apply(*rule-tac x=i* **in** *exI*)
apply *simp*
apply(*cut-tac t = aa!i* **in** *dterm-ML-comp-patD*, *simp*, *assumption*)
apply *clarsimp*
apply(*auto simp: rev-map*)
apply(*rule no-match-R-coincide-aux*)
prefer *2* **apply** *assumption*
apply (*metis in-set-conv-nth pattern-At-vecD*)
done

inductive *C-normal* :: *tm* ⇒ *bool* **where**
∀ *t*∈*set ts*. *C-normal t* ⟹ *C-normal*(*V x* •• *ts*) |
C-normal t ⟹ *C-normal*(Λ *t*) |
C-normal$_{ML}$ *v* ⟹ *C-normal*(*term v*) |
∀ *t*∈*set ts*. *C-normal t* ⟹ *no-match-R nm* (*map dterm ts*)
⟹ *C-normal*(*C nm* •• *ts*)

declare *C-normal.intros*[*simp*]

lemma *C-normal-term*[*simp*]: *C-normal*(*term v*) = *C-normal*$_{ML}$ *v*
apply (*auto*)
apply(*erule C-normal.cases*)
apply *auto*
done

lemma [*simp*]: *C-normal*(Λ *t*) = *C-normal t*
apply (*auto*)
apply(*erule C-normal.cases*)
apply *auto*
done

lemma [*simp*]: *C-normal*(*V x*)
using *C-normal.intros*(*1*)[*of* [] *x*]
by *simp*

lemma [*simp*]: *dterm* (*dterm*$_{ML}$ *v*) = *dterm*$_{ML}$ *v*
apply(*induct v rule*:*dterm-ML.induct*)
apply *simp-all*
done

lemma *u*⇒(*v*::*ml*) ⟹ *True* **and**
Red-ml-list-length: *vs* ⇒ *vs*′ ⟹ *length vs* = *length vs*′
by(*induct rule*: *Red-ml-Red-ml-list.inducts*) *simp-all*

lemma (*v*::*ml*) ⇒ *v*′ ⟹ *True* **and**
Red-ml-list-nth: (*vs*::*ml list*) ⇒ *vs*′
⟹ ∃ *v*′ *k*. *k*<*size vs* ∧ *vs*!*k* ⇒ *v*′ ∧ *vs*′ = *vs*[*k* := *v*′]
apply (*induct rule*: *Red-ml-Red-ml-list.inducts*)
apply (*auto split*:*nat.splits*)
done

lemma *Red-ml-list-pres-no-match*:
no-match$_{ML}$ *ps vs* ⟹ *vs* ⇒ *vs*′ ⟹ *no-match*$_{ML}$ *ps vs*′
proof(*induct ps vs arbitrary*: *vs*′ *rule*:*no-match-ML.induct*)
case (*1 vs os*)
show *?case* **using** *1*(*2*−*3*)
apply−

apply(*frule Red-ml-list-length*)
apply(*rotate-tac −2*)
apply(*subst* (*asm*) *no-match-ML.simps*)
apply *clarify*
apply(*rename-tac i nm nm′ us us′*)
apply(*subst no-match-ML.simps*)
apply(*rule-tac x=i* **in** *exI*)
apply (*simp*)
apply(*drule Red-ml-list-nth*)
apply *clarify*
apply(*rename-tac k*)
apply(*case-tac k = length os − Suc i*)
prefer *2*
apply(*rule-tac x=nm′* **in** *exI*)
apply(*rule-tac x=us′* **in** *exI*)
apply (*simp add*: *rev-nth nth-list-update*)
apply (*simp add*: *rev-nth*)
apply(*erule Red-ml.cases*)
apply *simp-all*
apply(*fastforce intro*: *1*(*1*) *simp add*:*rev-nth*)
done
qed

lemma *no-match-ML-subst-ML*[*rule-format*]:
  ∀ *v*∈*set vs*. ∀ *x*∈fv_{ML} *v*. *C-normal*$_{ML}$ (*σ x*) ⟹
    no-match$_{ML}$ *ps vs* ⟶ *no-match*$_{ML}$ *ps* (*map* (*subst*$_{ML}$ *σ*) *vs*)
apply(*induct ps vs rule*:*no-match-ML.induct*)
apply *simp*
apply(*subst* (*1 2*) *no-match-ML.simps*)
apply *clarsimp*
apply(*rule-tac x=i* **in** *exI*)
apply *simp*
apply(*rule-tac x=nm′* **in** *exI*)
apply(*rule-tac x=map* (*subst*$_{ML}$ *σ*) *vs′* **in** *exI*)
apply (*auto simp*:*rev-nth*)
apply(*erule-tac x = i* **in** *meta-allE*)
apply(*erule-tac x = nm′* **in** *meta-allE*)
apply(*erule-tac x = nm′* **in** *meta-allE*)
apply(*erule-tac x = vs* **in** *meta-allE*)
apply(*erule-tac x = vs′* **in** *meta-allE*)
apply *simp*
apply (*metis UN-I fv-ML.simps*(*5*) *in-set-conv-nth length-rev rev-nth set-rev*)
done

lemma *lift-is-CUD*:
  lift$_{ML}$ *k v* = C_U *nm vs′* ⟹ ∃ *vs*. *v* = C_U *nm vs* ∧ *vs′* = *map* (*lift*$_{ML}$ *k*) *vs*
by(*cases v*) *auto*

lemma *no-match-ML-lift-ML*:

$no\text{-}match_{ML}\ ps\ (map\ (lift_{ML}\ k)\ vs) = no\text{-}match_{ML}\ ps\ vs$
apply(*induct ps vs rule*:*no-match-ML.induct*)
apply *simp*
apply(*subst* (*1 2*) *no-match-ML.simps*)
apply *rule*
apply *clarsimp*
apply(*rule-tac x*=*i* **in** *exI*)
apply (*simp add*:*rev-nth*)
apply(*drule lift-is-CUD*)
apply *fastforce*
apply *clarsimp*
apply(*rule-tac x*=*i* **in** *exI*)
apply *simp*
apply(*rule-tac x*=nm' **in** *exI*)
apply(*rule-tac x*=$map\ (lift_{ML}\ k)\ vs'$ **in** *exI*)
apply (*fastforce simp*:*rev-nth*)
done

lemma *C-normal-ML-lift-ML*: $C\text{-}normal_{ML}(lift_{ML}\ k\ v) = C\text{-}normal_{ML}\ v$
by(*induct v arbitrary*: *k rule*:*C-normal-ML.induct*)(*auto simp*:*no-match-ML-lift-ML*)

