Sorted Terms*

Akihisa Yamada

National Institute of Advanced Industrial Science and Technology, Japan

René Thiemann

University of Innsbruck, Austria

June 3, 2024

Abstract

This entry provides a basic library for many-sorted terms and algebras. We view sorted sets just as partial maps from elements to sorts, and define sorted set of terms reusing the data type from the existing library of (unsorted) first order terms. All the existing functionality, such as substitutions and contexts, can be reused without any modifications. We provide predicates stating what substitutions or contexts are considered sorted, and prove facts that they preserve sorts as expected.

Contents

*This research was partly supported by the Austrian Science Fund (FWF) project I 5943.

1 Introduction

This entry extends the First-Order Terms [1] entry with many-sorted terms. Instead of defining a new datatype for sorted terms, we just define sorted sets over the existing datatype of unsorted terms. We do not even introduce our type for sorted sets: we just view sorted sets as partial maps from elements to their sorts.

Part of the entry is presented in [2].

theory *Sorted-Sets*
  imports
    Main
    HOL−Library.FuncSet
    HOL−Library.Monad-Syntax
    Complete-Non-Orders.Binary-Relations
begin

2 Auxiliary Lemmas

lemma *ex-set-conv-ex-nth*:
  $(\exists x \in set\ xs.\ P\ x) = (\exists i.\ i < length\ xs \wedge P\ (xs\ !\ i))$
  by (*auto simp add*: *set-conv-nth*)

lemma *Ball-Pair-conv*: $(\forall (x,y) \in R.\ P\ x\ y) \longleftrightarrow (\forall x\ y.\ (x,y) \in R \longrightarrow P\ x\ y)$ **by** *auto*

lemma *Some-eq-bind-conv*: $(Some\ x = f \gg= g) = (\exists y.\ f = Some\ y \wedge g\ y = Some\ x)$
  by (*fold bind-eq-Some-conv*, *auto*)

lemma *length-le-nth-append*: $length\ xs \le n \Longrightarrow (xs@ys)!n = ys!(n-length\ xs)$
  by (*simp add*: *nth-append*)

lemma *list-all2-same-left*:

$\forall a' \in set\ as.\ a' = a \Longrightarrow list\text{-}all2\ r\ as\ bs \longleftrightarrow length\ as = length\ bs \wedge (\forall b \in set\ bs.\ r\ a\ b)$
by (*auto simp*: *list-all2-conv-all-nth all-set-conv-all-nth*)

lemma *list-all2-same-leftI*:
$\forall a' \in set\ as.\ a' = a \Longrightarrow length\ as = length\ bs \Longrightarrow \forall b \in set\ bs.\ r\ a\ b \Longrightarrow list\text{-}all2\ r\ as\ bs$
by (*auto simp*: *list-all2-same-left*)

lemma *list-all2-same-right*:
$\forall b' \in set\ bs.\ b' = b \Longrightarrow list\text{-}all2\ r\ as\ bs \longleftrightarrow length\ as = length\ bs \wedge (\forall a \in set\ as.\ r\ a\ b)$
by (*auto simp*: *list-all2-conv-all-nth all-set-conv-all-nth*)

lemma *list-all2-same-rightI*:
$\forall b' \in set\ bs.\ b' = b \Longrightarrow length\ as = length\ bs \Longrightarrow \forall a \in set\ as.\ r\ a\ b \Longrightarrow list\text{-}all2\ r\ as\ bs$
by (*auto simp*: *list-all2-same-right*)

lemma *list-all2-all-all*:
$\forall a \in set\ as.\ \forall b \in set\ bs.\ r\ a\ b \Longrightarrow list\text{-}all2\ r\ as\ bs \longleftrightarrow length\ as = length\ bs$
by (*auto simp*: *list-all2-conv-all-nth all-set-conv-all-nth*)

lemma *list-all2-indep1*:
$list\text{-}all2\ (\lambda a\ b.\ P\ b)\ as\ bs \longleftrightarrow length\ as = length\ bs \wedge (\forall b \in set\ bs.\ P\ b)$
by (*auto simp*: *list-all2-conv-all-nth all-set-conv-all-nth*)

lemma *list-all2-indep2*:
$list\text{-}all2\ (\lambda a\ b.\ P\ a)\ as\ bs \longleftrightarrow length\ as = length\ bs \wedge (\forall a \in set\ as.\ P\ a)$
by (*auto simp*: *list-all2-conv-all-nth all-set-conv-all-nth*)

lemma *list-all2-replicate*[*simp*]:
$list\text{-}all2\ r\ (replicate\ n\ x)\ ys \longleftrightarrow length\ ys = n \wedge (\forall y \in set\ ys.\ r\ x\ y)$
$list\text{-}all2\ r\ xs\ (replicate\ n\ y) \longleftrightarrow length\ xs = n \wedge (\forall x \in set\ xs.\ r\ x\ y)$
by (*auto simp*: *list-all2-conv-all-nth all-set-conv-all-nth*)

lemma *list-all2-choice-nth*: **assumes** $\forall i < length\ xs.\ \exists y.\ r\ (xs!i)\ y$ **shows** $\exists ys.\ list\text{-}all2\ r\ xs\ ys$
proof−
from *assms* **have** $\forall i \in \{0..<length\ xs\}.\ \exists y.\ r\ (xs!i)\ y$ **by** *auto*
from *finite-set-choice*[*OF - this*]
obtain f **where** $\forall i < length\ xs.\ r\ (xs\ !\ i)\ (f\ i)$ **by** (*auto simp*: *Ball-def*)
then have $list\text{-}all2\ r\ xs\ (map\ f\ [0..<length\ xs])$ **by** (*auto simp*: *list-all2-conv-all-nth*)
then show *?thesis* **by** *auto*
qed

lemma *list-all2-choice*: $\forall x \in set\ xs.\ \exists y.\ r\ x\ y \Longrightarrow \exists ys.\ list\text{-}all2\ r\ xs\ ys$
using *list-all2-choice-nth* **by** (*auto simp*: *all-set-conv-all-nth*)

lemma *list-all2-concat*:
list-all2 (*list-all2 r*) *ass bss* $\Longrightarrow$ *list-all2 r* (*concat ass*) (*concat bss*)
by (*induct rule*:*list-all2-induct*, *auto intro*!: *list-all2-appendI*)

lemma *those-eq-None*[*simp*]: *those as* = *None* $\longleftrightarrow$ *None* $\in$ *set as* **by** (*induct as*, *auto split*:*option.split*)

lemma *those-eq-Some*[*simp*]: *those xos* = *Some xs* $\longleftrightarrow$ *xos* = *map Some xs*
by (*induct xos arbitrary*:*xs*, *auto split*:*option.split-asm*)

lemma *those-map-Some*[*simp*]: *those* (*map Some xs*) = *Some xs* **by** *simp*

lemma *those-append*:
those (*as* @ *bs*) = *do* {*xs* $\leftarrow$ *those as*; *ys* $\leftarrow$ *those bs*; *Some* (*xs*@*ys*)}
by (*auto simp*: *those-eq-None split*: *bind-split*)

lemma *those-Cons*:
those (*a*#*as*) = *do* {*x* $\leftarrow$ *a*; *xs* $\leftarrow$ *those as*; *Some* (*x* # *xs*)}
by (*auto split*: *option.split bind-split*)

lemma *map-singleton-o*[*simp*]: (λx. [*x*]) $\circ$ *f* = (λx. [*f x*]) **by** *auto*

lemmas *list-3-cases* = *remdups-adj.cases*

lemma *in-set-updateD*: *x* $\in$ *set* (*xs*[*n* := *y*]) $\Longrightarrow$ *x* $\in$ *set xs* $\vee$ *x* = *y*
by (*auto dest*: *subsetD*[*OF set-update-subset-insert*])

lemma *map-nth'*: *length xs* = *n* $\Longrightarrow$ *map* (*nth xs*) [*0*..<*n*] = *xs*
using *map-nth* **by** *auto*

lemma *product-lists-map-map*: *product-lists* (*map* (*map f*) *xss*) = *map* (*map f*) (*product-lists xss*)
by (*induct xss*, *auto simp*: *Cons o-def map-concat*)

lemma (**in** *monoid-add*) *sum-list-concat*: *sum-list* (*concat xs*) = *sum-list* (*map sum-list xs*)
by (*induct xs*, *auto*)

context *semiring-1* **begin**

lemma *prod-list-map-sum-list-distrib*:
shows *prod-list* (*map sum-list xss*) = *sum-list* (*map prod-list* (*product-lists xss*))
by (*induct xss*, *simp-all add*: *map-concat o-def sum-list-concat sum-list-const-mult sum-list-mult-const*)

lemma *prod-list-sum-list-distrib*:
($\prod xs \leftarrow xss.\ \sum x \leftarrow xs.\ f\ x$) = ($\sum xs \leftarrow$ *product-lists xss*. $\prod x{\leftarrow}xs.\ f\ x$)
using *prod-list-map-sum-list-distrib*[*of map* (*map f*) *xss*]
by (*simp add*: *o-def product-lists-map-map*)

end

lemma *ball-set-bex-set-distrib*:
 $(\forall xs{\in}set\ xss.\ \exists x{\in}set\ xs.\ f\ x) \longleftrightarrow (\exists xs{\in}set\ (product\text{-}lists\ xss).\ \forall x{\in}set\ xs.\ f\ x)$
 by (*induct xss*, *auto*)

lemma *bex-set-ball-set-distrib*:
 $(\exists xs{\in}set\ xss.\ \forall x{\in}set\ xs.\ f\ x) \longleftrightarrow (\forall xs{\in}set\ (product\text{-}lists\ xss).\ \exists x{\in}set\ xs.\ f\ x)$
 by (*induct xss*, *auto*)

declare *upt-Suc*[*simp del*]

lemma *map-nth-Cons*: $map\ (nth\ (x\#xs))\ [0..{<}n] = (case\ n\ of\ 0 \Rightarrow []\ |\ Suc\ n \Rightarrow x\ \#\ map\ (nth\ xs)\ [0..{<}n])$
 by (*auto simp*:*map-upt-Suc split*: *nat.split*)

lemma *upt-0-Suc-Cons*: $[0..{<}Suc\ i] = 0\ \#\ map\ Suc\ [0..{<}i]$
 using *map-upt-Suc*[*of id*] **by** *simp*

lemma *upt-map-add*: $i \le j \Longrightarrow [i..{<}j] = map\ (\lambda k.\ k + i)\ [0..{<}j-i]$
 by (*simp add*: *map-add-upt*)

lemma *map-nth-append*:
 $map\ (nth\ (xs\ @\ ys))\ [0..{<}n] =$
 $(if\ n < length\ xs\ then\ map\ (nth\ xs)\ [0..{<}n]\ else\ xs\ @\ map\ (nth\ ys)\ [0..{<}n-length\ xs])$
 by (*induct xs arbitrary*: *n*, *auto simp*: *map-nth-Cons split*: *nat.split*)

lemma *all-dom*: $(\forall x \in dom\ f.\ P\ x) \longleftrightarrow (\forall x\ y.\ f\ x = Some\ y \longrightarrow P\ x)$ **by** *auto*

lemma *trancl-Collect*: $\{(x,y).\ r\ x\ y\}^+ = \{(x,y).\ tranclp\ r\ x\ y\}$
 by (*simp add*: *tranclp-unfold*)

lemma *restrict-submap*[*intro*!]: $A\ |`\ S \subseteq_m A$
 by (*auto simp*: *restrict-map-def map-le-def domIff*)

lemma *restrict-map-mono-left*: $A \subseteq_m A' \Longrightarrow A\ |`\ S \subseteq_m A'\ |`\ S$
 and *restrict-map-mono-right*: $S \subseteq S' \Longrightarrow A\ |`\ S \subseteq_m A\ |`\ S'$
 by (*auto simp*: *map-le-def*)

3 Sorted Sets and Maps

declare *domIff*[*iff del*]

