

Sorted Terms*

Akihisa Yamada

National Institute of Advanced Industrial Science and Technology,
Japan

René Thiemann

University of Innsbruck, Austria

March 22, 2026

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.

We further provide algorithms for computing emptiness, finiteness and cardinality of sorts.

Contents

1	Introduction	2
2	Auxiliary Lemmas	2
3	Sorted Sets and Maps	6
3.1	Maps between Sorted Sets	10
3.1.1	Sorted bijection	14
3.2	Sorted Images	15

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

4	Sorted Terms	18
4.1	Overloaded Notations	18
4.2	Sorted Signatures and Sorted Sets of Terms	19
4.3	Sorted Algebras	22
4.3.1	Term Algebras	24
4.3.2	Homomorphisms	25
4.4	Lifting Sorts	31
4.5	Collecting Variables via Evaluation	35
4.6	Ground Terms	36
4.6.1	Cardinality of Sorts	37
4.6.2	Enumerating Ground Terms	39
4.7	Subsignatures	40
5	Sorted Contexts	42
6	Basic Terms	46
7	Computing Nonempty and Infinite sorts	49
7.1	Deciding the nonemptiness of all sorts under consideration . .	50
7.2	Deciding infiniteness of a sort and computing cardinalities . .	53

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:
   $(\exists x \in \text{set } xs. P x) = (\exists i. i < \text{length } xs \wedge P (xs ! i))$ 
  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*: $(\text{Some } x = f \ggg g) = (\exists y. f = \text{Some } y \wedge g y = \text{Some } x)$
by (*fold bind-eq-Some-conv, auto*)

lemma *length-le-nth-append*: $\text{length } xs \leq n \implies (xs@ys)!n = ys!(n-\text{length } xs)$
by (*simp add: nth-append*)

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

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

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

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

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

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

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

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

lemma *list-all2-choice-nth*: **assumes** $\forall i < \text{length } xs. \exists y. r \ (xs!i) \ y$ **shows** $\exists ys. \text{list-all2 } r \ xs \ ys$

proof –

from *assms* **have** $\forall i \in \{0..<\text{length } xs\}. \exists y. r \ (xs!i) \ y$ **by** *auto*

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

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

lemma *list-all2-concat*:
list-all2 (list-all2 r) ass bss \implies 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 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*]: $(\lambda x. [x]) \circ f = (\lambda x. [f x])$ **by** *auto*

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

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

lemma *map-nth'*: $\text{length } xs = n \implies \text{map } (nth \ xs) \ [0.. n] = xs$
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 $\text{prod-list } (\text{map sum-list } xss) = \text{sum-list } (\text{map prod-list } (\text{product-lists } xss))$
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*:

$(\prod xs \leftarrow xss. \sum x \leftarrow xs. f x) = (\sum xs \leftarrow \text{product-lists } xss. \prod x \leftarrow xs. f x)$
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*:

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

lemma *bex-set-ball-set-distrib*:

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

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

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

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

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

lemma *map-nth-append*:

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

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

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

lemma *restrict-submap*[*intro!*]: $A \mid' S \subseteq_m A$

by (*auto simp: restrict-map-def map-le-def domIff*)

lemma *restrict-map-mono-left*: $A \subseteq_m A' \implies A \mid' S \subseteq_m A' \mid' S$

and *restrict-map-mono-right*: $S \subseteq S' \implies A \mid' S \subseteq_m A \mid' S'$

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* $\langle \langle (-) :/ (-) \text{ in/ } (-) \rangle \rangle$ [50,61,51]50)
where $a : \sigma \text{ in } A \equiv A \ a = \text{Some } \sigma$

abbreviation *all-hastype* $\sigma \ A \ P \equiv \forall a. a : \sigma \text{ in } A \longrightarrow P \ a$

abbreviation *ex-hastype* $\sigma \ A \ P \equiv \exists a. a : \sigma \text{ in } A \wedge P \ a$

syntax

all-hastype $:: 'pttrn \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \langle \langle \forall - :/ - \text{ in/ } -/ - \rangle \rangle$ [50,51,51,10]10)

ex-hastype $:: 'pttrn \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \langle \langle \exists - :/ - \text{ in/ } -/ - \rangle \rangle$ [50,51,51,10]10)

syntax-consts

all-hastype \Leftarrow *all-hastype* **and**

ex-hastype \Leftarrow *ex-hastype*

translations

$\forall a : \sigma \text{ in } A. e \Leftarrow \text{CONST } \text{all-hastype } \sigma \ A \ (\lambda a. e)$

$\exists a : \sigma \text{ in } A. e \Leftarrow \text{CONST } \text{ex-hastype } \sigma \ A \ (\lambda a. e)$

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** $a : \sigma \text{ in } A$ **shows** $a : \sigma' \text{ in } A \longleftrightarrow \sigma' = \sigma$
using *assms* **by** (*unfold hastype-def, auto*)

lemma *sset-eqI*: **assumes** $(\bigwedge a \ \sigma. a : \sigma \text{ in } A \longleftrightarrow a : \sigma \text{ in } B)$ **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*: $a \in \text{dom } A \longleftrightarrow (\exists \sigma. a : \sigma \text{ in } A)$ **by** (*auto simp: hastype-def domIff*)

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

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

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

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

lemma *all-dom-iff-all-hastype*: $(\forall x \in \text{dom } A. P x) \longleftrightarrow (\forall x \sigma. x : \sigma \text{ in } A \longrightarrow P x)$

by (*simp add: all-dom hastype-def*)

Explicitly sorted sets:

abbreviation *sort-annotated* $\equiv \text{Some} \circ \text{snd}$

lemma *hastype-in-Some[simp]*: $a : \sigma \text{ in } (\lambda x. \text{Some } (f x)) \longleftrightarrow \sigma = f a$

by (*auto simp: hastype-def*)

Listwise type judgement:

abbreviation *hastype-list* $\langle\langle(-) :_l (-) \text{ in/ } (-)\rangle [50,61,51]50\rangle$

where $as :_l \sigma s \text{ in } A \equiv \text{list-all2 } (\lambda a \sigma. a : \sigma \text{ in } A) as \sigma s$

lemma *has-same-type-list*:

$as :_l \sigma s \text{ in } A \Longrightarrow as :_l \sigma s' \text{ in } A \longleftrightarrow \sigma s' = \sigma s$

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*: $as :_l \sigma s \text{ in } A \longleftrightarrow \text{those } (\text{map } A as) = \text{Some } \sigma s$

proof (*induct as arbitrary: σs*)

case *Nil*

then show *?case* **by** *auto*

next

case *IH: (Cons a as σs)*

show *?case*

proof (*cases σs*)

case [*simp*]: *Nil*

show *?thesis* **by** (*auto split: option.split*)

next

case [*simp*]: (*Cons 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*: $xs :_l \sigma s \text{ in } A \Longrightarrow xs \in \text{lists } (\text{dom } A)$

by (*auto simp: list-all2-conv-all-nth in-set-conv-nth hastype-def*)

lemma *subset*: $A \subseteq_m A' \longleftrightarrow (\forall a \sigma. a : \sigma \text{ in } A \longrightarrow a : \sigma \text{ in } A')$

by (*auto simp: Ball-def map-le-def hastype-def domIff*)

lemmas *subsetI* = *subsubset[THEN iffD2, rule-format]*

lemmas *subsselD* = *subssel*[*THEN iffD1, rule-format*]

lemma *subssel-hastype-listD*: $A \subseteq_m A' \implies as :_l \sigma s \text{ in } A \implies as :_l \sigma s \text{ in } A'$
by (*auto simp: list-all2-conv-all-nth subsselD*)

lemma *has-same-type-in-subssel*:

$a : \sigma \text{ in } A' \implies A \subseteq_m A' \implies a : \sigma' \text{ in } A \implies \sigma' = \sigma$
by (*auto dest!: subsselD simp: has-same-type*)

lemma *has-same-type-in-dom-subssel*:

$a : \sigma \text{ in } A' \implies A \subseteq_m A' \implies a \in \text{dom } A \longleftrightarrow a : \sigma \text{ in } A$
by (*auto simp: in-dom-iff-ex-type dest: has-same-type-in-subssel*)

Restriction of partial map, also depending on the value.

definition *restrict-sset* $A P a \equiv$

$\text{do } \{ \sigma \leftarrow A a; \text{ if } P a \sigma \text{ then Some } \sigma \text{ else None } \}$

syntax *restrict-sset* :: $'pttrn \Rightarrow 'pttrn \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$
 $(\langle \{- : - \text{ in } - / - \rangle [50,51,50,0]1000)$

translations $\{a : \sigma \text{ in } A. P\} \equiv \text{CONST } \text{restrict-sset } A (\lambda a \sigma. P)$

lemma *hastype-in-restrict-sset*[*simp*]:

$a : \sigma \text{ in } \{a : \sigma \text{ in } A. P a \sigma\} \longleftrightarrow a : \sigma \text{ in } A \wedge P a \sigma$
by (*auto simp: restrict-sset-def hastype-def bind-eq-Some-conv*)

lemma *restrict-sset-cong*:

assumes $A = A'$
and $\bigwedge a \sigma. a : \sigma \text{ in } A \implies P a \sigma \longleftrightarrow P' a \sigma$
shows $\{a : \sigma \text{ in } A. P a \sigma\} = \{a : \sigma \text{ in } A'. P' a \sigma\}$
by (*auto intro!: sset-eqI simp: assms*)

lemma *restrict-sset-True*[*simp*]: $\{a : \sigma \text{ in } A. \text{True}\} = A$

by (*auto intro!: sset-eqI*)

lemma *dom-restrict-sset*: $\text{dom } \{a : \sigma \text{ in } A. P a \sigma\} = \{a. \exists \sigma. a : \sigma \text{ in } A \wedge P a \sigma\}$

by (*auto elim!: in-dom-hastypeE*)

lemma *hastype-restrict*: $a : \sigma \text{ in } A \mid' S \longleftrightarrow a \in S \wedge a : \sigma \text{ in } A$

by (*auto simp: restrict-map-def hastype-def*)

lemma *restrict-map-eq-restrict-sset*: $A \mid' S = \{x : \sigma \text{ in } A. x \in S\}$

by (*auto intro!: sset-eqI simp: hastype-restrict*)

lemma *hastype-the-simp*[*simp*]: $a : \sigma \text{ in } A \implies \text{the } (A a) = \sigma$

by (*auto*)

lemma *hastype-in-upd*[*simp*]: $x : \sigma \text{ in } A(y \mapsto \tau) \longleftrightarrow (\text{if } x = y \text{ then } \sigma = \tau \text{ else } x : \sigma \text{ in } A)$

by (*auto simp: hastype-def*)

lemma *all-set-hastype-iff-those*: $\forall a \in \text{set } as. a : \sigma \text{ in } A \implies$
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 \implies *Some a* | *Suc n* \implies *safe-nth as n*)

lemma *safe-nth-simp[simp]*: $i < \text{length } as \implies \text{safe-nth } as \ i = \text{Some } (as \ ! \ i)$
by (*induct as arbitrary:i, auto split:nat.split*)

lemma *safe-nth-None[simp]*:
 $\text{length } as \leq i \implies \text{safe-nth } as \ i = \text{None}$
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*:
 $\text{safe-nth } as \ i = \text{Some } a \implies (i < \text{length } as \implies as \ ! \ i = a \implies \text{thesis}) \implies \text{thesis}$
by (*cases i < length as, auto*)

lemma *dom-safe-nth[simp]*: $\text{dom } (\text{safe-nth } as) = \{0..<\text{length } as\}$
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]*: $i : \sigma \text{ in } \text{safe-nth } \sigma s \longleftrightarrow i < \text{length } \sigma s \wedge \sigma = \sigma s ! i$
by (*auto simp: hastype-def safe-nth*)