lemma *no-match-compR-Cons*:
$no\text{-}match\text{-}compR\ nm\ vs \Longrightarrow no\text{-}match\text{-}compR\ nm\ (v\ \#\ vs)$
apply *auto*
apply(*drule bspec*, *assumption*)
apply *simp*
apply(*subst* (*asm*) *no-match-ML.simps*)
apply(*subst no-match-ML.simps*)
apply *clarsimp*
apply(*rule-tac x*=*i* **in** *exI*)
apply (*simp add*:*nth-append*)
done

lemma *C-normal-ML-comp-open*: *pure t* $\Longrightarrow C\text{-}normal_{ML}(comp\text{-}open\ t)$
by (*induct pred*:*pure*) (*auto simp*:*comp-open-def*)

lemma *C-normal-compR-rhs*: $(nm, vs, v) \in compR \Longrightarrow C\text{-}normal_{ML}\ v$
by(*auto simp*: *compR-def image-def Bex-def pure-R C-normal-ML-comp-open*)

lemma *C-normal-ML-subst-ML*:
$C\text{-}normal_{ML}\ (subst_{ML}\ \sigma\ v) \Longrightarrow (\forall\, x{\in}fv_{ML}\ v.\ C\text{-}normal_{ML}\ (\sigma\ x))$
proof(*induct* σ *v rule*:*subst-ML.induct*)
case *4* **thus** *?case*
by(*simp del*:*apply-cons-ML*)(*force simp add*: *C-normal-ML-lift-ML*)

qed *auto*

lemma *C-normal-ML-subst-ML-iff*: $C\text{-}normal_{ML}\ v \Longrightarrow$

$C\text{-}normal_{ML}\ (subst_{ML}\ \sigma\ v) \longleftrightarrow (\forall x{\in}fv_{ML}\ v.\ C\text{-}normal_{ML}\ (\sigma\ x))$
proof(*induct* σ v *rule*:*subst-ML.induct*)
case *4* **thus** *?case*
by(*simp del*:*apply-cons-ML*)(*force simp add*: *C-normal-ML-lift-ML*)

next
case *5* **thus** *?case* **by** *simp* (*blast intro*: *no-match-ML-subst-ML*)
qed *auto*

lemma *C-normal-ML-inv*: $v \Rightarrow v' \Longrightarrow C\text{-}normal_{ML}\ v \Longrightarrow C\text{-}normal_{ML}\ v'$ **and**
$vs \Rightarrow vs' \Longrightarrow \forall v{\in}set\ vs.\ C\text{-}normal_{ML}\ v \Longrightarrow \forall v'{\in}set\ vs'.\ C\text{-}normal_{ML}\ v'$
apply(*induct rule*:*Red-ml-Red-ml-list.inducts*)
apply(*simp-all add*: *C-normal-ML-subst-ML-iff*)
apply(*metis C-normal-ML-subst-ML C-normal-compR-rhs*
fv-compR C-normal-ML-subst-ML-iff)
apply(*blast intro*!:*no-match-compR-Cons*)
apply(*blast dest*:*Red-ml-list-pres-no-match*)
done

lemma *Red-term-hnf-induct*[*consumes 1*]:
assumes $(t{::}tm) \Rightarrow t'$
$\bigwedge nm\ vs\ ts.\ P\ ((term\ (C_U\ nm\ vs)) \cdot\cdot ts)\ ((C\ nm \cdot\cdot map\ term\ (rev\ vs)) \cdot\cdot ts)$
$\bigwedge x\ vs\ ts.\ P\ (term\ (V_U\ x\ vs) \cdot\cdot ts)\ ((V\ x \cdot\cdot map\ term\ (rev\ vs)) \cdot\cdot ts)$
$\bigwedge vf\ vs\ n\ ts.$
$P\ (term\ (Clo\ vf\ vs\ n) \cdot\cdot ts)$
$((\Lambda\ (term\ (apply\ (lift\ 0\ (Clo\ vf\ vs\ n))\ (V_U\ 0\ []))))) \cdot\cdot ts)$
$\bigwedge t\ t'\ ts.\ [\![t \Rightarrow t';\ P\ t\ t']\!] \Longrightarrow P\ (\Lambda\ t \cdot\cdot ts)\ (\Lambda\ t' \cdot\cdot ts)$
$\bigwedge v\ v'\ ts.\ v \Rightarrow v' \Longrightarrow P\ (term\ v \cdot\cdot ts)\ (term\ v' \cdot\cdot ts)$
$\bigwedge x\ i\ t'\ ts.\ i{<}size\ ts \Longrightarrow ts!i \Rightarrow t' \Longrightarrow P\ (ts!i)\ (t')$
$\Longrightarrow P\ (V\ x \cdot\cdot ts)\ (V\ x \cdot\cdot ts[i{:=}t'])$
$\bigwedge nm\ i\ t'\ ts.\ i{<}size\ ts \Longrightarrow ts!i \Rightarrow t' \Longrightarrow P\ (ts!i)\ (t')$
$\Longrightarrow P\ (C\ nm \cdot\cdot ts)\ (C\ nm \cdot\cdot ts[i{:=}t'])$
$\bigwedge t\ i\ t'\ ts.\ i{<}size\ ts \Longrightarrow ts!i \Rightarrow t' \Longrightarrow P\ (ts!i)\ (t')$
$\Longrightarrow P\ (\Lambda\ t \cdot\cdot ts)\ (\Lambda\ t \cdot\cdot ts[i{:=}t'])$
$\bigwedge v\ i\ t'\ ts.\ i{<}size\ ts \Longrightarrow ts!i \Rightarrow t' \Longrightarrow P\ (ts!i)\ (t')$
$\Longrightarrow P\ (term\ v \cdot\cdot ts)\ (term\ v \cdot\cdot (ts[i{:=}t']))$
shows $P\ t\ t'$
proof−
{ fix *ts* **from** *assms* **have** $P\ (t \cdot\cdot ts)\ (t' \cdot\cdot ts)$
proof(*induct arbitrary*: *ts rule*:*Red-term.induct*)
case *term-C* **thus** *?case* **by** *metis*
next
case *term-V* **thus** *?case* **by** *metis*
next
case *term-Clo* **thus** *?case* **by** *metis*
next
case *ctxt-Lam* **thus** *?case* **by** *simp* (*metis foldl-Nil*)
next

```
    case (ctxt-At1 s s' t ts)
    thus ?case using ctxt-At1(2)[of t#ts] by simp
  next
    case (ctxt-At2 t t' s ts)
    { fix n rs assume s = V n •• rs
      hence ?case using ctxt-At2(8)[of size rs rs @ t # ts t' n] ctxt-At2
        by simp (metis foldl-Nil)
    } moreover
    { fix nm rs assume s = C nm •• rs
      hence ?case using ctxt-At2(9)[of size rs rs @ t # ts t' nm] ctxt-At2
        by simp (metis foldl-Nil)
    } moreover
    { fix r rs assume s = Λ r •• rs
      hence ?case using ctxt-At2(10)[of size rs rs @ t # ts t'] ctxt-At2
        by simp (metis foldl-Nil)
    } moreover
    { fix v rs assume s = term v •• rs
      hence ?case using ctxt-At2(11)[of size rs rs @ t # ts t'] ctxt-At2
        by simp (metis foldl-Nil)
    } ultimately show ?case using tm-vector-cases[of s] by blast
  qed
 }
 from this[of []] show ?thesis by simp
qed

corollary Red-term-hnf-cases[consumes 1]:
assumes (t::tm) ⇒ t'
 ⋀nm vs ts.
 t = term (C_U nm vs) •• ts ⟹ t' = (C nm •• map term (rev vs)) •• ts ⟹ P
 ⋀x vs ts.
 t = term (V_U x vs) •• ts ⟹ t' = (V x •• map term (rev vs)) •• ts ⟹ P
 ⋀vf vs n ts. t = term (Clo vf vs n) •• ts ⟹
   t' = Λ (term (apply (lift 0 (Clo vf vs n)) (V_U 0 []))) •• ts ⟹ P
 ⋀s s' ts. t = Λ s •• ts ⟹ t' = Λ s' •• ts ⟹ s ⇒ s' ⟹ P
 ⋀v v' ts. t = term v •• ts ⟹ t' = term v' •• ts ⟹ v ⇒ v' ⟹ P
 ⋀x i r' ts. i<size ts ⟹ ts!i ⇒ r'
   ⟹ t = V x •• ts ⟹ t' = V x •• ts[i:=r'] ⟹ P
 ⋀nm i r' ts. i<size ts ⟹ ts!i ⇒ r'
   ⟹ t = C nm •• ts ⟹ t' = C nm •• ts[i:=r'] ⟹ P
 ⋀s i r' ts. i<size ts ⟹ ts!i ⇒ r'
   ⟹ t = Λ s •• ts ⟹ t' = Λ s •• ts[i:=r'] ⟹ P
 ⋀v i r' ts. i<size ts ⟹ ts!i ⇒ r'
   ⟹ t = term v •• ts ⟹ t' = term v •• (ts[i:=r']) ⟹ P
shows P using assms
apply −
apply(induct rule:Red-term-hnf-induct)
apply metis+
done
```