We view sorted sets just as partial maps from elements to their sorts. We just introduce the following notation:

definition *hastype* (((-) :/ (-) *in*/ (-)) [*50*,*61*,*51*]*50*)
 where $a : \sigma\ in\ A \equiv A\ a = Some\ \sigma$

abbreviation *all-hastype* $\sigma\ A\ P \equiv \forall\, a.\ a : \sigma\ in\ A \longrightarrow P\ a$
abbreviation *ex-hastype* $\sigma\ A\ P \equiv \exists\, a.\ a : \sigma\ in\ A \wedge P\ a$

syntax
all-hastype :: $'pttrn \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ $(\forall\,\text{-}\ \text{:/}\ \text{-}\ in/\ \text{-./}\ \text{-}\ [50,51,51,10]10)$
ex-hastype :: $'pttrn \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ $(\exists\,\text{-}\ \text{:/}\ \text{-}\ in/\ \text{-./}\ \text{-}\ [50,51,51,10]10)$

translations
$\forall\, a : \sigma\ in\ A.\ e \rightleftharpoons CONST\ all\text{-}hastype\ \sigma\ A\ (\lambda a.\ e)$
$\exists\, a : \sigma\ in\ A.\ e \rightleftharpoons CONST\ ex\text{-}hastype\ \sigma\ A\ (\lambda a.\ e)$

lemmas *hastypeI* = *hastype-def*[*unfolded atomize-eq*, *THEN iffD2*]
lemmas *hastypeD*[*dest*] = *hastype-def*[*unfolded atomize-eq*, *THEN iffD1*]
lemmas *eq-Some-iff-hastype* = *hastype-def*[*symmetric*]

lemma *has-same-type*: **assumes** $a : \sigma\ in\ A$ **shows** $a : \sigma'\ in\ A \longleftrightarrow \sigma' = \sigma$
using *assms* **by** (*unfold hastype-def*, *auto*)

lemma *sset-eqI*: **assumes** $(\bigwedge a\ \sigma.\ a : \sigma\ in\ A \longleftrightarrow a : \sigma\ in\ B)$ **shows** $A = B$
proof (*intro ext*)
fix *a* **show** $A\ a = B\ a$ **using** *assms* **apply** (*cases A a*, *auto simp*: *hastype-def*)
by (*metis option.exhaust*)
qed

lemma *in-dom-iff-ex-type*: $a \in dom\ A \longleftrightarrow (\exists\,\sigma.\ a : \sigma\ in\ A)$ **by** (*auto simp*: *hastype-def domIff*)

lemma *in-dom-hastypeE*: $a \in dom\ A \Longrightarrow (\bigwedge\sigma.\ a : \sigma\ in\ A \Longrightarrow thesis) \Longrightarrow thesis$
by (*auto simp*: *hastype-def domIff*)

lemma *hastype-imp-dom*[*simp*]: $a : \sigma\ in\ A \Longrightarrow a \in dom\ A$ **by** (*auto simp*: *domIff*)

lemma *untyped-imp-not-hastype*: $A\ a = None \Longrightarrow \neg\ a : \sigma\ in\ A$ **by** *auto*

lemma *nex-hastype-iff*: $(\nexists\,\sigma.\ a : \sigma\ in\ A) \longleftrightarrow A\ a = None$ **by** (*auto simp*: *hastype-def*)

lemma *all-dom-iff-all-hastype*: $(\forall\, x \in dom\ A.\ P\ x) \longleftrightarrow (\forall\, x\ \sigma.\ x : \sigma\ in\ A \longrightarrow P\ x)$
by (*simp add*: *all-dom hastype-def*)

abbreviation *hastype-list* $(((\text{-})\ :_l/\ (\text{-})\ in/\ (\text{-}))\ [50,61,51]50)$
where $as :_l \sigma s\ in\ A \equiv list\text{-}all2\ (\lambda a\ \sigma.\ a : \sigma\ in\ A)\ as\ \sigma s$

lemma *has-same-type-list*:
$as :_l \sigma s\ in\ A \Longrightarrow as :_l \sigma s'\ in\ A \longleftrightarrow \sigma s' = \sigma s$
proof (*induct as arbitrary*: $\sigma s\ \sigma s'$)
case *Nil*
then show *?case* **by** *auto*

```
next
  case (Cons a as)
  then show ?case by (auto simp: has-same-type list-all2-Cons1)
qed

lemma hastype-list-iff-those: as :l σs in A ⟷ those (map A as) = Some σs
proof (induct as arbitrary:σs)
  case Nil
  then show ?case by auto
next
  case IH: (Cons a as σσs)
  show ?case
  proof (cases σσs)
    case [simp]: Nil
    show ?thesis by (auto split:option.split)
  next
    case [simp]: (Cons σ σs)
    from IH show ?thesis by (auto intro!:hastypeI split: option.split)
  qed
qed

lemmas hastype-list-imp-those[simp] = hastype-list-iff-those[THEN iffD1]

lemma hastype-list-imp-lists-dom: xs :l σs in A ⟹ xs ∈ lists (dom A)
  by (auto simp: list-all2-conv-all-nth in-set-conv-nth hastype-def)

lemma subsset: A ⊆m A' ⟷ (∀ a σ. a : σ in A ⟶ a : σ in A')
  by(auto simp: Ball-def map-le-def hastype-def domIff)

lemmas subssetI = subsset[THEN iffD2, rule-format]
lemmas subssetD = subsset[THEN iffD1, rule-format]

lemma subsset-hastype-listD: A ⊆m A' ⟹ as :l σs in A ⟹ as :l σs in A'
  by (auto simp: list-all2-conv-all-nth subssetD)

lemma has-same-type-in-subsset:
  a : σ in A' ⟹ A ⊆m A' ⟹ a : σ' in A ⟹ σ' = σ
  by (auto dest!: subssetD simp: has-same-type)

lemma has-same-type-in-dom-subsset:
  a : σ in A' ⟹ A ⊆m A' ⟹ a ∈ dom A ⟷ a : σ in A
  by (auto simp: in-dom-iff-ex-type dest: has-same-type-in-subsset)

lemma hastype-restrict: a : σ in A |‘ S ⟷ a ∈ S ∧ a : σ in A
  by (auto simp: restrict-map-def hastype-def)

lemma hastype-the-simp[simp]: a : σ in A ⟹ the (A a) = σ
  by (auto)
```

lemma hastype-in-Some[simp]: x : σ in Some ⟷ x = σ **by** (auto simp: hastype-def)

lemma hastype-in-upd[simp]: x : σ in A(y ↦ τ) ⟷ (if x = y then σ = τ else x : σ in A)
  by (auto simp: hastype-def)

lemma all-set-hastype-iff-those: ∀ a ∈ set as. a : σ in A ⟹
  those (map A as) = Some (replicate (length as) σ)
  by (induct as, auto)

The partial version of list nth:

primrec safe-nth **where**
    safe-nth [] - = None
  | safe-nth (a#as) n = (case n of 0 ⇒ Some a | Suc n ⇒ safe-nth as n)

lemma safe-nth-simp[simp]: i < length as ⟹ safe-nth as i = Some (as ! i)
  by (induct as arbitrary:i, auto split:nat.split)

lemma safe-nth-None[simp]:
  length as ≤ i ⟹ safe-nth as i = None
  by (induct as arbitrary:i, auto split:nat.split)

lemma safe-nth: safe-nth as i = (if i < length as then Some (as ! i) else None)
  by auto

lemma safe-nth-eq-SomeE:
 safe-nth as i = Some a ⟹ (i < length as ⟹ as ! i = a ⟹ thesis) ⟹ thesis
  by (cases i < length as, auto)

lemma dom-safe-nth[simp]: dom (safe-nth as) = {0..<length as}
  by (auto simp: domIff elim!: safe-nth-eq-SomeE)

lemma safe-nth-replicate[simp]:
  safe-nth (replicate n a) i = (if i < n then Some a else None)
  by auto

lemma safe-nth-append:
  safe-nth (ls@rs) i = (if i < length ls then Some (ls!i) else safe-nth rs (i − length ls))
  by (cases i < length (ls@rs), auto simp: nth-append)

lemma hastype-in-safe-nth[simp]: i : σ in safe-nth σs ⟷ i < length σs ∧ σ = σs!i
  by (auto simp: hastype-def safe-nth)

lemmas hastype-in-safe-nthE = safe-nth-eq-SomeE[folded hastype-def]

lemma hastype-in-o[simp]: a : σ in A ∘ f ⟷ f a : σ in A **by** (simp add: hastype-def)

definition *o-sset* (**infix** ∘*s* *55*) **where**
f ∘*s* *A* ≡ *map-option f* ∘ *A*

lemma *hastype-in-o-sset*: $a : \sigma'$ *in* $f \circ s\ A \longleftrightarrow (\exists \sigma.\ a : \sigma$ *in* $A \land \sigma' = f\ \sigma)$
by (*auto simp*: *o-sset-def hastype-def*)

lemma *hastype-in-o-ssetI*: $a : \sigma$ *in* $A \Longrightarrow f\ \sigma = \sigma' \Longrightarrow a : \sigma'$ *in* $f \circ s\ A$
by (*auto simp*: *o-sset-def hastype-def*)

lemma *hastype-in-o-ssetD*: $a : \tau$ *in* $f \circ s\ A \Longrightarrow \exists \sigma.\ a : \sigma$ *in* $A \land \tau = f\ \sigma$
by (*auto simp*: *o-sset-def hastype-def*)

lemma *hastype-in-o-ssetE*: $a : \tau$ *in* $f \circ s\ A \Longrightarrow (\bigwedge\sigma.\ a : \sigma$ *in* $A \Longrightarrow \tau = f\ \sigma \Longrightarrow$ *thesis*$) \Longrightarrow$ *thesis*
by (*auto simp*: *o-sset-def hastype-def*)

lemma *o-sset-restrict-sset-assoc*[*simp*]: $f \circ s\ (A\ |`\ X) = (f \circ s\ A)\ |`\ X$
by (*auto simp*: *o-sset-def restrict-map-def*)

lemma *id-o-sset*[*simp*]: *id* ∘*s* $A = A$
and *identity-o-sset*[*simp*]: $(\lambda x.\ x) \circ s\ A = A$
by (*auto simp*: *o-sset-def map-option.id map-option.identity*)

lemma *o-ssetI*: $A\ x =$ *Some* $y \Longrightarrow z = f\ y \Longrightarrow (f \circ s\ A)\ x =$ *Some* z **by** (*auto simp*: *o-sset-def*)

lemma *o-ssetE*: $(f \circ s\ A)\ x =$ *Some* $z \Longrightarrow (\bigwedge y.\ A\ x =$ *Some* $y \Longrightarrow z = f\ y \Longrightarrow$ *thesis*$) \Longrightarrow$ *thesis*
by (*auto simp*: *o-sset-def*)

lemma *dom-o-sset*[*simp*]: *dom* $(f \circ s\ A) =$ *dom* A
by (*auto intro*!: *o-ssetI elim*!: *o-ssetE simp*: *domIff*)

lemma *safe-nth-map*: *safe-nth* (*map f as*) = *f* ∘*s* *safe-nth as*
by (*auto simp*: *safe-nth o-sset-def*)

notation *Map.empty* (∅)
lemma *safe-nth-Nil*[*simp*]: *safe-nth* [] = ∅ **by** *auto*

lemma *o-sset-empty*[*simp*]: *f* ∘*s* ∅ = ∅ **by** (*auto simp*: *o-sset-def*)

lemma *hastype-in-empty*[*simp*]: $\neg x : \sigma$ *in* ∅ **by** (*auto simp*: *hastype-def*)

3.1 Maps between Sorted Sets

locale *sort-preserving* = **fixes** $f :: 'a \Rightarrow 'b$ **and** $A :: 'a \rightharpoonup 's$
assumes *same-value-imp-same-type*: $a : \sigma$ *in* $A \Longrightarrow b : \tau$ *in* $A \Longrightarrow f\ a = f\ b \Longrightarrow \sigma = \tau$

begin

lemma *same-value-imp-in-dom-iff*:
assumes *fafa'*: $f\ a = f\ a'$ **and** *a*: $a : \sigma\ \ in\ A$ **shows** *a'*: $a' \in\ dom\ A \longleftrightarrow a' : \sigma\ in\ A$
using *same-value-imp-same-type*[*OF a - fafa'*] **by** (*auto elim*!: *in-dom-hastypeE*)

end

lemma *sort-preserving-cong*:
$A = A' \Longrightarrow (\bigwedge a\ \sigma.\ a : \sigma\ \ in\ A \Longrightarrow f\ a = f'\ a) \Longrightarrow$ *sort-preserving* $f\ A \longleftrightarrow$ *sort-preserving* $f'\ A'$
by (*auto simp*: *sort-preserving-def*)

lemma *inj-on-dom-imp-sort-preserving*:
assumes *inj-on* f (*dom* A) **shows** *sort-preserving* $f\ A$
proof *unfold-locales*
fix $a\ b\ \sigma\ \tau$
assume *a*: $a : \sigma\ \ in\ A$ **and** *b*: $b : \tau\ \ in\ A$ **and** *eq*: $f\ a = f\ b$
with *inj-onD*[*OF assms*] **have** $a = b$ **by** *auto*
with $a\ b$ **show** $\sigma = \tau$ **by** (*auto simp*: *has-same-type*)
qed

lemma *inj-imp-sort-preserving*:
assumes *inj f* **shows** *sort-preserving* $f\ A$
using *assms* **by** (*auto intro*!: *inj-on-dom-imp-sort-preserving simp*: *inj-on-def*)

locale *sorted-map* =
fixes $f :: 'a \Rightarrow 'b$ **and** $A :: 'a \rightharpoonup 's$ **and** $B :: 'b \rightharpoonup 's$
assumes *sorted-map*: $\bigwedge a\ \sigma.\ a : \sigma\ \ in\ A \Longrightarrow f\ a : \sigma\ \ in\ B$
begin

lemma *target-has-same-type*: $a : \sigma\ \ in\ A \Longrightarrow f\ a : \tau\ \ in\ B \longleftrightarrow \sigma = \tau$
by (*auto simp*:*has-same-type dest*!: *sorted-map*)

lemma *target-dom-iff-hastype*:
$a : \sigma\ \ in\ A \Longrightarrow f\ a \in\ dom\ B \longleftrightarrow f\ a : \sigma\ \ in\ B$
by (*auto simp*: *in-dom-iff-ex-type target-has-same-type*)

lemma *source-dom-iff-hastype*:
$f\ a : \sigma\ \ in\ B \Longrightarrow a \in\ dom\ A \longleftrightarrow a : \sigma\ \ in\ A$
by (*auto simp*: *in-dom-iff-ex-type target-has-same-type*)

lemma *elim*:
assumes *a*: $(\bigwedge a\ \sigma.\ a : \sigma\ \ in\ A \Longrightarrow f\ a : \sigma\ \ in\ B) \Longrightarrow P$
shows P
using *a* **by** (*auto simp*: *sorted-map*)

sublocale *sort-preserving*

apply *unfold-locales*
by (*auto simp add*: *sorted-map dest*!: *target-has-same-type*)

lemma *funcset-dom*: $f : dom\ A \to dom\ B$
using *sorted-map*[*unfolded hastype-def*] **by** (*auto simp*: *domIff*)

lemma *sorted-map-list*: $as :_l \sigma s\ in\ A \Longrightarrow map\ f\ as :_l \sigma s\ in\ B$
by (*auto simp*: *list-all2-conv-all-nth sorted-map*)

lemma *in-dom*: $a \in dom\ A \Longrightarrow f\ a \in dom\ B$ **by** (*auto elim*!: *in-dom-hastypeE dest*!:*sorted-map*)

end

notation *sorted-map* $(\text{-} :_s (/\ \text{-} \to/\ \text{-})\ [50,51,51]50)$

abbreviation $all\text{-}sorted\text{-}map\ A\ B\ P \equiv \forall f.\ f :_s A \to B \longrightarrow P\ f$
abbreviation $ex\text{-}sorted\text{-}map\ A\ B\ P \equiv \exists f.\ f :_s A \to B \wedge P\ f$

syntax
$all\text{-}sorted\text{-}map :: 'pttrn \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a\ (\forall\ \text{-} :_s (/\ \text{-} \to/\ \text{-})./\ \text{-}\ [50,51,51,10]10)$
$ex\text{-}sorted\text{-}map :: 'pttrn \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a\ (\exists\ \text{-} :_s (/\ \text{-} \to/\ \text{-})./\ \text{-}\ [50,51,51,10]10)$

translations
$\forall f :_s A \to B.\ e \rightleftharpoons CONST\ all\text{-}sorted\text{-}map\ A\ B\ (\lambda f.\ e)$
$\exists f :_s A \to B.\ e \rightleftharpoons CONST\ ex\text{-}sorted\text{-}map\ A\ B\ (\lambda f.\ e)$

lemmas *sorted-mapI* = *sorted-map.intro*

lemma *sorted-mapD*: $f :_s A \to B \Longrightarrow a : \sigma\ in\ A \Longrightarrow f\ a : \sigma\ in\ B$
using *sorted-map.sorted-map*.

