

Normalization by Evaluation

Klaus Aehlig and Tobias Nipkow

December 12, 2009

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

types $vname = nat$
 $ml-vname = nat$

types $cname = int$

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 $tm = C\ cname\ | V\ vname\ | \Lambda\ tm\ | At\ tm\ tm$ (**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  $\Rightarrow$  bool where
  pure(C nm) |
  pure(V x) |
  Lam: pure t  $\Rightarrow$  pure( $\Lambda$  t) |
  pure s  $\Rightarrow$  pure t  $\Rightarrow$  pure(s·t)

```

```

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

```

```

lemma pure-Lam[simp]: pure( $\Lambda$  t) = pure t

```

```

proof
  assume pure( $\Lambda$  t) thus pure t
  proof cases qed auto
next
  assume pure t thus pure( $\Lambda$  t) by(rule Lam)
qed

```

Closed terms w.r.t. ML variables:

```

fun closed-ML :: nat  $\Rightarrow$  ml  $\Rightarrow$  bool (closedML) where
  closedML i (CML nm) = True |
  closedML i (VML X) = (X < i) |
  closedML i (AML v vs) = (closedML i v  $\wedge$  ( $\forall v \in$  set vs. closedML i v)) |
  closedML i (LamML v) = closedML (i+1) v |
  closedML i (CU nm vs) = ( $\forall v \in$  set vs. closedML i v) |
  closedML i (VU nm vs) = ( $\forall v \in$  set vs. closedML i v) |
  closedML i (Clo f vs n) = (closedML i f  $\wedge$  ( $\forall v \in$  set vs. closedML i v)) |
  closedML i (apply v w) = (closedML i v  $\wedge$  closedML i w)

```

```

fun closed-tm-ML :: nat  $\Rightarrow$  tm  $\Rightarrow$  bool (closedML) where
  closed-tm-ML i (r·s) = (closed-tm-ML i r  $\wedge$  closed-tm-ML i s) |
  closed-tm-ML i ( $\Lambda$  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  $\Rightarrow$  ml-vname set (fvML) where
  fvML (CML nm) = {} |
  fvML (VML X) = {X} |
  fvML (AML v vs) = fvML v  $\cup$  ( $\bigcup v \in$  set vs. fvML v) |

```

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

primrec $fv :: tm \Rightarrow vname\ set$ **where**

$$\begin{aligned}
fv (C\ nm) &= \{\} \mid \\
fv (V\ X) &= \{X\} \mid \\
fv (s \cdot t) &= fv\ s \cup fv\ t \mid \\
fv (\Lambda\ t) &= \{X. Suc\ X : fv\ t\}
\end{aligned}$$

1.1 Iterated Term Application

abbreviation $foldl\text{-}At$ (**infix** $\cdot\cdot$ 90) **where**

$$t \cdot\cdot ts \equiv foldl\ (op\ \cdot)\ t\ ts$$

Auxiliary measure function:

primrec $depth\text{-}At :: tm \Rightarrow nat$

where

$$\begin{aligned}
depth\text{-}At(C\ nm) &= 0 \\
\mid\ depth\text{-}At(V\ x) &= 0 \\
\mid\ depth\text{-}At(s \cdot t) &= depth\text{-}At\ s + 1 \\
\mid\ depth\text{-}At(\Lambda\ t) &= 0 \\
\mid\ depth\text{-}At(term\ v) &= 0
\end{aligned}$$

lemma $depth\text{-}At\text{-}foldl$:

$$depth\text{-}At(s \cdot\cdot ts) = depth\text{-}At\ s + size\ ts$$

by (*induct* ts *arbitrary: s*) *simp-all*

lemma $foldl\text{-}At\text{-}eq\text{-}lemma$: $size\ ts = size\ ts' \implies$

$$s \cdot\cdot ts = s' \cdot\cdot ts' \iff s = s' \wedge ts = ts'$$

by (*induct* *arbitrary: s s'* *rule:list-induct2*) *simp-all*

lemma $foldl\text{-}At\text{-}eq\text{-}length$:

$$s \cdot\cdot ts = s \cdot\cdot ts' \implies length\ ts = length\ ts'$$

apply(*subgoal-tac* $depth\text{-}At(s \cdot\cdot ts) = depth\text{-}At(s \cdot\cdot ts')$)

apply(*erule thin-rl*)

apply (*simp add:depth-At-foldl*)

apply *simp*

done

lemma $foldl\text{-}At\text{-}eq[simp]$: $s \cdot\cdot ts = s \cdot\cdot ts' \iff ts = ts'$

apply(*rule*)

prefer 2 **apply** *simp*

apply(*blast dest:foldl-At-eq-lemma foldl-At-eq-length*)

done

lemma $term\text{-}eq\text{-}foldl\text{-}At[simp]$:

$$term\ v = t \cdot\cdot ts \iff t = term\ v \wedge ts = []$$

```

by (induct ts arbitrary:t) auto

lemma At-eq-foldl-At[simp]:
   $r \cdot s = t \cdot ts \iff$ 
  (if ts=[] then  $t = r \cdot s$  else  $s = \text{last } ts \wedge r = t \cdot \text{butlast } ts$ )
apply (induct ts arbitrary:t)
apply fastsimp
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]:
   $t \cdot ts = r \cdot s \iff$ 
  (if ts=[] then  $t = r \cdot s$  else  $s = \text{last } ts \wedge r = t \cdot \text{butlast } ts$ )
by(metis At-eq-foldl-At)

```

```

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

```

```

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

```

```

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

```

```

fun atomic-tm :: tm  $\Rightarrow$  bool where
  atomic-tm(s · t) = False |
  atomic-tm(-) = True

```

```

fun head-tm where
  head-tm(s · 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]*: $head-tm(s \cdot\cdot ts) = head-tm\ s$
by(*induct ts arbitrary: s*) *auto*

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

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

declare
 $tm-eq-iff[of\ h \cdot\cdot ts, standard, simp]$
 $tm-eq-iff[of\ -\ h \cdot\cdot ts, standard, simp]$

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*: $t = head-tm\ t \cdot\cdot args-tm\ t$
by (*subst tm-eq-iff*) (*auto simp: atomic-tm-head-tm head-tm-idem args-tm-head-tm*)

lemma *tm-vector-cases*:
 $(\exists n\ ts. t = V\ n \cdot\cdot ts) \vee$
 $(\exists nm\ ts. t = C\ nm \cdot\cdot ts) \vee$
 $(\exists t'\ ts. t = \Lambda\ t' \cdot\cdot ts) \vee$
 $(\exists v\ ts. t = term\ v \cdot\cdot ts)$
apply(*induct t*)
apply *simp-all*
apply (*metis append-is-Nil-conv args-tm.simps(2) args-tm.simps(5) args-tm-foldl-At*
butlast.simps(2) butlast-append butlast-snoc eta-head-args head-tm.simps(2) head-tm.simps(4)
head-tm.simps(5) head-tm-foldl-At last-snoc list.simps(2) not-Cons-self rotate1-is-Nil-conv
rotate-simps self-append-conv self-append-conv2 tm.simps(12) tm.simps(22))
done

lemma *fv-head-C[simp]*: $fv\ (t \cdot\cdot ts) = fv\ t \cup (\bigcup_{t \in set\ ts} fv\ t)$
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* $\lambda i\ v.\ P\ v$, *standard*]

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 ($s.t$) = $(lift\ i\ s).(lift\ i\ t)$ |
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### $\equiv \lambda i.$ *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### $\equiv \lambda i.$ *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 σ ($C\ nm$) = $C\ nm$
| *subst* σ ($V\ x$) = $\sigma\ x$
| *subst* σ ($\Lambda\ t$) = $\Lambda (subst\ (V\ 0\ ##\ \sigma)\ t)$
| *subst* σ ($s.t$) = $(subst\ \sigma\ s) \cdot (subst\ \sigma\ t)$

fun *subst-ML* :: (*nat* \Rightarrow *ml*) \Rightarrow *ml* \Rightarrow *ml* (*subst_{ML}*) **where**

subst_{ML} σ ($C_{ML} nm$) = $C_{ML} nm$ |
subst_{ML} σ ($V_{ML} X$) = $\sigma\ X$ |

$subst_{ML} \sigma (A_{ML} v vs) = A_{ML} (subst_{ML} \sigma v) (map (subst_{ML} \sigma) vs) \mid$
 $subst_{ML} \sigma (Lam_{ML} v) = Lam_{ML} (subst_{ML} (V_{ML} 0 \#\#\sigma) v) \mid$
 $subst_{ML} \sigma (C_U nm vs) = C_U nm (map (subst_{ML} \sigma) vs) \mid$
 $subst_{ML} \sigma (V_U x vs) = V_U x (map (subst_{ML} \sigma) vs) \mid$
 $subst_{ML} \sigma (Clo v vs n) = Clo (subst_{ML} \sigma v) (map (subst_{ML} \sigma) vs) n \mid$
 $subst_{ML} \sigma (apply u v) = apply (subst_{ML} \sigma u) (subst_{ML} \sigma v)$

abbreviation

$subst-decr :: nat \Rightarrow tm \Rightarrow nat \Rightarrow tm$ **where**
 $subst-decr k t \equiv \lambda n. \text{if } n < k \text{ then } V n \text{ else if } n = k \text{ then } t \text{ else } V(n - 1)$

abbreviation

$subst-decr-ML :: nat \Rightarrow ml \Rightarrow nat \Rightarrow ml$ **where**
 $subst-decr-ML k v \equiv \lambda n. \text{if } n < k \text{ then } V_{ML} n \text{ else if } n = k \text{ then } v \text{ else } V_{ML}(n - 1)$

abbreviation

$subst1 :: tm \Rightarrow tm \Rightarrow nat \Rightarrow tm$ $((-/[-'/-]) [300, 0, 0] 300)$ **where**
 $s[t/k] \equiv subst (subst-decr k t) s$

abbreviation

$subst1-ML :: ml \Rightarrow ml \Rightarrow nat \Rightarrow ml$ $((-/[-'/-]) [300, 0, 0] 300)$ **where**
 $u[v/k] \equiv subst_{ML} (subst-decr-ML k v) u$

lemma *apply-cons[simp]*:

$(t\#\#\sigma) i = (\text{if } i = 0 \text{ then } t::tm \text{ else lift } 0 (\sigma(i - 1)))$

by(*simp add: cons-def split:nat.split*)

lemma *apply-cons-ML[simp]*:

$(v\#\#\sigma) i = (\text{if } i = 0 \text{ then } v::ml \text{ else lift}_{ML} 0 (\sigma(i - 1)))$

by(*simp add: cons-ML-def split:nat.split*)

lemma *lift-foldl-At[simp]*:

$lift k (s \cdot\cdot ts) = (lift k s) \cdot\cdot (map (lift k) ts)$

by(*induct ts arbitrary:s*) *simp-all*

lemma *lift-lift-ml: fixes v :: ml shows*

$i < k+1 \implies lift (Suc k) (lift i v) = lift i (lift k v)$

by(*induct i v rule:lift-ml.induct*)

simp-all

lemma *lift-lift-tm: fixes t :: tm shows*

$i < k+1 \implies lift (Suc k) (lift i t) = lift i (lift k t)$

by(*induct t arbitrary: i rule:lift-tm.induct*)(*simp-all add:lift-lift-ml*)

lemma *lift-lift-ML:*

$i < k+1 \implies lift_{ML} (Suc k) (lift_{ML} i v) = lift_{ML} i (lift_{ML} k v)$

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]:

$V_{ML} 0 \#\# subst_decr_ML k v = subst_decr_ML (Suc k) (lift_{ML} 0 v)$
by(rule ext)(simp add:cons-ML-def split:nat.split)

lemma *shift-subst-decr*[simp]:

$V 0 \#\# subst_decr k t = subst_decr (Suc k) (lift 0 t)$
by(rule ext)(simp add:cons-def split:nat.split)

lemma *lift-comp-subst-decr*[simp]:

$lift 0 o subst_decr_ML k v = subst_decr_ML k (lift 0 v)$
by(rule ext) simp

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

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

lemma *lift-Pure-tms*[simp]: $pure t \implies pure (lift k t)$
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 σv 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*:

$\forall v k. lift_ml 0 (u[v/k]) = (lift_ml 0 u)[lift 0 v/k]$
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*:

$lift_{ML} k (subst_{ML} \sigma v) =$
 $subst_{ML} (\lambda i. \text{if } i < k \text{ then } lift_{ML} k (\sigma i) \text{ else if } i = k \text{ then } V_{ML} k \text{ else } lift_{ML} k$

```

( $\sigma(i - 1)$ ) ( $\text{lift}_{ML} k v$ )
  (is - =  $\text{subst}_{ML} (?insrt k \sigma) (\text{lift}_{ML} k v)$ )
apply ( $\text{induct } k v \text{ arbitrary: } \sigma k \text{ rule: lift-ML.induct}$ )
apply ( $\text{simp-all add: o-assoc lift-o-shift}$ )
apply ( $\text{subgoal-tac } V_{ML} 0 \#\#\ ?insrt k \sigma = ?insrt (Suc k) (V_{ML} 0 \#\#\ \sigma)$ )
  apply  $\text{simp}$ 
apply ( $\text{simp add: expand-fun-eq lift-lift-ML cons-ML-def split:nat.split}$ )
done

```

corollary *subst-cons-lift*:

```

 $\text{subst}_{ML} (V_{ML} 0 \#\#\ \sigma) o (\text{lift}_{ML} 0) = \text{lift}_{ML} 0 o (\text{subst}_{ML} \sigma)$ 
apply ( $\text{rule ext}$ )
apply ( $\text{simp add: lift-ML-subst-ML}$ )
apply ( $\text{subgoal-tac } (V_{ML} 0 \#\#\ \sigma) = (\lambda i. \text{if } i = 0 \text{ then } V_{ML} 0 \text{ else } \text{lift}_{ML} 0 (\sigma (i - 1)))$ )
  apply  $\text{simp}$ 
apply ( $\text{rule ext, simp}$ )
done

```

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

lemma *subst-ML-id*:

```

 $\text{closed}_{ML} k v \implies \forall i < k. \sigma i = V_{ML} i \implies \text{subst}_{ML} \sigma v = v$ 
apply ( $\text{induct } \sigma v \text{ arbitrary: } k \text{ rule: subst-ML.induct}$ )
apply ( $\text{auto simp add: list-eq-iff-nth-eq}$ )
  apply ( $\text{simp add: Ball-def}$ )
  apply ( $\text{erule-tac } x=vs!i \text{ in meta-allE}$ )
  apply ( $\text{erule-tac } x=k \text{ in meta-allE}$ )
  apply ( $\text{erule-tac } x=k \text{ in meta-allE}$ )
  apply  $\text{simp}$ 
  apply ( $\text{erule-tac } x=vs!i \text{ in meta-allE}$ )
  apply ( $\text{erule-tac } x=k \text{ in meta-allE}$ )
  apply  $\text{simp}$ 
  apply ( $\text{erule-tac } x=vs!i \text{ in meta-allE}$ )
  apply ( $\text{erule-tac } x=k \text{ in meta-allE}$ )
  apply  $\text{simp}$ 
  apply ( $\text{erule-tac } x=vs!i \text{ in meta-allE}$ )
  apply ( $\text{erule-tac } x=k \text{ in meta-allE}$ )
  apply  $\text{simp}$ 
done

```

corollary *subst-ML-id2[simp]*: $\text{closed}_{ML} 0 v \implies \text{subst}_{ML} \sigma v = v$
using *subst-ML-id[where k=0]* **by** *simp*

lemma *subst-ML-coincidence*:

```

 $\text{closed}_{ML} k v \implies \forall i < k. \sigma i = \sigma' i \implies \text{subst}_{ML} \sigma v = \text{subst}_{ML} \sigma' v$ 
by ( $\text{induct } \sigma v \text{ arbitrary: } k \sigma' \text{ rule: subst-ML.induct}$ ) auto

```

lemma *subst-ML-comp*:

$subst_{ML} \sigma (subst_{ML} \sigma' v) = subst_{ML} (subst_{ML} \sigma \circ \sigma') v$
apply (*induct* $\sigma' v$ *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*:

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

lemma *closed-tm-ML-foldl-At*:

$closed_{ML} k (t \cdot\cdot ts) \iff closed_{ML} k t \wedge (\forall t \in set\ ts. closed_{ML} k t)$
by (*induct* *ts* *arbitrary*: *t*) *simp-all*

lemma *closed-ML-lift[*simp*]*:

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

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

lemma *closed-ML-subst-ML*:

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

lemma *closed-ML-subst-ML2*:

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

lemma *subst-foldl[*simp*]*:

$subst\ \sigma (s \cdot\cdot ts) = (subst\ \sigma\ s) \cdot\cdot (map\ (subst\ \sigma)\ ts)$
by (*induct* *ts* *arbitrary*: s) *auto*

lemma *subst-V*: $pure\ t \implies subst\ V\ t = t$

by (*induct* *pred*:*pure*) *simp-all*

lemma *lift-subst-aux*:

$pure\ t \implies \forall i < k. \sigma' i = lift\ k (\sigma\ i) \implies$
 $\forall i \geq k. \sigma'(Suc\ i) = lift\ k (\sigma\ i) \implies$
 $\sigma' k = V\ k \implies lift\ k (subst\ \sigma\ t) = subst\ \sigma' (lift\ k\ t)$
apply (*induct* *arbitrary*: $\sigma\ \sigma' k$ *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  $\implies$  lift 0 (subst  $\sigma$  t) = subst (V 0 ##  $\sigma$ ) (lift 0 t)
by (simp add: lift-subst-aux lift-lift-ml)

```

lemma *subst-comp*:

```

  pure t  $\implies$   $\forall i$ . pure( $\sigma'$  i)  $\implies$ 
   $\sigma'' = (\lambda i$ . subst  $\sigma$  ( $\sigma'$  i))  $\implies$  subst  $\sigma$  (subst  $\sigma'$  t) = subst  $\sigma''$  t
apply(induct arbitrary: $\sigma$   $\sigma'$   $\sigma''$  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  $\equiv$   $\forall t \in \text{set } ts$ . pattern t |
  pat-V: pattern(V X) |
  pat-C: patterns ts  $\implies$  pattern(C nm .. ts)

```

```

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

```

```

lemma pattern-At'D12: pattern r  $\implies$  r = (s  $\cdot$  t)  $\implies$  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:split-if-asm)

```

(metis eta-head-args in-set-butlastD pattern.pat-C)
qed

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

lemma pattern-At-vecD: pattern($s \cdot\cdot ts$) \implies patterns ts
apply(induct ts rule:rev-induct)
apply simp
apply (fastsimp dest!:pattern-AtD12)
done

lemma pattern-At-decomp: pattern($s \cdot t$) $\implies \exists nm ss. s = C nm \cdot\cdot ss$
proof(induct s arbitrary: t)
case (At $s1 s2$) **show** ?case

using pattern-AtD12[OF At.premis] At(1)[of $s2$]
by clarsimp (metis append-is-Nil-conv butlast-snoc last-snoc not-Cons-self)
qed (auto elim!: pattern.cases split:split-if-asm)

2.2 Reduction of λ -terms

The source program:

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

inductive-set

Red-tm :: $(tm * tm) set$
and *red-tm* :: $[tm, tm] \Rightarrow bool$ (**infixl** $\rightarrow 50$)
where
 $s \rightarrow t \equiv (s, t) \in Red\text{-}tm$
— β -reduction
 $| (\Lambda t) \cdot s \rightarrow t[s/0]$
— η -expansion
 $| t \rightarrow \Lambda ((lift\ 0\ t) \cdot (V\ 0))$
— Rewriting
 $| (nm, ts, t) : R \implies (C\ nm) \cdot\cdot (map\ (subst\ \sigma)\ ts) \rightarrow subst\ \sigma\ t$
 $| t \rightarrow t' \implies \Lambda\ t \rightarrow \Lambda\ t'$
 $| s \rightarrow s' \implies s \cdot t \rightarrow s' \cdot t$
 $| t \rightarrow t' \implies s \cdot t \rightarrow s \cdot t'$

abbreviation

reds-tm :: $[tm, tm] \Rightarrow bool$ (**infixl** $\rightarrow^* 50$) **where**
 $s \rightarrow^* t \equiv (s, t) \in Red\text{-}tm\ \hat{*}$

inductive-set

Reds-tm-list :: $(tm\ list * tm\ list)\ set$

```

and reds-tm-list :: [tm list, tm list] ⇒ bool (infixl →* 50)
where
  ss →* ts ≡ (ss, ts) ∈ Reds-tm-list
| [] →* []
| ts →* ts' ⇒ t →* t' ⇒ t#ts →* t'#ts'

declare Reds-tm-list.intros[simp]

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

lemma Red-tm-append: rs →* rs' ⇒ ts →* ts' ⇒ rs @ ts →* rs' @ ts'
by(induct set: Reds-tm-list) auto

lemma Red-tm-rev: ts →* ts' ⇒ rev ts →* rev ts'
by(induct set: Reds-tm-list) (auto simp:Red-tm-append)

lemma red-Lam[simp]: t →* t' ⇒ Λ t →* 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 →* t' ⇒ t · s →* t' · s
apply(induct rule:rtrancl-induct)
apply(simp-all)
apply(blast intro: rtrancl-into-rtrancl Red-tm.intros)
done

lemma red-At2[simp]: t →* t' ⇒ s · t →* s · 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 →* ts' ⇒ s →* s' ⇒ s .. ts →* s' .. 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  $\implies$  rev ps ! i =  $C_U$  nm vs
 $\implies$  listsum (map size vs) < listsum (map size ps)
apply(subgoal-tac  $C_U$  nm vs : set ps)
apply(drule listsum-map-remove1[of - - size])
apply (simp add:list-size-conv-listsum)
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-matchML ps os =
  ( $\exists i < \min$  (size os) (size ps).
     $\exists nm\ nm'\ vs\ vs'. (rev\ ps)!i = C_U\ nm\ vs \wedge (rev\ os)!i = C_U\ nm'\ vs' \wedge$ 
      ( $nm=nm' \implies no-match_{ML}\ vs\ vs'$ ))

```

by *pat-completeness auto*

termination

```

apply(relation measure(%(vs::ml list,-).  $\sum v \leftarrow vs.$  size v))
apply (auto simp:termination-no-match-ML)
done

```

abbreviation

```

no-match-compR nm os  $\equiv$ 
   $\forall (nm',ps,v) \in compR. nm=nm' \implies no-match_{ML}\ ps\ os$ 

```

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  $\Rightarrow$  50)
and red-ml-list :: [ml list, ml list] => bool (infixl  $\Rightarrow$  50)
and reds-ml :: [ml, ml] => bool (infixl  $\Rightarrow^*$  50)

```

where

```

s  $\Rightarrow$  t  $\equiv$  (s, t)  $\in$  Red-ml
| ss  $\Rightarrow$  ts  $\equiv$  (ss, ts)  $\in$  Red-ml-list
| s  $\Rightarrow^*$  t  $\equiv$  (s, t)  $\in$  Red-ml*
— ML  $\beta$ -reduction
|  $A_{ML} (Lam_{ML}\ u)\ [v] \Rightarrow u[v/0]$ 
— Execution of a compiled rewrite rule
| (nm,vs,v) : compR  $\implies \forall i. closed_{ML}\ 0\ (\sigma\ i) \implies$ 
   $A_{ML} (C_{ML}\ nm)\ (map\ (subst_{ML}\ \sigma)\ vs) \Rightarrow subst_{ML}\ \sigma\ v$ 
— default rule:
|  $\forall i. closed_{ML}\ 0\ (\sigma\ i)$ 
 $\implies vs = map\ V_{ML}\ [0..<arity\ nm] \implies vs' = map\ (subst_{ML}\ \sigma)\ vs$ 