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

lemma *hastype-in-o[simp]*: $a : \sigma \text{ in } A \circ f \longleftrightarrow f \ a : \sigma \text{ in } A$ **by** (*simp add: hastype-def*)

definition *o-sset* (**infix** $\langle \circ s \rangle$ 55) **where**
 $f \circ s \ A \equiv \text{map-option } f \circ A$

lemma *hastype-in-o-sset*: $a : \sigma' \text{ in } f \circ_s A \iff (\exists \sigma. a : \sigma \text{ in } A \wedge \sigma' = f \sigma)$
by (*auto simp: o-sset-def hastype-def*)

lemma *hastype-in-o-ssetI*: $a : \sigma \text{ in } A \implies f \sigma = \sigma' \implies a : \sigma' \text{ in } f \circ_s A$
by (*auto simp: o-sset-def hastype-def*)

lemma *hastype-in-o-ssetD*: $a : \tau \text{ in } f \circ_s A \implies \exists \sigma. a : \sigma \text{ in } A \wedge \tau = f \sigma$
by (*auto simp: o-sset-def hastype-def*)

lemma *hastype-in-o-ssetE*: $a : \tau \text{ in } f \circ_s A \implies (\bigwedge \sigma. a : \sigma \text{ in } A \implies \tau = f \sigma \implies \text{thesis}) \implies \text{thesis}$
by (*auto simp: o-sset-def hastype-def*)

lemma *o-sset-restrict-sset-assoc[simp]*: $f \circ_s (A \upharpoonright X) = (f \circ_s A) \upharpoonright X$
by (*auto simp: o-sset-def restrict-map-def*)

lemma *id-o-sset[simp]*: $\text{id} \circ_s A = A$
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*: $A x = \text{Some } y \implies z = f y \implies (f \circ_s A) x = \text{Some } z$ **by** (*auto simp: o-sset-def*)

lemma *o-ssetE*: $(f \circ_s A) x = \text{Some } z \implies (\bigwedge y. A x = \text{Some } y \implies z = f y \implies \text{thesis}) \implies \text{thesis}$
by (*auto simp: o-sset-def*)

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

lemma *safe-nth-map*: $\text{safe-nth } (\text{map } f \text{ as}) = f \circ_s \text{safe-nth as}$
by (*auto simp: safe-nth o-sset-def*)

notation *Map.empty* ($\langle \emptyset \rangle$)

lemma *safe-nth-Nil[simp]*: $\text{safe-nth } [] = \emptyset$ **by** *auto*

lemma *o-sset-empty[simp]*: $f \circ_s \emptyset = \emptyset$ **by** (*auto simp: o-sset-def*)

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

3.1 Maps between Sorted Sets

locale *sort-preserving* = **fixes** $f :: 'a \Rightarrow 'b$ **and** $A :: 'a \rightarrow 's$

assumes *same-value-imp-same-type*: $a : \sigma \text{ in } A \implies b : \tau \text{ in } A \implies f a = f b \implies \sigma = \tau$

begin

lemma *same-value-imp-in-dom-iff*:

assumes *fafa'*: $f a = f a'$ **and** $a : a : \sigma \text{ in } A$ **shows** *a'*: $a' \in \text{dom } A \iff a' : \sigma$

in A
using *same-value-imp-same-type*[*OF a - fafa'*] **by** (*auto elim!*: *in-dom-hastypeE*)

lemma *sort-preserving-subset*:
assumes $A' \subseteq_m A$
shows *sort-preserving f A'*
using *same-value-imp-same-type*
apply *unfold-locales* **by** (*auto dest!*: *subsetD*[*OF assms*])

end

lemma *sort-preserving-cong*:
 $A = A' \implies (\bigwedge a \sigma. a : \sigma \text{ in } A \implies f a = f' a) \implies \text{sort-preserving } f A \longleftrightarrow \text{sort-preserving } f' A'$
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 $a : a : \sigma \text{ in } A$ **and** $b : b : \tau \text{ in } A$ **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** $A :: 'a \rightarrow 's$ **and** $B :: 'b \rightarrow 's$
assumes *sorted-map*: $\bigwedge a \sigma. a : \sigma \text{ in } A \implies f a : \sigma \text{ in } B$
begin

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

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

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

lemma *elim*:
assumes $a : (\bigwedge a \sigma. a : \sigma \text{ in } A \implies f a : \sigma \text{ in } B) \implies P$
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*: $f : \text{dom } A \rightarrow \text{dom } B$

using *sorted-map[unfolded hastype-def]* **by** (*auto simp: domIff*)

lemma *sorted-map-list*: $as :_1 \sigma s \text{ in } A \implies \text{map } f \text{ as } :_1 \sigma s \text{ in } B$

by (*auto simp: list-all2-conv-all-nth sorted-map*)

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

end

notation *sorted-map* ($\langle \cdot :_s (/ \ - \rightarrow / \ -) \rangle$ [50,51,51]50)

abbreviation *all-sorted-map* $A B P \equiv \forall f. f :_s A \rightarrow B \longrightarrow P f$

abbreviation *ex-sorted-map* $A B P \equiv \exists f. f :_s A \rightarrow B \wedge P f$

syntax

all-sorted-map :: $'pttrn \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \langle \forall \cdot :_s (/ \ - \rightarrow / \ -) \rangle \rightarrow$ [50,51,51,10]10)

ex-sorted-map :: $'pttrn \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \langle \exists \cdot :_s (/ \ - \rightarrow / \ -) \rangle \rightarrow$ [50,51,51,10]10)

translations

$\forall f :_s A \rightarrow B. e \equiv \text{CONST } \text{all-sorted-map } A B (\lambda f. e)$

$\exists f :_s A \rightarrow B. e \equiv \text{CONST } \text{ex-sorted-map } A B (\lambda f. e)$

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

lemma *sorted-mapD*: $f :_s A \rightarrow B \implies a : \sigma \text{ in } A \implies f a : \sigma \text{ in } B$

using *sorted-map.sorted-map*.

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

lemma *assumes* $f :_s A \rightarrow B$

shows *sorted-map-o*: $g :_s B \rightarrow C \implies g \circ f :_s A \rightarrow C$

and *sorted-map-cmono*: $A' \subseteq_m A \implies f :_s A' \rightarrow B$

and *sorted-map-mono*: $B \subseteq_m B' \implies f :_s A \rightarrow B'$

using *assms* **by** (*auto intro!:sorted-mapI dest!:subsssetD sorted-mapD*)

lemma *sorted-map-cong*:

$(\bigwedge a \sigma. a : \sigma \text{ in } A \implies f a = f' a) \implies$

$A = A' \implies$

$(\bigwedge a \sigma. a : \sigma \text{ in } A \implies f a : \sigma \text{ in } B \iff f a : \sigma \text{ in } B') \implies$

$f :_s A \rightarrow B \iff f' :_s A' \rightarrow B'$

by (*auto simp: sorted-map-def*)

```

lemma sorted-choice:
  assumes  $\forall a \sigma. a : \sigma \text{ in } A \longrightarrow (\exists b : \sigma \text{ in } B. P a b)$ 
  shows  $\exists f :_s A \rightarrow B. (\forall a \in \text{dom } A. P a (f a))$ 
proof –
  have  $\forall a \in \text{dom } A. \exists b. A a = B b \wedge P a b$ 
  proof
    fix  $a$  assume  $a \in \text{dom } A$ 
    then obtain  $\sigma$  where  $a : a : \sigma \text{ in } A$  by (auto elim!: in-dom-hastypeE)
    with assms obtain  $b$  where  $b : b : \sigma \text{ in } B$  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  $f : \forall x \in \text{dom } A. A x = B (f x) \wedge P x (f x)$ 
  by auto
  have  $f :_s A \rightarrow B$ 
  proof
    fix  $a \sigma$  assume  $a : a : \sigma \text{ in } A$ 
    then have  $a \in \text{dom } A$  by auto
    with  $f$  have  $A a = B (f a)$  by auto
    with  $a$  show  $f a : \sigma \text{ in } B$  by (auto simp: hastype-def)
  qed
  with  $f$  show ?thesis by auto
qed

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

lemma sorted-map-comp-nth:
   $\alpha :_s (f \text{ os safe-nth } (a \# as)) \rightarrow A \longleftrightarrow \alpha \ 0 : f a \text{ in } A \wedge (\alpha \circ \text{Suc} :_s (f \text{ os safe-nth } as) \rightarrow A)$ 
  (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  $0 : \alpha \ 0 : f a \text{ in } A$  and  $\alpha \circ \text{Suc} :_s f \text{ os safe-nth } as \rightarrow A$  by auto
  then
  have  $*$ :  $i' < \text{length } as \implies \alpha (\text{Suc } i') : f (as!i') \text{ in } A$  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

```

3.1.1 Sorted bijection

locale *sorted-surjection* = *sorted-map* +
 assumes *surj*: $f \text{ ' } \text{dom } A = \text{dom } B$
begin

lemma *hastype-in-target-iff*: $b : \sigma \text{ in } B \longleftrightarrow (\exists a : \sigma \text{ in } A. b = f a)$

proof *safe*

assume *b*: $b : \sigma \text{ in } B$

then have $b \in f \text{ ' } \text{dom } A$ **by** (*auto simp: surj*)

then obtain *a* **where** $a \in \text{dom } A$ $b = f a$ **by** *auto*

with *b* **show** $\exists a : \sigma \text{ in } A. b = f a$

by (*auto intro! exI[of - a] simp: source-dom-iff-hastype*)

qed (*simp add: sorted-map*)

lemma *image-of-sort*: $f \text{ ' } \{a. a : \sigma \text{ in } A\} = \{b. b : \sigma \text{ in } B\}$

by (*auto simp: hastype-in-target-iff*)

lemma *all-in-target-iff*: $(\forall b : \sigma \text{ in } B. P b) \longleftrightarrow (\forall a : \sigma \text{ in } A. P (f a))$

by (*auto simp: hastype-in-target-iff*)

end

locale *sorted-bijection* = *sorted-map* +
 assumes *bij*: *bij-betw* *f* (*dom* *A*) (*dom* *B*)
begin

lemma *inj*: *inj-on* *f* (*dom* *A*)

using *bij* **by** (*auto simp: bij-betw-def*)

sublocale *sorted-surjection*

proof

show $f \text{ ' } \text{dom } A = \text{dom } B$ **using** *bij* **by** (*auto simp: bij-betw-def*)

qed

thm *inj-on-subset*[*OF inj*]

lemma *bij-betw-sort*: *bij-betw* *f* $\{a. a : \sigma \text{ in } A\}$ $\{b. b : \sigma \text{ in } B\}$

by (*auto simp: bij-betw-def sorted-map hastype-in-target-iff intro: inj-on-subset[OF inj]*)

end

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

assumes *inhabited*: $\bigwedge \sigma. \exists a. a : \sigma \text{ in } A$

begin

lemma *some-hastype*:

$(\text{SOME } a. a : \sigma \text{ in } A) : \sigma \text{ in } A$

using *inhabited*[*of* σ] **by** (*auto intro: someI*)

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

3.2 Sorted Images

The partial version of *The* operator.