lemmas *sorted-mapE* = *sorted-map.elim*

lemma assumes $f :_s A \to B$
shows *sorted-map-o*: $g :_s B \to C \Longrightarrow g \circ f :_s A \to C$
and *sorted-map-cmono*: $A' \subseteq_m A \Longrightarrow f :_s A' \to B$
and *sorted-map-mono*: $B \subseteq_m B' \Longrightarrow f :_s A \to B'$
using *assms* **by** (*auto intro*!:*sorted-mapI dest*!:*subssetD sorted-mapD*)

lemma *sorted-map-cong*:
$(\bigwedge a\ \sigma.\ a : \sigma\ in\ A \Longrightarrow f\ a = f'\ a) \Longrightarrow$
$A = A' \Longrightarrow$
$(\bigwedge a\ \sigma.\ a : \sigma\ in\ A \Longrightarrow f\ a : \sigma\ in\ B \longleftrightarrow f\ a : \sigma\ in\ B') \Longrightarrow$
$f :_s A \to B \longleftrightarrow f' :_s A' \to B'$
by (*auto simp*: *sorted-map-def*)

lemma *sorted-choice*:
assumes $\forall a\ \sigma.\ a : \sigma\ in\ A \longrightarrow (\exists b : \sigma\ in\ B.\ P\ a\ b)$

shows $\exists f :_s A \to B.\ (\forall a \in dom\ A.\ P\ a\ (f\ a))$
proof−
have $\forall a \in dom\ A.\ \exists b.\ A\ a = B\ b \wedge P\ a\ b$
proof
fix *a* **assume** $a \in dom\ A$
then obtain σ **where** *a*: $a : \sigma\ in\ A$ **by** (*auto elim*!: *in-dom-hastypeE*)
with *assms* **obtain** *b* **where** *b*: $b : \sigma\ in\ B$ **and** *P*: $P\ a\ b$ **by** *auto*
with *a* **have** $A\ a = B\ b$ **by** (*auto simp*: *hastype-def*)
with *P* **show** $\exists b.\ A\ a = B\ b \wedge P\ a\ b$ **by** *auto*
qed
from *bchoice*[*OF this*] **obtain** *f* **where** *f*: $\forall x \in dom\ A.\ A\ x = B\ (f\ x) \wedge P\ x\ (f\ x)$ **by** *auto*
have $f :_s A \to B$
proof
fix $a\ \sigma$ **assume** *a*: $a : \sigma\ in\ A$
then have $a \in dom\ A$ **by** *auto*
with *f* **have** $A\ a = B\ (f\ a)$ **by** *auto*
with *a* **show** $f\ a : \sigma\ in\ B$ **by** (*auto simp*: *hastype-def*)
qed
with *f* **show** *?thesis* **by** *auto*
qed

lemma *sorted-map-empty*[*simp*]: $f :_s \emptyset \to A$
by (*auto simp*: *sorted-map-def*)

lemma *sorted-map-comp-nth*:
$\alpha :_s (f \circ s\ safe\text{-}nth\ (a\#as)) \to A \longleftrightarrow \alpha\ 0 : f\ a\ in\ A \wedge (\alpha \circ Suc :_s (f \circ s\ safe\text{-}nth\ as) \to A)$
(**is** $?l \longleftrightarrow ?r$)
proof
assume *?l*
from *sorted-mapD*(*1*)[*OF this*, *of 0*] *sorted-mapD*(*1*)[*OF this*, *of Suc* -]
show *?r*
apply (*intro conjI sorted-map.intro*, *unfold hastype-in-o-sset*)
by (*auto simp*: *hastype-def*)
next
assume *r*: *?r*
then have *0*: $\alpha\ 0 : f\ a\ in\ A$ **and** $\alpha \circ Suc :_s f \circ s\ safe\text{-}nth\ as \to A$ **by** *auto*
then
have $*$: $i' < length\ as \Longrightarrow \alpha\ (Suc\ i') : f\ (as!i')\ in\ A$ **for** i'
apply (*elim sorted-mapE*)
apply (*unfold hastype-in-o-sset*)
apply (*auto simp*:*sorted-map-def hastype-def*)**.**
with *0* **show** *?l*
by (*intro sorted-map.intro*, *unfold hastype-in-o-sset*, *unfold hastype-def*,*auto split*:*nat.split-asm elim*:*safe-nth-eq-SomeE*)
qed

locale *inhabited* = **fixes** *A*

assumes *inhabited*: $\bigwedge\sigma.\ \exists\, a.\ a : \sigma\ in\ A$
begin

lemma *ex-sorted-map*: $\exists\, \alpha.\ \alpha :_s V \to A$
proof (*unfold sorted-map-def*, *intro choice allI*)
fix *v*
from *inhabited*
obtain *a* **where** $\forall\, \sigma.\ v : \sigma\ in\ V \longrightarrow a : \sigma\ in\ A$
apply (*cases V v*)
apply (*auto dest*: *untyped-imp-not-hastype*)[*1*]
apply *force***.**
then show $\exists\, y.\ \forall\, \sigma.\ v : \sigma\ in\ V \longrightarrow y : \sigma\ in\ A$
by (*intro exI*[*of - a*], *auto*)
qed

end

3.2 Sorted Images

The partial version of *The* operator.

definition *safe-The* $P \equiv$ *if* $\exists!x.\ P\ x$ *then Some* (*The P*) *else None*

lemma *safe-The-eq-Some*: *safe-The* $P =$ *Some* $x \longleftrightarrow P\ x \wedge (\forall\, x'.\ P\ x' \longrightarrow x' = x)$
apply (*unfold safe-The-def*)
apply (*cases* $\exists!x.\ P\ x$)
apply (*metis option.sel the-equality*)
by *auto*

lemma *safe-The-eq-None*: *safe-The* $P =$ *None* $\longleftrightarrow \neg(\exists!x.\ P\ x)$
by (*auto simp*: *safe-The-def*)

lemma *safe-The-False*[*simp*]: *safe-The* ($\lambda x.$ *False*) $=$ *None*
by (*auto simp*: *safe-The-def*)

definition *sorted-image* :: $('a \Rightarrow 'b) \Rightarrow ('a \rightharpoonup 's) \Rightarrow 'b \rightharpoonup 's$ (**infixr** *‘s 90*) **where**
(f ‘s A) $b \equiv$ *safe-The* ($\lambda\sigma.\ \exists\, a : \sigma\ in\ A.\ f\ a = b$)

lemma *hastype-in-imageE*:
assumes $fx : \sigma\ in\ f$ ‘s X
and $\bigwedge x.\ x : \sigma\ in\ X \Longrightarrow fx = f\ x \Longrightarrow thesis$
shows *thesis*
using *assms* **by** (*auto simp*: *hastype-def sorted-image-def safe-The-eq-Some*)

lemma *in-dom-imageE*:
$b \in dom$ (f ‘s A) $\Longrightarrow (\bigwedge a\ \sigma.\ a : \sigma\ in\ A \Longrightarrow b = f\ a \Longrightarrow thesis) \Longrightarrow thesis$
by (*elim in-dom-hastypeE hastype-in-imageE*)

context *sort-preserving* **begin**

lemma *hastype-in-imageI*: $a : \sigma\ in\ A \Longrightarrow b = f\ a \Longrightarrow b : \sigma\ in\ f\ `^s\ A$
by (*auto simp*: *hastype-def sorted-image-def safe-The-eq-Some*)
(*meson eq-Some-iff-hastype same-value-imp-same-type*)

lemma *hastype-in-imageI2*: $a : \sigma\ in\ A \Longrightarrow f\ a : \sigma\ in\ f\ `^s\ A$
using *hastype-in-imageI* **by** *simp*

lemma *hastype-in-image*: $b : \sigma\ in\ f\ `^s\ A \longleftrightarrow (\exists\, a : \sigma\ in\ A.\ f\ a = b)$
by (*auto elim*!: *hastype-in-imageE intro*!: *hastype-in-imageI*)

lemma *in-dom-imageI*: $a \in dom\ A \Longrightarrow b = f\ a \Longrightarrow b \in dom\ (f\ `^s\ A)$
by (*auto intro*!: *hastype-imp-dom hastype-in-imageI elim*!: *in-dom-hastypeE*)

lemma *in-dom-imageI2*: $a \in dom\ A \Longrightarrow f\ a \in dom\ (f\ `^s\ A)$
by (*auto intro*!: *in-dom-imageI*)

lemma *hastype-list-in-image*: $bs :_l \sigma s\ in\ f\ `^s\ A \longleftrightarrow (\exists\, as.\ as :_l \sigma s\ in\ A \wedge map\ f\ as = bs)$
by (*auto simp*: *list-all2-conv-all-nth hastype-in-image Skolem-list-nth intro*!:*nth-equalityI*)

lemma *dom-image*[*simp*]: $dom\ (f\ `^s\ A) = f\ `\ dom\ A$
by (*auto intro*!: *map-le-implies-dom-le in-dom-imageI elim*!: *in-dom-imageE*)

sublocale *to-image*: *sorted-map f A* $f\ `^s\ A$
apply *unfold-locales* **by** (*auto intro*!: *hastype-in-imageI*)

lemma *sorted-map-iff-image-subset*:
$f :_s A \to B \longleftrightarrow f\ `^s\ A \subseteq_m B$
by (*auto intro*!: *subssetI sorted-mapI hastype-in-imageI elim*!: *hastype-in-imageE sorted-mapE dest*!:*subssetD*)

end

lemma *sort-preserving-o*:
assumes *f*: *sort-preserving f A* **and** *g*: *sort-preserving g* $(f\ `^s\ A)$
shows *sort-preserving* $(g \circ f)\ A$
proof (*intro sort-preserving.intro*, *unfold o-def*)
interpret *f*: *sort-preserving* **using** *f*.
interpret *g*: *sort-preserving g* $f\ `^s\ A$ **using** *g*.
fix $a\ b\ \sigma\ \tau$
assume *a*: $a : \sigma\ in\ A$ **and** *b*: $b : \tau\ in\ A$ **and** *eq*: $g\ (f\ a) = g\ (f\ b)$
from *a b* **have** $g\ (f\ a) : \sigma\ in\ g\ `^s\ f\ `^s\ A\ g\ (f\ b) : \tau\ in\ g\ `^s\ f\ `^s\ A$
by (*auto intro*!: *g.hastype-in-imageI f.hastype-in-imageI*)
with *eq* **show** $\sigma = \tau$ **by** (*auto simp*: *has-same-type*)
qed

lemma *sorted-image-image*:
assumes *f*: *sort-preserving f A* **and** *g*: *sort-preserving g* $(f\ `^s\ A)$

shows *g* ‘s *f* ‘s *A* = (*g* ∘ *f*) ‘s *A*
proof−
interpret *f*: *sort-preserving* **using** *f*.
interpret *g*: *sort-preserving g f* ‘s *A* **using** *g*.
interpret *gf*: *sort-preserving* ‹*g* ∘ *f*› *A* **using** *sort-preserving-o*[*OF f g*].
show *?thesis*
by (*auto elim*!: *hastype-in-imageE*
intro!: *sset-eqI gf.hastype-in-imageI g.hastype-in-imageI f.hastype-in-imageI*)
qed

context *sorted-map* **begin**

lemma *image-subsset*[*intro*!]: *f* ‘s *A* $\subseteq_m$ *B*
by (*auto intro*!: *subssetI sorted-map elim*!: *hastype-in-imageE*)

lemma *dom-image-subset*[*intro*!]: *f* ‘ *dom A* ⊆ *dom B*
using *map-le-implies-dom-le*[*OF image-subsset*] **by** *simp*

end

lemma *sorted-image-cong*: (⋀*a σ*. *a* : *σ in A* ⟹ *f a* = *f′ a*) ⟹ *f* ‘s *A* = *f′* ‘s *A*
by (*auto 0 3 intro*!: *ext arg-cong*[*of - - safe-The*] *simp*: *sorted-image-def*)

lemma *inj-on-dom-imp-sort-preserving-inv-into*:
assumes *inj*: *inj-on f* (*dom A*) **shows** *sort-preserving* (*inv-into* (*dom A*) *f*) (*f* ‘s
A)
by (*unfold-locales*, *auto elim*!: *hastype-in-imageE simp*: *inv-into-f-f*[*OF inj*] *has-same-type*)

lemma *inj-imp-sort-preserving-inv*:
assumes *inj*: *inj f* **shows** *sort-preserving* (*inv f*) (*f* ‘s *A*)
by (*unfold-locales*, *auto elim*!: *hastype-in-imageE simp*: *inv-into-f-f*[*OF inj*] *has-same-type*)

lemma *inj-on-dom-imp-inv-into-image-cancel*:
assumes *inj*: *inj-on f* (*dom A*)
shows *inv-into* (*dom A*) *f* ‘s *f* ‘s *A* = *A*
proof−
interpret *f*: *sort-preserving f A* **using** *inj-on-dom-imp-sort-preserving*[*OF inj*].
interpret *f′*: *sort-preserving* ‹*inv-into* (*dom A*) *f*› ‹*f* ‘s *A*›
using *inj-on-dom-imp-sort-preserving-inv-into*[*OF inj*].
show *?thesis*
by (*auto intro*!: *sset-eqI f′.hastype-in-imageI f.hastype-in-imageI elim*!: *hastype-in-imageE*
simp: *inj*)
qed

lemma *inj-imp-inv-image-cancel*:
assumes *inj*: *inj f*
shows *inv f* ‘s *f* ‘s *A* = *A*
proof−
interpret *f*: *sort-preserving f A* **using** *inj-imp-sort-preserving*[*OF inj*].