lemma [*simp*]: *C-normal*(*term v* •• *ts*) ⟷ *C-normal*$_{ML}$ *v* ∧ *ts* = []
by(*fastforce elim*: *C-normal.cases*)

lemma [*simp*]: *C-normal*(Λ *t* •• *ts*) ⟷ *C-normal t* ∧ *ts* = []
by(*fastforce elim*: *C-normal.cases*)

lemma [*simp*]: *C-normal*(*C nm* •• *ts*) ⟷
(∀ *t*∈*set ts*. *C-normal t*) ∧ *no-match-R nm* (*map dterm ts*)
by(*fastforce elim*: *C-normal.cases*)

lemma [*simp*]: *C-normal*(*V x* •• *ts*) ⟷ (∀ *t* ∈ *set ts*. *C-normal t*)
by(*fastforce elim*: *C-normal.cases*)

lemma *no-match-ML-lift*:
no-match$_{ML}$ *ps vs* ⟶ *no-match*$_{ML}$ *ps* (*map* (*lift k*) *vs*)
apply(*induct ps vs rule*:*no-match-ML.induct*)
apply *simp*
apply(*subst* (*1 2*) *no-match-ML.simps*)
apply *clarsimp*
apply(*rule-tac x*=*i* **in** *exI*)
apply *simp*
apply(*rule-tac x*=*nm'* **in** *exI*)
apply(*rule-tac x*=*map* (*lift k*) *vs'* **in** *exI*)
apply (*fastforce simp*:*rev-nth*)
done

lemma *no-match-compR-lift*:
no-match-compR nm vs ⟹ *no-match-compR nm* (*map* (*lift k*) *vs*)
by (*fastforce simp*: *no-match-ML-lift*)

lemma [*simp*]: *C-normal*$_{ML}$ *v* ⟹ *C-normal*$_{ML}$(*lift k v*)
apply(*induct v arbitrary*:*k rule*:*lift-ml.induct*)
apply(*simp-all add*:*no-match-compR-lift*)
done

declare [[*simp-depth-limit* = *10*]]