```

$\implies \text{no-match-compR } nm \text{ } vs'$
 $\implies A_{ML} (C_{ML} nm) vs' \Rightarrow \text{subst}_{ML} \sigma (C_U nm vs)$
 — Equations for function **apply**
 $| \text{apply-Clo1: } \text{apply} (Clo f vs (Suc 0)) v \Rightarrow A_{ML} f (v \# vs)$
 $| \text{apply-Clo2: } n > 0 \implies$
 $\text{apply} (Clo f vs (Suc n)) v \Rightarrow Clo f (v \# vs) n$
 $| \text{apply-C: } \text{apply} (C_U nm vs) v \Rightarrow C_U nm (v \# vs)$
 $| \text{apply-V: } \text{apply} (V_U x vs) v \Rightarrow V_U x (v \# vs)$
 — Context rules
 $| \text{ctxt-C: } vs \Rightarrow vs' \implies C_U nm vs \Rightarrow C_U nm vs'$
 $| \text{ctxt-V: } vs \Rightarrow vs' \implies V_U x vs \Rightarrow V_U x vs'$
 $| \text{ctxt-Clo1: } f \Rightarrow f' \implies Clo f vs n \Rightarrow Clo f' vs n$
 $| \text{ctxt-Clo3: } vs \Rightarrow vs' \implies Clo f vs n \Rightarrow Clo f vs' n$
 $| \text{ctxt-apply1: } s \Rightarrow s' \implies \text{apply } s t \Rightarrow \text{apply } s' t$
 $| \text{ctxt-apply2: } t \Rightarrow t' \implies \text{apply } s t \Rightarrow \text{apply } s t'$
 $| \text{ctxt-A-ML1: } f \Rightarrow f' \implies A_{ML} f vs \Rightarrow A_{ML} f' vs$
 $| \text{ctxt-A-ML2: } vs \Rightarrow vs' \implies A_{ML} f vs \Rightarrow A_{ML} f vs'$
 $| \text{ctxt-list1: } v \Rightarrow v' \implies v \# vs \Rightarrow v' \# vs$
 $| \text{ctxt-list2: } vs \Rightarrow vs' \implies v \# vs \Rightarrow v \# vs'$

inductive-set

$Red\text{-term} :: (tm * tm) \text{set}$
and $red\text{-term} :: [tm, tm] \Rightarrow bool$ (**infixl** \Rightarrow 50)
and $reds\text{-term} :: [tm, tm] \Rightarrow bool$ (**infixl** \Rightarrow^* 50)

where

$s \Rightarrow t \equiv (s, t) \in Red\text{-term}$
 $| s \Rightarrow^* t \equiv (s, t) \in Red\text{-term}^*$
 — function term
 $| \text{term-C: } \text{term} (C_U nm vs) \Rightarrow (C nm) \cdot \cdot (\text{map term } (rev vs))$
 $| \text{term-V: } \text{term} (V_U x vs) \Rightarrow (V x) \cdot \cdot (\text{map term } (rev vs))$
 $| \text{term-Clo: } \text{term}(Clo vf vs n) \Rightarrow \Lambda (\text{term} (\text{apply} (\text{lift } 0 (Clo vf vs n)) (V_U 0 [])))$
 — context rules
 $| \text{ctxt-Lam: } t \Rightarrow t' \implies \Lambda t \Rightarrow \Lambda t'$
 $| \text{ctxt-At1: } s \Rightarrow s' \implies s \cdot t \Rightarrow s' \cdot t$
 $| \text{ctxt-At2: } t \Rightarrow t' \implies s \cdot t \Rightarrow s \cdot t'$
 $| \text{ctxt-term: } v \Rightarrow v' \implies \text{term } v \Rightarrow \text{term } v'$

3 Kernel

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

fun $size' :: ml \Rightarrow nat$ **where**

$size' (C_{ML} nm) = 1 \mid$
 $size' (V_{ML} X) = 1 \mid$
 $size' (A_{ML} v vs) = (size' v + (\sum v \leftarrow vs. size' v)) + 1 \mid$
 $size' (Lam_{ML} v) = size' v + 1 \mid$
 $size' (C_U nm vs) = (\sum v \leftarrow vs. size' v) + 1 \mid$
 $size' (V_U nm vs) = (\sum v \leftarrow vs. size' v) + 1 \mid$

$size' (Clo\ f\ vs\ n) = (size'\ f + (\sum v \leftarrow vs. size'\ v)) + 1 \mid$
 $size' (apply\ v\ w) = (size'\ v + size'\ w) + 1$

lemma *listsum-size'[simp]*:
 $v \in set\ vs \implies size'\ v < Suc(listsum\ (map\ size'\ vs))$
by(*induct vs*)(*auto*)

corollary *cor-listsum-size'[simp]*:
 $v \in set\ vs \implies size'\ v < Suc(m + listsum\ (map\ size'\ vs))$
using *listsum-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 = listsum]*)
apply(*rule map-ext*)
apply *simp*
apply(*rule arg-cong[where f = listsum]*)
apply(*rule map-ext*)
apply *simp*
apply(*rule arg-cong[where f = listsum]*)
apply(*rule map-ext*)
apply *simp*
apply(*rule arg-cong[where f = listsum]*)
apply(*rule map-ext*)
apply *simp*
done

lemma *size'-subst-ML[simp]*:
 $\forall i\ j. size'(\sigma\ i) = 1 \implies size'\ (subst_{ML}\ \sigma\ v) = size'\ v$
apply(*induct v arbitrary:\sigma rule:size'.induct*)
apply *simp-all*
apply(*rule arg-cong[where f = listsum]*)
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 = listsum]*)
apply(*rule map-ext*)
apply *simp*
apply(*rule arg-cong[where f = listsum]*)
apply(*rule map-ext*)
apply *simp*
apply(*rule arg-cong[where f = listsum]*)
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 = listsum])
  apply(rule map-ext)
  apply simp
  apply(rule arg-cong[where f = listsum])
  apply(rule map-ext)
  apply simp
  apply(rule arg-cong[where f = listsum])
  apply(rule map-ext)
  apply simp
apply(rule arg-cong[where f = listsum])
apply(rule map-ext)
apply simp
done

```

```

function kernel :: ml ⇒ tm (! 300) where
(CML nm)! = C nm |
(AML v vs)! = v! .. (map kernel (rev vs)) |
(LamML v)! =  $\Lambda$  (((lift 0 v)[VU 0 []/0])!) |
(CU nm vs)! = (C nm) .. (map kernel (rev vs)) |
(VU x vs)! = (V x) .. (map kernel (rev vs)) |
(Clo f vs n)! = f! .. (map kernel (rev vs)) |
(apply v w)! = v! · (w!) |
(VML 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!)
| ( $\Lambda$  t)! =  $\Lambda$ (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:

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

lemma pure-kernel: fixes $v :: ml$ **shows** $\text{closed}_{ML} \ 0 \ v \implies \text{pure}(v!)$

proof(*induct v rule:kernel.induct*)

case ($\exists v$)

hence $\text{closed}_{ML} \ (\text{Suc } 0) \ (\text{lift } 0 \ v)$ **by** *simp*

then have $\text{subst}_{ML} \ (\lambda n. \ V_U \ 0 \ []) \ (\text{lift } 0 \ v) = \text{lift } 0 \ v[V_U \ 0 \ []/0]$

by(*rule subst-ML-coincidence simp*)

moreover have $\text{closed}_{ML} \ 0 \ (\text{subst}_{ML} \ (\lambda n. \ V_U \ 0 \ []) \ (\text{lift } 0 \ v))$

by(*simp add: closed-ML-subst-ML*)

ultimately have $\text{closed}_{ML} \ 0 \ (\text{lift } 0 \ v[V_U \ 0 \ []/0])$ **by** *simp*

thus $?case$ **using** $\exists(1)$ **by** (*simp add:pure-foldl*)

qed (*simp-all add:pure-foldl*)

corollary subst-V-kernel: fixes $v :: ml$ **shows**

$\text{closed}_{ML} \ 0 \ v \implies \text{subst } V \ (v!) = v!$

by (*metis pure-kernel subst-V*)

lemma kernel-lift-tm: fixes $v :: ml$ **shows**

$\text{closed}_{ML} \ 0 \ v \implies (\text{lift } i \ v)! = \text{lift } i \ (v!)$

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) $\circ (\lambda n. \text{if } n = 0 \text{ then } V_U \ 0 \ [] \ \text{else } V_{ML} \ (n - 1)) =$*
($\lambda n. \text{if } n = 0 \text{ then } V_U \ 0 \ [] \ \text{else } V_{ML} \ (n - 1))$))

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 substitution lemma *kernel-subst1* below.

```
fun subst-ml :: (nat  $\Rightarrow$  nat)  $\Rightarrow$  ml  $\Rightarrow$  ml where
  subst-ml  $\sigma$  (CML nm) = CML nm |
  subst-ml  $\sigma$  (VML X) = VML X |
  subst-ml  $\sigma$  (AML v vs) = AML (subst-ml  $\sigma$  v) (map (subst-ml  $\sigma$ ) vs) |
  subst-ml  $\sigma$  (LamML v) = LamML (subst-ml  $\sigma$  v) |
  subst-ml  $\sigma$  (CU nm vs) = CU nm (map (subst-ml  $\sigma$ ) vs) |
  subst-ml  $\sigma$  (VU x vs) = VU ( $\sigma$  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)
```

lemma lift-ML-subst-ml:

```
  liftML k (subst-ml  $\sigma$  v) = subst-ml  $\sigma$  (liftML k v)
apply (induct  $\sigma$  v arbitrary: k rule:subst-ml.induct)
apply (simp-all add:list-eq-iff-nth-eq)
done
```

lemma subst-ml-subst-ML:

```
  subst-ml  $\sigma$  (substML  $\sigma'$  v) = substML (subst-ml  $\sigma$  o  $\sigma'$ ) (subst-ml  $\sigma$  v)
apply (induct  $\sigma'$  v arbitrary:  $\sigma$  rule:subst-ML.induct)
apply (simp-all add:list-eq-iff-nth-eq)
apply (subgoal-tac (subst-ml  $\sigma'$  o VML 0 ##  $\sigma$ ) = VML 0 ## (subst-ml  $\sigma'$  o
 $\sigma$ ))
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 ( $\lambda n.$  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  $\sigma$  (subst-ml  $\sigma'$  v) = subst-ml ( $\sigma$  o  $\sigma'$ ) v
by (induct  $\sigma'$  v rule:subst-ml.induct) (simp-all add:list-eq-iff-nth-eq)
```

lemma subst-kernel:

```
  closedML 0 v  $\implies$  subst ( $\lambda n.$  V( $\sigma$  n)) (v!) = (subst-ml  $\sigma$  v)!
apply (induct v arbitrary:  $\sigma$  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( $\sigma$  k) in meta-alle)
apply (erule-tac meta-impE)
```

```

apply(rule closed-ML-subst-ML2[where  $k = \text{Suc } 0$ ])
apply (metis closed-ML-lift)
apply simp
apply(subgoal-tac ( $\lambda n. V(\text{case } n \text{ of } 0 \Rightarrow 0 \mid \text{Suc } k \Rightarrow \text{Suc } (\sigma k)) = (V\ 0 \ \#\#$ 
( $\lambda n. V(\sigma n)))$ ))
apply (simp add:subst-ml-subst-ML)
defer
apply(simp add:expand-fun-eq split:nat.split)
apply(simp add:lift-is-subst-ml subst-ml-comp)
apply(rule arg-cong[where  $f = \text{kernel}$ ])
apply(subgoal-tac ( $\text{nat-case } 0 (\lambda k. \text{Suc } (\sigma k)) \circ \text{Suc} = \text{Suc } \circ \sigma$ )
prefer 2 apply(simp add:expand-fun-eq split:nat.split)
apply(subgoal-tac (subst-ml ( $\text{nat-case } 0 (\lambda k. \text{Suc } (\sigma k))$ )  $\circ$ 
( $\lambda n. \text{if } n = 0 \text{ then } V_U\ 0 \ \square \ \text{else } V_{ML}\ (n - 1)$ ))
= ( $\lambda n. \text{if } n = 0 \text{ then } V_U\ 0 \ \square \ \text{else } V_{ML}\ (n - 1)$ ))
apply simp
apply(simp add: expand-fun-eq)
done