interpret f': *sort-preserving* ‹*inv f*› ‹f ‘‘^{s} A› **using** *inj-imp-sort-preserving-inv*[*OF inj*].
show *?thesis*
by (*auto intro*!: *sset-eqI f'.hastype-in-imageI f.hastype-in-imageI elim*!: *hastype-in-imageE simp*: *inj*)
qed

definition *sorted-Imagep* (**infixr** ‘‘^{s} *90*)
where $((\sqsubseteq)$ ‘‘^{s} $A)$ $b \equiv$ *safe-The* $(\lambda\sigma.\ \exists\, a : \sigma$ *in* $A.\ a \sqsubseteq b)$ **for** r (**infix** $\sqsubseteq$ *50*)

lemma *untyped-hastypeE*: $A\ a = None \Longrightarrow a : \sigma$ *in* $A \Longrightarrow$ *thesis*
by (*auto simp*: *hastype-def*)

end

4 Sorted Terms

theory *Sorted-Terms*
imports *Sorted-Sets First-Order-Terms.Term*
begin

4.1 Overloaded Notations

consts *vars* :: $'a \Rightarrow 'b$ *set*

adhoc-overloading *vars vars-term*

consts *map-vars* :: $('a \Rightarrow 'b) \Rightarrow 'c \Rightarrow 'd$

adhoc-overloading *map-vars map-term* $(\lambda x.\ x)$

lemma *map-term-eq-Var*: *map-term* $F\ V\ s = Var\ y \longleftrightarrow (\exists\, x.\ s = Var\ x \wedge y = V\ x)$
by (*cases s*, *auto*)

lemma *map-vars-id-iff*: *map-vars* $f\ s = s \longleftrightarrow (\forall\, x \in$ *vars-term* $s.\ f\ x = x)$
by (*induct s*, *auto simp*: *list-eq-iff-nth-eq all-set-conv-all-nth*)

lemma *map-var-term-id*[*simp*]: *map-term* $(\lambda x.\ x)$ *id* = *id* **by** (*auto simp*: *id-def*[*symmetric*] *term.map-id*)

lemma *map-term-eq-Fun*:
map-term $F\ V\ s = Fun\ g\ ts \longleftrightarrow (\exists f\ ss.\ s = Fun\ f\ ss \wedge g = F\ f \wedge ts = map$ (*map-term* $F\ V$) $ss)$
by (*cases s*, *auto*)

declare *domIff*[*iff del*]

4.2 Sorted Signatures and Sorted Sets of Terms

We view a sorted signature as a partial map that assigns an output sort to the pair of a function symbol and a list of input sorts.

type-synonym ('f,'s) *ssig* = 'f × 's *list* ⇀ 's

definition *hastype-in-ssig* :: 'f ⇒ 's *list* ⇒ 's ⇒ ('f,'s) *ssig* ⇒ *bool*
(- : - → - *in* - [50,61,61,50]50)
where *f* : σ*s* → τ *in F* ≡ *F* (*f*,σ*s*) = *Some* τ

lemmas *hastype-in-ssigI* = *hastype-in-ssig-def*[*unfolded atomize-eq*, *THEN iffD2*]
lemmas *hastype-in-ssigD* = *hastype-in-ssig-def*[*unfolded atomize-eq*, *THEN iffD1*]

lemma *hastype-in-ssig-imp-dom*:
assumes *f* : σ*s* → τ *in F* **shows** (*f*,σ*s*) ∈ *dom F*
using *assms* **by** (*auto simp*: *hastype-in-ssig-def domIff*)

lemma *has-same-type-ssig*:
assumes *f* : σ*s* → τ *in F* **and** *f* : σ*s* → τ' *in F* **shows** τ = τ'
using *assms* **by** (*auto simp*: *hastype-in-ssig-def*)

lemma *hastype-restrict-ssig*: *f* : σ*s* → τ *in F* |' *S* ⟷ (*f*,σ*s*) ∈ *S* ∧ *f* : σ*s* → τ *in F*
by (*auto simp*: *restrict-map-def hastype-in-ssig-def*)

lemma *subssigI*: **assumes** ⋀*f* σ*s* τ. *f* : σ*s* → τ *in F* ⟹ *f* : σ*s* → τ *in F'*
shows $F \subseteq_m F'$
using *assms* **by** (*auto simp*: *map-le-def hastype-in-ssig-def dom-def*)

lemma *subssigD*: **assumes** *FF*: $F \subseteq_m F'$ **and** *f* : σ*s* → τ *in F* **shows** *f* : σ*s* → τ *in F'*
using *assms* **by** (*auto simp*: *map-le-def hastype-in-ssig-def dom-def*)

The sorted set of terms:

primrec *Term* (𝒯'(-,-')) **where**
𝒯(*F*,*V*) (*Var v*) = *V v*
| 𝒯(*F*,*V*) (*Fun f ss*) =
(*case those* (*map* 𝒯(*F*,*V*) *ss*) *of None* ⇒ *None* | *Some* σ*s* ⇒ *F* (*f*,σ*s*))

lemma *Var-hastype*[*simp*]: *Var v* : σ *in* 𝒯(*F*,*V*) ⟷ *v* : σ *in V*
by (*auto simp*: *hastype-def*)

lemma *Fun-hastype*:
Fun f ss : τ *in* 𝒯(*F*,*V*) ⟷ (∃σ*s*. *f* : σ*s* → τ *in F* ∧ *ss* $:_l$ σ*s in* 𝒯(*F*,*V*))
apply (*unfold hastype-list-iff-those*)
by (*auto simp*: *hastype-in-ssig-def hastype-def split*:*option.split-asm*)

lemma *Fun-in-dom-imp-arg-in-dom*: *Fun f ss* ∈ *dom* 𝒯(*F*,*V*) ⟹ *s* ∈ *set ss* ⟹ *s* ∈ *dom* 𝒯(*F*,*V*)

by (*auto simp*: *in-dom-iff-ex-type Fun-hastype list-all2-conv-all-nth in-set-conv-nth*)

lemma *Fun-hastypeI*: $f : \sigma s \to \tau$ *in* $F \Longrightarrow ss :_l \sigma s$ *in* $\mathcal{T}(F,V) \Longrightarrow$ *Fun f ss* : τ *in* $\mathcal{T}(F,V)$
by (*auto simp*: *Fun-hastype*)

lemma *hastype-in-Term-induct*[*case-names Var Fun*, *induct pred*]:
assumes *s*: $s : \sigma$ *in* $\mathcal{T}(F,V)$
and *V*: $\bigwedge v\ \sigma.\ v : \sigma$ *in* $V \Longrightarrow P$ (*Var v*) σ
and *F*: $\bigwedge f\ ss\ \sigma s\ \tau.$
$f : \sigma s \to \tau$ *in* $F \Longrightarrow ss :_l \sigma s$ *in* $\mathcal{T}(F,V) \Longrightarrow$ *list-all2 P ss* $\sigma s \Longrightarrow P$ (*Fun f ss*) τ
shows $P\ s\ \sigma$
proof (*insert s*, *induct s arbitrary*: σ *rule*:*term.induct*)
case (*Var v* σ)
with *V*[*of v* σ] **show** *?case* **by** *auto*
next
case (*Fun f ss* τ)
then obtain σs **where** *f*: $f : \sigma s \to \tau$ *in* F **and** *ss*: $ss :_l \sigma s$ *in* $\mathcal{T}(F,V)$ **by** (*auto simp*:*Fun-hastype*)
show *?case*
proof (*rule F*[*OF f ss*], *unfold list-all2-conv-all-nth*, *safe*)
from *ss* **show** *len*: *length ss* = *length* σs **by** (*auto dest*: *list-all2-lengthD*)
fix *i* **assume** *i*: $i <$ *length ss*
with *ss* **have** $*$: *ss* ! $i : \sigma s$! i *in* $\mathcal{T}(F,V)$ **by** (*auto simp*: *list-all2-conv-all-nth*)
from *i* **have** *ssi*: *ss* ! $i \in$ *set ss* **by** *auto*
from *Fun*(*1*)[*OF this* $*$]
show *P* (*ss* ! *i*) (σs ! *i*)**.**
qed
qed

lemma *in-dom-Term-induct*[*case-names Var Fun*, *induct pred*]:
assumes *s*: $s \in$ *dom* $\mathcal{T}(F,V)$
assumes *V*: $\bigwedge v\ \sigma.\ v : \sigma$ *in* $V \Longrightarrow P$ (*Var v*)
assumes *F*: $\bigwedge f\ ss\ \sigma s\ \tau.$
$f : \sigma s \to \tau$ *in* $F \Longrightarrow ss :_l \sigma s$ *in* $\mathcal{T}(F,V) \Longrightarrow \forall s \in$ *set ss*. $P\ s \Longrightarrow P$ (*Fun f ss*)
shows $P\ s$
proof−
from *s* **obtain** σ **where** $s : \sigma$ *in* $\mathcal{T}(F,V)$ **by** (*auto elim*!:*in-dom-hastypeE*)
then show *?thesis*
by (*induct rule*: *hastype-in-Term-induct*, *auto intro*!: *V F simp*: *list-all2-indep2*)
qed

lemma *Term-mono-left*: **assumes** *FF*: $F \subseteq_m F'$ **shows** $\mathcal{T}(F,V) \subseteq_m \mathcal{T}(F',V)$
proof (*intro subsetI*, *elim hastype-in-Term-induct*, *goal-cases*)
case (*1 a* σ *v* σ')
then show *?case* **by** *auto*
next
case (*2 a* σ *f ss* σs τ)

then show *?case*
by (*auto intro*!:*exI*[*of* - σs] *dest*!: *subssigD*[*OF FF*] *simp*: *Fun-hastype*)
qed

lemmas *hastype-in-Term-mono-left* = *Term-mono-left*[*THEN subssetD*]

lemmas *dom-Term-mono-left* = *Term-mono-left*[*THEN map-le-implies-dom-le*]

lemma *Term-mono-right*: **assumes** *VV*: $V \subseteq_m V'$ **shows** $\mathcal{T}(F,V) \subseteq_m \mathcal{T}(F,V')$
proof (*intro subsetI*, *elim hastype-in-Term-induct*, *goal-cases*)
case (*1 a* σ *v* σ')
with *VV* **show** *?case* **by** (*auto dest*!:*subssetD*)
next
case (*2 a* σ *f ss* σs τ)
then show *?case*
by (*auto intro*!:*exI*[*of* - σs] *simp*: *Fun-hastype*)
qed

lemmas *hastype-in-Term-mono-right* = *Term-mono-right*[*THEN subssetD*]

lemmas *dom-Term-mono-right* = *Term-mono-right*[*THEN map-le-implies-dom-le*]

lemmas *Term-mono* = *map-le-trans*[*OF Term-mono-left Term-mono-right*]

lemmas *hastype-in-Term-mono* = *Term-mono*[*THEN subssetD*]

lemmas *dom-Term-mono* = *Term-mono*[*THEN map-le-implies-dom-le*]

lemma *hastype-in-Term-restrict-vars*: *s* : σ *in* $\mathcal{T}(F,\ V\ |\text{`}\ vars\ s) \longleftrightarrow$ *s* : σ *in* $\mathcal{T}(F,V)$
(**is** *?l s* $\longleftrightarrow$ *?r s*)
proof (*rule iffI*)
assume *?l s*
from *hastype-in-Term-mono-right*[*OF restrict-submap this*]
show *?r s*.
next
show *?r s* $\Longrightarrow$ *?l s*
proof (*induct rule*: *hastype-in-Term-induct*)
case (*Var v* σ)
then show *?case* **by** (*auto simp*:*hastype-restrict*)
next
case (*Fun f ss* σs τ)
have *ss* $:_l$ σs *in* $\mathcal{T}(F,V\ |\text{`}\ vars\ (Fun\ f\ ss))$
apply (*rule list.rel-mono-strong*[*OF Fun*(*3*) *hastype-in-Term-mono-right*])
by (*auto intro*: *restrict-map-mono-right*)
with *Fun* **show** *?case*
by (*auto simp*:*Fun-hastype*)
qed
qed

lemma *hastype-in-Term-imp-vars*: $s : \sigma\ in\ \mathcal{T}(F,V) \Longrightarrow v \in vars\ s \Longrightarrow v \in dom$
V
proof (*induct s σ rule*: *hastype-in-Term-induct*)
  case (*Var v σ*)
  then show *?case* **by** *auto*
next
  case (*Fun f ss σs τ*)
  then obtain *i* **where** *i*: $i <$ *length ss* **and** *v*: $v \in$ *vars* (*ss*!*i*) **by** (*auto simp*:*in-set-conv-nth*)
  from *Fun(3) i v*
  show *?case* **by** (*auto simp*: *list-all2-conv-all-nth*)
qed

lemma *in-dom-Term-imp-vars*: $s \in dom\ \mathcal{T}(F,V) \Longrightarrow v \in vars\ s \Longrightarrow v \in dom\ V$
  by (*auto elim*!: *in-dom-hastypeE simp*: *hastype-in-Term-imp-vars*)

lemma *hastype-in-Term-imp-vars-subset*: $t : s\ in\ \mathcal{T}(F,V) \Longrightarrow vars\ t \subseteq dom\ V$
  by (*auto dest*: *hastype-in-Term-imp-vars*)

interpretation *Var*: *sorted-map Var V* $\mathcal{T}(F,V)$ **for** *F V* **by** (*auto intro*!: *sorted-mapI*)