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

lemma *safe-The-cong*[*cong*]:
assumes *eq*: $\bigwedge x. P x \longleftrightarrow Q x$
shows *safe-The* $P = \text{safe-The } Q$
using *ext[of P Q, OF eq]* **by** *simp*

lemma *safe-The-eq-Some*: *safe-The* $P = \text{Some } x \longleftrightarrow P x \wedge (\forall x'. P x' \longrightarrow x' = x)$
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* $P = \text{None} \longleftrightarrow \neg(\exists!x. P x)$
by (*auto simp: safe-The-def*)

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

definition *sorted-image* $:: ('a \Rightarrow 'b) \Rightarrow ('a \rightarrow 's) \Rightarrow 'b \rightarrow 's$ (**infixr** $\langle ^{is} \rangle 90$) **where**
 $(f \langle ^{is} \rangle A) b \equiv \text{safe-The } (\lambda \sigma. \exists a : \sigma \text{ in } A. f a = b)$

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

lemma *in-dom-image-hastypeE*:
 $b \in \text{dom } (f \langle ^{is} \rangle A) \Longrightarrow (\bigwedge a \sigma. a : \sigma \text{ in } A \Longrightarrow b = f a \Longrightarrow \text{thesis}) \Longrightarrow \text{thesis}$

by (*elim in-dom-hastypeE hastype-in-imageE*)

lemma *in-dom-imageE*:
 assumes $x: x \in \text{dom } (f \text{ } ^{\text{cs}} A)$ and *main*: $\bigwedge a. a \in \text{dom } A \implies x = f a \implies \textit{thesis}$
 shows *thesis*
proof –
 from x obtain $a \sigma$ where $a: a : \sigma$ in A and $xf: x = f a$ by (*auto elim!: in-dom-image-hastypeE*)
 from *main*[*OF hastype-imp-dom*[*OF a*] *xf*] show *thesis*.
qed

context *sort-preserving* **begin**

lemma *hastype-in-imageI*: $a : \sigma$ in $A \implies b = f a \implies b : \sigma$ in $f \text{ } ^{\text{cs}} A$
 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 $A \implies f a : \sigma$ in $f \text{ } ^{\text{cs}} A$
 using *hastype-in-imageI* by *simp*

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

lemma *in-dom-imageI*: $a \in \text{dom } A \implies b = f a \implies b \in \text{dom } (f \text{ } ^{\text{cs}} A)$
 by (*auto intro!: hastype-imp-dom hastype-in-imageI elim!: in-dom-hastypeE*)

lemma *in-dom-imageI2*: $a \in \text{dom } A \implies f a \in \text{dom } (f \text{ } ^{\text{cs}} A)$
 by (*auto intro!: in-dom-imageI*)

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

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

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

lemma *sorted-map-iff-image-subset*:
 $f :_s A \rightarrow B \longleftrightarrow f \text{ } ^{\text{cs}} A \subseteq_m B$
 by (*auto intro!: subsetI sorted-mapI hastype-in-imageI elim!: hastype-in-imageE sorted-mapE dest!:subsetD*)

end

lemma *sort-preserving-o*:
 assumes f : *sort-preserving* $f A$ and g : *sort-preserving* $g (f \text{ } ^{\text{cs}} A)$
 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 is A using g.
  fix a b σ τ
  assume a: a : σ in A and b: b : τ in A and eq: g (f a) = g (f b)
  from a b have g (f a) : σ in g is f is A g (f b) : τ in g is f is A
    by (auto intro!: g.hastype-in-imageI f.hastype-in-imageI)
  with eq show σ = τ by (auto simp: has-same-type)
qed

lemma sorted-image-image:
  assumes f: sort-preserving f A and g: sort-preserving g (f is A)
  shows g is f is A = (g ∘ f) is A
proof -
  interpret f: sort-preserving using f.
  interpret g: sort-preserving g f is 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-subset[intro!]: f is A ⊆m B
  by (auto intro!: subsetI sorted-map elim!: hastype-in-imageE)

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

end

lemma sorted-image-cong: (∧ a σ. a : σ in A ⇒ f a = f' a) ⇒ f is A = f' is A
  by (auto 0 3 intro!: arg-cong[of - - safe-The] simp: fun-eq-iff 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 is 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 is 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 is f is A = A
proof -
  interpret f: sort-preserving f A using inj-on-dom-imp-sort-preserving[OF inj].

```

interpret f' : *sort-preserving* $\langle \text{inv-into } (\text{dom } A) f \rangle \langle f \text{ }^{\text{as}} A \rangle$
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 $\text{inv } f \text{ }^{\text{as}} f \text{ }^{\text{as}} A = A$

proof –

interpret f : *sort-preserving* $f A$ **using** *inj-imp-sort-preserving*[*OF inj*].

interpret f' : *sort-preserving* $\langle \text{inv } f \rangle \langle f \text{ }^{\text{as}} A \rangle$ **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** $\langle \text{ }^{\text{as}} \rangle$ 90)

where ($\langle \sqsubseteq \rangle \text{ }^{\text{as}} A$) $b \equiv \text{safe-The } (\lambda \sigma. \exists a : \sigma \text{ in } A. a \sqsubseteq b)$ **for** r (**infix** $\langle \sqsubseteq \rangle$ 50)

lemma *untyped-hastypeE*: $A a = \text{None} \implies a : \sigma \text{ in } A \implies \text{thesis}$

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* :: $'a \Rightarrow 'b \text{ set}$

adhoc-overloading *vars* $\rightleftharpoons \text{vars-term}$

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

adhoc-overloading *map-vars* $\rightleftharpoons \text{map-term } (\lambda x. x)$

lemma *map-term-eq-Var*: $\text{map-term } F V s = \text{Var } y \iff (\exists x. s = \text{Var } x \wedge y = V x)$

by (*cases s, auto*)

lemma *map-vars-id-iff*: $\text{map-vars } f s = s \iff (\forall x \in \text{vars-term } s. f x = x)$

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 ($\exists f ss. s = \text{Fun } f ss \wedge g = F f \wedge ts = \text{map}$
(*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* \times *'s list* \rightarrow *'s*

definition *fun-hastype* :: *f* \Rightarrow *'s* \Rightarrow *'t* \Rightarrow (*f* \times *'s* \rightarrow *'t*) \Rightarrow *bool*
($\langle (- : /- / \rightarrow /- \text{ in } / -) \rangle$ [50,61,61,50]50)
where *f* : $\sigma \rightarrow \tau$ *in F* \equiv *F* (*f, \sigma*) = *Some* τ

lemmas *fun-hastypeI* = *fun-hastype-def*[*unfolded atomize-eq, THEN iffD2*]

lemmas *fun-hastypeD* = *fun-hastype-def*[*unfolded atomize-eq, THEN iffD1*]

lemma *fun-hastype-imp-dom*[simp]:

assumes *f* : $\sigma \rightarrow \tau$ *in F* **shows** (*f, \sigma*) \in *dom F*
using *assms* **by** (*auto simp: fun-hastype-def domIff*)

lemma *in-dom-fun-hastypeE*:

assumes (*f, \sigma*) \in *dom F* **and** $\bigwedge \tau. f : \sigma \rightarrow \tau$ *in F* \implies *thesis* **shows** *thesis*
using *assms* **by** (*auto simp: fun-hastype-def dom-def*)

lemma *fun-has-same-type*:

assumes *f* : $\sigma \rightarrow \tau$ *in F* **and** *f* : $\sigma \rightarrow \tau'$ *in F* **shows** $\tau = \tau'$
using *assms* **by** (*auto simp: fun-hastype-def*)

lemma *fun-hastype-empty*[simp]: $\neg f : \sigma \rightarrow \tau$ *in* \emptyset

by (*auto simp: fun-hastype-def*)

lemma *fun-hastype-upd*: *f* : $\sigma \rightarrow \tau$ *in F* (*f', \sigma'*) \mapsto τ' \longleftrightarrow

(*if* *f* = *f'* \wedge $\sigma = \sigma'$ *then* $\tau = \tau'$ *else* *f* : $\sigma \rightarrow \tau$ *in F*)

by (*auto simp: fun-hastype-def*)

lemma *fun-hastype-restrict*: *f* : $\sigma \rightarrow \tau$ *in F* | *S* \longleftrightarrow (*f, \sigma*) \in *S* \wedge *f* : $\sigma \rightarrow \tau$ *in F*

by (*auto simp: restrict-map-def fun-hastype-def*)

lemma *subssigI*: **assumes** $\bigwedge f \sigma \tau. f : \sigma \rightarrow \tau$ *in F* \implies *f* : $\sigma \rightarrow \tau$ *in F'*

shows $F \subseteq_m F'$

using *assms* **by** (*auto simp: map-le-def fun-hastype-def dom-def*)

lemma *subsigD*: **assumes** $FF: F \subseteq_m F'$ **and** $f : \sigma \rightarrow \tau$ **in** F **shows** $f : \sigma \rightarrow \tau$ **in** F'

using *assms* **by** (*auto simp: map-le-def fun-hastype-def dom-def*)

The sorted set of terms:

primrec *Term* ($\langle \mathcal{T}'(-, -) \rangle$) **where**

$\mathcal{T}(F, V) (\text{Var } v) = V v$

| $\mathcal{T}(F, V) (\text{Fun } f \text{ } ss) =$

(*case those (map* $\mathcal{T}(F, V) \text{ } ss)$ *of* $\text{None} \Rightarrow \text{None} \mid \text{Some } \sigma s \Rightarrow F (f, \sigma s)$)

lemma *Var-hastype[simp]*: $\text{Var } v : \sigma$ **in** $\mathcal{T}(F, V) \iff v : \sigma$ **in** V

by (*auto simp: hastype-def*)

lemma *Fun-hastype*:

$\text{Fun } f \text{ } ss : \tau$ **in** $\mathcal{T}(F, V) \iff (\exists \sigma s. f : \sigma s \rightarrow \tau$ **in** $F \wedge ss :_l \sigma s$ **in** $\mathcal{T}(F, V))$

apply (*unfold hastype-list-iff-those*)

by (*auto simp: fun-hastype-def hastype-def split.option.split-asm*)

lemma *Fun-in-dom-imp-arg-in-dom*: $\text{Fun } f \text{ } ss \in \text{dom } \mathcal{T}(F, V) \implies s \in \text{set } ss \implies s \in \text{dom } \mathcal{T}(F, V)$

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

lemma *Fun-hastypeI*: $f : \sigma s \rightarrow \tau$ **in** $F \implies ss :_l \sigma s$ **in** $\mathcal{T}(F, V) \implies \text{Fun } f \text{ } ss : \tau$ **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** $V \implies P (\text{Var } v) \sigma$

and $F: \bigwedge f \text{ } ss \sigma s \tau.$

$f : \sigma s \rightarrow \tau$ **in** $F \implies ss :_l \sigma s$ **in** $\mathcal{T}(F, V) \implies \text{list-all2 } P \text{ } ss \text{ } \sigma s \implies P (\text{Fun } f \text{ } ss) \tau$

shows $P s \sigma$

proof (*insert s, induct s arbitrary: σ rule:term.induct*)

case ($\text{Var } v \sigma$)

with $V[\text{of } v \sigma]$ **show** *?case* **by** *auto*

next

case ($\text{Fun } f \text{ } ss \tau$)

then obtain σs **where** $f : f : \sigma s \rightarrow \tau$ **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** $\text{len: length } ss = \text{length } \sigma s$ **by** (*auto dest: list-all2-lengthD*)

fix i **assume** $i: i < \text{length } ss$

with ss **have** $*$: $ss ! i : \sigma s ! i$ **in** $\mathcal{T}(F, V)$ **by** (*auto simp: list-all2-conv-all-nth*)

from i **have** $ssi: ss ! i \in \text{set } ss$ **by** *auto*

from $\text{Fun}(1)[\text{OF this } *]$

show $P (ss ! i) (\sigma s ! i).$

qed
qed

lemma *in-dom-Term-induct*[*case-names Var Fun, induct pred*]:

assumes $s: s \in \text{dom } \mathcal{T}(F, V)$

assumes $V: \bigwedge v \sigma. v : \sigma \text{ in } V \implies P (\text{Var } v)$

assumes $F: \bigwedge f \text{ ss } \sigma s \tau.$

$f : \sigma s \rightarrow \tau \text{ in } F \implies \text{ss} :_1 \sigma s \text{ in } \mathcal{T}(F, V) \implies \forall s \in \text{set ss}. P s \implies P (F \text{un } f \text{ ss})$

shows $P s$

proof –

from s **obtain** σ **where** $s : \sigma \text{ in } \mathcal{T}(F, V)$ **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 ($1 a \sigma v \sigma'$)

then show *?case* **by** *auto*

next

case ($2 a \sigma f \text{ ss } \sigma s \tau$)

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 subsetD*]

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 ($1 a \sigma v \sigma'$)

with VV **show** *?case* **by** (*auto dest!:subsetD*)

next

case ($2 a \sigma f \text{ ss } \sigma s \tau$)

then show *?case*

by (*auto intro!:exI[of - σs] simp: Fun-hastype*)

qed

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

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 subsetD*]

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