```

lemma if-cong0: *If $x\ y\ z = \text{If } x\ y\ z$*
by simp

lemma kernel-subst1:

```

closedML 0 v  $\implies$  closedML (Suc 0) u  $\implies$ 
kernel(u[v/0]) = (kernel((lift 0 u)[VU 0  $\square$ /0]))[v!/0]
proof(induct u arbitrary:v rule:kernel.induct)
case ( $\exists w$ )
show ?case (is ?L = ?R)
proof -
have ?L =  $\Lambda(\text{lift } 0 (w[\text{lift}_{ML}\ 0\ v/\text{Suc } 0])[V_U\ 0\ \square/0] !)$ 
by (simp cong:if-cong0)
also have ... =  $\Lambda((\text{lift } 0\ w)[\text{lift}_{ML}\ 0\ (\text{lift } 0\ v)/\text{Suc } 0][V_U\ 0\ \square/0] !)$ 
by(simp only: lift-subst-ML1 lift-lift-ML-comm)
also have ... =  $\Lambda(\text{subst}_{ML} (\lambda n. \text{if } n=0 \text{ then } V_U\ 0 \ \square \ \text{else}$ 
 $\text{if } n=\text{Suc } 0 \text{ then } \text{lift } 0\ v \ \text{else } V_{ML}\ (n - 2)) (\text{lift } 0\ w) !)$ 
apply simp
apply(rule arg-cong[where  $f = \text{kernel}$ ])
apply(rule subst-ML-comp2)
using 3
apply auto
done
also have ... =  $\Lambda((\text{lift } 0\ w)[V_U\ 0\ \square/0][\text{lift } 0\ v/0] !)$ 
apply simp
apply(rule arg-cong[where  $f = \text{kernel}$ ])
apply(rule subst-ML-comp2[symmetric])
using 3
apply auto
done
also have ... =  $\Lambda((\text{lift-ml } 0 ((\text{lift-ml } 0\ w)[V_U\ 0\ \square/0]))[V_U\ 0\ \square/0]![\text{lift } 0$ 

```

```

v)!/0])
  apply(rule arg-cong[where f =  $\Lambda$ ])
  apply(rule 3(1))
  apply (metis closed-ML-lift 3(2))
  apply(subgoal-tac closedML (Suc(Suc 0)) w)
  defer
  using 3
  apply force
  apply(subgoal-tac closedML (Suc (Suc 0)) (lift 0 w))
  defer
  apply(erule closed-ML-lift)
  apply(erule closed-ML-subst-ML2)
  apply simp
  done
also have ... =  $\Lambda((\text{lift-ml } 0 (\text{lift-ml } 0 w)[V_U 1 \ \square/0])[V_U 0 \ \square/0]!(\text{lift } 0 v)!/0])$ 
(is - = ?M)
  apply(subgoal-tac lift-ml 0 (lift-ml 0 w[V_U 0 \ \square/0])[V_U 0 \ \square/0] =
    lift-ml 0 (lift-ml 0 w)[V_U 1 \ \square/0][V_U 0 \ \square/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 =  $\Lambda(\text{subst } (V 0 \ \#\# \ \text{subst-decr } 0 (v!))$ 
  (lift 0 (lift-ml 0 w[V_U 0 \ \square/Suc 0])[V_U 0 \ \square/0]!))
  apply(subgoal-tac (VML 0  $\#\#$  ( $\lambda n.$  if n = 0 then VU 0  $\square$  else VML (n -
Suc 0))) = subst-decr-ML (Suc 0) (VU 0  $\square$ ))
  apply(simp cong:if-cong)
  apply(simp add:expand-fun-eq cons-ML-def split:nat.splits)
  done
also have ... =  $\Lambda(\text{subst } (V 0 \ \#\# \ \text{subst-decr } 0 (v!))$ 
  ((lift 0 (lift-ml 0 w)[V_U 1 \ \square/Suc 0][V_U 0 \ \square/0]!))
  apply(subgoal-tac lift 0 (lift 0 w[V_U 0 \ \square/Suc 0]) = lift 0 (lift 0 w)[V_U 1
\square/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 \ \square/Suc 0][V_U 0 \ \square/0] =
  (lift 0 (lift-ml 0 w))[V_U 0 \ \square/0][V_U 1 \ \square/0] (is ?l = ?r)
proof -
  have ?l = substML ( $\lambda n.$  if n= 0 then VU 0  $\square$  else if n = 1 then VU 1  $\square$  else
VML (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  $\Lambda(\text{subst } (V 0 \ \#\# \ \text{subst-decr } 0 (v!)) (\text{?r } !)) = ?M$ 

```

```

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–
    def pi == λn::nat. if n = 0 then 1 else if n = 1 then 0 else n
    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 expand-fun-eq)
      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* \Rightarrow *nat*

primrec *compile* :: *tm* \Rightarrow (*nat* \Rightarrow *ml*) \Rightarrow *ml*

where

```

  compile (V x)  $\sigma$  =  $\sigma$  x
| compile (C nm)  $\sigma$  = Clo (CML nm) [] (arity nm)
| compile (s · t)  $\sigma$  = apply (compile s  $\sigma$ ) (compile t  $\sigma$ )
| compile ( $\Lambda$  t)  $\sigma$  = Clo (LamML (compile t (VML 0 ##  $\sigma$ ))) [] 1

```

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

definition *comp-open* *t* = *compile* *t* *V_{ML}*

abbreviation *comp-fixed* *t* \equiv *compile* *t* ($\lambda i. V_U i$ [])

Compiled rules:

lemma *size-args-less-size-tm*[*simp*]: $s \in \text{set } (\text{args-tm } t) \implies \text{size } s < \text{size } t$

by(*induct* *t*) *auto*

fun *comp-pat* **where**

```

comp-pat t =
  (case head-tm t of
    C nm  $\Rightarrow$  CU nm (map comp-pat (rev (args-tm t)))
  | V X  $\Rightarrow$  VML 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]*:
 $comp\text{-}pat(C\ nm\ \cdot\ ts) = C_U\ nm\ (map\ comp\text{-}pat\ (rev\ ts))$
by(*simp add:comp-pat.simps*)

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

defs *compR-def*:
 $compR \equiv (\lambda(nm,ts,t). (nm, map\ comp\text{-}pat\ (rev\ ts), comp\text{-}open\ t))\ 'R$

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

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

lemma *fv-compR-aux*:
 $(nm,ts,t') : R \implies x \in fv_{ML}\ (comp\text{-}open\ t')$
 $\implies \exists t \in set\ ts. x \in fv_{ML}(comp\text{-}pat\ t)$
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*:
 $(nm,vs,v) : compR \implies x \in fv_{ML}\ v \implies \exists u \in set\ vs. x \in fv_{ML}\ u$
by(*fastsimp simp add:compR-def image-def dest:fv-compR-aux*)

lemma *lift-compile*:
 $pure\ t \implies \forall \sigma\ k. lift\ k\ (compile\ t\ \sigma) = compile\ t\ (lift\ k\ \circ\ \sigma)$
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*:
 $pure\ t \implies subst_{ML}\ \sigma'\ (compile\ t\ \sigma) = compile\ t\ (subst_{ML}\ \sigma'\ \circ\ \sigma)$
apply(*induct arbitrary: \sigma\ \sigma'\ pred:pure*)
apply *simp-all*
apply(*erule-tac x = V_{ML}\ 0 ## \sigma' 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:

$pure\ t \implies \forall i. \sigma\ i = V_U\ i \ [] \implies (compile\ t\ \sigma)! = t$
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} ($\lambda n. \text{if } n = 0 \text{ then } V_U\ 0 \ [] \text{ else } V_{ML}\ (n - 1)) \circ$
 $(lift\ 0 \circ V_{ML}\ 0 \#\# \sigma) = (\lambda a. V_U\ a \ [])$))
apply(simp)
apply(rule ext)
apply(simp)
done

lemma kernel-subst-ML-pat:

$pure\ t \implies pattern\ t \implies \forall i. closed_{ML}\ 0\ (\sigma\ i) \implies$
 $(subst_{ML}\ \sigma\ (comp-pat\ t))! = subst\ (kernel\ \circ\ \sigma)\ t$
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:

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

lemma kernel-subst-ML-pat-map:

$\forall t \in set\ ts. pure\ t \implies patterns\ ts \implies \forall i. closed_{ML}\ 0\ (\sigma\ i) \implies$
 $map\ kernel\ (map\ (subst_{ML}\ \sigma)\ (map\ comp-pat\ ts)) =$

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

lemma compR-Red-tm: (nm, vs, v) : compR $\implies \forall i. \text{closed}_{ML} 0 (\sigma i)$
 $\implies C \text{ nm} \cdot (\text{map} (\text{subst}_{ML} \sigma) (\text{rev vs}))! \rightarrow^* (\text{subst}_{ML} \sigma v)!$
 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: $s = t \implies t \rightarrow t' \implies s \rightarrow t'$
 by simp

Soundness of reduction:

theorem fixes $v :: ml$ shows Red-ml-sound:

$v \Rightarrow v' \implies \text{closed}_{ML} 0 v \implies v! \rightarrow^* v'! \wedge \text{closed}_{ML} 0 v'$ and
 $vs \Rightarrow vs' \implies \forall v \in \text{set } vs. \text{closed}_{ML} 0 v \implies$
 $vs! \rightarrow^* vs'! \wedge (\forall v' \in \text{set } vs'. \text{closed}_{ML} 0 v')$

proof (induct rule: Red-ml-Red-ml-list.inducts)

fix u v

let ?v = $A_{ML} (\text{Lam}_{ML} u) [v]$

assume cl: $\text{closed}_{ML} 0 (A_{ML} (\text{Lam}_{ML} u) [v])$

let ?u' = $(\text{lift-ml } 0 u)[V_U 0 []/0]$

have ?v! = $(\Lambda((?u')!)) \cdot (v !)$ **by** simp

also have $\dots \rightarrow (?u'!)[v!/0]$ (**is** $\rightarrow ?R$) **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 $\text{kernel}(A_{ML} (\text{Lam}_{ML} u) [v]) \rightarrow^* \text{kernel}(u[v/0])$ (**is** ?A)

by (rule r-into-rtrancl)

moreover have $\text{closed}_{ML} 0 (u[v/0])$ (**is** ?C)

proof –

let ? σ = $\lambda n. \text{if } n = 0 \text{ then } v \text{ else } V_{ML} (n - 1)$

let ? σ' = $\lambda n. v$

have clu: $\text{closed}_{ML} (\text{Suc } 0) u$ and clv: $\text{closed}_{ML} 0 v$ **using** cl **by** simp+

have $\text{closed}_{ML} 0 (\text{subst}_{ML} ?\sigma' u)$

by (metis closed-ML-subst-ML clv)

hence $\text{closed}_{ML} 0 (\text{subst}_{ML} ?\sigma u)$

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. closedML 0 (σ i) and compR: (nm, vs, v) ∈ compR
have map (subst V) (map (substML σ) (rev vs)!) = map (substML 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 (substML σ) (rev vs))!) →* (substML σ v)!
  by(simp add:subst-V-kernel closed-ML-subst-ML[OF σ])
hence AML (CML nm) (map (substML σ) vs)! →* substML σ v! (is ?A)
  by(simp add:rev-map)
moreover
have closedML 0 (substML 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' ⇒ closedML 0 t ⇒ kernelt t →* kernelt t' ∧ closedML 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)) (VU 0 [])))!
  using term-Clo
  by(simp del:lift-foldl-At add: r-into-rtrancl Red-tm.intros(2))
moreover
have closedML 0 (Λ (term (apply (lift 0 (Clo vf vs n)) (VU 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' ⇒ closedML 0 t ⇒ t! →* t'! ∧ closedML 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*:

$\text{pure } t \implies \forall i. \text{closed}_{ML} \ n \ (\sigma \ i) \implies \text{closed}_{ML} \ n \ (\text{compile } t \ \sigma)$

proof(*induct arbitrary:n σ pred:pure*)

case (*Lam t*)

have $1: \forall i. \text{closed}_{ML} \ (\text{Suc } n) \ ((V_{ML} \ 0 \ \#\# \ \sigma) \ i)$ **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*: $\text{closed}_{ML} \ 0 \ (\text{term } (\text{comp-fixed } t))$

by (*simp add: closed-ML-compile[OF $\langle \text{pure } t \rangle$]*)

have $(\text{term } (\text{comp-fixed } t))! = t$

using *kernel-compile[OF $\langle \text{pure } t \rangle$]* **by** *simp*

moreover **have** $\text{term } (\text{comp-fixed } t)! \rightarrow^* t'$

using *kernel-inv[OF assms(2) ML-cl]* **by** *auto*

ultimately **have** $t \rightarrow^* t'$ **by** *simp*

thus *?thesis* **using** *kernel-pure[OF $\langle \text{pure } t' \rangle$]* **by** *simp*

qed

6 Normal Forms

inductive *normal* :: $tm \Rightarrow bool$ **where**

$\forall t \in \text{set } ts. \text{normal } t \implies \text{normal}(V \ x \ \cdot\cdot \ ts) \mid$

$\text{normal } t \implies \text{normal}(\Lambda \ t) \mid$

$\forall t \in \text{set } ts. \text{normal } t \implies$

$\forall \sigma. \forall (nm', ls, r) \in R. \neg(nm = nm' \wedge \text{take } (\text{size } ls) \ ts = \text{map } (\text{subst } \sigma) \ ls)$

$\implies \text{normal}(C \ nm \ \cdot\cdot \ ts)$

fun *C-normal-ML* :: $ml \Rightarrow bool$ (*C'-normal_{ML}*) **where**

$C\text{-normal}_{ML}(C_U \ nm \ vs) =$

$((\forall v \in \text{set } vs. C\text{-normal}_{ML} \ v) \wedge \text{no-match-compR } nm \ vs) \mid$

$C\text{-normal}_{ML} \ (C_{ML} \ -) = \text{True} \mid$

$C\text{-normal}_{ML} \ (V_{ML} \ -) = \text{True} \mid$

$C\text{-normal}_{ML} \ (A_{ML} \ v \ vs) = (C\text{-normal}_{ML} \ v \wedge (\forall v \in \text{set } vs. C\text{-normal}_{ML} \ v)) \mid$

$C\text{-normal}_{ML} \ (\text{Lam}_{ML} \ v) = C\text{-normal}_{ML} \ v \mid$

$C\text{-normal}_{ML} \ (V_U \ x \ vs) = (\forall v \in \text{set } vs. C\text{-normal}_{ML} \ v) \mid$

$C\text{-normal}_{ML} \ (C_{lo} \ v \ vs \ -) = (C\text{-normal}_{ML} \ v \wedge (\forall v \in \text{set } vs. C\text{-normal}_{ML} \ v)) \mid$

$C\text{-normal}_{ML} \ (\text{apply } u \ v) = (C\text{-normal}_{ML} \ u \wedge C\text{-normal}_{ML} \ v)$

fun *size-tm* :: $tm \Rightarrow nat$ **where**

$\text{size-tm} \ (C \ -) = 1 \mid$

$\text{size-tm} \ (\text{At } s \ t) = \text{size-tm } s + \text{size-tm } t + 1 \mid$

$\text{size-tm} \ - = 0$

lemma *size-tm-foldl-At*: $\text{size-tm}(t \cdot\cdot ts) = \text{size-tm } t + \text{list-size size-tm } ts$
by (*induct ts arbitrary:t*) *auto*

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

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

function *no-match* :: *tm list* \Rightarrow *tm list* \Rightarrow *bool* **where**
no-match ps ts =
 $(\exists i < \min (\text{size } ts) (\text{size } ps).$
 $\exists \text{ nm nm}' \text{ rs rs}'. \text{ ps} ! i = (C \text{ nm}) \cdot\cdot \text{ rs} \wedge \text{ ts} ! i = (C \text{ nm}') \cdot\cdot \text{ rs}' \wedge$
 $(\text{nm} = \text{nm}' \longrightarrow \text{no-match } \text{rs } \text{rs}'))$
by *pat-completeness auto*
termination
apply(*relation measure*($\%(ts::tm \text{ list}, -). \sum t \leftarrow ts. \text{size-tm } t$))
apply (*auto simp:termination-no-match*)
done

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

abbreviation
no-match-R nm ts $\equiv \forall (\text{nm}', \text{ps}, t) \in R. \text{nm} = \text{nm}' \longrightarrow \text{no-match } \text{ps } ts$

lemma *no-match*: $\text{no-match } \text{ps } ts \implies \neg(\exists \sigma. \text{map } (\text{subst } \sigma) \text{ps} = ts)$
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 *fastsimp*
done
qed

lemma *no-match-take*: $\text{no-match } \text{ps } ts \implies \text{no-match } \text{ps } (\text{take } (\text{size } \text{ps}) \text{ts})$
apply(*subst (asm) no-match.simps*)
apply(*subst no-match.simps*)
apply *fastsimp*
done

fun *dterm-ML* :: *ml* \Rightarrow *tm* (*dterm_{ML}*) **where**
dterm_{ML} (*C_U nm vs*) = $C \text{ nm } \cdot\cdot \text{map } \text{dterm}_{ML} (\text{rev } vs) \mid$

$dterm_{ML} - = V 0$

fun $dterm :: tm \Rightarrow tm$ **where**

$dterm (V n) = V n$ |
 $dterm (C nm) = C nm$ |
 $dterm (s \cdot t) = dterm s \cdot dterm t$ |
 $dterm (\Lambda t) = \Lambda (dterm t)$ |
 $dterm (term v) = dterm_{ML} v$

lemma $dterm\text{-}pure[simp]$: $pure\ t \Longrightarrow dterm\ t = t$
by ($induct\ pred: pure$) *auto*

lemma $map\text{-}dterm\text{-}pure[simp]$: $\forall t \in set\ ts. pure\ t \Longrightarrow map\ dterm\ ts = ts$
by ($induct\ ts$) *auto*

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

lemma $dterm\text{-}foldl\text{-}At[simp]$: $dterm(t \cdot\cdot ts) = dterm\ t \cdot\cdot map\ dterm\ ts$
by($induct\ ts\ arbitrary: t$) *auto*

lemma $no\text{-}match\text{-}coincide$:

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

```

apply(rule-tac x=map dtermML (rev vs) in exI)
apply(rule-tac x=map dtermML (rev vs') in exI)
apply(simp)
done

```

lemma *dterm-ML-comp-patD*:

```

  pattern t  $\implies$  dtermML (comp-pat t) = C nm .. rs  $\implies$   $\exists$  ts. t = C nm .. ts
by(induct pred:pattern) simp-all

```

lemma *no-match-R-coincide-aux*[rule-format]: patterns ts \implies

```

  no-match (map (dtermML  $\circ$  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)  $\implies$  no-match-R nm (map dtermML 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 dtermML vs])
apply(rule-tac x=i in exI)
apply simp
apply(cut-tac t = aal!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* \Rightarrow *bool* **where**
 $\forall t \in \text{set } ts. C\text{-normal } t \Longrightarrow C\text{-normal}(V\ x \ \cdot\cdot\ ts) \mid$
 $C\text{-normal } t \Longrightarrow C\text{-normal}(\Lambda\ t) \mid$
 $C\text{-normal}_{ML}\ v \Longrightarrow C\text{-normal}(\text{term } v) \mid$
 $\forall t \in \text{set } ts. C\text{-normal } t \Longrightarrow \text{no-match-R } nm\ (\text{map } d\text{term } ts)$
 $\Longrightarrow C\text{-normal}(C\ nm \ \cdot\cdot\ ts)$

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

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

lemma [*simp*]: $C\text{-normal}(\Lambda\ t) = C\text{-normal } t$
apply (*auto*)
apply (*erule C-normal.cases*)
apply *auto*
done

lemma [*simp*]: $C\text{-normal}(V\ x)$
using *C-normal.intros*(1)[*of [] x*]
by *simp*

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

lemma $u \Rightarrow (v::ml) \Longrightarrow \text{True}$ **and**
Red-ml-list-length: $vs \Rightarrow vs' \Longrightarrow \text{length } vs = \text{length } vs'$
by (*induct rule: Red-ml-Red-ml-list.inducts*) *simp-all*

lemma $(v::ml) \Rightarrow v' \Longrightarrow \text{True}$ **and**
Red-ml-list-nth: $(vs::ml\ \text{list}) \Rightarrow vs'$
 $\Longrightarrow \exists v' k. k < \text{size } vs \wedge vs!k \Rightarrow v' \wedge vs' = vs[k := v']$
apply (*induct rule: Red-ml-Red-ml-list.inducts*)
apply (*auto split:nat.splits*)
done

lemma *Red-ml-list-pres-no-match*:
 $\text{no-match}_{ML}\ ps\ vs \Longrightarrow vs \Rightarrow vs' \Longrightarrow \text{no-match}_{ML}\ ps\ vs'$
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(fastsimp intro: 1(1) simp add:rev-nth)
done
qed

```

```

lemma no-match-ML-subst-ML[rule-format]:
   $\forall v \in \text{set } vs. \forall x \in \text{fv}_{ML} v. C\text{-normal}_{ML}(\sigma x) \implies$ 
   $\text{no-match}_{ML} ps vs \longrightarrow \text{no-match}_{ML} ps (\text{map}(\text{subst}_{ML} \sigma) vs)$ 
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 (substML  $\sigma$ ) 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:
   $\text{lift}_{ML} k v = C_U nm vs' \implies \exists vs. v = C_U nm vs \wedge vs' = \text{map}(\text{lift}_{ML} k) vs$ 
by(cases v) auto

```

```

lemma no-match-ML-lift-ML:

```

```

    no-matchML ps (map (liftML k) vs) = no-matchML 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 fastsimp
  apply clarsimp
  apply(rule-tac x=i in exI)
  apply simp
  apply(rule-tac x=nm' in exI)
  apply(rule-tac x=map (liftML k) vs' in exI)
  apply (fastsimp simp:rev-nth)
  done

```

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

lemma *no-match-compR-Cons*:
 $no\text{-match}\text{-compR} nm vs \implies no\text{-match}\text{-compR} nm (v \# vs)$
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*: $pure t \implies C\text{-normal}_{ML}(\text{comp-open } t)$
by (induct pred:pure) (auto simp:comp-open-def)

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

lemma *C-normal-ML-subst-ML*:
 $C\text{-normal}_{ML}(\text{subst}_{ML} \sigma v) \implies (\forall x \in \text{fv}_{ML} v. C\text{-normal}_{ML}(\sigma x))$
proof(induct σv 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*: $C\text{-normal}_{ML} v \implies$