4.3 Sorted Algebras

locale *sorted-algebra-syntax* =
  fixes F :: ($'f$,$'s$) *ssig* **and** A :: $'a \rightharpoonup 's$ **and** I :: $'f \Rightarrow 'a\ list \Rightarrow 'a$

locale *sorted-algebra* = *sorted-algebra-syntax* +
  assumes *sort-matches*: $f : \sigma s \to \tau\ in\ F \Longrightarrow as :_l \sigma s\ in\ A \Longrightarrow I\ f\ as : \tau\ in\ A$
begin

context
  fixes α V
  assumes α: $\alpha :_s V \to A$
begin

lemma *eval-hastype*:
  assumes *s*: $s : \sigma\ in\ \mathcal{T}(F,V)$ **shows** $I[\![s]\!]\alpha : \sigma\ in\ A$
  by (*insert s*, *induct s σ rule*: *hastype-in-Term-induct*,
    auto simp: *sorted-mapD*[*OF α*] *intro*!: *sort-matches simp*: *list-all2-conv-all-nth*)

interpretation *eval*: *sorted-map* $\lambda s.\ I[\![s]\!]\alpha\ \mathcal{T}(F,V)\ A$
  by (*auto intro*!: *sorted-mapI eval-hastype*)

lemmas *eval-sorted-map* = *eval.sorted-map-axioms*
lemmas *eval-dom* = *eval.in-dom*
lemmas *map-eval-hastype* = *eval.sorted-map-list*
lemmas *eval-has-same-type* = *eval.target-has-same-type*
lemmas *eval-dom-iff-hastype* = *eval.target-dom-iff-hastype*
lemmas *dom-iff-hastype* = *eval.source-dom-iff-hastype*

end

lemmas *eval-hastype-vars* =
eval-hastype[*OF - hastype-in-Term-restrict-vars*[*THEN iffD2*]]

lemmas *eval-has-same-type-vars* =
eval-has-same-type[*OF - hastype-in-Term-restrict-vars*[*THEN iffD2*]]

end

lemma *sorted-algebra-cong*:
assumes $F = F'$ **and** $A = A'$
and $\bigwedge f\ \sigma s\ \tau\ as.\ f : \sigma s \to \tau\ in\ F' \Longrightarrow as :_l \sigma s\ in\ A' \Longrightarrow I\ f\ as = I'\ f\ as$
shows *sorted-algebra F A I* = *sorted-algebra F' A' I'*
using *assms* **by** (*auto simp*: *sorted-algebra-def*)

4.3.1 Term Algebras

The sorted set of terms constitutes a sorted algebra, in which evaluation is substitution.

interpretation *term*: *sorted-algebra F* $\mathcal{T}(F,V)$ *Fun* **for** *F V*
apply (*unfold-locales*)
by (*auto simp*: *Fun-hastype*)

Sorted substitution preserves type:

lemma *subst-hastype*: $\vartheta :_s X \to \mathcal{T}(F,V) \Longrightarrow s : \sigma\ in\ \mathcal{T}(F,X) \Longrightarrow s\cdot\vartheta : \sigma\ in\ \mathcal{T}(F,V)$
using *term.eval-hastype*.

lemmas *subst-hastype-imp-dom-iff* = *term.dom-iff-hastype*
lemmas *subst-hastype-vars* = *term.eval-hastype-vars*
lemmas *subst-has-same-type* = *term.eval-has-same-type*
lemmas *subst-same-vars* = *eval-same-vars*[*of - - - Fun*]
lemmas *subst-map-vars* = *eval-map-vars*[*of Fun*]
lemmas *subst-o* = *eval-o*[*of Fun*]
lemmas *subst-sorted-map* = *term.eval-sorted-map*
lemmas *map-subst-hastype* = *term.map-eval-hastype*

lemma *subst-compose-sorted-map*:
assumes $\vartheta :_s X \to \mathcal{T}(F,Y)$ **and** $\varrho :_s Y \to \mathcal{T}(F,Z)$
shows $\vartheta \circ_s \varrho :_s X \to \mathcal{T}(F,Z)$
using *assms* **by** (*simp add*: *sorted-map-def subst-compose subst-hastype*)

lemma *subst-hastype-iff-vars*:
assumes $\forall x \in vars\ s.\ \forall \sigma.\ \vartheta\ x : \sigma\ in\ \mathcal{T}(F,W) \longleftrightarrow x : \sigma\ in\ V$
shows $s \cdot \vartheta : \sigma\ in\ \mathcal{T}(F,W) \longleftrightarrow s : \sigma\ in\ \mathcal{T}(F,V)$
proof (*insert assms*, *induct s arbitrary*: σ)

case (*Var x*)
then show *?case* **by** (*auto intro*!: *hastypeI*)
next
case (*Fun f ss* τ)
then show *?case* **by** (*simp add*:*Fun-hastype list-all2-conv-all-nth cong*:*map-cong*)
qed

lemma *subst-in-dom-imp-var-in-dom*:
assumes $s{\cdot}\vartheta \in dom\ \mathcal{T}(F,V)$ **and** $x \in vars\ s$ **shows** $\vartheta\ x \in dom\ \mathcal{T}(F,V)$
using *assms*
proof (*induction s*)
case (*Var v*)
then show *?case* **by** *auto*
next
case (*Fun f ss*)
then obtain *s* **where** *s*: $s \in set\ ss$ **and** $s{\cdot}\vartheta : dom\ \mathcal{T}(F,V)$ **and** *xs*: $x \in vars\ s$
by (*auto dest*!: *Fun-in-dom-imp-arg-in-dom*)
from *Fun.IH*[*OF this*]
show *?case*.
qed

lemma *subst-sorted-map-restrict-vars*:
assumes ϑ: $\vartheta :_s X \to \mathcal{T}(F,V)$ **and** *WV*: $W \subseteq_m V$ **and** $s\vartheta$: $s{\cdot}\vartheta \in dom\ \mathcal{T}(F,W)$
shows $\vartheta :_s X\ |`\ vars\ s \to \mathcal{T}(F,W)$
proof (*safe intro*!: *sorted-mapI dest*!: *hastype-restrict*[*THEN iffD1*])
fix $x\ \sigma$ **assume** *xs*: $x \in vars\ s$ **and** $x\sigma$: $x : \sigma\ in\ X$
from *sorted-mapD*[*OF* ϑ $x\sigma$] **have** $x\vartheta\sigma$: $\vartheta\ x : \sigma\ in\ \mathcal{T}(F,V)$ **by** *auto*
from *subst-in-dom-imp-var-in-dom*[*OF* $s\vartheta$ *xs*]
obtain σ' **where** $\vartheta\ x : \sigma'\ in\ \mathcal{T}(F,W)$ **by** (*auto simp*: *in-dom-iff-ex-type*)
with *hastype-in-Term-mono*[*OF map-le-refl WV this*] $x\vartheta\sigma$
show $\vartheta\ x : \sigma\ in\ \mathcal{T}(F,W)$ **by** (*auto simp*: *has-same-type*)
qed

4.3.2 Homomorphisms

locale *sorted-distributive* =
sort-preserving φ *A* + *source*: *sorted-algebra F A I* **for** $F\ \varphi\ A\ I\ J$ +
assumes *distrib*: $f : \sigma s \to \tau\ in\ F \Longrightarrow as :_l \sigma s\ in\ A \Longrightarrow \varphi\ (I\ f\ as) = J\ f\ (map\ \varphi\ as)$
begin

lemma *distrib-eval*:
assumes α: $\alpha :_s V \to A$ **and** *s*: $s : \sigma\ in\ \mathcal{T}(F,V)$
shows $\varphi\ (I[\![s]\!]\alpha) = J[\![s]\!](\varphi \circ \alpha)$
proof (*insert s*, *induct rule*: *hastype-in-Term-induct*)
case (*Var v* σ)
then show *?case* **by** *auto*
next
case (*Fun f ss* σs τ)

note *ty* = *source.map-eval-hastype*[*OF* α *Fun*(*2*)]
from *Fun*(*3*)[*unfolded list-all2-indep2*] *distrib*[*OF Fun*(*1*) *ty*]
show *?case* **by** (*auto simp*: *o-def cong*:*map-cong*)
qed

The image of a distributive map forms a sorted algebra.

sublocale *image*: *sorted-algebra F* φ *'s* *A J*
proof (*unfold-locales*)
fix *f σs τ bs*
assume *f*: *f* : *σs* → *τ in F* **and** *bs*: *bs* :$_l$ *σs in* φ *'s* *A*
from *bs*[*unfolded hastype-list-in-image*]
obtain *as* **where** *as*: *as* :$_l$ *σs in A* **and** *asbs*: *map* φ *as* = *bs* **by** *auto*
show *J f bs* : *τ in* φ *'s* *A*
apply (*rule hastype-in-imageI*)
apply (*fact source.sort-matches*[*OF f as*])
by (*auto simp*: *distrib*[*OF f as*] *asbs*)
qed

end

lemma *sorted-distributive-cong*:
fixes *A A′* :: *′a* ⇀ *′s* **and** φ :: *′a* ⇒ *′b* **and** *I* :: *′f* ⇒ *′a list* ⇒ *′a*
assumes φ: ⋀*a σ*. *a* : *σ in A* ⟹ φ *a* = φ*′ a*
and *A*: *A* = *A′*
and *I*: ⋀*f σs τ as*. *f* : *σs* → *τ in F* ⟹ *as* :$_l$ *σs in A* ⟹ *I f as* = *I′ f as*
and *J*: ⋀*f σs τ as*. *f* : *σs* → *τ in F* ⟹ *as* :$_l$ *σs in A* ⟹ *J f* (*map* φ *as*) = *J′ f* (*map* φ *as*)
shows *sorted-distributive F* φ *A I J* = *sorted-distributive F* φ*′ A′ I′ J′*
proof−
{ **fix** *A A′* :: *′a* ⇀ *′s* **and** φ φ*′* :: *′a* ⇒ *′b* **and** *I I′* :: *′f* ⇒ *′a list* ⇒ *′a* **and** *J J′* :: *′f* ⇒ *′b list* ⇒ *′b*
assume φ: ⋀*a σ*. *a* : *σ in A* ⟹ φ *a* = φ*′ a*
have *map-eq*: *as* :$_l$ *σs in A* ⟹ *map* φ *as* = *map* φ*′ as* **for** *as σs*
by (*auto simp*: *list-eq-iff-nth-eq* φ *dest*:*list-all2-nthD*)
{ **assume** *A*: *A* = *A′*
and *I*: ⋀*f σs τ as*. *f* : *σs* → *τ in F* ⟹ *as* :$_l$ *σs in A′* ⟹ *I f as* = *I′ f as*
and *J*: ⋀*f σs τ as*. *f* : *σs* → *τ in F* ⟹ *as* :$_l$ *σs in A′* ⟹ *J f* (*map* φ *as*) = *J′ f* (*map* φ *as*)
{ **assume** *hom*: *sorted-distributive F* φ*′ A′ I′ J′*
from *hom* **interpret** *sorted-distributive F* φ*′ A′ I′ J′*.
interpret *I*: *sorted-algebra F A I*
using *source.sort-matches A I* **by** (*auto intro*!: *sorted-algebra.intro*)
have *sorted-distributive F* φ *A I J*
proof (*intro sorted-distributive.intro sorted-distributive-axioms.intro I.sorted-algebra-axioms*)
show *sort-preserving* φ *A* **using** *sort-preserving-axioms*[*folded A*] φ
by (*simp cong*: *sort-preserving-cong*)
fix *f σs τ as*
assume *f*: *f* : *σs* → *τ in F* **and** *as*: *as* :$_l$ *σs in A*

```
        from distrib[OF f as[unfolded A]] φ as I.sort-matches[OF f as]
          I[OF f as[unfolded A]]
         show φ (I f as) = J f (map φ as) by (auto simp: map-eq[symmetric] A
intro!: J[OF f, symmetric])
       qed
     }
   }
   note this map-eq
  }
  note * = this(1) and map-eq = this(2)
  from map-eq[unfolded atomize-imp atomize-all, folded atomize-imp] φ
  have map-eq: as :l σs in A ⟹ map φ as = map φ' as for as σs by metis
  show ?thesis
  proof (rule iffI)
    assume pre: sorted-distributive F φ A I J
    show sorted-distributive F φ' A' I' J'
      apply (rule *[rotated -1, OF pre])
      using assms by (auto simp: map-eq)
  next
    assume pre: sorted-distributive F φ' A' I' J'
    show sorted-distributive F φ A I J
      apply (rule *[rotated -1, OF pre])
      using assms by auto
  qed
qed

lemma sorted-distributive-o:
  assumes sorted-distributive F φ A I J and sorted-distributive F ψ (φ `s A) J K
  shows sorted-distributive F (ψ ∘ φ) A I K
proof−
  interpret φ: sorted-distributive F φ A I J + ψ: sorted-distributive F ψ φ `s A J
K using assms.
  interpret sort-preserving ψ∘φ A by (rule sort-preserving-o; unfold-locales)
  show ?thesis
    apply (unfold-locales)
    by (simp add: φ.distrib ψ.distrib[OF - φ.to-image.sorted-map-list])
qed

locale sorted-homomorphism = sorted-distributive F φ A I J + sorted-map φ A
B +
  target: sorted-algebra F B J for F φ A I B J
begin
end

lemma sorted-homomorphism-o:
  assumes sorted-homomorphism F φ A I B J and sorted-homomorphism F ψ B
J C K
  shows sorted-homomorphism F (ψ ∘ φ) A I C K
proof−
```

interpret φ: *sorted-homomorphism F* φ *A I B J* + ψ: *sorted-homomorphism F* ψ *B J C K* **using** *assms*.
  interpret *sorted-map* $\psi \circ \varphi$ *A C*
    using *sorted-map-o*[*OF* φ*.sorted-map-axioms* ψ*.sorted-map-axioms*].
  show *?thesis*
    apply (*unfold-locales*)
    by (*simp add*: φ*.distrib* ψ*.distrib*[*OF* - φ*.sorted-map-list*])
qed

context *sorted-algebra* **begin**

context fixes α *V* **assumes** *sorted*: $\alpha :_s V \to A$
begin

The term algebra is free in all F-algebras; that is, every assignment $\alpha :_s V \to A$ is extended to a homomorphism $\lambda s.\ I[\![s]\!]\alpha$.