lemma *Red-term-pres-no-match*:
⟦*i* < *length ts*; *ts* ! *i* ⇒ *t'*; *no-match ps dts*; *dts* = (*map dterm ts*)⟧
⟹ *no-match ps* (*map dterm* (*ts*[*i* := *t'*]))
proof(*induct ps dts arbitrary*: *ts i t' rule*:*no-match.induct*)
case (*1 ps dts ts i t'*)
from ‹*no-match ps dts*› ‹*dts* = *map dterm ts*›
obtain *j nm nm' rs rs'* **where** *ob*: *j* < *size ts j* < *size ps*
ps!*j* = *C nm* •• *rs dterm* (*ts*!*j*) = *C nm'* •• *rs'*
nm = *nm'* ⟶ *no-match rs rs'*
by (*subst* (*asm*) *no-match.simps*) *fastforce*
show *?case*

```
  proof (subst no-match.simps)
    show ∃k<min (length (map dterm (ts[i := t']))) (length ps).
       ∃nm nm' rs rs'. ps!k  = C nm •• rs ∧
         map dterm (ts[i := t']) ! k = C nm' •• rs' ∧
        (nm = nm' ⟶ no-match rs rs')
      (is ∃k < ?m. ?P k)
    proof−
      { assume [simp]: j=i
        have ∃rs'. dterm t' = C nm' •• rs' ∧ (nm = nm' ⟶ no-match rs rs')
          using ‹ts ! i ⇒ t'›
        proof(cases rule:Red-term-hnf-cases)
          case (5 v v' ts'')
          then obtain vs where [simp]:
            v = C_U nm' vs rs' = map dterm_ML (rev vs) @ map dterm ts''
            using ob by(cases v) auto
          obtain vs' where [simp]: v' = C_U nm' vs' vs ⇒ vs'
            using ‹v⇒v'› by(rule Red-ml.cases) auto
          obtain v' k where [arith]: k<size vs and vs!k ⇒ v'
            and [simp]: vs' = vs[k := v']
            using Red-ml-list-nth[OF ‹vs⇒vs'›] by fastforce
          show ?thesis (is ∃rs'. ?P rs' ∧ ?Q rs')
          proof
          let ?rs' = map dterm ((map term (rev vs) @ ts'')[(size vs − k − 1):=term
v'])
            have ?P ?rs' using ob 5
              by(simp add: list-update-append map-update[symmetric] rev-update)
            moreover have ?Q ?rs'
              apply rule
              apply(rule 1.hyps[OF - ob(3)])
              using 1.prems 5 ob
               apply (auto simp:nth-append rev-nth ctxt-term[OF ‹vs!k ⇒ v'›] simp
del: map-map)
              done
            ultimately show ?P ?rs' ∧ ?Q ?rs' ..
          qed
        next
          case (7 nm'' k r' ts'')
          show ?thesis (is ∃rs'. ?P rs')
          proof
            show ?P(map dterm (ts''[k := r']))
              using 7 ob
              apply clarsimp
              apply(rule 1.hyps[OF - ob(3)])
              using 7 1.prems ob apply auto
              done
          qed
        next
          case (9 v k r' ts'')
          then obtain vs where [simp]: v = C_U nm' vs rs' = map dterm_ML (rev
```

```
vs) @ map dterm ts''
          using ob by(cases v) auto
        show ?thesis (is ∃ rs'. ?P rs' ∧ ?Q rs')
        proof
          let ?rs' = map dterm ((map term (rev vs) @ ts'')[k+size vs:=r'])
          have ?P ?rs' using ob 9 by (auto simp: list-update-append)
          moreover have ?Q ?rs'
            apply rule
            apply(rule 1.hyps[OF - ob(3)])
            using 9 1.prems ob by (auto simp:nth-append simp del: map-map)
          ultimately show ?P ?rs' ∧ ?Q ?rs' ..
        qed
      qed (insert ob, auto simp del: map-map)
    }
    hence ∃ rs'. dterm (ts[i := t'] ! j) = C nm' •• rs' ∧ (nm = nm' ⟶ no-match
rs rs')
      using ‹i < size ts› ob by(simp add:nth-list-update)
    hence ?P j using ob by auto
    moreover have j < ?m using ‹j < length ts› ‹j < size ps› by simp
    ultimately show ?thesis by blast
  qed
 qed
qed

declare [[simp-depth-limit = 50]]

lemma Red-term-pres-no-match-it:
 ⟦ ∀ i < length ts. (ts ! i, ts' ! i) : Red-term ^^ (ns!i);
   size ts' = size ts; size ns = size ts;
   no-match ps (map dterm ts)⟧
  ⟹ no-match ps (map dterm ts')
proof(induct sum-list ns arbitrary: ts ns)
 case 0
 hence ∀ i < size ts. ns!i = 0 by simp
 with 0 show ?case by simp (metis nth-equalityI)
next
 case (Suc n)
 then have sum-list ns ≠ 0 by arith
 then obtain k l where k<size ts and [simp]: ns!k = Suc l
   by simp (metis ‹length ns = length ts› gr0-implies-Suc in-set-conv-nth)
 let ?ns = ns[k := l]
 have n = sum-list ?ns using ‹Suc n = sum-list ns› ‹k<size ts› ‹size ns = size
ts›
   by (simp add:sum-list-update)
 obtain t' where ts!k ⇒ t' (t', ts'!k) : Red-term^^l
   using Suc(3) ‹k<size ts› ‹size ns = size ts› ‹ns!k = Suc l›
   by (metis relpow-Suc-E2)
 then have 1: ∀ i<size(ts[k:=t']). (ts[k:=t']!i, ts'!i) : Red-term^^(?ns!i)
   using Suc(3) ‹k<size ts› ‹size ns = size ts›
```

by (*auto simp add*:*nth-list-update*)
note *nm1* = *Red-term-pres-no-match*[*OF* ‹*k*<*size ts*› ‹*ts*!*k* ⇒ *t'*› ‹*no-match ps* (*map dterm ts*)›]
show *?case* **by**(*rule Suc*(*1*)[*OF* ‹*n* = *sum-list ?ns*› *1* - - *nm1*])
(*simp-all add*: ‹*size ts'* = *size ts*› ‹*size ns* = *size ts*›)
qed

lemma *Red-term-pres-no-match-star*:
assumes ∀ *i* < *length*(*ts*::*tm list*). *ts* ! *i* ⇒∗ *ts'* ! *i* **and** *size ts'* = *size ts*
and *no-match ps* (*map dterm ts*)
shows *no-match ps* (*map dterm ts'*)
proof−
let *?P* = %*ns*. *size ns* = *size ts* ∧
(∀ *i* < *length ts*.(*ts*!*i*, *ts'*!*i*) : *Red-term*^^(*ns*!*i*))
have ∃ *ns*. *?P ns* **using** *assms*(*1*)
by(*subst Skolem-list-nth*[*symmetric*])
(*simp add*:*rtrancl-power*)
from *someI-ex*[*OF this*] **show** *?thesis*
by(*fast intro*: *Red-term-pres-no-match-it*[*OF* - *assms*(*2*) - *assms*(*3*)])
qed

lemma *not-pure-term*[*simp*]: ¬ *pure*(*term v*)
proof
assume *pure*(*term v*) **thus** *False*
by *cases*
qed

abbreviation *RedMLs* :: *tm list* ⇒ *tm list* ⇒ *bool* (**infix** ‹[⇒∗]› *50*) **where**
ss [⇒∗] *ts* ≡ *size ss* = *size ts* ∧ (∀ *i*<*size ss*. *ss*!*i* ⇒∗ *ts*!*i*)

fun *C-U-args* :: *tm* ⇒ *tm list* (‹*C*$_U$'-*args*›) **where**
C$_U$-*args*(*s* · *t*) = *C*$_U$-*args s* @ [*t*] |
C$_U$-*args*(*term*(*C*$_U$ *nm vs*)) = *map term* (*rev vs*) |
C$_U$-*args* - = []

lemma [*simp*]: *C*$_U$-*args*(*C nm* •• *ts*) = *ts*
by (*induct ts rule*:*rev-induct*) *auto*

lemma *redts-term-cong*: *v* ⇒∗ *v'* ⟹ *term v* ⇒∗ *term v'*
apply(*erule converse-rtrancl-induct*)
apply(*rule rtrancl-refl*)
apply(*fast intro*: *converse-rtrancl-into-rtrancl dest*: *ctxt-term*)
done

lemma *C-Red-term-ML*:
v ⇒ *v'* ⟹ *C-normal*$_{ML}$ *v* ⟹ *dterm*$_{ML}$ *v* = *C nm* •• *ts*
⟹ *dterm*$_{ML}$ *v'* = *C nm* •• *map dterm* (*C*$_U$-*args*(*term v'*)) ∧