```

lemma hastype-in-Term-restrict-vars:  $s : \sigma$  in  $\mathcal{T}(F, V \mid \text{' vars } s) \longleftrightarrow s : \sigma$  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  $s \implies$  ?l  $s$ 
  proof (induct rule: hastype-in-Term-induct)
    case (Var  $v \sigma$ )
    then show ?case by (auto simp:hastype-restrict)
  next
    case (Fun  $f ss \sigma s \tau$ )
    have  $ss :_l \sigma s$  in  $\mathcal{T}(F, V \mid \text{' vars } (Fun f ss))$ 
    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  $\mathcal{T}(F, V) \implies v \in \text{vars } s \implies v \in \text{dom } V$ 
proof (induct s  $\sigma$  rule: hastype-in-Term-induct)
  case (Var  $v \sigma$ )
  then show ?case by auto
next
  case (Fun  $f ss \sigma s \tau$ )
  then obtain  $i$  where  $i : i < \text{length } ss$  and  $v : v \in \text{vars } (ss!i)$  by (auto simp:in-set-conv-nth)
  from Fun(3) i v
  show ?case by (auto simp: list-all2-conv-all-nth)
qed

```

```

lemma hastype-in-Term-imp-vars-subset:  $s : \sigma$  in  $\mathcal{T}(F, V) \implies \text{vars } s \subseteq \text{dom } V$ 
by (auto dest!: hastype-in-Term-imp-vars)

```

```

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

```

```

lemma in-dom-Term-vars-subset:  $s \in \text{dom } \mathcal{T}(F, V) \implies \text{vars } s \subseteq \text{dom } V$ 
by (auto dest!: in-dom-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  $F :: ('f, 's) \text{ssig}$  and  $A :: 'a \rightarrow 's$  and  $I :: 'f \Rightarrow 'a \text{ list} \Rightarrow 'a$ 

```

```

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

context
  fixes  $\alpha V$ 
  assumes  $\alpha : \alpha :_s V \rightarrow A$ 
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  $\sigma$  rule: hastype-in-Term-induct,
    auto simp: sorted-mapD[OF  $\alpha$ ] 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]]

lemma eval-subst-sorted-map:
  assumes  $\vartheta : \vartheta :_s X \rightarrow \mathcal{T}(F, V)$  and  $\alpha : \alpha :_s V \rightarrow A$ 
  shows  $I[\vartheta]_s \alpha :_s X \rightarrow A$ 
proof
  fix  $x \sigma$  assume  $x : \sigma$  in  $X$ 
  from sorted-mapD[OF  $\vartheta$  this]
  show  $(I[\vartheta]_s \alpha) x : \sigma$  in  $A$  by (auto simp: eval-subst-def intro!: eval-hastype[OF  $\alpha$ ])
qed

lemmas eval-Term-empty-hastype = eval-hastype[OF sorted-map-empty]
lemmas map-eval-Term-empty-hastype = map-eval-hastype[OF sorted-map-empty]
lemmas eval-Term-empty-sorted-map = eval-sorted-map[OF sorted-map-empty]

end

```

lemma *sorted-algebra-cong*:
assumes $F = F'$ **and** $A = A'$
and $\bigwedge f \sigma s \tau \text{ as. } f : \sigma s \rightarrow \tau \text{ in } F' \implies \text{as} :_l \sigma s \text{ in } A' \implies I f \text{ as} = I' f \text{ as}$
shows *sorted-algebra* $F A I = \text{sorted-algebra } F' A' I'$
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*: $\vartheta :_s X \rightarrow \mathcal{T}(F, V) \implies s : \sigma \text{ in } \mathcal{T}(F, X) \implies s \cdot \vartheta : \sigma \text{ in } \mathcal{T}(F, V)$
using *term.eval-hastype*.

lemma *subst-compose-sorted-map*:
 $\vartheta :_s X \rightarrow \mathcal{T}(F, Y) \implies \varrho :_s Y \rightarrow \mathcal{T}(F, Z) \implies \vartheta \circ_s \varrho :_s X \rightarrow \mathcal{T}(F, Z)$
using *term.eval-subst-sorted-map*.

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-hastype-iff-vars*:
assumes $\forall x \in \text{vars } s. \forall \sigma. \vartheta x : \sigma \text{ in } \mathcal{T}(F, W) \longleftrightarrow x : \sigma \text{ in } V$
shows $s \cdot \vartheta : \sigma \text{ in } \mathcal{T}(F, W) \longleftrightarrow s : \sigma \text{ in } \mathcal{T}(F, V)$
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 $s \cdot \vartheta \in \text{dom } \mathcal{T}(F, V)$ **and** $x \in \text{vars } s$ **shows** $\vartheta x \in \text{dom } \mathcal{T}(F, V)$
using *assms*
proof (*induction s*)
case (*Var v*)

```

then show ?case by auto
next
case (Fun f ss)
then obtain s where s: s ∈ set ss and s·∅ : dom T(F,V) and xs: x ∈ 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 ∅: ∅ :s X → T(F,V) and WV: W ⊆m V and s∅: s·∅ ∈ dom T(F,W)
  shows ∅ :s X |c vars s → T(F,W)
proof (safe intro!: sorted-mapI dest!: hastype-restrict[THEN iffD1])
  fix x σ assume xs: x ∈ vars s and xσ: x : σ in X
  from sorted-mapD[OF ∅ xσ] have x∅σ: ∅ x : σ in T(F,V) by auto
  from subst-in-dom-imp-var-in-dom[OF s∅ xs]
  obtain σ' where ∅ x : σ' in T(F,W) by (auto simp: in-dom-iff-ex-type)
  with hastype-in-Term-mono[OF map-le-refl WV this] x∅σ
  show ∅ x : σ in T(F,W) by (auto simp: has-same-type)
qed

```

4.3.2 Homomorphisms

```

locale sorted-distributive =
  sort-preserving ∅ A + source: sorted-algebra F A I for F ∅ A I J +
  assumes distrib: f : σs → τ in F ⇒ as :l σs in A ⇒ ∅ (I f as) = J f (map
∅ as)
begin

```

```

lemma distrib-eval:
  assumes α: α :s V → A and s: s : σ in T(F,V)
  shows ∅ (I[s]α) = J[s](∅ ∘ α)
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 ∅s A J
proof (unfold-locales)
  fix f σs τ bs
  assume f: f : σs → τ in F and bs: bs :l σs in ∅s A
  from bs[unfolded hastype-list-in-image]
  obtain as where as: as :l σs in A and asbs: map ∅ as = bs by auto

```

```

show J f bs : τ in φ as 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 σ s τ as. f : σ s → τ in F ⇒ as :l σ s in A ⇒ I f as = I' f as
    and J: ⋀ f σ s τ as. f : σ s → τ in F ⇒ as :l σ s 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 σ s in A ⇒ map φ as = map φ' as for as σ s
      by (auto simp: list-eq-iff-nth-eq φ dest: list-all2-nthD)
    { assume A: A = A'
      and I: ⋀ f σ s τ as. f : σ s → τ in F ⇒ as :l σ s in A' ⇒ I f as = I' f as
      and J: ⋀ f σ s τ as. f : σ s → τ in F ⇒ as :l σ s 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 σ s τ as
          assume f: f : σ s → τ in F and as: as :l σ s 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 σ s in A ⇒ map φ as = map φ' as for as σ s by metis

```

```

show ?thesis
proof (rule iffI)
  assume pre: sorted-distributive F  $\varphi$  A I J
  show sorted-distributive F  $\varphi'$  A' I' J'
  apply (rule *[rotated -1, OF pre])
  using assms by (auto simp: map-eq)
next
  assume pre: sorted-distributive F  $\varphi'$  A' I' J'
  show sorted-distributive F  $\varphi$  A I J
  apply (rule *[rotated -1, OF pre])
  using assms by auto
qed
qed

lemma sorted-distributive-o:
  assumes sorted-distributive F  $\varphi$  A I J and sorted-distributive F  $\psi$  ( $\varphi$ cs A) J K
  shows sorted-distributive F ( $\psi \circ \varphi$ ) A I K
proof -
  interpret  $\varphi$ : sorted-distributive F  $\varphi$  A I J +  $\psi$ : sorted-distributive F  $\psi$   $\varphi$ cs A J K using assms.
  interpret sort-preserving  $\psi \circ \varphi$  A by (rule sort-preserving-o; unfold-locales)
  show ?thesis
  apply (unfold-locales)
  by (simp add:  $\varphi$ .distrib  $\psi$ .distrib[OF -  $\varphi$ .to-image.sorted-map-list])
qed

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

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

context sorted-algebra begin

context fixes  $\alpha$  V assumes sorted:  $\alpha$  :s V  $\rightarrow$  A

```

begin

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

interpretation *sorted-map* $\alpha V A$ **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 $A A' :: 'a \rightarrow 's$ **and** $\varphi :: 'a \Rightarrow 'b$ **and** $I :: 'f \Rightarrow 'a \text{ list} \Rightarrow 'a$

assumes $\varphi: \bigwedge a \sigma. a : \sigma \text{ in } A \Longrightarrow \varphi a = \varphi' a$

and $A: A = A'$

and $I: \bigwedge f \sigma s \tau as. f : \sigma s \rightarrow \tau \text{ in } F \Longrightarrow as :_l \sigma s \text{ in } A \Longrightarrow I f as = I' f as$

and $B: B = B'$

and $J: \bigwedge f \sigma s \tau bs. f : \sigma s \rightarrow \tau \text{ in } F \Longrightarrow bs :_l \sigma s \text{ in } B \Longrightarrow J f bs = J' f bs$

shows *sorted-homomorphism* $F \varphi A I B J = \text{sorted-homomorphism } F \varphi' A' I' B' J'$ (**is** $?l \longleftrightarrow ?r$)

proof

assume $?l$

then interpret *sorted-homomorphism* $F \varphi A I B J$.

have $J': as :_l \sigma s \text{ in } A' \Longrightarrow J f (\text{map } \varphi as) = J' f (\text{map } \varphi as)$ **if** $f: f : \sigma s \rightarrow \tau$
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 $J': as :_l \sigma s \text{ in } A' \Longrightarrow J f (\text{map } \varphi' as) = J' f (\text{map } \varphi' as)$ **if** $f: f : \sigma s \rightarrow \tau$
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 $?$ = *sorted-map-cong*[*THEN iffD1, rotated -1, OF sorted-map-axioms*]

show $?l$

apply (*intro sorted-homomorphism.intro* $* ?$)

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 σ τ

assume a: a : σ in $\mathcal{T}(F,A)$ and b: b : τ 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 σ s σ)

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 τ s where ff2: ff : τ s \rightarrow τ in F and tt: tt :_i τ s in $\mathcal{T}(F,A)$

by (*auto simp: Fun Fun-hastype*)

from IH have *lenss*: *length* ss = *length* σ s by (*auto simp: list-all2-lengthD*)

have σ s = τ s

proof (*unfold list-eq-iff-nth-eq, safe*)

from *lensstt* tt IH show *len2*: *length* σ s = *length* τ s by (*auto simp: list-all2-lengthD*)

fix i assume i < *length* σ s

with *lenss* have i: i < *length* ss by auto

show σ s ! i = τ s ! i

proof (*rule list-all2-nthD*[*OF IH*](\exists) i, *rule-format*)

from i *lenss* *lensstt* *arg-cong*[*OF eq, of λ 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 : τ s ! i in $\mathcal{T}(F,A)$ by auto

qed

qed

with ff2 *Fun* IH.*hyps*(1) show $\sigma = \tau$ by (*auto simp: fun-hastype-def*)

qed

qed

qed

lemma *map-vars-image-Term*: *map-vars* f ^{cs} $\mathcal{T}(F,A)$ = $\mathcal{T}(F, f$ ^{cs} A) (is ?L = ?R)

proof (*intro sset-eqI*)