$C\text{-normal}_{ML} (\text{subst}_{ML} \sigma v) \longleftrightarrow (\forall x \in \text{fv}_{ML} v. C\text{-normal}_{ML} (\sigma x))$
proof(*induct* σv *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
 $vs \Rightarrow vs' \Longrightarrow \forall v \in \text{set } vs. C\text{-normal}_{ML} v \Longrightarrow \forall v' \in \text{set } vs'. C\text{-normal}_{ML} v'$
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 $(t::tm) \Rightarrow t'$
 $\bigwedge nm \text{ vs } ts. P ((\text{term } (C_U \text{ nm } \text{ vs})) \cdot\cdot ts) ((C \text{ nm } \cdot\cdot \text{map term } (\text{rev } \text{ vs})) \cdot\cdot ts)$
 $\bigwedge x \text{ vs } ts. P (\text{term } (V_U x \text{ vs}) \cdot\cdot ts) ((V x \cdot\cdot \text{map term } (\text{rev } \text{ vs})) \cdot\cdot ts)$
 $\bigwedge \text{vf } \text{vs } n \text{ ts.}$
 $P (\text{term } (Clo \text{ vf } \text{ vs } n) \cdot\cdot ts)$
 $((\Lambda (\text{term } (\text{apply } (\text{lift } 0 (Clo \text{ vf } \text{ vs } n)) (V_U 0 [])))) \cdot\cdot ts)$
 $\bigwedge t \text{ t}' \text{ ts. } \llbracket t \Rightarrow t'; P t t' \rrbracket \Longrightarrow P (\Lambda t \cdot\cdot ts) (\Lambda t' \cdot\cdot ts)$
 $\bigwedge v \text{ v}' \text{ ts. } v \Rightarrow v' \Longrightarrow P (\text{term } v \cdot\cdot ts) (\text{term } v' \cdot\cdot ts)$
 $\bigwedge x \text{ i } t' \text{ ts. } i < \text{size } ts \Longrightarrow \text{ts!}i \Rightarrow t' \Longrightarrow P (\text{ts!}i) (t')$
 $\Longrightarrow P (V x \cdot\cdot ts) (V x \cdot\cdot \text{ts}[i:=t'])$
 $\bigwedge nm \text{ i } t' \text{ ts. } i < \text{size } ts \Longrightarrow \text{ts!}i \Rightarrow t' \Longrightarrow P (\text{ts!}i) (t')$
 $\Longrightarrow P (C \text{ nm } \cdot\cdot ts) (C \text{ nm } \cdot\cdot \text{ts}[i:=t'])$
 $\bigwedge t \text{ i } t' \text{ ts. } i < \text{size } ts \Longrightarrow \text{ts!}i \Rightarrow t' \Longrightarrow P (\text{ts!}i) (t')$
 $\Longrightarrow P (\Lambda t \cdot\cdot ts) (\Lambda t \cdot\cdot \text{ts}[i:=t'])$
 $\bigwedge v \text{ i } t' \text{ ts. } i < \text{size } ts \Longrightarrow \text{ts!}i \Rightarrow t' \Longrightarrow P (\text{ts!}i) (t')$
 $\Longrightarrow P (\text{term } v \cdot\cdot ts) (\text{term } v \cdot\cdot (\text{ts}[i:=t']))$

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 $(t::tm) \Rightarrow t'$

$\bigwedge nm \ vs \ ts.$

$t = \text{term } (C_U \ nm \ vs) \cdot ts \Longrightarrow t' = (C \ nm \cdot \text{map term } (\text{rev } vs)) \cdot ts \Longrightarrow P$

$\bigwedge x \ vs \ ts.$

$t = \text{term } (V_U \ x \ vs) \cdot ts \Longrightarrow t' = (V \ x \cdot \text{map term } (\text{rev } vs)) \cdot ts \Longrightarrow P$

$\bigwedge vf \ vs \ n \ ts. t = \text{term } (Clo \ vf \ vs \ n) \cdot ts \Longrightarrow$

$t' = \Lambda (\text{term } (\text{apply } (\text{lift } 0 \ (Clo \ vf \ vs \ n)) \ (V_U \ 0 \ []))) \cdot ts \Longrightarrow P$

$\bigwedge s \ s' \ ts. t = \Lambda \ s \cdot ts \Longrightarrow t' = \Lambda \ s' \cdot ts \Longrightarrow s \Rightarrow s' \Longrightarrow P$

$\bigwedge v \ v' \ ts. t = \text{term } v \cdot ts \Longrightarrow t' = \text{term } v' \cdot ts \Longrightarrow v \Rightarrow v' \Longrightarrow P$

$\bigwedge x \ i \ r' \ ts. i < \text{size } ts \Longrightarrow ts!i \Rightarrow r'$

$\Longrightarrow t = V \ x \cdot ts \Longrightarrow t' = V \ x \cdot ts[i:=r'] \Longrightarrow P$

$\bigwedge nm \ i \ r' \ ts. i < \text{size } ts \Longrightarrow ts!i \Rightarrow r'$

$\Longrightarrow t = C \ nm \cdot ts \Longrightarrow t' = C \ nm \cdot ts[i:=r'] \Longrightarrow P$

$\bigwedge s \ i \ r' \ ts. i < \text{size } ts \Longrightarrow ts!i \Rightarrow r'$

$\Longrightarrow t = \Lambda \ s \cdot ts \Longrightarrow t' = \Lambda \ s \cdot ts[i:=r'] \Longrightarrow P$

$\bigwedge v \ i \ r' \ ts. i < \text{size } ts \Longrightarrow ts!i \Rightarrow r'$

$\Longrightarrow t = \text{term } v \cdot ts \Longrightarrow t' = \text{term } v \cdot (ts[i:=r']) \Longrightarrow P$

shows P using *assms*

apply –

apply (*induct rule:Red-term-hnf-induct*)

apply *metis+*

done

lemma [simp]: $C\text{-normal}(\text{term } v \cdot\cdot ts) \longleftrightarrow C\text{-normal}_{ML} v \wedge ts = []$
by(fastsimp elim: C-normal.cases)

lemma [simp]: $C\text{-normal}(\Lambda t \cdot\cdot ts) \longleftrightarrow C\text{-normal } t \wedge ts = []$
by(fastsimp elim: C-normal.cases)

lemma [simp]: $C\text{-normal}(C \text{ nm} \cdot\cdot ts) \longleftrightarrow$
 $(\forall t \in \text{set } ts. C\text{-normal } t) \wedge \text{no-match-R nm } (\text{map } dterm \text{ } ts)$
by(fastsimp elim: C-normal.cases)

lemma [simp]: $C\text{-normal}(V x \cdot\cdot ts) \longleftrightarrow (\forall t \in \text{set } ts. C\text{-normal } t)$
by(fastsimp elim: C-normal.cases)

lemma no-match-ML-lift:
 $\text{no-match}_{ML} ps \text{ vs} \longrightarrow \text{no-match}_{ML} ps (\text{map } (\text{lift } k) \text{ } vs)$
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 (fastsimp simp: rev-nth)
done

lemma no-match-compR-lift:
 $\text{no-match-compR nm } vs \Longrightarrow \text{no-match-compR nm } (\text{map } (\text{lift } k) \text{ } vs)$
by (fastsimp simp: no-match-ML-lift)

lemma [simp]: $C\text{-normal}_{ML} v \Longrightarrow C\text{-normal}_{ML}(\text{lift } k \text{ } v)$
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:
 $[[i < \text{length } ts; ts ! i \Rightarrow t'; \text{no-match } ps \text{ } dts; dts = (\text{map } dterm \text{ } ts)]]$
 $\Longrightarrow \text{no-match } ps (\text{map } dterm \text{ } (ts[i := t']))$
proof(induct ps dts arbitrary: ts i t' rule:no-match.induct)
case (1 ps dts ts i t')
from $\langle \text{no-match } ps \text{ } dts \rangle \langle dts = \text{map } dterm \text{ } ts \rangle$
obtain $j \text{ nm } nm' \text{ rs } rs'$ **where** $ob: j < \text{size } ts \ j < \text{size } ps$
 $ps!j = C \text{ nm} \cdot\cdot rs \text{ } dterm \text{ } (ts!j) = C \text{ nm}' \cdot\cdot rs'$
 $nm = nm' \longrightarrow \text{no-match } rs \text{ } rs'$
by (subst (asm) no-match.simps) fastsimp
show ?case

```

proof (subst no-match.simps)
  show  $\exists k < \min (\text{length } (\text{map } \text{dterm } (ts[i := t']))) (\text{length } ps)$ .
     $\exists nm \ nm' \ rs \ rs'. \ ps!k = C \ nm \ \cdot \ rs \ \wedge$ 
       $\text{map } \text{dterm } (ts[i := t'])!k = C \ nm' \ \cdot \ rs' \ \wedge$ 
       $(nm = nm' \longrightarrow \text{no-match } rs \ rs')$ 
    (is  $\exists k < ?m. ?P \ k$ )
proof -
  { assume [simp]:  $j=i$ 
    have  $\exists rs'. \ \text{dterm } t' = C \ nm' \ \cdot \ rs' \ \wedge (nm = nm' \longrightarrow \text{no-match } rs \ rs')$ 
      using  $\langle ts \ ! \ i \Rightarrow t' \rangle$ 
    proof(cases rule:Red-term-hnf-cases)
      case (5  $v \ v' \ ts''$ )
        then obtain  $vs$  where [simp]:
           $v = C_U \ nm' \ vs \ rs' = \text{map } \text{dterm}_{ML} (\text{rev } vs) \ @ \ \text{map } \text{dterm } ts''$ 
          using ob by(cases  $v$ ) auto
        obtain  $vs'$  where [simp]:  $v' = C_U \ nm' \ vs' \ vs \Rightarrow vs'$ 
          using  $\langle v \Rightarrow v' \rangle$  by(rule Red-ml.cases) auto
        obtain  $v' \ k$  where [arith]:  $k < \text{size } vs$  and  $vs!k \Rightarrow v'$ 
          and [simp]:  $vs' = vs[k := v']$ 
          using Red-ml-list-nth[OF  $\langle v \Rightarrow v' \rangle$ ] by fastsimp
        show thesis (is  $\exists rs'. ?P \ rs' \ \wedge \ ?Q \ rs'$ )
        proof
          let  $?rs' = \text{map } \text{dterm } ((\text{map } \text{term } (\text{rev } vs) \ @ \ ts'')[(\text{size } vs - k - 1) := \text{term}$ 
             $v'])$ 
          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  $\langle vs!k \Rightarrow v' \rangle$ ] simp
              del: map-map)
          done
          ultimately show  $?P \ ?rs' \ \wedge \ ?Q \ ?rs' \ ..$ 
        qed
      next
      case (7  $nm'' \ k \ r' \ ts''$ )
        show thesis (is  $\exists rs'. ?P \ rs'$ )
        proof
          show  $?P(\text{map } \text{dterm } (ts''[k := r']))$ 
            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]:  $v = C_U \ nm' \ vs \ rs' = \text{map } \text{dterm}_{ML} (\text{rev}$ 

```

```

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.prem1 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 prem1 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 listsimps-eq-0-iff: listsum ns = (0::nat) ↔ (∀ n ∈ set ns. n = 0)
by(metis listsum sum-eq-0-conv)

```

```

lemma elem-le-listsum-nat: k < size ns ⇒ ns!k ≤ listsum(ns::nat list)
apply(induct ns arbitrary: k)
apply(auto simp:nth-Cons')
apply(metis diff-less less-imp-diff-less nat-neq-iff not-less-eq trans-le-add2)
done

```

```

lemma listsum-update: k < size ns ⇒
  listsum (ns[k := (n::nat)]) = listsum ns + n - ns!k
apply(induct ns arbitrary:k)
apply (auto split:nat.split)
apply(drule elem-le-listsum-nat)
apply arith
done

```

```

lemma Red-term-pres-no-match-it:
  [∀ i < length ts. (ts ! i, ts' ! i) : Red-term ^^ (ns!i);