interpretation *sorted-map* α *V A* **using** *sorted*.

interpretation *eval*: *sorted-map* $\langle\lambda s.\ I[\![s]\!]\alpha\rangle$ $\langle\mathcal{T}(F,V)\rangle$ *A* **using** *eval-sorted-map*[*OF sorted*].

interpretation *eval*: *sorted-homomorphism F* $\langle\lambda s.\ I[\![s]\!]\alpha\rangle$ $\langle\mathcal{T}(F,V)\rangle$ *Fun A I*
  apply (*unfold-locales*) **by** *auto*

lemmas *eval-sorted-homomorphism* = *eval.sorted-homomorphism-axioms*

end

end

lemma *sorted-homomorphism-cong*:
  fixes $A\ A' :: 'a \rightharpoonup 's$ **and** $\varphi :: 'a \Rightarrow 'b$ **and** $I :: 'f \Rightarrow 'a\ list \Rightarrow 'a$
  assumes φ: $\bigwedge a\ \sigma.\ a : \sigma\ in\ A \Longrightarrow \varphi\ a = \varphi'\ a$
    and *A*: $A = A'$
    and *I*: $\bigwedge f\ \sigma s\ \tau\ as.\ f : \sigma s \to \tau\ in\ F \Longrightarrow as :_l \sigma s\ in\ A \Longrightarrow I\ f\ as = I'\ f\ as$
    and *B*: $B = B'$
    and *J*: $\bigwedge f\ \sigma s\ \tau\ bs.\ f : \sigma s \to \tau\ in\ F \Longrightarrow bs :_l \sigma s\ in\ B \Longrightarrow J\ f\ bs = J'\ f\ bs$
  shows *sorted-homomorphism F* φ *A I B J* = *sorted-homomorphism F* $\varphi'\ A'\ I'\ B'\ J'$ (**is** $?l \longleftrightarrow ?r$)
proof
  assume *?l*
  then interpret *sorted-homomorphism F* φ *A I B J*.
  have J': $as :_l \sigma s\ in\ A' \Longrightarrow J\ f\ (map\ \varphi\ as) = J'\ f\ (map\ \varphi\ as)$ **if** *f*: $f : \sigma s \to \tau$ *in F* **for** $f\ \sigma s\ \tau\ as$
    apply (*rule J*[*OF f*]) **using** *A B sorted-map-list* **by** *auto*
  note $*$ = *sorted-distributive-cong*[*THEN iffD1*, *rotated* -1, *OF sorted-distributive-axioms*]
  show *?r*
    apply (*intro sorted-homomorphism.intro* $*$)
    using *assms* J' *sorted-map-axioms target.sorted-algebra-axioms*

by (*simp-all cong*: *sorted-map-cong sorted-algebra-cong*)
next
assume *?r*
then interpret *sorted-homomorphism* $F\ \varphi'\ A'\ I'\ B'\ J'$.
have J': *as* $:_l\ \sigma s$ *in* $A' \Longrightarrow J\ f\ (map\ \varphi'\ as) = J'\ f\ (map\ \varphi'\ as)$ **if** *f*: $f : \sigma s \to \tau$ *in F* **for** $f\ \sigma s\ \tau\ as$
apply (*rule J*[*OF f*]) **using** *A B sorted-map-list* φ **by** *auto*
note $* =$ *sorted-distributive-cong*[*THEN iffD1*, *rotated* -1, *OF sorted-distributive-axioms*]
note $2 =$ *sorted-map-cong*[*THEN iffD1*, *rotated* -1, *OF sorted-map-axioms*]
show *?l*
apply (*intro sorted-homomorphism.intro* $*$ *2*)
using *assms* J' *target.sorted-algebra-axioms*
by (*simp-all cong*: *sorted-distributive-cong sorted-algebra-cong*)
qed

context *sort-preserving* **begin**

lemma *sort-preserving-map-vars*: *sort-preserving* (*map-vars f*) $\mathcal{T}(F,A)$
proof
fix $a\ b\ \sigma\ \tau$
assume *a*: $a : \sigma$ *in* $\mathcal{T}(F,A)$ **and** *b*: $b : \tau$ *in* $\mathcal{T}(F,A)$ **and** *eq*: *map-vars f a* $=$ *map-vars f b*
from *a b eq* **show** $\sigma = \tau$
proof (*induct arbitrary*: τ *b*)
case (*Var x* σ)
then show *?case* **by** (*cases b*, *auto simp*: *same-value-imp-same-type*)
next
case *IH*: (*Fun ff ss* $\sigma s\ \sigma$)
show *?case*
proof (*cases b*)
case (*Var y*)
with *IH* **show** *?thesis* **by** *auto*
next
case (*Fun gg tt*)
with *IH* **have** *eq*: *map* (*map-vars f*) *ss* $=$ *map* (*map-vars f*) *tt* **by** (*auto simp*: *id-def*)
from *arg-cong*[*OF this*, *of length*] **have** *lensstt*: *length ss* $=$ *length tt* **by** *auto*
with *IH* **obtain** τs **where** *ff2*: *ff* : $\tau s \to \tau$ *in F* **and** *tt*: *tt* $:_l\ \tau s$ *in* $\mathcal{T}(F,A)$
by (*auto simp*: *Fun Fun-hastype*)
from *IH* **have** *lenss*: *length ss* $=$ *length* σs **by** (*auto simp*: *list-all2-lengthD*)
have $\sigma s = \tau s$
proof (*unfold list-eq-iff-nth-eq*, *safe*)
from *lensstt tt IH* **show** *len2*: *length* σs $=$ *length* τs **by** (*auto simp*: *list-all2-lengthD*)
fix *i* **assume** $i <$ *length* σs
with *lenss* **have** *i*: $i <$ *length ss* **by** *auto*
show $\sigma s\ !\ i = \tau s\ !\ i$
proof(*rule list-all2-nthD*[*OF IH*(*3*) *i*, *rule-format*])
from *i lenss lensstt arg-cong*[*OF eq*, *of* $\lambda xs.\ xs!i$]

```
          show map-vars f (ss ! i) = map-vars f (tt ! i) by auto
          from i lensstt list-all2-nthD[OF tt]
          show tt ! i : τs ! i in 𝒯(F,A) by auto
        qed
      qed
      with ff2 Fun IH.hyps(1) show σ = τ by (auto simp: hastype-in-ssig-def)
    qed
  qed
qed

lemma map-vars-image-Term: map-vars f ‘s 𝒯(F,A) = 𝒯(F,f ‘s A) (is ?L = ?R)
proof (intro sset-eqI)
  interpret map-vars: sort-preserving map-term (λx. x) f 𝒯(F,A) using sort-preserving-map-vars.
  fix a σ
  show a : σ in ?L ⟷ a : σ in ?R
  proof (induct a arbitrary: σ)
    case (Var x)
    then show ?case
     by (auto simp: map-vars.hastype-in-image map-term-eq-Var hastype-in-image)
       (metis Var-hastype)
  next
    case IH: (Fun ff as)
    show ?case
     proof (unfold Fun-hastype map-vars.hastype-in-image map-term-eq-Fun, safe
dest!: Fun-hastype[THEN iffD1])
      fix ss σs
      assume as: as = map (map-vars f) ss and ff: ff : σs → σ in F and ss: ss
:l σs in 𝒯(F,A)
      from ss have map (map-vars f) ss :l σs in map-vars f ‘s 𝒯(F,A)
        by (auto simp: map-vars.hastype-list-in-image)
      with IH[unfolded as]
      have map (map-vars f) ss :l σs in 𝒯(F,f ‘s A)
        by (auto simp: list-all2-conv-all-nth)
      with ff
      show ∃σs. ff : σs → σ in F ∧ map (map-vars f) ss :l σs in 𝒯(F,f ‘s A) by
auto
    next
      fix σs assume ff: ff : σs → σ in F and as: as :l σs in 𝒯(F,f ‘s A)
      with IH have as :l σs in map-vars f ‘s 𝒯(F,A)
        by (auto simp: map-vars.hastype-in-image list-all2-conv-all-nth)
      then obtain ss where ss: ss :l σs in 𝒯(F,A) and as: as = map (map-vars
f) ss
        by (auto simp: map-vars.hastype-list-in-image)
      from ss ff have a: Fun ff ss : σ in 𝒯(F,A) by (auto simp: Fun-hastype)
      show ∃a. a : σ in 𝒯(F,A) ∧ (∃fa ss. a = Fun fa ss ∧ ff = fa ∧ as = map
(map-vars f) ss)
        apply (rule exI[of - Fun ff ss])
        using a as by auto
    qed
```

qed
qed

end

context *sorted-map* **begin**

lemma *sorted-map-map-vars*: *map-vars f* $:_s$ $\mathcal{T}(F,A) \to \mathcal{T}(F,B)$
proof−
interpret *map-vars*: *sort-preserving* ‹*map-vars f*› ‹$\mathcal{T}(F,A)$› **using** *sort-preserving-map-vars*.
show *?thesis*
apply (*unfold map-vars.sorted-map-iff-image-subset*)
apply (*unfold map-vars-image-Term*)
apply (*rule Term-mono-right*)
using *image-subsset*.
qed

end

4.4 Lifting Sorts

By 'uni-sorted' we mean the situation where there is only one sort (). This situation is isomorphic to sets.

definition *unisorted A a* ≡ *if a* ∈ *A then Some* () *else None*

lemma *unisorted-eq-Some*[*simp*]: *unisorted A a* = *Some* σ ⟷ *a* ∈ *A*
and *unisorted-eq-None*[*simp*]: *unisorted A a* = *None* ⟷ *a* ∉ *A*
and *hastype-in-unisorted*[*simp*]: *a* : σ *in unisorted A* ⟷ *a* ∈ *A*
by (*auto simp*: *unisorted-def hastype-def*)

lemma *hastype-list-in-unisorted*[*simp*]: *as* $:_l$ σ*s in unisorted A* ⟷ *length as* = *length* σ*s* ∧ *set as* ⊆ *A*
by (*auto simp*: *list-all2-conv-all-nth dest*: *all-nth-imp-all-set*)

lemma *dom-unisorted*[*simp*]: *dom* (*unisorted A*) = *A*
by (*auto simp*: *unisorted-def domIff split*:*if-split-asm*)

lemma *unisorted-map*[*simp*]:
f $:_s$ *unisorted A* → τ ⟷ *f* : *A* → *dom* τ
f $:_s$ σ → *unisorted B* ⟷ *f* : *dom* σ → *B*
by (*auto simp*: *sorted-map-def hastype-def domIff*)

lemma *image-unisorted*[*simp*]: *f* '^{s} *unisorted A* = *unisorted* (*f* ' *A*)
by (*auto intro*!:*sset-eqI simp*: *hastype-def sorted-image-def safe-The-eq-Some*)

definition *unisorted-sig* :: ('*f*×*nat*) *set* ⇒ ('*f*,*unit*) *ssig*
where *unisorted-sig F* ≡ λ(*f*,σ*s*). *if* (*f*, *length* σ*s*) ∈ *F then Some* () *else None*

lemma *in-unisorted-sig*[*simp*]: *f* : σ*s* → τ *in unisorted-sig F* ⟷ (*f*,*length* σ*s*) ∈