```

interpret map-vars: sort-preserving map-term  $(\lambda x. x) f \mathcal{T}(F,A)$  using sort-preserving-map-vars.
fix a  $\sigma$ 
show  $a : \sigma$  in ?L  $\longleftrightarrow$   $a : \sigma$  in ?R
proof (induct a arbitrary:  $\sigma$ )
  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  $\sigma s$ 
    assume as:  $as = \text{map} (\text{map-vars } f) ss$  and ff:  $ff : \sigma s \rightarrow \sigma$  in F and ss:  $ss$ 
 $:_i \sigma s$  in  $\mathcal{T}(F,A)$ 
    from ss have  $\text{map} (\text{map-vars } f) ss :_i \sigma s$  in  $\text{map-vars } f \text{ } ^{as} \mathcal{T}(F,A)$ 
    by (auto simp: map-vars.hastype-list-in-image)
    with IH[unfolded as]
    have  $\text{map} (\text{map-vars } f) ss :_i \sigma s$  in  $\mathcal{T}(F, f \text{ } ^{as} A)$ 
    by (auto simp: list-all2-conv-all-nth)
    with ff
    show  $\exists \sigma s. ff : \sigma s \rightarrow \sigma$  in F  $\wedge \text{map} (\text{map-vars } f) ss :_i \sigma s$  in  $\mathcal{T}(F, f \text{ } ^{as} A)$  by
auto
  next
  fix  $\sigma s$  assume ff:  $ff : \sigma s \rightarrow \sigma$  in F and as:  $as :_i \sigma s$  in  $\mathcal{T}(F, f \text{ } ^{as} A)$ 
  with IH have  $as :_i \sigma s$  in  $\text{map-vars } f \text{ } ^{as} \mathcal{T}(F,A)$ 
  by (auto simp: map-vars.hastype-in-image list-all2-conv-all-nth)
  then obtain ss where  $ss :_i \sigma s$  in  $\mathcal{T}(F,A)$  and as:  $as = \text{map} (\text{map-vars}$ 
f) ss
  by (auto simp: map-vars.hastype-list-in-image)
  from ss ff have  $a : \text{Fun } ff \text{ } ss : \sigma$  in  $\mathcal{T}(F,A)$  by (auto simp: Fun-hastype)
  show  $\exists a. a : \sigma$  in  $\mathcal{T}(F,A) \wedge (\exists fa \text{ } ss. a = \text{Fun } fa \text{ } ss \wedge ff = fa \wedge as = \text{map}$ 
(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:  $\text{map-vars } f :_s \mathcal{T}(F,A) \rightarrow \mathcal{T}(F,B)$ 
proof –
interpret map-vars: sort-preserving  $\langle \text{map-vars } f \rangle \langle \mathcal{T}(F,A) \rangle$  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-subset*.
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 $a \equiv$ if $a \in A$ then *Some* () else *None*

lemma *unisorted-eq-Some*[*simp*]: *unisorted* A $a =$ *Some* $\sigma \longleftrightarrow a \in A$
and *unisorted-eq-None*[*simp*]: *unisorted* A $a =$ *None* $\longleftrightarrow a \notin A$
and *hastype-in-unisorted*[*simp*]: $a : \sigma$ in *unisorted* $A \longleftrightarrow a \in A$
by (*auto simp: unisorted-def hastype-def*)

lemma *hastype-list-in-unisorted*[*simp*]: $as :_l \sigma s$ in *unisorted* $A \longleftrightarrow$ *length* $as =$
length $\sigma s \wedge$ *set* $as \subseteq A$
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* $A \rightarrow \tau \longleftrightarrow f : A \rightarrow$ *dom* τ
 $f :_s \sigma \rightarrow$ *unisorted* $B \longleftrightarrow f :$ *dom* $\sigma \rightarrow B$
by (*auto simp: sorted-map-def hastype-def domIff*)

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

definition *unisorted-sig* :: ($f \times \text{nat}$) *set* \Rightarrow (f, unit) *ssig*
where *unisorted-sig* $F \equiv \lambda(f, \sigma s). \text{if } (f, \text{length } \sigma s) \in F \text{ then } \text{Some } () \text{ else } \text{None}$

lemma *in-unisorted-sig*[*simp*]: $f : \sigma s \rightarrow \tau$ in *unisorted-sig* $F \longleftrightarrow (f, \text{length } \sigma s) \in$
 F
by (*auto simp: unisorted-sig-def fun-hastype-def*)

inductive-set *uTerm* ($\langle \mathfrak{T}'(-, -) \rangle [1, 1] 1000$) **for** F V **where**
 $\text{Var } v \in \mathfrak{T}(F, V)$ **if** $v \in V$
 $|\ \forall s \in \text{set } ss. s \in \mathfrak{T}(F, V) \implies \text{Fun } f ss \in \mathfrak{T}(F, V)$ **if** $(f, \text{length } ss) \in F$

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

lemma *Fun-in-Term*[*simp*]: $\text{Fun } f ss \in \mathfrak{T}(F, V) \longleftrightarrow (f, \text{length } ss) \in F \wedge \text{set } ss \subseteq$
 $\mathfrak{T}(F, V)$

```

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]:
   $s : \sigma$  in  $\mathcal{T}(\text{unisorted-sig } F, \text{ unisorted } V) \longleftrightarrow s \in \mathfrak{T}(F, V)$ 
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:  $\mathcal{T}(\text{unisorted-sig } F, \text{ unisorted } V) = \text{ unisorted } \mathfrak{T}(F, V)$ 
by (auto intro!: sset-eqI)

locale algebra =
  fixes  $F :: ('f \times \text{ nat}) \text{ set}$  and  $A :: 'a \text{ set}$  and  $I$ 
  assumes closed:  $(f, \text{ length } as) \in F \implies \text{ set } as \subseteq A \implies I f as \in A$ 
begin
end

lemma unisorted-algebra: sorted-algebra (unisorted-sig F) (unisorted A) I  $\longleftrightarrow$ 
  algebra F A I
  (is ?l  $\longleftrightarrow$  ?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  $\langle \text{ unisorted-sig } F \rangle \langle \text{ unisorted } A \rangle I$ .
  show ?r
    proof unfold-locales
      fix f as assume f:  $(f, \text{ length } as) \in F$  and asA:  $\text{ set } as \subseteq A$ 
      from f have f : replicate (length as) ()  $\rightarrow$  () in unisorted-sig F by auto
      from sort-matches[OF this] asA
      show I f as  $\in A$  by auto
    qed
qed

context algebra begin

interpretation unisorted: sorted-algebra  $\langle \text{ unisorted-sig } F \rangle \langle \text{ unisorted } A \rangle I$ 
apply (unfold unisorted-algebra)..

```

lemma *eval-closed*: $\alpha : V \rightarrow A \implies s \in \mathfrak{T}(F, V) \implies I[[s]]\alpha \in A$
using *unsorted.eval-hastype*[of α unsorted V] **by** *simp*

end

locale *distributive* =
source: algebra $F A I$ **for** $F \varphi A I J +$
assumes *distrib*: $(f, \text{length } as) \in F \implies \text{set } as \subseteq A \implies \varphi (I f as) = J f (\text{map } \varphi as)$

lemma *unsorted-distributive*:
sorted-distributive (*unsorted-sig* F) φ (*unsorted* A) $I J \longleftrightarrow$
distributive $F \varphi A I J$ (**is** $?l \longleftrightarrow ?r$)

proof

assume $?r$
then interpret *distributive*.
show $?l$
apply (*intro sorted-distributive.intro unsorted-algebra*[*THEN iffD2*])
apply (*unfold-locales*)
by (*auto intro!*: *distrib simp: list-all2-same-right*)

next

assume $?l$
then interpret *sorted-distributive* \langle *unsorted-sig* $F \rangle \varphi \langle$ *unsorted* $A \rangle I J$.
from *source.sorted-algebra-axioms*
interpret *source*: algebra $F A I$ **by** (*unfold unsorted-algebra*)
show $?r$
proof *unfold-locales*
fix $f as$
show $(f, \text{length } as) \in F \implies \text{set } as \subseteq A \implies \varphi (I f as) = J f (\text{map } \varphi as)$
using *distrib*[of f replicate ($\text{length } as$) () - as]
by *auto*

qed

qed

locale *homomorphism* =
distributive $F \varphi A I J +$ *target*: algebra $F B J$ **for** $F \varphi A I B J +$
assumes *funcset*: $\varphi : A \rightarrow B$

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

lemma *homomorphism-cong*:

assumes $\varphi: \bigwedge a. a \in A \implies \varphi a = \varphi' a$
and $A: A = A'$
and $I: \bigwedge f as. (f, \text{length } as) \in F \implies I f as = I' f as$

and $B: B = B'$
and $J: \bigwedge f bs. (f, \text{length } bs) \in F \implies J f bs = J' f bs$
shows *homomorphism* $F \varphi A I B J = \text{homomorphism } F \varphi' A' I' B' J'$
proof –
note *sorted-homomorphism-cong*
[**where** $F = \text{unsorted-sig } F$ **and** $A = \text{unsorted } A$ **and** $A' = \text{unsorted } A'$ **and**
 $B = \text{unsorted } B$ **and** $B' = \text{unsorted } B'$]
note $*$ = *this[unfolded unsorted-homomorphism]*
show *?thesis* **apply** (*rule* $*$)
by (*auto simp: A B φ I J list-all2-same-right*)
qed

context *algebra* **begin**

interpretation *unsorted: sorted-algebra* $\langle \text{unsorted-sig } F \rangle \varphi \langle \text{unsorted } A \rangle I$
apply (*unfold unsorted-algebra*)..

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

end

context *homomorphism* **begin**

interpretation *unsorted: sorted-homomorphism* $\langle \text{unsorted-sig } F \rangle \varphi \langle \text{unsorted } A \rangle I \langle \text{unsorted } B \rangle J$
apply (*unfold unsorted-homomorphism*)..