C_U-*args*(*term v*) [⇒∗] C_U-*args*(*term v′*) ∧
ts = *map dterm* (C_U-*args*(*term v*)) **and**
(*vs*:: *ml list*) ⇒ *vs′* ⟹ *i* < *length vs* ⟹ *vs* ! *i* ⇒∗ *vs′* ! *i*
apply(*induct arbitrary*: *nm ts* **and** *i rule*:*Red-ml-Red-ml-list.inducts*)
apply(*simp-all add*:*Red-ml-list-length del*: *map-map*)
apply(*frule Red-ml-list-length*)
apply(*simp add*: *redts-term-cong rev-nth del*: *map-map*)
apply(*simp add*:*nth-Cons′ r-into-rtrancl del*: *map-map*)
apply(*simp add*:*nth-Cons′*)
done

lemma *C-normal-subterm*:
C-normal t ⟹ *dterm t* = *C nm* •• *ts* ⟹ *s* ∈ *set*(C_U-*args t*) ⟹ *C-normal s*
apply(*induct rule*: *C-normal.induct*)
apply *auto*
apply(*case-tac v*)
apply *auto*
done

lemma *C-normal-subterms*:
C-normal t ⟹ *dterm t* = *C nm* •• *ts* ⟹ *ts* = *map dterm* (C_U-*args t*)
apply(*induct rule*: *C-normal.induct*)
apply *auto*
apply(*case-tac v*)
apply *auto*
done

lemma *C-redt*: *t* ⇒ *t′* ⟹ *C-normal t* ⟹
C-normal t′ ∧ (*dterm t* = *C nm* •• *ts* ⟶
(∃ *ts′*. *ts′* = *map dterm* (C_U-*args t′*) ∧ *dterm t′* = *C nm* •• *ts′* ∧
C_U-*args t* [⇒∗] C_U-*args t′*))
apply(*induct arbitrary*: *ts nm rule*:*Red-term-hnf-induct*)
apply (*simp-all del*: *map-map*)
apply (*metis no-match-R-coincide rev-rev-ident*)
apply *rule*
apply (*metis C-normal-ML-inv*)
apply *clarify*
apply(*drule* (*2*) *C-Red-term-ML*)
apply *clarsimp*
apply *clarsimp*
apply (*metis insert-iff subsetD set-update-subset-insert*)
apply *clarsimp*
apply(*rule*)
apply (*metis insert-iff subsetD set-update-subset-insert*)
apply *rule*
apply *clarify*
apply(*drule bspec*, *assumption*)
apply *simp*

apply(*subst no-match.simps*)
apply(*subst* (*asm*) *no-match.simps*)
apply *clarsimp*
apply(*rename-tac j nm nm′ rs rs′*)
apply(*rule-tac x=j* **in** *exI*)
apply *simp*
apply(*case-tac i=j*)
apply(*erule-tac x=rs′* **in** *meta-allE*)
apply(*erule-tac x=nm′* **in** *meta-allE*)
apply (*clarsimp simp*: *all-set-conv-all-nth*)
apply(*metis C-normal-subterms Red-term-pres-no-match-star*)
apply (*auto simp*:*nth-list-update*)
done

lemma *C-redts*: $t \Rightarrow* t' \Longrightarrow$ *C-normal* $t \Longrightarrow$
C-normal $t' \wedge$ (*dterm* $t = C\ nm \bullet\bullet ts \longrightarrow$
$(\exists ts'.$ *dterm* $t' = C\ nm \bullet\bullet ts' \wedge C_U$*-args* $t\ [\Rightarrow*]\ C_U$*-args* $t' \wedge$
$ts' =$ *map dterm* (C_U*-args* t')))
apply(*induct arbitrary*: *nm ts rule*:*converse-rtrancl-induct*)
apply *simp*
using *tm-vector-cases*[*of t′*]
apply(*elim disjE*)
apply *clarsimp*
apply *clarsimp*
apply *clarsimp*
apply *clarsimp*
apply(*case-tac v*)
apply *simp*
apply *simp*
apply *simp*
apply *simp*
apply *clarsimp*
apply *simp*
apply *simp*
apply *simp*
apply *simp*
apply(*frule-tac nm=nm* **and** *ts=ts* **in** *C-redt*)
apply *assumption*
apply *clarify*
apply *rule*
apply *metis*
apply *clarify*
apply *simp*
apply *rule*
apply (*metis rtrancl-trans*)
done