F
by (*auto simp*: *unisorted-sig-def hastype-in-ssig-def*)

inductive-set *uTerm* ($\mathfrak{T}'$(-,-′) [*1*,*1*]*1000*) **for** *F V* **where**
Var v ∈ $\mathfrak{T}(F,V)$ **if** *v* ∈ *V*
| ∀ *s* ∈ *set ss*. *s* ∈ $\mathfrak{T}(F,V)$ ⟹ *Fun f ss* ∈ $\mathfrak{T}(F,V)$ **if** (*f*,*length ss*) ∈ *F*

lemma *Var-in-Term*[*simp*]: *Var x* ∈ $\mathfrak{T}(F,V)$ ⟷ *x* ∈ *V*
using *uTerm.cases* **by** (*auto intro*: *uTerm.intros*)

lemma *Fun-in-Term*[*simp*]: *Fun f ss* ∈ $\mathfrak{T}(F,V)$ ⟷ (*f*,*length ss*) ∈ *F* ∧ *set ss* ⊆ $\mathfrak{T}(F,V)$
apply (*unfold subset-iff*)
apply (*fold Ball-def*)
by (*metis* (*no-types*, *lifting*) *term.distinct*(*1*) *term.inject*(*2*) *uTerm.simps*)

lemma *hastype-in-unisorted-Term*[*simp*]:
s : *σ in* $\mathcal{T}$(*unisorted-sig F*, *unisorted V*) ⟷ *s* ∈ $\mathfrak{T}(F,V)$
proof (*induct s*)
case (*Var x*)
then show *?case* **by** *auto*
next
case (*Fun f ss*)
then show *?case*
by (*auto simp*: *in-dom-iff-ex-type Fun-hastype list-all2-indep2*
intro!: *exI*[*of - replicate* (*length ss*) ()])
qed

lemma *unisorted-Term*: $\mathcal{T}$(*unisorted-sig F*, *unisorted V*) = *unisorted* $\mathfrak{T}(F,V)$
by (*auto intro*!: *sset-eqI*)

locale *algebra* =
fixes *F* :: (′*f* × *nat*) *set* **and** *A* :: ′*a set* **and** *I*
assumes *closed*: (*f*, *length as*) ∈ *F* ⟹ *set as* ⊆ *A* ⟹ *I f as* ∈ *A*
begin
end

lemma *unisorted-algebra*: *sorted-algebra* (*unisorted-sig F*) (*unisorted A*) *I* ⟷ *algebra F A I*
(**is** *?l* ⟷ *?r*)
proof
assume *?r*
then interpret *algebra*.
show *?l*
apply *unfold-locales* **by** (*auto simp*: *list-all2-indep2 intro*!: *closed*)
next
assume *?l*
then interpret *sorted-algebra* ‹*unisorted-sig F*› ‹*unisorted A*› *I*.
show *?r*

```
  proof unfold-locales
    fix f as assume f: (f, length as) ∈ F and asA: set as ⊆ A
    from f have f : replicate (length as) () → () in unisorted-sig F by auto
    from sort-matches[OF this] asA
    show I f as ∈ A by auto
  qed
qed

context algebra begin

interpretation unisorted: sorted-algebra ‹unisorted-sig F› ‹unisorted A› I
  apply (unfold unisorted-algebra)..

lemma eval-closed: α : V → A ⟹ s ∈ 𝔗(F,V) ⟹ I⟦s⟧α ∈ A
  using unisorted.eval-hastype[of α unisorted V] by simp

end

locale distributive =
  source: algebra F A I for F φ A I J +
  assumes distrib: (f, length as) ∈ F ⟹ set as ⊆ A ⟹ φ (I f as) = J f (map
φ as)

lemma unisorted-distributive:
  sorted-distributive (unisorted-sig F) φ (unisorted A) I J ⟷
   distributive F φ A I J (is ?l ⟷ ?r)
proof
  assume ?r
  then interpret distributive.
  show ?l
    apply (intro sorted-distributive.intro unisorted-algebra[THEN iffD2])
    apply (unfold-locales)
    by (auto intro!: distrib simp: list-all2-same-right)
next
  assume ?l
  then interpret sorted-distributive ‹unisorted-sig F› φ ‹unisorted A› I J.
  from source.sorted-algebra-axioms
  interpret source: algebra F A I by (unfold unisorted-algebra)
  show ?r
  proof unfold-locales
    fix f as
    show (f, length as) ∈ F ⟹ set as ⊆ A ⟹ φ (I f as) = J f (map φ as)
      using distrib[of f replicate (length as) () - as]
      by auto
  qed
qed

locale homomorphism =
  distributive F φ A I J + target: algebra F B J for F φ A I B J +
```

assumes *funcset*: $\varphi : A \to B$

lemma *unisorted-homomorphism*:
sorted-homomorphism (*unisorted-sig F*) φ (*unisorted A*) *I* (*unisorted B*) *J* $\longleftrightarrow$
homomorphism F φ *A I B J* (**is** *?l* $\longleftrightarrow$ *?r*)
by (*auto simp*: *sorted-homomorphism-def unisorted-distributive unisorted-algebra homomorphism-def homomorphism-axioms-def*)

lemma *homomorphism-cong*:
assumes φ: $\bigwedge a.\ a \in A \Longrightarrow \varphi\ a = \varphi'\ a$
and *A*: $A = A'$
and *I*: $\bigwedge f\ as.\ (f,\ length\ as) \in F \Longrightarrow I\ f\ as = I'\ f\ as$
and *B*: $B = B'$
and *J*: $\bigwedge f\ bs.\ (f,\ length\ bs) \in F \Longrightarrow J\ f\ bs = J'\ f\ bs$
shows *homomorphism F* φ *A I B J* = *homomorphism F* φ' A' I' B' J'
proof−
note *sorted-homomorphism-cong*
[**where** *F* = *unisorted-sig F* **and** *A* = *unisorted A* **and** *A′* = *unisorted A′* **and** *B* = *unisorted B* **and** *B′* = *unisorted B′*]
note ∗ = *this*[*unfolded unisorted-homomorphism*]
show *?thesis* **apply** (*rule* ∗)
by (*auto simp*: *A B* φ *I J list-all2-same-right*)
qed

context *algebra* **begin**

interpretation *unisorted*: *sorted-algebra* ‹*unisorted-sig F*› ‹*unisorted A*› *I*
apply (*unfold unisorted-algebra*)..

lemma *eval-homomorphism*: $\alpha : V \to A \Longrightarrow$ *homomorphism F* $(\lambda s.\ I[\![s]\!]\alpha)$ $\mathfrak{T}(F,V)$ *Fun A I*
apply (*fold unisorted-homomorphism*)
apply (*fold unisorted-Term*)
apply (*rule unisorted.eval-sorted-homomorphism*)
by *auto*

end

context *homomorphism* **begin**

interpretation *unisorted*: *sorted-homomorphism* ‹*unisorted-sig F*› φ ‹*unisorted A*› *I* ‹*unisorted B*› *J*
apply (*unfold unisorted-homomorphism*)..

lemma *distrib-eval*: $\alpha : V \to A \Longrightarrow s \in \mathfrak{T}(F,V) \Longrightarrow \varphi\ (I[\![s]\!]\alpha) = J[\![s]\!](\varphi \circ \alpha)$
using *unisorted.distrib-eval*[*of - unisorted V*] **by** *simp*

end

By ‘unsorted’ we mean the situation where any element has the unique type

().

lemma *Term-UNIV*[*simp*]: $\mathfrak{T}$(*UNIV*,*UNIV*) = *UNIV*
proof−
  have *s* ∈ $\mathfrak{T}$(*UNIV*,*UNIV*) **for** *s* **by** (*induct s*, *auto*)
  then show *?thesis* **by** *auto*
qed

When the carrier is unsorted, any interpretation forms an algebra.

interpretation *unsorted*: *algebra UNIV UNIV I*
  rewrites ⋀*a*. *a* ∈ *UNIV* ⟷ *True*
    and ⋀*P0*. (*True* ⟹ *P0*) ≡ *Trueprop P0*
    and ⋀*P0*. (*True* ⟹ *PROP P0*) ≡ *PROP P0*
    and ⋀*P0 P1*. (*True* ⟹ *PROP P1* ⟹ *P0*) ≡ (*PROP P1* ⟹ *P0*)
  for *F I*
 apply *unfold-locales* **by** *auto*

interpretation *unsorted.eval*: *homomorphism UNIV* λ*s*. *I*⟦*s*⟧α *UNIV Fun UNIV I*
  rewrites ⋀*a*. *a* ∈ *UNIV* ⟷ *True*
    and ⋀*X*. *X* ⊆ *UNIV* ⟷ *True*
    and ⋀*P0*. (*True* ⟹ *P0*) ≡ *Trueprop P0*
    and ⋀*P0*. (*True* ⟹ *PROP P0*) ≡ *PROP P0*
    and ⋀*P0 P1*. (*True* ⟹ *PROP P1* ⟹ *P0*) ≡ (*PROP P1* ⟹ *P0*)
  for *I*
  using *unsorted.eval-homomorphism*[*of - UNIV*] **by** *auto*

Evaluation distributes over evaluations in the term algebra, i.e., substitutions.

lemma *subst-eval*: *I*⟦*s*·ϑ⟧α = *I*⟦*s*⟧(λ*x*. *I*⟦ϑ *x*⟧α)
  using *unsorted.eval.distrib-eval*[*of - UNIV*, *unfolded o-def*]
  by *auto*

lemmas *subst-subst* = *subst-eval*[*of Fun*]

4.4.1 Collecting Variables via Evaluation

definition *var-list-term t* ≡ (λ*f*. *concat*)⟦*t*⟧(λ*v*. [*v*])

lemma *var-list-Fun*[*simp*]: *var-list-term* (*Fun f ss*) = *concat* (*map var-list-term ss*)
  and *var-list-Var*[*simp*]: *var-list-term* (*Var x*) = [*x*]
  by (*simp-all add*: *var-list-term-def*[*abs-def*])

lemma *set-var-list*[*simp*]: *set* (*var-list-term s*) = *vars s*
  by (*induct s*, *auto simp*: *var-list-term-def*)

lemma *eval-subset-Un-vars*:
  assumes ∀*f as*. *foo* (*I f as*) ⊆ ⋃(*foo* ‘ *set as*)
  shows *foo* (*I*⟦*s*⟧α) ⊆ (⋃*x*∈*vars-term s*. *foo* (α *x*))
proof (*induct s*)

case (*Var x*)
show *?case* **by** *simp*
next
case (*Fun f ss*)
have *foo* ($I⟦Fun\ f\ ss⟧\alpha$) = *foo* (*I f* (*map* ($\lambda s.\ I⟦s⟧\alpha$) *ss*)) **by** *simp*
also note *assms*[*rule-format*]
also have $\bigcup$ (*foo* ‘ *set* (*map* ($\lambda s.\ I⟦s⟧\alpha$) *ss*)) = ($\bigcup s \in set\ ss.\ foo\ (I⟦s⟧\alpha)$) **by** *simp*
also have ... $\subseteq$ ($\bigcup s \in set\ ss.\ (\bigcup\ x \in vars\text{-}term\ s.\ foo\ (\alpha\ x))$)
apply (*rule UN-mono*)
using *Fun* **by** *auto*
finally show *?case* **by** *simp*
qed

4.4.2 Ground terms

lemma *hastype-in-Term-empty-imp-vars*: $s : \sigma\ in\ \mathcal{T}(F,\emptyset) \Longrightarrow vars\ s = \{\}$
by (*auto dest*: *hastype-in-Term-imp-vars-subset*)

lemma *hastype-in-Term-empty-imp-vars-subst*: $s : \sigma\ in\ \mathcal{T}(F,\emptyset) \Longrightarrow vars\ (s \cdot \vartheta) = \{\}$
by (*auto simp*: *vars-term-subst-apply-term hastype-in-Term-empty-imp-vars*)

lemma *ground-Term-iff*: $s : \sigma\ in\ \mathcal{T}(F,V) \land ground\ s \longleftrightarrow s : \sigma\ in\ \mathcal{T}(F,\emptyset)$
using *hastype-in-Term-restrict-vars*[*of s σ F V*]
by (*auto simp*: *hastype-in-Term-empty-imp-vars ground-vars-term-empty*)

lemma *hastype-in-Term-empty-imp-subst*:
$s : \sigma\ in\ \mathcal{T}(F,\emptyset) \Longrightarrow s \cdot \vartheta : \sigma\ in\ \mathcal{T}(F,V)$
by (*rule subst-hastype*, *auto*)

locale *subsignature* = **fixes** *F G* :: (*'f*,*'s*) *ssig* **assumes** *subssig*: $F \subseteq_m G$
begin

lemmas *Term-subsset* = *Term-mono-left*[*OF subssig*]
lemmas *hastype-in-Term-sub* = *Term-subsset*[*THEN subssetD*]

lemma *subsignature*: $f : \sigma s \to \tau\ in\ F \Longrightarrow f : \sigma s \to \tau\ in\ G$
using *subssig* **by** (*auto dest*: *subssigD*)

end

locale *subsignature-algebra* = *subsignature* + *super*: *sorted-algebra G*
begin

sublocale *sorted-algebra F A I*
apply *unfold-locales*
using *super.sort-matches*[*OF subssigD*[*OF subssig*]] **by** *auto*

end

locale *subalgebra* = *sorted-algebra* F A I + *super*: *sorted-algebra* G B J +
subsignature F G
for F :: $('f,'s)$ *ssig* **and** A :: $'a \rightharpoonup 's$ **and** I
and G :: $('f,'s)$ *ssig* **and** B :: $'a \rightharpoonup 's$ **and** J +
assumes *subcar*: $A \subseteq_m B$
assumes *subintp*: $f : \sigma s \to \tau$ *in* $F \Longrightarrow as :_l \sigma s$ *in* $A \Longrightarrow I\ f\ as = J\ f\ as$
begin

lemma *subcarrier*: $a : \sigma$ *in* $A \Longrightarrow a : \sigma$ *in* B
using *subcar* **by** (*auto dest*: *subssetD*)

lemma *subeval*:
assumes *s*: $s : \sigma$ *in* $\mathcal{T}(F,V)$ **and** α: $\alpha :_s V \to A$ **shows** $J[\![s]\!]\alpha = I[\![s]\!]\alpha$
proof (*insert s*, *induct rule*: *hastype-in-Term-induct*)
case (*Var v* σ)
then show *?case* **by** *auto*
next
case (*Fun f ss* σs τ)
then show *?case*
by (*auto simp*: *list-all2-indep2 cong*:*map-cong intro*!:*subintp*[*symmetric*] *map-eval-hastype*
α)
qed

end

lemma *term-subalgebra*:
assumes *FG*: $F \subseteq_m G$ **and** *VW*: $V \subseteq_m W$
shows *subalgebra* F $\mathcal{T}(F,V)$ *Fun* G $\mathcal{T}(G,W)$ *Fun*
apply *unfold-locales*
using *FG VW Term-mono*[*OF FG VW*] **by** *auto*