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

end

By ‘unsorted’ we mean the situation where any element has the unique type $()$.

lemma *Term-UNIV[simp]*: $\mathfrak{T}(UNIV, UNIV) = UNIV$ (**is** $?l = -$)
proof –
have $s \in ?l$ **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 $\bigwedge a. a \in UNIV \longleftrightarrow \text{True}$
and $\bigwedge P0. (\text{True} \implies P0) \equiv \text{Trueprop } P0$

and $\bigwedge P0. (True \implies PROP P0) \equiv PROP P0$
and $\bigwedge P0 P1. (True \implies PROP P1 \implies P0) \equiv (PROP P1 \implies P0)$
for $F I$
apply *unfold-locales* **by** *auto*

interpretation *unsorted.eval*: homomorphism $UNIV \lambda s. I[s]\alpha$ $UNIV Fun UNIV I$

rewrites $\bigwedge a. a \in UNIV \longleftrightarrow True$
and $\bigwedge X. X \subseteq UNIV \longleftrightarrow True$
and $\bigwedge P0. (True \implies P0) \equiv Trueprop P0$
and $\bigwedge P0. (True \implies PROP P0) \equiv PROP P0$
and $\bigwedge P0 P1. (True \implies PROP P1 \implies P0) \equiv (PROP P1 \implies P0)$
for I
using *unsorted.eval-homomorphism*[of - $UNIV$] **by** *auto*

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

lemma *subst-eval*: $I[s \cdot \vartheta]\alpha = I[s](\lambda x. I[\vartheta x]\alpha)$
using *unsorted.eval.distrib-eval*[of - $UNIV$, *unfolded o-def*]
by *auto*

4.5 Collecting Variables via Evaluation

definition *var-list-term* $t \equiv (\lambda f. concat)[t](\lambda v. [v])$

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 $\forall f as. foo (I f as) \subseteq \bigcup (foo \text{ ' set } as)$
shows $foo (I[s]\alpha) \subseteq (\bigcup_{x \in vars\text{-term } s. foo (\alpha x))$

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 $\bigcup (foo \text{ ' set } (map (\lambda s. I[s]\alpha) ss)) = (\bigcup_{s \in set ss. foo (I[s]\alpha))$ **by** *simp*

also have $\dots \subseteq (\bigcup_{s \in set ss. (\bigcup_{x \in vars\text{-term } s. foo (\alpha x))}$

apply (*rule UN-mono*)

using *Fun* **by** *auto*

finally show *?case* **by** *simp*

qed

4.6 Ground Terms

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

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

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

lemma *Term-empty-imp-ground*: $s : \sigma$ in $\mathcal{T}(F, \emptyset) \implies \text{ground } s$
using *Term-empty-iff*[*of s σ*] **by** *auto*

lemmas *subst-Term-empty-hastype = term.eval-Term-empty-hastype*

lemma *in-dom-Term-empty-imp-subst*:
 $s \in \text{dom } \mathcal{T}(F, \emptyset) \implies s \cdot \vartheta \in \text{dom } \mathcal{T}(F, V)$
proof (*elim in-dom-hastypeE*)
fix σ **assume** $s : \sigma$ in $\mathcal{T}(F, \emptyset)$
from *subst-Term-empty-hastype*[*OF this, of ϑ V*]
show $s \cdot \vartheta \in \text{dom } \mathcal{T}(F, V)$ **by** *auto*
qed

lemma *eval-ground-eq*: $\text{ground } s \implies I[s]\alpha = I[s]\alpha'$
apply (*induct rule: ground.induct*)
by (*auto cong: map-cong*)

lemmas *eval-Term-empty-eq = eval-ground-eq*[*OF Term-empty-imp-ground*]

lemmas *subst-Term-empty-eq = eval-Term-empty-eq*[**where** $I = \text{Fun}$]

lemma *subst-Term-empty-id*:
assumes $s : \sigma$ in $\mathcal{T}(F, \emptyset)$ **shows** $s \cdot \vartheta = s$
using *subst-Term-empty-eq*[*OF s, of Var*] **by** *simp*

lemma *subst-subst-Term-empty*:
 $s : \sigma$ in $\mathcal{T}(F, \emptyset) \implies s \cdot \vartheta \cdot \rho = s \cdot \text{undefined}$
apply (*unfold subst-subst*)
using *subst-Term-empty-eq*.

lemma *in-dom-Term-empty-subst-id*:
 $s \in \text{dom } \mathcal{T}(F, \emptyset) \implies s \cdot \vartheta = s$
by (*auto elim: in-dom-hastypeE simp: subst-Term-empty-id*)

lemma *in-dom-Term-empty-subst-subst*:
 $s \in \text{dom } \mathcal{T}(F, \emptyset) \implies s \cdot \vartheta \cdot \rho = s \cdot \text{undefined}$
apply (*elim in-dom-hastypeE*)
using *subst-subst-Term-empty*.

lemma *map-eval-Term-empty-eq*: $ss :_l \sigma s$ in $\mathcal{T}(F, \emptyset) \implies [I[s]\alpha. s \leftarrow ss] = [I[s]\alpha'. s \leftarrow ss]$

by (*auto 0 3 simp: in-set-conv-nth list-all2-conv-all-nth eval-Term-empty-eq*)

lemma *map-subst-Term-empty-eq*: $ss :_l \sigma s$ in $\mathcal{T}(F, \emptyset) \implies [s.\vartheta. s \leftarrow ss] = [s.\varrho. s \leftarrow ss]$

using *map-eval-Term-empty-eq*.

lemma *map-subst-Term-empty-id*: $ss :_l \sigma s$ in $\mathcal{T}(F, \emptyset) \implies [s.\vartheta. s \leftarrow ss] = ss$

using *map-subst-Term-empty-eq[of - - - Var]* **by** *simp*

lemma *map-subst-subst-Term-empty*:

$ss :_l \sigma s$ in $\mathcal{T}(F, \emptyset) \implies [s.\vartheta.\varrho. s \leftarrow ss] = [s.undefined. s \leftarrow ss]$

apply (*unfold subst-subst*)

using *map-subst-Term-empty-eq*.

context fixes $\vartheta :: 'v \Rightarrow ('f, 'w)$ term **begin**

interpretation *sorted-bijection* $\lambda s. s.\vartheta$ $\mathcal{T}(F, \emptyset)$ $\mathcal{T}(F, \emptyset)$

proof

show *bij-betw* ($\lambda s. s.\vartheta$) (*dom* $\mathcal{T}(F, \emptyset)$) (*dom* $\mathcal{T}(F, \emptyset)$)

proof (*intro bij-betwI*)

show ($\lambda s. s.undefined$) : *dom* $\mathcal{T}(F, \emptyset) \rightarrow$ *dom* $\mathcal{T}(F, \emptyset)$

by (*auto simp: in-dom-Term-empty-imp-subst*)

qed (*auto simp del: subst-subst-compose*)

simp: subst-subst subst-Term-empty-id in-dom-Term-empty-subst-id in-dom-Term-empty-imp-subst)

qed (*auto simp: subst-Term-empty-hastype*)

lemmas *sorted-bijection-Term-empty = sorted-bijection-axioms*

lemmas *bij-betw-dom-Term-empty = bij*

lemmas *bij-betw-sort-Term-empty = bij-betw-sort*

lemma *all-Term-empty-subst-iff*:

$(\forall s : \sigma$ in $\mathcal{T}(F, \emptyset). P (s.\vartheta)) \iff (\forall s : \sigma$ in $\mathcal{T}(F, \emptyset). P s)$

by (*simp add: all-in-target-iff*)

end

Canonically, let us use unit as the type of variables for ground terms.

abbreviation *gTerm* ($\langle \mathcal{T}'(-) \rangle$) **where** $\mathcal{T}(F) \equiv \mathcal{T}(F, \lambda x::unit. None)$

4.6.1 Cardinality of Sorts

The emptiness, finiteness, and cardinality of a sort w.r.t. a signature is those of the set of ground terms of that sort.

definition *empty-sort* **where**

$$\text{empty-sort } F \sigma \longleftrightarrow \{s. s : \sigma \text{ in } \mathcal{T}(F)\} = \{\}$$

definition *finite-sort* **where**

$$\text{finite-sort } F \sigma \longleftrightarrow \text{finite } \{s. s : \sigma \text{ in } \mathcal{T}(F)\}$$

definition *card-of-sort* **where**

$$\text{card-of-sort } F \sigma = \text{card } \{s. s : \sigma \text{ in } \mathcal{T}(F)\}$$

The definitions fix the type of the variables (that never occur) to unit. We prove that the choice of the type is irrelevant.

lemma *finite-sort*: $\text{finite } \{s. s : \sigma \text{ in } \mathcal{T}(F, \emptyset)\} \longleftrightarrow \text{finite-sort } F \sigma$

apply (*unfold finite-sort-def*)

using *bij-betw-finite[OF bij-betw-sort-Term-empty]*.

lemma *card-of-sort*: $\text{card } \{s. s : \sigma \text{ in } \mathcal{T}(F, \emptyset)\} = \text{card-of-sort } F \sigma$

apply (*unfold card-of-sort-def*)

using *bij-betw-same-card[OF bij-betw-sort-Term-empty]*.

lemma *empty-sort*: $\{s. s : \sigma \text{ in } \mathcal{T}(F, \emptyset)\} = \{\} \longleftrightarrow \text{empty-sort } F \sigma$

apply (*unfold empty-sort-def*)

by (*metis card-eq-0-iff card-of-sort finite.emptyI finite-sort*)

lemma *empty-sortD[simp]*: $\text{empty-sort } F \sigma \Longrightarrow \neg s : \sigma \text{ in } \mathcal{T}(F, \emptyset)$

using *empty-sort[of σ F] by auto*

lemma *empty-sort-imp-card[simp]*: $\text{empty-sort } F \sigma \Longrightarrow \text{card-of-sort } F \sigma = 0$

by (*auto simp: card-of-sort-def*)

lemma *empty-sort-imp-finite[simp]*: $\text{empty-sort } F \sigma \Longrightarrow \text{finite-sort } F \sigma$

by (*auto simp: finite-sort-def*)

lemma *empty-sortI*: $(\bigwedge s. \neg s : \sigma \text{ in } \mathcal{T}(F, \emptyset)) \Longrightarrow \text{empty-sort } F \sigma$

using *empty-sort[of σ F] by auto*

lemma *not-empty-sortE*: $\neg \text{empty-sort } F \sigma \Longrightarrow (\bigwedge s. s : \sigma \text{ in } \mathcal{T}(F, \emptyset) \Longrightarrow \text{thesis}) \Longrightarrow \text{thesis}$

using *empty-sort[of σ F] by auto*

lemma *finite-sort-bij*:

assumes *fin*: *finite-sort* $F \sigma$

shows $\exists f. \text{bij-betw } f \{s. s : \sigma \text{ in } \mathcal{T}(F, \emptyset)\} \{0..<\text{card-of-sort } F \sigma\}$

proof –

from *ex-bij-betw-finite-nat[OF fin[unfolded finite-sort-def]]*

obtain *h* **where**

bij-betw *h* $\{t. t : \sigma \text{ in } \mathcal{T}(F)\} \{0..<\text{card-of-sort } F \sigma\}$

by (*auto simp add: card-of-sort*)

from *bij-betw-trans[OF bij-betw-sort-Term-empty this]*

show *?thesis* **by auto**

qed

4.6.2 Enumerating Ground Terms

definition *index-of-term* $F =$

$(\text{SOME } f. \forall \sigma. \text{finite-sort } F \ \sigma \longrightarrow \text{bij-betw } f \ \{t. t : \sigma \text{ in } \mathcal{T}(F, \emptyset)\} \ \{0..<\text{card-of-sort } F \ \sigma\})$

definition *term-of-index* $F \ \sigma = \text{inv-into } \{t. t : \sigma \text{ in } \mathcal{T}(F, \emptyset)\} \ (\text{index-of-term } F)$

lemma *index-of-term-bij*:

assumes $\text{fin}: \text{finite-sort } F \ \sigma$

shows $\text{bij-betw } (\text{index-of-term } F) \ \{t. t : \sigma \text{ in } \mathcal{T}(F, \emptyset)\} \ \{0..<\text{card-of-sort } F \ \sigma\}$