lemma *no-match-preserved*:

```
  ∀ t∈set ts. C-normal t ⟹ ts [⇒*] ts'
   ⟹ no-match ps os ⟹ os = map dterm ts ⟹ no-match ps (map dterm ts')
proof(induct ps os arbitrary: ts ts' rule: no-match.induct)
  case (1 ps os)
  obtain i nm nm' ps' os' where a: ps!i = C nm •• ps' i < size ps
      i < size os os!i = C nm' •• os' nm=nm' ⟶ no-match ps' os'
    using 1(4) no-match.simps[of ps os] by fastforce
  note 1(5)[simp]
  have C-normal (ts ! i) using 1(2) ‹i < size os› by auto
  have ts!i ⇒* ts'!i using 1(3) ‹i < size os› by auto
  have dterm (ts ! i) = C nm' •• os' using ‹os!i = C nm' •• os'› ‹i < size os›
    by (simp add:nth-map)
  with C-redts [OF ‹ts!i ⇒* ts'!i› ‹C-normal (ts!i)›]
    C-normal-subterm[OF ‹C-normal (ts!i)›]
    C-normal-subterms[OF ‹C-normal (ts!i)›]
  obtain ss' rs rs' :: tm list where b: ∀ t∈set rs. C-normal t
    dterm (ts' ! i) = C nm' •• ss' length rs = length rs'
    ∀ i<length rs. rs ! i ⇒* rs' ! i ss' = map dterm rs' os' = map dterm rs
    by fastforce
  show ?case
    apply(subst no-match.simps)
    apply(rule-tac x=i in exI)
    using 1(2−5) a b
    apply clarsimp
    apply(rule 1(1)[of i nm' - nm' map dterm rs rs])
    apply simp-all
    done
qed

lemma Lam-Red-term-itE:
  (Λ t, t') : Red-term^^i ⟹ ∃ t''. t' = Λ t'' ∧ (t,t'') : Red-term^^i
apply(induct i arbitrary: t')apply simp
apply(erule relpow-Suc-E)
apply(erule Red-term.cases)
apply (simp-all)
apply blast+
done


lemma Red-term-it: (V x •• rs, r) : Red-term^^i
  ⟹ ∃ ts is. r = V x •• ts ∧ size ts = size rs & size is = size rs ∧
       (∀ j<size ts. (rs!j, ts!j) : Red-term^^(is!j) ∧ is!j <= i)
proof(induct i arbitrary:rs)
  case 0
  moreover
  have ∃ is. length is = length rs ∧
   (∀ j<size rs. (rs!j, rs!j) ∈ Red-term ^^ is!j ∧ is!j = 0) (is ∃ is. ?P is)
  proof
    show ?P(replicate (size rs) 0) by simp
```

```
  qed
  ultimately show ?case by auto
next
  case (Suc i rs)
  from ‹(V x •• rs, r) ∈ Red-term ^^ Suc i›
  obtain r' where r': V x •• rs ⇒ r' and (r',r) ∈ Red-term ^^ i
    by (metis relpow-Suc-D2)
  from r' have ∃k<size rs. ∃s. rs!k ⇒ s ∧ r' = V x •• rs[k:=s]
  proof(induct rs arbitrary: r' rule:rev-induct)
    case Nil thus ?case by(fastforce elim: Red-term.cases)
  next
    case (snoc r rs)
    hence (V x •• rs) • r ⇒ r' by simp
    thus ?case
    proof(cases rule:Red-term.cases)
      case (ctxt-At1 s')
     then obtain k s'' where aux: k<length rs rs ! k ⇒ s'' s' = V x •• rs[k := s'']
        using snoc(1) by force
      show ?thesis (is ∃k < ?n. ∃s. ?P k s)
      proof−
        have k<?n ∧ ?P k s'' using ctxt-At1 aux
        by (simp add:nth-append) (metis last-snoc butlast-snoc list-update-append1)
        thus ?thesis by blast
      qed
    next
      case (ctxt-At2 t')
      show ?thesis (is ∃k < ?n. ∃s. ?P k s)
      proof−
        have size rs<?n ∧ ?P (size rs) t' using ctxt-At2 by simp
        thus ?thesis by blast
      qed
    qed
  qed
  then obtain k s where k<size rs rs!k ⇒ s and [simp]: r' = V x •• rs[k:=s] by
metis
  from Suc(1)[of rs[k:=s]] ‹(r',r) ∈ Red-term ^^ i›
  show ?case using ‹k<size rs› ‹rs!k ⇒ s›
    apply auto
    apply(rule-tac x=is[k := Suc(is!k)] in exI)
    apply (auto simp:nth-list-update)
    apply(erule-tac x=k in allE)
    apply auto
    apply (metis relpow-Suc-I2 relpow.simps(2))
    done
qed

lemma C-Red-term-it: (C nm •• rs, r) : Red-term^^i
  ⟹ ∃ts is. r = C nm •• ts ∧ size ts = size rs ∧ size is = size rs ∧
       (∀j<size ts. (rs!j, ts!j) ∈ Red-term^^(is!j) ∧ is!j ≤ i)
```

```
proof(induct i arbitrary:rs)
  case 0
  moreover
  have ∃ is. length is = length rs ∧
   (∀ j<size rs. (rs!j, rs!j) ∈ Red-term ^^ is!j ∧ is!j = 0) (is ∃ is. ?P is)
  proof
    show ?P(replicate (size rs) 0) by simp
  qed
  ultimately show ?case by auto
next
  case (Suc i rs)
  from ‹(C nm ·· rs, r) ∈ Red-term ^^ Suc i›
  obtain r' where r': C nm ·· rs ⇒ r' and (r',r) ∈ Red-term ^^ i
    by (metis relpow-Suc-D2)
  from r' have ∃ k<size rs. ∃ s. rs!k ⇒ s ∧ r' = C nm ·· rs[k:=s]
  proof(induct rs arbitrary: r' rule:rev-induct)
    case Nil thus ?case by(fastforce elim: Red-term.cases)
  next
    case (snoc r rs)
    hence (C nm ·· rs) · r ⇒ r' by simp
    thus ?case
    proof(cases rule:Red-term.cases)
      case (ctxt-At1 s')
      then obtain k s'' where aux: k<length rs rs ! k ⇒ s'' s' = C nm ·· rs[k :=
s'']
        using snoc(1) by force
      show ?thesis (is ∃ k < ?n. ∃ s. ?P k s)
      proof−
        have k<?n ∧ ?P k s'' using ctxt-At1 aux
        by (simp add:nth-append) (metis last-snoc butlast-snoc list-update-append1)
        thus ?thesis by blast
      qed
    next
      case (ctxt-At2 t')
      show ?thesis (is ∃ k < ?n. ∃ s. ?P k s)
      proof−
        have size rs<?n ∧ ?P (size rs) t' using ctxt-At2 by simp
        thus ?thesis by blast
      qed
    qed
  qed
  then obtain k s where k<size rs rs!k ⇒ s and [simp]: r' = C nm ·· rs[k:=s]
by metis
  from Suc(1)[of rs[k:=s]] ‹(r',r) ∈ Red-term ^^ i›
  show ?case using ‹k<size rs› ‹rs!k ⇒ s›
    apply auto
    apply(rule-tac x=is[k := Suc(is!k)] in exI)
    apply (auto simp:nth-list-update)
    apply(erule-tac x=k in allE)
```

```
    apply auto
    apply (metis relpow-Suc-I2 relpow.simps(2))
    done
qed