An algebra where every element has a representation:

locale *sorted-algebra-constant* = *sorted-algebra-syntax* +
fixes *const*
assumes *vars-const*[*simp*]: $\bigwedge d.$ *vars* (*const d*) = {}
assumes *eval-const*[*simp*]: $\bigwedge d\ \alpha.$ $I[\![\mathit{const}\ d]\!]\alpha = d$
begin

lemma *eval-subst-const*[*simp*]: $I[\![e \cdot (\mathit{const} \circ \alpha)]\!]\beta = I[\![e]\!]\alpha$
by (*induct e*, *auto simp*:*o-def intro*!:*arg-cong*[*of* - - *I* -])

lemma *eval-upd-as-subst*: $I[\![e]\!]\alpha(x{:=}a) = I[\![e \cdot \mathit{Var}(x{:=}\mathit{const}\ a)]\!]\alpha$
by (*induct e*, *auto simp*: *o-def intro*: *arg-cong*[*of* - - *I* -])

end

context *sorted-algebra-syntax* **begin**

definition *constant-at f σs i* ≡
  $\forall$ *as b. as* $:_l$ *σs in A* ⟶ *A b* = *A* (*as*!*i*) ⟶ *I f* (*as*[*i*:=*b*]) = *I f as*

lemma *constant-atI*[*intro*]:
  assumes ⋀*as b. as* $:_l$ *σs in A* ⟹ *A b* = *A* (*as*!*i*) ⟹ *I f* (*as*[*i*:=*b*]) = *I f as*
  shows *constant-at f σs i* **using** *assms* **by** (*auto simp*: *constant-at-def*)

lemma *constant-atD*:
  constant-at f σs i ⟹ *as* $:_l$ *σs in A* ⟹ *A b* = *A* (*as*!*i*) ⟹ *I f* (*as*[*i*:=*b*]) = *I f*
as
  by (*auto simp*: *constant-at-def*)

lemma *constant-atE*[*elim*]:
  assumes *constant-at f σs i*
    and (⋀*as b. as* $:_l$ *σs in A* ⟹ *A b* = *A* (*as*!*i*) ⟹ *I f* (*as*[*i*:=*b*]) = *I f as*) ⟹
thesis
  shows *thesis* **using** *assms* **by** (*auto simp*: *constant-at-def*)

definition *constant-term-on s x* ≡ $\forall$ *α a. I*⟦*s*⟧*α*(*x*:=*a*) = *I*⟦*s*⟧*α*

lemma *constant-term-onI*:
  assumes ⋀*α a. I*⟦*s*⟧*α*(*x*:=*a*) = *I*⟦*s*⟧*α* **shows** *constant-term-on s x*
  using *assms* **by** (*auto simp*: *constant-term-on-def*)

lemma *constant-term-onD*:
  assumes *constant-term-on s x* **shows** *I*⟦*s*⟧*α*(*x*:=*a*) = *I*⟦*s*⟧*α*
  using *assms* **by** (*auto simp*: *constant-term-on-def*)

lemma *constant-term-onE*:
  assumes *constant-term-on s x* **and** (⋀*α a. I*⟦*s*⟧*α*(*x*:=*a*) = *I*⟦*s*⟧*α*) ⟹ *thesis*
  shows *thesis* **using** *assms* **by** (*auto simp*: *constant-term-on-def*)

lemma *constant-term-on-extra-var*: *x* ∉ *vars s* ⟹ *constant-term-on s x*
  by (*auto intro*!: *constant-term-onI simp*: *eval-with-fresh-var*)

lemma *constant-term-on-eq*:
  assumes *st*: *I*⟦*s*⟧ = *I*⟦*t*⟧ **and** *s*: *constant-term-on s x* **shows** *constant-term-on t*
x
  using *s fun-cong*[*OF st*] **by** (*auto simp*: *constant-term-on-def*)

definition *constant-term s* ≡ $\forall$ *x. constant-term-on s x*

lemma *constant-termI*: **assumes** ⋀*x. constant-term-on s x* **shows** *constant-term*
s
  using *assms* **by** (*auto simp*: *constant-term-def*)

lemma *ground-imp-constant*: *vars s* = {} ⟹ *constant-term s*
  by (*auto intro*!: *constant-termI constant-term-on-extra-var*)

end

end

5 Sorted Contexts

theory *Sorted-Contexts*
  imports
    First-Order-Terms.Subterm-and-Context
    Sorted-Terms
begin

lemma *subt-in-dom*:
  assumes *s*: $s \in dom\ \mathcal{T}(F,V)$ **and** *st*: $s \trianglerighteq t$ **shows** $t \in dom\ \mathcal{T}(F,V)$
  using *st s*
proof (*induction*)
  case (*refl t*)
  then show *?case*.
next
  case (*subt u ss t f*)
  from *Fun-in-dom-imp-arg-in-dom*[*OF* ‹*Fun f ss* $\in$ *dom* $\mathcal{T}(F,V)$› ‹*u* $\in$ *set ss*›] *subt.IH*
  show *?case* **by** *auto*
qed

inductive *has-type-context* :: $('f,'s)$ *ssig* $\Rightarrow$ $('v \rightharpoonup 's) \Rightarrow 's \Rightarrow ('f,'v)ctxt \Rightarrow 's \Rightarrow$ *bool*
   for $F\ V\ \sigma$ **where**
  Hole: *has-type-context* $F\ V\ \sigma$ *Hole* σ
| *More*: $f : \sigma b\ @\ \varrho\ \#\ \sigma a \to \tau\ in\ F \Longrightarrow$
  $bef :_l \sigma b\ in\ \mathcal{T}(F,V) \Longrightarrow$ *has-type-context* $F\ V\ \sigma\ C\ \varrho \Longrightarrow aft :_l \sigma a\ in\ \mathcal{T}(F,V) \Longrightarrow$
  has-type-context $F\ V\ \sigma$ (*More f bef C aft*) τ

hide-fact (**open**) *Hole More*

abbreviation *has-type-context'* (((-) $:_c$/ (-) $\to$ (-) *in*/ $\mathcal{T}'$(-,-')) [*50*,*61*,*51*,*51*,*51*]*50*)
  where $C :_c \sigma \to \tau\ in\ \mathcal{T}(F,V) \equiv$ *has-type-context* $F\ V\ \sigma\ C\ \tau$

lemma *hastype-context-apply*:
  assumes $C :_c \sigma \to \tau\ in\ \mathcal{T}(F,V)$ **and** $t : \sigma\ in\ \mathcal{T}(F,V)$
  shows $C\langle t\rangle : \tau\ in\ \mathcal{T}(F,V)$
  using *assms*
proof *induct*
  case (*More f* $\sigma b\ \varrho\ \sigma a\ \tau$ *bef C aft*)
  show *?case* **unfolding** *ctxt-apply-term.simps*

proof (*intro Fun-hastypeI*[*OF More*(*1*)])
show *bef* @ $C\langle t\rangle$ # *aft* $:_l$ σb @ ϱ # σa *in* $\mathcal{T}(F,V)$
using *More*(*2*,*5*) *More*(*4*)[*OF More*(*6*)]
by (*simp add*: *list-all2-appendI*)
qed
qed *auto*

lemma *hastype-context-decompose*:
assumes $C\langle t\rangle : \tau$ *in* $\mathcal{T}(F,V)$
shows $\exists\ \sigma.\ C :_c \sigma \to \tau$ *in* $\mathcal{T}(F,V) \wedge t : \sigma$ *in* $\mathcal{T}(F,V)$
using *assms*
proof (*induct C arbitrary*: τ)
case *Hole*
then show *?case* **by** (*auto intro*: *has-type-context.Hole*)
next
case (*More f bef C aft* τ)
from *More*(*2*) **have** *Fun f* (*bef* @ $C\langle t\rangle$ # *aft*) : τ *in* $\mathcal{T}(F,V)$ **by** *auto*
from *this*[*unfolded Fun-hastype*] **obtain** σs **where**
f: $f : \sigma s \to \tau$ *in F* **and** *list*: *bef* @ $C\langle t\rangle$ # *aft* $:_l$ σs *in* $\mathcal{T}(F,V)$
by *auto*
from *list* **have** *len*: *length* σs = *length bef* + *Suc* (*length aft*)
by (*simp add*: *list-all2-conv-all-nth*)
let *?i* = *length bef*
from *len* **have** *?i* < *length* σs **by** *auto*
hence *id*: *take ?i* σs @ σs ! *?i* # *drop* (*Suc ?i*) σs = σs
by (*metis id-take-nth-drop*)
from *list* **have** *Ct*: $C\langle t\rangle : \sigma s$! *?i in* $\mathcal{T}(F,V)$
by (*metis* (*no-types*, *lifting*) *list-all2-Cons1 list-all2-append1 nth-append-length*)
from *list* **have** *bef*: *bef* $:_l$ *take ?i* σs *in* $\mathcal{T}(F,V)$
by (*metis* (*no-types*, *lifting*) *append-eq-conv-conj list-all2-append1*)
from *list* **have** *aft*: *aft* $:_l$ *drop* (*Suc ?i*) σs *in* $\mathcal{T}(F,V)$
by (*metis* (*no-types*, *lifting*) *Cons-nth-drop-Suc append-eq-conv-conj drop-all length-greater-0-conv linorder-le-less-linear list.rel-inject*(*2*) *list.simps*(*3*) *list-all2-append1*)
from *More*(*1*)[*OF Ct*] **obtain** σ **where** *C*: $C :_c \sigma \to \sigma s$! *?i in* $\mathcal{T}(F,V)$ **and** *t*: $t : \sigma$ *in* $\mathcal{T}(F,V)$
by *auto*
show *?case*
by (*intro exI*[*of - σ*] *conjI has-type-context.More*[*OF - bef - aft*, *of -* σs ! *?i*] *C t*, *unfold id*, *rule f*)
qed

lemma *apply-ctxt-in-dom-imp-in-dom*:
assumes $C\langle t\rangle \in$ *dom* $\mathcal{T}(F,V)$
shows $t \in$ *dom* $\mathcal{T}(F,V)$
apply (*rule subt-in-dom*[*OF assms*]) **by** *simp*

lemma *apply-ctxt-hastype-imp-hastype-context*:
assumes *C*: $C\langle t\rangle : \tau$ *in* $\mathcal{T}(F,V)$ **and** *t*: $t : \sigma$ *in* $\mathcal{T}(F,V)$
shows $C :_c \sigma \to \tau$ *in* $\mathcal{T}(F,V)$

```
  using hastype-context-decompose[OF C] t by (auto simp: has-same-type)

lemma subst-apply-ctxt-sorted:
  assumes C :c σ → τ in 𝒯(F,X) and ϑ :s X → 𝒯(F,V)
  shows C ·c ϑ :c σ → τ in 𝒯(F,V)
  using assms
proof(induct arbitrary: ϑ rule: has-type-context.induct)
  case (Hole)
  then show ?case by (simp add: has-type-context.Hole)
next
  case (More f σb ϱ σa τ bef C aft)
  have fssig: f : σb @ ϱ # σa → τ in F using More(1) .
  have bef:bef :l σb in 𝒯(F,X) using More(2) .
  have Cssig:C :c σ → ϱ in 𝒯(F,X) using More(3) .
  have aft:aft :l σa in 𝒯(F,X) using More(5) .
  have theta:ϑ :s X → 𝒯(F,V) using More(6) .
  hence ctheta:C ·c ϑ :c σ → ϱ in 𝒯(F,V) using More(4) by simp
  have len-bef:length bef = length σb using bef list-all2-iff by blast
  have len-aft:length aft = length σa using aft list-all2-iff by blast
  { fix i
    assume len-i:i < length σb
    hence bef ! i · ϑ : σb ! i in 𝒯(F,V)
    proof –
      have bef ! i : σb ! i in 𝒯(F,X) using bef
        by (simp add: len-i list-all2-conv-all-nth)
      from subst-hastype[OF theta this]
      show ?thesis.
    qed
  } note * = this
  have mb: map (λt. t · ϑ) bef :l σb in 𝒯(F,V) using length-map
    by (auto simp:* theta bef list-all2-conv-all-nth len-bef)
  { fix i
    assume len-i:i < length σa
    hence aft ! i · ϑ : σa ! i in 𝒯(F,V)
    proof –
      have aft ! i : σa ! i in 𝒯(F,X) using aft
        by (simp add: len-i list-all2-conv-all-nth)
      from subst-hastype[OF theta this]
      show ?thesis.
    qed
  } note ** = this
  have ma: map (λt. t · ϑ) aft :l σa in 𝒯(F,V) using length-map
    by (auto simp:** theta aft list-all2-conv-all-nth len-aft)
  show More f bef C aft ·c ϑ :c σ → τ in 𝒯(F,V)
    by (auto intro!: has-type-context.intros fssig simp:ctheta mb ma)
qed

end
```

References

[1] C. Sternagel and R. Thiemann. First-order terms. *Archive of Formal Proofs*, February 2018. https://isa-afp.org/entries/First_Order_Terms.html, Formal proof development.

[2] R. Thiemann and A. Yamada. A verified algorithm for deciding pattern completeness. In J. Rehof, editor, *9th International Conference on Formal Structures for Computation and Deduction, FSCD 2024, July 10-13, 2024, Tallinn, Estonia*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. To appear.