(is $\text{bij-betw} - (?T \ \sigma) \ (?I \ \sigma)$ **)**

proof –

have $\forall \sigma \in \text{Collect } (\text{finite-sort } F). \exists f. \text{bij-betw } f \ (?T \ \sigma) \ (?I \ \sigma)$

by $(\text{auto intro!}: \text{finite-sort-bij})$

from $\text{bchoice}[OF \ \text{this}]$

obtain f **where** $f: \bigwedge \sigma. \text{finite-sort } F \ \sigma \implies \text{bij-betw } (f \ \sigma) \ (?T \ \sigma) \ (?I \ \sigma)$

by auto

define g **where** $g = (\lambda t. f \ (\text{the } (\mathcal{T}(F, \emptyset) \ t)) \ t)$

have $\forall \sigma. \text{finite-sort } F \ \sigma \longrightarrow \text{bij-betw } g \ (?T \ \sigma) \ (?I \ \sigma)$

by $(\text{auto simp}: g\text{-def intro!}: \text{bij-betw-cong}[THEN \text{iffD1}, OF \ f])$

then have $\exists g. \forall \sigma. \text{finite-sort } F \ \sigma \longrightarrow \text{bij-betw } g \ (?T \ \sigma) \ (?I \ \sigma)$

by auto

from $\text{someI-ex}[OF \ \text{this}, \text{folded } \text{index-of-term-def}] \ \text{fin}$

show $?thesis$ **by** auto

qed

lemma *term-of-index-of-term*:

assumes $t: t : \sigma \text{ in } \mathcal{T}(F, \emptyset)$ **and** $\text{fin}: \text{finite-sort } F \ \sigma$

shows $\text{term-of-index } F \ \sigma \ (\text{index-of-term } F \ t) = t$

apply $(\text{unfold } \text{term-of-index-def})$

apply $(\text{rule } \text{bij-betw-inv-into-left}[OF \ \text{index-of-term-bij}])$

using assms **by** auto

lemma *index-of-term-of-index*:

assumes $\text{fin}: \text{finite-sort } F \ \sigma$ **and** $n < \text{card-of-sort } F \ \sigma$

shows $\text{index-of-term } F \ (\text{term-of-index } F \ \sigma \ n) = n$

apply $(\text{unfold } \text{term-of-index-def})$

apply $(\text{rule } \text{bij-betw-inv-into-right}[OF \ \text{index-of-term-bij}])$

using assms **by** auto

lemma *term-of-index-bij*:

assumes $\text{fin}: \text{finite-sort } F \ \sigma$

shows $\text{bij-betw } (\text{term-of-index } F \ \sigma) \ \{0..<\text{card-of-sort } F \ \sigma\} \ \{t. t : \sigma \text{ in } \mathcal{T}(F, \emptyset)\}$

by $(\text{simp add}: \text{bij-betw-inv-into } \text{fin } \text{index-of-term-bij } \text{term-of-index-def})$

4.7 Subsignatures

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

lemmas *Term-subssset* = *Term-mono-left*[*OF subssig*]

lemmas *hastype-in-Term-sub* = *Term-subssset*[*THEN subsssetD*]

lemma *subsignature*: $f : \sigma s \rightarrow \tau$ in $F \implies f : \sigma s \rightarrow \tau$ 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 $F :: ('f, 's)\ \text{ssig}$ **and** $A :: 'a \rightarrow 's$ **and** I

and $G :: ('f, 's)\ \text{ssig}$ **and** $B :: 'a \rightarrow 's$ **and** J +

assumes *subcar*: $A \subseteq_m B$

assumes *subintp*: $f : \sigma s \rightarrow \tau$ in $F \implies as :_l \sigma s$ in $A \implies I f as = J f as$

begin

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

lemma *subeval*:

assumes $s : s : \sigma$ in $\mathcal{T}(F, V)$ **and** $\alpha : \alpha :_s V \rightarrow A$ **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* $\sigma s \tau$)

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* $F\ \mathcal{T}(F, V)\ \text{Fun}\ G\ \mathcal{T}(G, W)\ \text{Fun}$

apply *unfold-locales*
using *FG VW Term-mono[OF FG VW]* **by** *auto*

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

definition *constant-at f $\sigma s i$* \equiv
 $\forall as\ b. as :_i \sigma s\ in\ A \longrightarrow A\ b = A\ (as!i) \longrightarrow If\ (as[i:=b]) = If\ as$

lemma *constant-atI[intro]*:
assumes $\bigwedge as\ b. as :_i \sigma s\ in\ A \Longrightarrow A\ b = A\ (as!i) \Longrightarrow If\ (as[i:=b]) = If\ as$
shows *constant-at f $\sigma s i$* **using** *assms* **by** (*auto simp: constant-at-def*)

lemma *constant-atD*:
 $constant-at\ f\ \sigma s\ i \Longrightarrow as :_i \sigma s\ in\ A \Longrightarrow A\ b = A\ (as!i) \Longrightarrow If\ (as[i:=b]) = If\ as$
by (*auto simp: constant-at-def*)

lemma *constant-atE[elim]*:
assumes *constant-at f $\sigma s i$*
and $(\bigwedge as\ b. as :_i \sigma s\ in\ A \Longrightarrow A\ b = A\ (as!i) \Longrightarrow If\ (as[i:=b]) = If\ as) \Longrightarrow$
thesis
shows *thesis* **using** *assms* **by** (*auto simp: constant-at-def*)

definition *constant-term-on s x* $\equiv \forall \alpha\ a. I[s]\alpha(x:=a) = I[s]\alpha$

lemma *constant-term-onI*:
assumes $\bigwedge \alpha\ a. I[s]\alpha(x:=a) = I[s]\alpha$ **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** $I[s]\alpha(x:=a) = I[s]\alpha$
using *assms* **by** (*auto simp: constant-term-on-def*)

lemma *constant-term-onE*:
assumes *constant-term-on s x* **and** $(\bigwedge \alpha\ a. I[s]\alpha(x:=a) = I[s]\alpha) \Longrightarrow$ *thesis*
shows *thesis* **using** *assms* **by** (*auto simp: constant-term-on-def*)

lemma *constant-term-on-extra-var: x \notin vars s \Longrightarrow 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* $\equiv \forall x. constant-term-on\ s\ x$

lemma *constant-termI*: **assumes** $\bigwedge x. constant-term-on\ s\ x$ **shows** *constant-term s*

```

using assms by (auto simp: constant-term-def)

lemma ground-imp-constant: vars  $s = \{\}$   $\implies$  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

```

We introduce the sort signature for abstract contexts:

```

fun aContext where
  aContext  $F A$  (Hole, $\sigma$ ) = Some  $\sigma$ 
| aContext  $F A$  (More  $f$   $ls$   $C$   $rs$ ,  $\sigma$ ) = do {
   $\rho s \leftarrow$  those (map  $A$   $ls$ );
   $\mu \leftarrow$  aContext  $F A$  ( $C$ , $\sigma$ );
   $\nu s \leftarrow$  those (map  $A$   $rs$ );
   $F$  ( $f$ ,  $\rho s @ \mu \# \nu s$ )}

```

Term contexts are abstract contexts in the term algebra.

```

abbreviation Context ( $\langle \langle \mathcal{C}'(-,-) \rangle \rangle [1,1]1000$ ) where
   $\mathcal{C}(F, V) \equiv$  aContext  $F \mathcal{T}(F, V)$ 

```

```

lemma Hole-hastype[simp]: Hole :  $\sigma \rightarrow \tau$  in aContext  $F A \iff \sigma = \tau$ 
and More-hastype: More  $f$   $ls$   $C$   $rs$  :  $\sigma \rightarrow \tau$  in aContext  $F A \iff (\exists \rho s \mu \nu s.$ 
   $f : \rho s @ \mu \# \nu s \rightarrow \tau$  in  $F \wedge$ 
   $ls ;_1 \rho s$  in  $A \wedge$ 
   $C : \sigma \rightarrow \mu$  in aContext  $F A \wedge$ 
   $rs ;_1 \nu s$  in  $A)$ 
by (auto simp: hastype-list-iff-those bind-eq-Some-conv fun-hastype-def
  intro!: hastypeI)

```

```

lemma More-hastypeI:
  assumes  $f : \rho s @ \mu \# \nu s \rightarrow \tau$  in  $F$ 
  and  $ls ;_1 \rho s$  in  $A$ 
  and  $C : \sigma \rightarrow \mu$  in aContext  $F A$ 
  and  $rs ;_1 \nu s$  in  $A$ 
  shows More  $f$   $ls$   $C$   $rs$  :  $\sigma \rightarrow \tau$  in aContext  $F A$ 
  using assms by (auto simp: More-hastype)

```

```

lemma hastype-aContext-induct[consumes 1, case-names Hole More]:

```

assumes $C: C : \sigma \rightarrow \tau$ in *aContext F A*
and hole: $P \square \sigma$
and more: $\bigwedge f \mu s \varrho \nu s \tau ls C rs.$
 $f : \mu s @ \varrho \# \nu s \rightarrow \tau$ in $F \implies$
 $ls :_l \mu s$ in $A \implies$
 $C : \sigma \rightarrow \varrho$ in *aContext F A* \implies
 $P C \varrho \implies$
 $rs :_l \nu s$ in $A \implies$
 $P (More f ls C rs) \tau$
shows $P C \tau$
using C
proof (*induct C arbitrary: τ*)
case *Hole*
with hole show *?case by auto*
next
case (*More f ls C rs*)
from $\langle More f ls C rs : \sigma \rightarrow \tau$ in *aContext F A* \rangle
[*unfolded More-hastype*]
obtain $\varrho s \mu \nu s$
where $f: f : \varrho s @ \mu \# \nu s \rightarrow \tau$ in F
and $ls: ls :_l \varrho s$ in A
and $C: C : \sigma \rightarrow \mu$ in *aContext F A*
and $rs: rs :_l \nu s$ in A **by auto**
show *?case*
using $More(1)[OF C] more[OF f ls C - rs]$
by (*auto simp: bind-eq-Some-conv*)
qed

lemma *hastype-aContext-cases*[*consumes 1, case-names Hole More*]:

assumes $C: C : \sigma \rightarrow \tau$ in *aContext F A*
and hole: $C = \square \implies thesis$
and more: $\bigwedge f \mu s \varrho \nu s ls D rs.$
 $C = More f ls D rs \implies$
 $f : \mu s @ \varrho \# \nu s \rightarrow \tau$ in $F \implies$
 $ls :_l \mu s$ in $A \implies$
 $D : \sigma \rightarrow \varrho$ in *aContext F A* \implies
 $rs :_l \nu s$ in $A \implies$
thesis
shows *thesis*
proof (*cases C*)
case *Hole*
with hole show *?thesis by auto*
next
case (*More f ls D rs*)
from C [*unfolded this More-hastype*]
obtain $\varrho s \mu \nu s$
where $f: f : \varrho s @ \mu \# \nu s \rightarrow \tau$ in F
and $ls: ls :_l \varrho s$ in A
and $D: D : \sigma \rightarrow \mu$ in *aContext F A*

and $rs :_l \nu s$ in A **by** *auto*
show *?thesis*
using *more[OF More f ls D rs]*.
qed

lemma (*in sorted-map*) *map-args-actxt-hastype*:
assumes $C : \sigma \rightarrow \tau$ in *aContext F A*
shows *map-args-actxt f C : $\sigma \rightarrow \tau$ in aContext F B*
using *assms*
apply (*induct C arbitrary: τ*)
by (*auto dest!: sorted-map-list simp: More-hastype*)

context *sorted-algebra* **begin**

lemma *intp-ctxt-hastype*:
assumes $C : C : \sigma \rightarrow \tau$ in *aContext F A* **and** $a : a : \sigma$ in A
shows $I\langle C; a \rangle : \tau$ in A
using C
proof (*induct arbitrary: τ*)
case *Hole*
with a **show** *?case by simp*
next
case (*More f ls C rs*)
then show *?case by (auto intro!: sort-matches list-all2-appendI simp: More-hastype)*
qed

lemma *ctxt-has-same-type*:
assumes $C : C : \sigma \rightarrow \tau$ in *aContext F A* **and** $a : \sigma$ in A
shows $I\langle C; a \rangle : \tau' \text{ in } A \longleftrightarrow \tau' = \tau$
using *assms by (auto simp: has-same-type intp-ctxt-hastype)*

lemma *eval-ctxt-hastype*:
assumes $C : C : \sigma \rightarrow \tau$ in $\mathcal{C}(F, V)$ **and** $\alpha : \alpha :_s V \rightarrow A$
shows $I\llbracket C \rrbracket_c \alpha : \sigma \rightarrow \tau$ in *aContext F A*
using *sorted-map.map-args-actxt-hastype[OF eval-sorted-map[OF α] C]*.