```

$size\ ts' = size\ ts; size\ ns = size\ ts;$
 $no-match\ ps\ (map\ dterm\ ts)]$
 $\implies no-match\ ps\ (map\ dterm\ ts')$
proof(*induct* $n \equiv listsum\ ns$ *arbitrary: ts ns*)
case 0
hence $\forall i < size\ ts. ns!i = 0$ **by** (*simp add:listsimps-eq-0-iff*)
with 0 **show** ?*case* **by** *simp (metis nth-equalityI)*
next
case (*Suc n*)
then have $listsum\ ns \neq 0$ **by** *arith*
then obtain $k\ l$ **where** $k < size\ ts$ **and** [*simp*]: $ns!k = Suc\ l$
by(*simp add:listsimps-eq-0-iff*)
(*metis Suc(4) gr0-implies-Suc in-set-conv-nth*)
let ?*ns* = $ns[k := l]$
have $n = listsum\ ?ns$ **using** *Suc(6)* $\langle k < size\ ts \rangle \langle size\ ns = size\ ts \rangle$
by (*simp add:listsum-update*)
obtain t' **where** $ts!k \Rightarrow t'$ ($t', ts!k$) : $Red-term\ ^\wedge\ l$
using *Suc(2)* $\langle k < size\ ts \rangle \langle size\ ns = size\ ts \rangle \langle ns!k = Suc\ l \rangle$
by (*metis rel-pow-Suc-E2*)
then have $1: \forall i < size(ts[k:=t']). (ts[k:=t']!i, ts!i) : Red-term\ ^\wedge\ (ns!i)$
using *Suc(2)* $\langle k < size\ ts \rangle \langle size\ ns = size\ ts \rangle$
by (*auto simp add:nth-list-update*)
note $nm1 = Red-term-pres-no-match[OF\ \langle k < size\ ts \rangle \langle ts!k \Rightarrow t' \rangle\ Suc(5)]$
show ?*case* **by**(*rule Suc(1)[OF 1 - nm1 \langle n = listsum\ ?ns \rangle]*)
(*simp-all add: \langle size\ ts' = size\ ts \rangle \langle size\ ns = size\ ts \rangle*)
qed

lemma *Red-term-pres-no-match-star*:
assumes $\forall i < length(ts::tm\ list). ts\ !\ i \Rightarrow * ts'\ !\ i$ **and** $size\ ts' = size\ ts$
and $no-match\ ps\ (map\ dterm\ ts)$
shows $no-match\ ps\ (map\ dterm\ ts')$
proof–
let ?*P* = $\%ns. size\ ns = size\ ts \wedge$
 $(\forall i < length\ ts. (ts!i, ts!i) : Red-term\ ^\wedge\ (ns!i))$
have $\exists ns. ?P\ ns$ **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]*: $\neg pure(term\ v)$
proof
assume $pure(term\ v)$ **thus** *False*
proof cases **qed** *auto*
qed

abbreviation $RedMLs :: tm\ list \Rightarrow tm\ list \Rightarrow bool$ (**infix** $[\Rightarrow*]$ 50) **where**

$ss \ [\Rightarrow^*] \ ts \equiv \text{size } ss = \text{size } ts \wedge (\forall i < \text{size } ss. ss!i \Rightarrow^* ts!i)$

fun $C\text{-}U\text{-args} :: \text{tm} \Rightarrow \text{tm list } (C_U'\text{-args})$ **where**
 $C_U\text{-args}(s \cdot t) = C_U\text{-args } s \ @ \ [t] \ |$
 $C_U\text{-args}(\text{term}(C_U \text{ nm } vs)) = \text{map term } (\text{rev } vs) \ |$
 $C_U\text{-args } - = []$

lemma $[simp]$: $C_U\text{-args}(C \text{ nm } \cdot\cdot \ ts) = ts$
by $(\text{induct } ts \ \text{rule:rev-induct}) \ \text{auto}$

lemma redts-term-cong : $v \Rightarrow^* v' \Longrightarrow \text{term } v \Rightarrow^* \text{term } v'$
apply $(\text{erule converse-rtrancl-induct})$
apply $(\text{rule rtrancl-refl})$
apply $(\text{fast intro: converse-rtrancl-into-rtrancl dest: ctat-term})$
done

lemma $C\text{-Red-term-ML}$:

$v \Rightarrow v' \Longrightarrow C\text{-normal}_{ML} \ v \Longrightarrow \text{dterm}_{ML} \ v = C \ \text{nm} \ \cdot\cdot \ ts$
 $\Longrightarrow \text{dterm}_{ML} \ v' = C \ \text{nm} \ \cdot\cdot \ \text{map dterm } (C_U\text{-args}(\text{term } v')) \wedge$
 $(\forall t \in \text{set}(C_U\text{-args}(\text{term } v)). \ C\text{-normal } t) \wedge$
 $C_U\text{-args}(\text{term } v) \ [\Rightarrow^*] \ C_U\text{-args}(\text{term } v') \wedge$
 $ts = \text{map dterm } (C_U\text{-args}(\text{term } v))$ **and**
 $(vs :: \text{ml list}) \Rightarrow vs' \Longrightarrow i < \text{length } vs \Longrightarrow vs \ ! \ i \Rightarrow^* vs' \ ! \ i$
apply $(\text{induct arbitrary: nm } ts \ \text{and } i \ \text{rule:Red-ml-Red-ml-list.inducts})$
apply $(\text{simp-all add:Red-ml-list-length del: map-map})$
apply $(\text{frule Red-ml-list-length})$
apply $(\text{simp add: redts-term-cong rev-nth del: map-map})$
apply $(\text{simp add:nth-Cons' r-into-rtrancl del: map-map})$
apply $(\text{simp add:nth-Cons'})$
done

lemma $C\text{-redt}$: $t \Rightarrow t' \Longrightarrow C\text{-normal } t \Longrightarrow$

$C\text{-normal } t' \wedge (\text{dterm } t = C \ \text{nm} \ \cdot\cdot \ ts \longrightarrow$
 $(\exists ts'. \ ts' = \text{map dterm } (C_U\text{-args } t') \wedge \text{dterm } t' = C \ \text{nm} \ \cdot\cdot \ ts' \ \&$
 $(\forall t \in \text{set}(C_U\text{-args } t). \ C\text{-normal } t) \wedge$
 $C_U\text{-args } t \ [\Rightarrow^*] \ C_U\text{-args } t' \wedge ts = \text{map dterm } (C_U\text{-args } t)))$
apply $(\text{induct arbitrary: } ts \ \text{nm rule:Red-term-hnf-induct})$
apply $(\text{simp-all del: map-map})$
apply $(\text{metis no-match-R-coincide rev-rev-ident})$
apply clarsimp
apply rule
apply $(\text{metis } C\text{-normal-ML-inv})$
apply clarify
apply $(\text{drule } (2) \ C\text{-Red-term-ML})$
apply clarsimp
apply clarsimp
apply $(\text{metis List.set.simps}(2) \ \text{insert-code mem-def predicate1D set-update-subset-insert})$
apply clarsimp

```

apply(rule)
apply (metis List.set.simps(2) insert-code mem-def predicate1D 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 simp
apply(rule-tac x=nm' in exI)
apply(erule-tac x=rs' in meta-allE)
apply(erule-tac x=nm' in meta-allE)
apply clarsimp
apply(metis Red-term-pres-no-match-star)
apply (auto simp:nth-list-update)
done

```

```

lemma C-redts:  $t \Rightarrow^* t' \implies C\text{-normal } t \implies$ 
   $C\text{-normal } t' \wedge (dterm\ t = C\ nm \ \cdot\ \cdot\ ts \longrightarrow$ 
   $(\exists\ ts'.\ dterm\ t' = C\ nm \ \cdot\ \cdot\ ts' \wedge$ 
   $(\forall\ t \in set(C_U\ args\ t). C\text{-normal } t) \wedge C_U\ args\ t [\implies^*] C_U\ args\ t' \wedge$ 
   $ts' = map\ dterm\ (C_U\ args\ t') \wedge ts = map\ dterm\ (C_U\ args\ t)))$ 
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
apply simp
apply (metis rtrancl-trans)
done

```

lemma no-match-preserved:

```

   $\forall t \in \text{set } ts. C\text{-normal } t \implies ts \ [\Rightarrow^*] \ ts'$ 
   $\implies \text{no-match } ps \ os \implies os = \text{map } dterm \ ts \implies \text{no-match } ps \ (\text{map } dterm \ ts')$ 
proof(induct ps os arbitrary: ts ts' rule: no-match.induct)
  case (1 ps os)
  obtain i nm nm' ps' os' where ps!i = C nm .. ps' i < size ps i < size os os!i
  = C nm' .. os' nm=nm'  $\longrightarrow$  no-match ps' os'
  using 1(4) no-match.simps[of ps os] by fastsimp
  note 1(5)[simp]
  have C-normal (ts ! i) using 1(2) (i < size os) by auto
  have ts!i  $\Rightarrow^*$  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)
  from C-redts [OF (ts!i  $\Rightarrow^*$  ts'!i) (C-normal (ts!i))] this obtain ss' rs rs' :: tm
  list
  where  $\forall t \in \text{set } rs. C\text{-normal } t$ 
  dterm (ts' ! i) = C nm' .. ss' length rs = length rs'
   $\forall i < \text{length } rs. rs ! i \Rightarrow^* rs' ! i \ ss' = \text{map } dterm \ rs' \ os' = \text{map } dterm \ rs$ 
  by fastsimp
show ?case
  apply(subst no-match.simps)
  apply(rule-tac x=i in exI)
  using prems
  apply clarsimp
  apply(rule 1(1)[of i nm' - nm' map dterm rs rs])
  apply simp-all
  done

```

qed

lemma Lam-Red-term-itE:

```

   $(\Lambda t, t') : \text{Red-term } \hat{\hat{i}} \implies \exists t''. t' = \Lambda t'' \wedge (t, t') : \text{Red-term } \hat{\hat{i}}$ 
apply(induct i arbitrary: t')apply simp
apply(erule rel-pow-Suc-E)
apply(erule Red-term.cases)
apply (simp-all)
apply blast+
done

```

lemma *Red-term-it*: $(V x \cdot rs, r) : Red-term \hat{\hat{i}}$
 $\implies \exists ts \ is. r = V x \cdot ts \wedge size\ ts = size\ rs \ \& \ size\ is = size\ rs \wedge$
 $(\forall j < size\ ts. (rs!j, ts!j) : Red-term \hat{\hat{(is!j)}} \wedge is!j \leq i)$
proof(*induct i arbitrary:rs*)
case 0
moreover
have $\exists is. length\ is = length\ rs \wedge$
 $(\forall j < size\ rs. (rs!j, rs!j) \in Red-term \hat{\hat{is!j}} \wedge is!j = 0)$ (**is** $\exists is. ?P\ is$)
proof
show $?P(replicate\ (size\ rs)\ 0)$ **by** *simp*
qed
ultimately show *?case* **by** *auto*
next
case (*Suc i rs*)
from $\langle (V x \cdot rs, r) \in Red-term \hat{\hat{Suc\ i}} \rangle$
obtain r' **where** $r' : V x \cdot rs \Rightarrow r'$ **and** $(r', r) \in Red-term \hat{\hat{i}}$
by (*metis rel-pow-Suc-D2*)
from r' **have** $\exists k < size\ rs. \exists s. rs!k \Rightarrow s \wedge r' = V x \cdot rs[k:=s]$
proof(*induct rs arbitrary: r' rule:rev-induct*)
case *Nil* **thus** *?case* **by**(*fastsimp elim: Red-term.cases*)
next
case (*snoc r rs*)
hence $(V x \cdot rs) \cdot r \Rightarrow r'$ **by** *simp*
thus *?case*
proof(*cases rule:Red-term.cases*)
case (*ctxt-At1 s s' t*)
then obtain $k\ s''$ **where** $k < length\ rs \ rs!k \Rightarrow s'' \wedge s' = V x \cdot rs[k := s'']$
using *snoc(1)* **by** *force*
then show *?thesis* (**is** $\exists k < ?n. \exists s. ?P\ k\ s$)
proof-
have $k < ?n \wedge ?P\ k\ s''$ **using** *prems*
by (*simp add:nth-append*) (*metis last-snoc butlast-snoc list-update-append1*)
thus *?thesis* **by** *blast*
qed
next
case (*ctxt-At2 t t' s*)
then show *?thesis* (**is** $\exists k < ?n. \exists s. ?P\ k\ s$)
proof-
have $size\ rs < ?n \wedge ?P\ (size\ rs)\ t'$ **using** *prems* **by** *simp*
thus *?thesis* **by** *blast*
qed
qed *auto*
qed
then obtain $k\ s$ **where** $k < size\ rs \ rs!k \Rightarrow s$ **and** [*simp*]: $r' = V x \cdot rs[k:=s]$ **by**
metis
from *Suc(1)*[*of rs[k:=s]*] $\langle (r', r) \in Red-term \hat{\hat{i}} \rangle$
show *?case* **using** $\langle k < size\ rs \rangle \langle rs!k \Rightarrow s \rangle$
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 rel-pow-Suc-I2 relpow.simps(2))
done
qed