lemma pure-At[simp]: pure(s • t) ⟷ pure s ∧ pure t
by(fastforce elim: pure.cases)

lemma pure-foldl-At[simp]: pure(s •• ts) ⟷ pure s ∧ (∀ t∈set ts. pure t)
by(induct ts arbitrary: s) auto

lemma nbe-C-normal-ML:
  assumes term v ⇒* t' C-normal_ML v pure t' shows normal t'
proof −
  { fix t t' i v
    assume (t,t') : Red-term^^i
    hence t = term v ⟹ C-normal_ML v ⟹ pure t' ⟹ normal t'
    proof(induct i arbitrary: t t' v rule:less-induct)
    case (less k)
    show ?case
    proof (cases k)
      case 0 thus ?thesis using less by auto
    next
      case (Suc i)
      then obtain i' s where t ⇒ s and red: (s,t') : Red-term^^i' and [arith]: i'
<= i
        by (metis eq-imp-le less(5) Suc relpow-Suc-D2)
      hence term v ⇒ s using Suc less by simp
      thus ?thesis
      proof cases
        case (term-C nm vs)
        with less have 0:no-match-compR nm vs by auto
        let ?n = size vs
        have 1: (C nm •• map term (rev vs),t') : Red-term^^i'
          using term-C ‹(s,t') : Red-term^^i'› by simp
        with C-Red-term-it[OF 1]
        obtain ts ks where [simp]: t' = C nm •• ts
          and sz: size ts = ?n ∧ size ks = ?n ∧
          (∀ i<?n. (term((rev vs)!i), ts!i) : Red-term^^(ks!i) ∧ ks ! i ≤ i')
          by(auto cong:conj-cong)
        have pure-ts: ∀ t∈set ts. pure t using ‹pure t'› by simp
        { fix i assume i<size vs
          moreover hence (term((rev vs)!i), ts!i) : Red-term^^(ks!i) by(metis sz)
          ultimately have normal (ts!i)
            apply −
            apply(rule less(1))
            prefer 5 apply assumption
            using sz Suc apply fastforce
```

```
      apply(rule refl)
      using less term-C
      apply(auto)
      apply (metis in-set-conv-nth length-rev set-rev)
      apply (metis in-set-conv-nth pure-ts sz)
      done
  } note 2 = this
  have 3: no-match-R nm (map dterm (map term (rev vs)))
   apply(subst map-dterm-term)
   apply(rule no-match-R-coincide) using 0 by simp
  have 4: map term (rev vs) [⇒*] ts
  proof −
   have (C nm •• map term (rev vs),t'): Red-term^^i'
    using red term-C by auto
   from C-Red-term-it[OF this] obtain ts' is where t' = C nm •• ts'
    and length ts' = ?n ∧ length is =?n ∧
     (∀ j< ?n. (map term (rev vs) ! j, ts' ! j) ∈ Red-term ^^ is ! j ∧ is ! j ≤
i')
    using sz by auto
   from ‹t' = C nm •• ts'› ‹t' = C nm •• ts› have ts = ts' by simp
   show ?thesis using sz by (auto simp: rtrancl-is-UN-relpow)
  qed
  have 5: ∀ t∈set(map term vs). C-normal t
   using less term-C by auto
  have no-match-R nm (map dterm ts)
   apply auto
   apply(subgoal-tac no-match aa (map dterm (map term (rev vs))))
   prefer 2
   using 3 apply blast
    using 4 5 no-match-preserved[OF - - - refl, of map term (rev vs) ts] by
simp
  hence 6: no-match-R nm ts by(metis map-dterm-pure[OF pure-ts])
  then show normal t'
   apply(simp)
   apply(rule normal.intros(3))
   using 2 sz apply(fastforce simp:set-conv-nth)
   apply auto
   apply(subgoal-tac no-match aa (take (size aa) ts))
   apply (metis no-match)
   apply(fastforce intro:no-match-take)
   done
 next
  case (term-V x vs)
  let ?n = size vs
  have 1: (V x •• map term (rev vs),t') : Red-term^^i'
   using term-V ‹(s,t') : Red-term^^i'› by simp
  with Red-term-it[OF 1] obtain ts is where [simp]: t' = V x •• ts
   and 2: length ts = ?n ∧
    length is = ?n ∧ (∀ j<?n. (term (rev vs ! j), ts ! j) ∈ Red-term ^^ is ! j ∧
```

```
        is ! j ≤ i')
      by (auto cong:conj-cong)
    have ∀ j<?n. normal(ts!j)
    proof(clarify)
      fix j assume 0: j < ?n
      then have is!j < k using ‹k=Suc i› 2 by auto
      have red: (term (rev vs ! j), ts ! j) ∈ Red-term ^^ is ! j using ‹j < ?n› 2
by auto
      have pure: pure (ts ! j) using ‹pure t'› 0 2 by auto
      have Cnm: C-normal_ML (rev vs ! j) using less term-V
        by simp (metis 0 in-set-conv-nth length-rev set-rev)
      from less(1)[OF ‹is!j < k› refl Cnm pure red] show normal(ts!j) .
    qed
    note 3=this
    show ?thesis by simp (metis normal.intros(1) in-set-conv-nth 2 3)
  next
    case (term-Clo f vs n)
    let ?u = apply (lift 0 (Clo f vs n)) (V_U 0 [])
    from term-Clo ‹(s,t') : Red-term^^i'›
    obtain t'' where [simp]: t' = Λ t'' and 1: (term ?u, t'') : Red-term^^i'
      by(metis Lam-Red-term-itE)
    have i' < k using ‹k = Suc i› by arith
    have pure t'' using ‹pure t'› by simp
    have C-normal_ML ?u using less term-Clo by(simp)
    from less(1)[OF ‹i' < k› refl ‹C-normal_ML ?u› ‹pure t''› 1]
    show ?thesis by(simp add:normal.intros)
  next
    case (ctxt-term u')
    have i' < k using ‹k = Suc i› by arith
      have C-normal_ML u' by (rule C-normal-ML-inv) (insert less ctxt-term,
simp-all)
    have (term u', t') ∈ Red-term ^^ i' using red ctxt-term by auto
   from less(1)[OF ‹i' < k› refl ‹C-normal_ML u'› ‹pure t'› this] show ?thesis
.
  qed
 qed
qed
}
thus ?thesis using assms(2−) rtrancl-imp-relpow[OF assms(1)] by blast
qed

lemma C-normal-ML-compile:
  pure t ⟹ ∀ i. C-normal_ML(σ i) ⟹ C-normal_ML (compile t σ)
by(induct t arbitrary: σ) (simp-all add: C-normal-ML-lift-ML)

corollary nbe-normal:
  pure t ⟹ term(comp-fixed t) ⇒* t' ⟹ pure t' ⟹ normal t'
apply(erule nbe-C-normal-ML)
apply(simp add: C-normal-ML-compile)
```