lemma C-Red-term-it: (C nm .. rs, r) : Red-term ^^ i
   $\implies \exists ts\ is. r = C\ nm \ \cdot\ ts \wedge size\ ts = size\ rs \wedge size\ is = size\ rs \wedge$ 
   $(\forall j < size\ ts. (rs!j, ts!j) \in Red-term\ \hat{\wedge}(is!j) \wedge is!j \leq i)$ 
proof(induct i arbitrary:rs)
  case 0
  moreover
  have  $\exists is. length\ is = length\ rs \wedge$ 
   $(\forall j < size\ rs. (rs!j, rs!j) \in Red-term\ \hat{\wedge}\ is!j \wedge is!j = 0)$  (is  $\exists is. ?P\ is$ )
  proof
    show  $?P(replicate\ (size\ rs)\ 0)$  by simp
  qed
  ultimately show ?case by auto
next
  case (Suc i rs)
  from  $\langle (C\ nm \ \cdot\ rs, r) \in Red-term\ \hat{\wedge}\ Suc\ i \rangle$ 
  obtain r' where  $r': C\ nm \ \cdot\ rs \Rightarrow r'$  and  $(r', r) \in Red-term\ \hat{\wedge}\ i$ 
  by (metis rel-pow-Suc-D2)
  from r' have  $\exists k < size\ rs. \exists s. rs!k \Rightarrow s \wedge r' = C\ nm \ \cdot\ rs[k := s]$ 
  proof(induct rs arbitrary: r' rule:rev-induct)
    case Nil thus ?case by(fastsimp elim: Red-term.cases)
  next
  case (snoc r rs)
  hence  $(C\ nm \ \cdot\ rs) \cdot r \Rightarrow r'$  by simp
  thus ?case
  proof(cases rule:Red-term.cases)
    case (ctxt-At1 s s' t)
    then obtain k s'' where  $k < length\ rs \wedge rs!k \Rightarrow s'' \wedge s' = C\ nm \ \cdot\ rs[k := s'']$ 
    using snoc(1) by force
    then show ?thesis (is  $\exists k < ?n. \exists s. ?P\ k\ s$ )
    proof-
      have  $k < ?n \wedge ?P\ k\ s''$  using prems
      by (simp add:nth-append) (metis last-snoc butlast-snoc list-update-append1)
      thus ?thesis by blast
    qed
  next
  case (ctxt-At2 t t' s)
  then show ?thesis (is  $\exists k < ?n. \exists s. ?P\ k\ s$ )
  proof-
    have  $size\ rs < ?n \wedge ?P\ (size\ rs)\ t'$  using prems by simp
    thus ?thesis by blast
  qed
qed auto

```

```

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 rel-pow-Suc-I2 relpow.simps(2))
done
qed

```

lemma *pure-At*[simp]: $\text{pure}(s \cdot t) \longleftrightarrow \text{pure } s \wedge \text{pure } t$
by(*fastsimp elim: pure.cases*)

lemma *pure-foldl-At*[simp]: $\text{pure}(s \cdot\cdot ts) \longleftrightarrow \text{pure } s \wedge (\forall t \in \text{set } ts. \text{pure } t)$
by(*induct ts arbitrary: s auto*)

lemma *nbe-C-normal-ML*:

assumes $\text{term } v \Rightarrow^* t'$ *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-normalML 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 (s,t') : Red-term ^ i' and [arith]: i' ≤ i
  by (metis eq-imp-le less(5) Suc rel-pow-Suc-D2)
  hence term v ⇒ s using Suc less by simp
  thus ?thesis
  proof cases
  case (term-C nm vs)
  have 0: no-match-compR nm vs using prems 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:  $\forall t \in \text{set } ts. \text{pure } t$  using  $\langle \text{pure } t' \rangle$  by simp
{ fix  $i$  assume  $i < \text{size } vs$ 
  moreover hence  $(\text{term}((\text{rev } vs)!i), ts!i) : \text{Red-term}^{\wedge}(ks!i)$  by  $(\text{metis } sz)$ 
  ultimately have normal  $(ts!i)$ 
  apply -
  apply  $(\text{rule } \text{less}(1))$ 
  prefer 5 apply assumption
  using  $sz$  Suc apply fastsimp
  apply  $(\text{rule } \text{refl})$ 
  using  $\text{less term-C}$ 
  apply auto
  apply  $(\text{metis } \text{in-set-conv-nth } \text{length-rev } \text{set-rev})$ 
  apply  $(\text{metis } \text{in-set-conv-nth } \text{pure-ts } sz)$ 
  done
} note 2 = this
have 3: no-match-R  $nm$   $(\text{map } \text{dterm } (\text{map } \text{term } (\text{rev } vs)))$ 
  apply  $(\text{subst } \text{map-dterm-term})$ 
  apply  $(\text{rule } \text{no-match-R-coincide})$  using 0 by simp
have 4:  $\text{map } \text{term } (\text{rev } vs) [\Rightarrow^*] ts$ 
proof -
  have  $(C \text{ nm} \cdot \text{map } \text{term } (\text{rev } vs), t') : \text{Red-term}^{\wedge} i'$  using prems by auto
  from C-Red-term-it [OF this] obtain  $ts'$  is
  where  $t' = C \text{ nm} \cdot ts'$   $\text{length } ts' = ?n \wedge \text{length } is = ?n \wedge$ 
   $(\forall j < ?n. (\text{map } \text{term } (\text{rev } vs) ! j, ts' ! j) \in \text{Red-term}^{\wedge} is ! j \wedge is ! j \leq$ 
i')
  using prems by auto
  from  $\langle t' = C \text{ nm} \cdot ts' \rangle \langle t' = C \text{ nm} \cdot ts \rangle$  have  $ts = ts'$  by simp
  show thesis using prems by  $(\text{auto } \text{simp: } \text{rtrancl-is-UN-rel-pow})$ 
qed
have 5:  $\forall t \in \text{set}(\text{map } \text{term } vs). C\text{-normal } t$  using prems by auto
have no-match-R  $nm$   $(\text{map } \text{dterm } ts)$ 
  apply auto
  apply  $(\text{subgoal-tac } \text{no-match } aa (\text{map } \text{dterm } (\text{map } \text{term } (\text{rev } vs))))$ 
  prefer 2
  using 3 apply blast
  using 4 5 no-match-preserved [OF - - - refl, of  $\text{map } \text{term } (\text{rev } vs) ts]$  by
simp
hence 6: no-match-R  $nm$   $ts$  by  $(\text{metis } \text{map-dterm-pure} [\text{OF } \text{pure-ts}])$ 
then show normal  $t'$ 
  apply simp
  apply  $(\text{rule } \text{normal.intros}(3))$ 
  using 2  $sz$  apply  $(\text{fastsimp } \text{simp:set-conv-nth})$ 
  apply auto
  apply  $(\text{subgoal-tac } \text{no-match } aa (\text{take } (\text{size } aa) ts))$ 
  apply  $(\text{metis } \text{no-match})$ 
  apply  $(\text{fastsimp } \text{intro:no-match-take})$ 
  done
next
case  $(\text{term-V } x \text{ vs})$ 

```

```

    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-normalML (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)) (VU 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-normalML ?u using less term-Clo by (simp)
      from less(1)[OF (i' < k) refl (C-normalML ?u) (pure t'') 1] show ?thesis
    by (simp add:normal.intros)
  next
    case (ctxt-term u u')
    have i' < k using (k = Suc i) by arith
    have C-normalML u' by (rule C-normal-ML-inv)
      (insert less ctxt-term, simp-all)
    have (term u', t') ∈ Red-term ^^ i' using prems by auto
    from less(1)[OF (i' < k) refl (C-normalML u') (pure t') this] show ?thesis
  .
  qed auto
  qed
  qed
}
}
thus ?thesis using assms(2-) rtrancl-imp-rel-pow[OF assms(1)] by blast
qed

```

lemma C-normal-ML-compile:

$pure\ t \implies \forall i. C\text{-normal}_{ML}(\sigma\ i) \implies C\text{-normal}_{ML}\ (compile\ t\ \sigma)$

by(*induct t arbitrary: σ*) (*simp-all add: C-normal-ML-lift-ML*)

corollary *nbe-normal*:

pure t \implies term(comp-fixed t) \Rightarrow^ t' \implies pure t' \implies 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 $C_U s :: ml \Rightarrow bool$ **where**
 $C_U s(C_U nm vs) = (size vs = arity nm \wedge (\forall v \in set vs. C_U s v)) \mid$
 $C_U s - = False$

lemma *size-foldl-At*: $size(C nm \cdot ts) = size ts + listsum(map size ts)$
by(*induct ts rule:rev-induct*) *auto*

lemma *termination-linpats*:

i < length ts \implies ts!i = C nm \cdot ts'
 $\implies length ts' + listsum(map size ts') < length ts + listsum(map size ts)$
apply(*subgoal-tac C nm \cdot ts' : set ts*)
prefer 2 **apply** (*metis in-set-conv-nth*)
apply(*drule listsum-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* $\Rightarrow bool$ **where**
 $linpats ts \longleftrightarrow$
 $(ALL i < size ts. (EX x. ts!i = V x) \mid$
 $(EX nm ts'. ts!i = C nm \cdot ts' \wedge arity nm = size ts' \wedge linpats ts')) \&$
 $(ALL i < size ts. ALL j < size ts. i \neq j \longrightarrow fv(ts!i) \cap fv(ts!j) = \{\})$
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*:

$size\ xs = size\ ys \implies (xs=ys) = (ALL\ i < size\ xs.\ xs!i = ys!i)$
by (*metis nth-equalityI*)

lemma *pattern-subst-ML-coincidence*:
 $pattern\ t \implies ALL\ i:fv\ t.\ \sigma\ i = \sigma'\ i$
 $\implies subst_ML\ \sigma\ (comp_pat\ t) = subst_ML\ \sigma'\ (comp_pat\ t)$
by(*induct pred:pattern*) *auto*

lemma *linpats-pattern*: $linpats\ ts \implies patterns\ ts$

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 $(EX\ x.\ t = V\ x) \mid (EX\ nm\ ts' . t = C\ nm \ \cdot\ ts' \wedge arity\ nm = size\ ts'$

& $linpats\ ts')$

(**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*:

$length\ ps = length\ vs \implies no_match_{ML}\ ps\ (rev\ vs) \implies no_match_{ML}\ (rev\ ps)\ vs$

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 (*fastsimp simp:rev-nth*)

done

lemma *no-match-ML-aux*:

$ALL\ v : set\ cvs.\ C_{Us}\ v \implies linpats\ ps \implies size\ ps = size\ cvs \implies$

$\forall\ \sigma.\ map\ (subst_{ML}\ \sigma)\ (map\ comp_pat\ ps) \neq cvs \implies$

$no_match_{ML}\ (map\ comp_pat\ ps)\ cvs$

apply(*induct ps arbitrary: cvs rule:linpats.induct*)

apply(*frule linpats-pattern*)

apply(*subst (asm) linpats.simps*) **back**

apply *auto*

apply(*case-tac ALL i < size ts. EX sigma. subst_{ML} sigma (comp-pat (ts!i)) = cvs!i*)

apply(*clarsimp simp:Skolem-list-nth*)

apply(*rename-tac sigma*)

apply(*erule-tac x=%x. (sigma!(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-mp)
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  $\sim (EX x. ts!i = V x)$ )
  prefer 2
  apply fastsimp
apply(subgoal-tac EX nm ts'. ts!i = C nm .. ts' & size ts' = arity nm & linpats
ts'))
  prefer 2
  apply fastsimp
apply clarsimp
apply(rule-tac x=nm in exI)
apply(subgoal-tac EX nm' vs'. cvs!i = CU nm' vs' & size vs' = arity nm' & (ALL
v' : set vs'. CU 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-matchML (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-map 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.