apply *assumption*
done

7 Refinements

We ensure that all occurrences of C_U *nm vs* satisfy the invariant *size vs* = *arity nm*.

A constructor value:

fun $C_U s$ :: *ml* ⇒ *bool* **where**
$C_U s(C_U$ *nm vs*$)$ = (*size vs* = *arity nm* ∧ (∀ *v*∈*set vs*. $C_U s$ *v*)) |
$C_U s$ - = *False*

lemma *size-foldl-At*: *size*(*C nm* •• *ts*) = *size ts* + *sum-list*(*map size ts*)
by(*induct ts rule*:*rev-induct*) *auto*

lemma *termination-linpats*:
i < *length ts* ⟹ *ts*!*i* = *C nm* •• *ts*′
⟹ *length ts*′ + *sum-list* (*map size ts*′) < *length ts* + *sum-list* (*map size ts*)
apply(*subgoal-tac C nm* •• *ts*′ : *set ts*)
prefer *2* **apply** (*metis in-set-conv-nth*)
apply(*drule sum-list-map-remove1*[*of* - - *size*])
apply(*simp add*:*size-foldl-At*)
apply (*metis gr-implies-not0 length-0-conv*)
done

Linear patterns:

function *linpats* :: *tm list* ⇒ *bool* **where**
linpats ts ⟷
(∀ *i*<*size ts*. (∃ *x*. *ts*!*i* = *V x*) ∨
(∃ *nm ts*′. *ts*!*i* = *C nm* •• *ts*′ ∧ *arity nm* = *size ts*′ ∧ *linpats ts*′)) ∧
(∀ *i*<*size ts*. ∀ *j*<*size ts*. *i*≠*j* ⟶ *fv*(*ts*!*i*) ∩ *fv*(*ts*!*j*) = {})
by *pat-completeness auto*
termination
apply(*relation measure*(%*ts*. *size ts* + (*SUM t*<−*ts*. *size t*)))
apply (*auto simp*:*termination-linpats*)
done

declare *linpats.simps*[*simp del*]

lemma *eq-lists-iff-eq-nth*:
size xs = *size ys* ⟹ (*xs*=*ys*) = (∀ *i*<*size xs*. *xs*!*i* = *ys*!*i*)
by (*metis nth-equalityI*)

lemma *pattern-subst-ML-coincidence*:
pattern t ⟹ ∀ *i*∈*fv t*. *σ i* = *σ*′ *i*
⟹ *subst-ML σ* (*comp-pat t*) = *subst-ML σ*′ (*comp-pat t*)

by(*induct pred*:*pattern*) *auto*

lemma *linpats-pattern*: *linpats ts* $\Longrightarrow$ *patterns ts*
proof(*induct ts rule*:*linpats.induct*)
case (*1 ts*)
show *?case*
proof
fix *t* **assume** *t* : *set ts*
then obtain *i* **where** *i* < *size ts* **and** [*simp*]: *t* = *ts*!*i*
by (*auto simp*: *in-set-conv-nth*)
hence (∃ *x*. *t* = *V x*) ∨ (∃ *nm ts*′. *t* = *C nm* ·· *ts*′ ∧ *arity nm* = *size ts*′ & *linpats ts*′)
(**is** *?V* | *?C*)
using *1*(*2*) **by**(*simp add*:*linpats.simps*[*of ts*])
thus *pattern t*
proof
assume *?V* **thus** *?thesis* **by**(*auto simp*:*pat-V*)
next
assume *?C* **thus** *?thesis* **using** *1*(*1*) ‹*i* < *size ts*›
by *auto* (*metis pat-C*)
qed
qed
qed

lemma *no-match-ML-swap-rev*:
length ps = *length vs* $\Longrightarrow$ *no-match*$_{ML}$ *ps* (*rev vs*) $\Longrightarrow$ *no-match*$_{ML}$ (*rev ps*) *vs*
apply(*clarsimp simp*: *no-match-ML.simps*[*of ps*] *no-match-ML.simps*[*of* - *vs*])
apply(*rule-tac x*=*size ps* − *i* − *1* **in** *exI*)
apply (*fastforce simp*:*rev-nth*)
done

lemma *no-match-ML-aux*:
∀ *v* ∈ *set cvs*. *C*$_U$ *s v* $\Longrightarrow$ *linpats ps* $\Longrightarrow$ *size ps* = *size cvs* $\Longrightarrow$
∀ *σ*. *map* (*subst*$_{ML}$ *σ*) (*map comp-pat ps*) ≠ *cvs* $\Longrightarrow$
no-match$_{ML}$ (*map comp-pat ps*) *cvs*
apply(*induct ps arbitrary*: *cvs rule*:*linpats.induct*)
apply(*frule linpats-pattern*)
apply(*subst* (*asm*) *linpats.simps*) **back**
apply *auto*
apply(*case-tac* ∀ *i*<*size ts*. ∃ *σ*. *subst*$_{ML}$ *σ* (*comp-pat* (*ts*!*i*)) = *cvs*!*i*)
apply(*clarsimp simp*:*Skolem-list-nth*)
apply(*rename-tac σs*)
apply(*erule-tac x*=%*x*. (*σs*!(*THE i*. *i*<*size ts* & *x* : *fv*(*ts*!*i*)))*x* **in** *allE*)
apply(*clarsimp simp*:*eq-lists-iff-eq-nth*)
apply(*rotate-tac* −*3*)
apply(*erule-tac x*=*i* **in** *allE*)
apply *simp*
apply(*rotate-tac* −*1*)
apply(*drule sym*)

apply *simp*
apply(*erule contrapos-np*)
apply(*rule pattern-subst-ML-coincidence*)
apply (*metis in-set-conv-nth*)
apply *clarsimp*
apply(*rule-tac a=i* **in** *theI2*)
apply *simp*
apply (*metis disjoint-iff-not-equal*)
apply (*metis disjoint-iff-not-equal*)
apply *clarsimp*
apply(*subst no-match-ML.simps*)
apply(*rule-tac x=size ts − i − 1* **in** *exI*)
apply *simp*
apply *rule*
apply *simp*
apply(*subgoal-tac* ¬(∃ *x*. *ts*!*i* = *V x*))
prefer *2*
apply *fastforce*
apply(*subgoal-tac* ∃ *nm ts*′. *ts*!*i* = *C nm* •• *ts*′ & *size ts*′ = *arity nm* & *linpats ts*′)
prefer *2*
apply *fastforce*
apply *clarsimp*
apply(*rule-tac x=nm* **in** *exI*)
apply(*subgoal-tac* ∃ *nm*′ *vs*′. *cvs*!*i* = C_U *nm*′ *vs*′ & *size vs*′ = *arity nm*′ & (∀ *v*′ ∈ *set vs*′. C_U *s v*′))
prefer *2*
apply(*drule-tac x=cvs*!*i* **in** *bspec*)
apply *simp*
apply(*case-tac cvs*!*i*)
apply *simp-all*
apply (*clarsimp simp*:*rev-nth rev-map*[*symmetric*])
apply(*erule-tac x=i* **in** *meta-allE*)
apply(*erule-tac x=nm*′ **in** *meta-allE*)
apply(*erule-tac x=ts*′ **in** *meta-allE*)
apply(*erule-tac x=rev vs*′ **in** *meta-allE*)
apply *simp*
apply(*subgoal-tac no-match*$_{ML}$ (*map comp-pat ts*′) (*rev vs*′))
apply(*rule no-match-ML-swap-rev*)
apply *simp*
apply *assumption*
apply(*erule-tac meta-mp*)
apply (*metis rev-rev-ident*)
done

References

[1] Klaus Aehlig, Florian Haftmann, and Tobias Nipkow. A compiled implementation of normalization by evaluation. In Ait Mohamed, Munoz, and Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2008)*, volume 5170 of *LNCS*, pages 39–54. Springer, 2008. www.in.tum.de/~nipkow/pubs/tphols08.html.