end

lemmas *apply-ctxt-hastype = term.intp-ctxt-hastype*
lemmas *subst-ctxt-hastype = term.eval-ctxt-hastype*

lemma *subt-in-dom*:
assumes $s : s \in \text{dom } \mathcal{T}(F, V)$ **and** $st : s \geq t$ **shows** $t \in \text{dom } \mathcal{T}(F, V)$
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* $\langle \text{Fun } f \text{ ss} \in \text{dom } \mathcal{T}(F, V) \rangle \langle u \in \text{set } \text{ss} \rangle$]
subt.IH
show *?case by auto*
qed

lemma *hastype-context-decompose*:

assumes $C\langle t \rangle : \tau$ in $\mathcal{T}(F, V)$
shows $\exists \sigma. C : \sigma \rightarrow \tau$ in $\mathcal{C}(F, V) \wedge t : \sigma$ in $\mathcal{T}(F, V)$
using *assms*
proof (*induct C arbitrary: τ*)
case *Hole*
then show *?case by auto*
next
case (*More f bef C aft τ*)
from *More(2)* **have** $\text{Fun } f \text{ (bef @ } C\langle t \rangle \# \text{aft)} : \tau$ in $\mathcal{T}(F, V)$ **by** *auto*
from *this[unfolded Fun-hastype]* **obtain** σs **where**
 $f : f : \sigma s \rightarrow \tau$ in F **and** $\text{list: bef @ } C\langle t \rangle \# \text{aft} :_i \sigma s$ in $\mathcal{T}(F, V)$
by *auto*
from *list* **have** $\text{len: length } \sigma s = \text{length bef} + \text{Suc (length aft)}$
by (*simp add: list-all2-conv-all-nth*)
let $?i = \text{length bef}$
from *len* **have** $?i < \text{length } \sigma s$ **by** *auto*
hence $\text{id: take } ?i \sigma s @ \sigma s ! ?i \# \text{drop (Suc } ?i) \sigma s = \sigma s$
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** $\text{bef: bef} :_i \text{take } ?i \sigma s$ in $\mathcal{T}(F, V)$
by (*metis (no-types, lifting) append-eq-conv-conj list-all2-append1*)
from *list* **have** $\text{aft: aft} :_i \text{drop (Suc } ?i) \sigma s$ 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** $C : C : \sigma \rightarrow \sigma s ! ?i$ in $\mathcal{C}(F, V)$ **and** $t : t : \sigma$ in $\mathcal{T}(F, V)$
by *auto*
show *?case*
by (*intro exI[of - σ] conjI More-hastypeI[OF - bef - aft, of - $\sigma s ! ?i$] C t, unfold id, rule f*)
qed

lemma *apply-ctxt-in-dom-imp-in-dom*:

assumes $C\langle t \rangle \in \text{dom } \mathcal{T}(F, V)$
shows $t \in \text{dom } \mathcal{T}(F, V)$
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 $C : \sigma \rightarrow \tau$ in $\mathcal{C}(F, V)$
using *hastype-context-decompose[OF C] t* **by** (*auto simp: has-same-type*)

```

end
theory Basic-Terms
  imports Sorted-Contexts
begin

```

```

lemma map-le-map-add-left:
  assumes disj: dom m ∩ dom n = {}
  shows m ⊆m m ++ n
  using map-le-map-add map-add-comm[OF disj] by auto

```

6 Basic Terms

Given two signatures C and D , a term $f(s_1, \dots, s_n)$ is called a basic term if $f : [\sigma_1, \dots, \sigma_n] \rightarrow \tau$ in D and $s_1 : \sigma_1, \dots, s_n : \sigma_n$ in $\mathcal{T}(C, X)$. We define the sorted set of basic terms as follows.

```

fun Basic-Term :: ('f, 's) ssig ⇒ ('f, 's) ssig ⇒ ('v → 's) ⇒ ('f, 'v) term → 's
  (TB'(-, -))
  where TB(C, D, X) (Var x) = None
        | TB(C, D, X) (Fun f ss) = (case those (map T(C, X) ss) of Some σs ⇒ D (f, σs)
        | - ⇒ None)

```

abbreviation *ground-Basic-Term* (T_B'(-, -)) **where**
 T_B(C, D) ≡ T_B(C, D, λx. None) :: ('f, unit) term → 's

```

lemma Var-hastype-Basic:
  Var x : σ in TB(C, D, X) ⟷ False by (simp add: hastype-def)

```

```

lemma Fun-hastype-Basic:
  Fun f ss : τ in TB(C, D, X) ⟷ (∃ σs. f : σs → τ in D ∧ ss :l σs in T(C, X))
  apply (unfold hastype-list-iff-those)
  by (auto simp: hastype-def fun-hastype-def split: option.split-asm)

```

```

lemma hastype-Basic:
  s : τ in TB(C, D, X) ⟷ (∃ f ss σs. s = Fun f ss ∧ f : σs → τ in D ∧ ss :l σs
  in T(C, X))
  by (cases s, auto simp: Var-hastype-Basic Fun-hastype-Basic)

```

```

lemma in-dom-Basic:
  s ∈ dom TB(C, D, X) ⟷ (∃ f ss σs τ. s = Fun f ss ∧ f : σs → τ in D ∧ ss :l
  σs in T(C, X))
  by (auto simp: in-dom-iff-ex-type hastype-Basic)

```

```

lemma hastype-BasicE:
  assumes s : τ in TB(C, D, X)
  and ⋀ f ss σs. s = Fun f ss ⇒ f : σs → τ in D ⇒ ss :l σs in T(C, X) ⇒
  thesis
  shows thesis

```

using *assms* by (*auto simp: hastype-Basic*)

lemma *hastype-BasicI*:

$s = \text{Fun } f \text{ } ss \implies f : \sigma s \rightarrow \tau \text{ in } D \implies ss :_{\iota} \sigma s \text{ in } \mathcal{T}(C, X) \implies s : \tau \text{ in } \mathcal{T}_B(C, D, X)$
by (*auto simp: hastype-Basic*)

lemma *Basic-mono*:

assumes $D : D \subseteq_m D'$ and $C : C \subseteq_m C'$ and $X : X \subseteq_m X'$

shows $\mathcal{T}_B(C, D, X) \subseteq_m \mathcal{T}_B(C', D', X')$

proof (*safe intro!: subsetI elim!: hastype-BasicE*)

fix $f \text{ } ss \text{ } \sigma s \text{ } \tau$ assume $f : f : \sigma s \rightarrow \tau \text{ in } D$ and $ss : ss :_{\iota} \sigma s \text{ in } \mathcal{T}(C, X)$

from *subssigD*[*OF D f*] *Term-mono*[*OF C X*, *THEN subset-hastype-listD*, *OF ss*]

show $\text{Fun } f \text{ } ss : \tau \text{ in } \mathcal{T}_B(C', D', X')$

by (*intro hastype-BasicI*[*OF refl*])

qed

Basic terms are terms of the joint signature, if the signatures are disjoint.

lemma *Basic-Term*:

assumes *disj*: $\text{dom } C \cap \text{dom } D = \{\}$

shows $\mathcal{T}_B(C, D, X) \subseteq_m \mathcal{T}(C++D, X)$

proof (*intro subsetI*)

fix $s \text{ } \sigma$

assume $s : s : \sigma \text{ in } \mathcal{T}_B(C, D, X)$

show $s : \sigma \text{ in } \mathcal{T}(C++D, X)$

proof (*cases s*)

case (*Var x1*)

with s show *?thesis* by (*auto simp: Var-hastype-Basic*)

next

case *sfss*: (*Fun f ss*)

with s obtain τs where $ss : ss :_{\iota} \tau s \text{ in } \mathcal{T}(C, X)$ and $f : f : \tau s \rightarrow \sigma \text{ in } D$

by (*auto simp: Fun-hastype-Basic*)

from *disj* have $C : C \subseteq_m C++D$ by (*simp add: map-le-map-add-left*)

from *Term-mono-left*[*OF C*, *THEN subsetD*] *ss*

have *ssF*: $ss :_{\iota} \tau s \text{ in } \mathcal{T}(C++D, X)$ by (*metis subset-hastype-listD subsetI*)

have $D \subseteq_m C++D$ by *auto*

from *subssigD*[*OF this f*] have *fF*: $f : \tau s \rightarrow \sigma \text{ in } C++D$.

with *ssF* have $\text{Fun } f \text{ } ss : \sigma \text{ in } \mathcal{T}(C++D, X)$ by (*auto simp: Fun-hastype*)

then show *?thesis* by (*auto simp: sfss*)

qed

qed

Basic terms are preserved under constructor substitution.

lemma *subst-Basic-Term*:

assumes $\vartheta : \vartheta :_s X \rightarrow \mathcal{T}(C, V)$ and $s : s : \tau \text{ in } \mathcal{T}_B(C, D, X)$

shows $s \cdot \vartheta : \tau \text{ in } \mathcal{T}_B(C, D, V)$

proof –

from *s*[*unfolded hastype-Basic*]

obtain $f \text{ } ss \text{ } \sigma s$ where $s : s = \text{Fun } f \text{ } ss$ and $f : f : \sigma s \rightarrow \tau \text{ in } D$ and $ss : ss :_{\iota} \sigma s$

```

in  $\mathcal{T}(C, X)$ 
  by (auto simp: hastype-Basic)
  from map-subst-hastype[OF  $\vartheta$  ss] f
  show ?thesis by (auto simp: hastype-Basic s)
qed

```

```

lemma in-dom-Basic-Term-imp-vars:  $s \in \text{dom } \mathcal{T}_B(C, D, V) \implies x \in \text{vars } s \implies x \in \text{dom } V$ 
  apply (elim in-dom-hastypeE)
  apply (cases s)
  by (auto simp: Fun-hastype-Basic list-all2-conv-all-nth in-set-conv-nth dest!: hastype-in-Term-imp-vars)

```

```

lemma in-dom-Basic-empty-subst-id:
  assumes  $s \in \text{dom } \mathcal{T}_B(C, D, \emptyset)$ 
  shows  $s \cdot \vartheta = s$ 
  using assms
  by (auto elim!: in-dom-hastypeE hastype-BasicE simp: o-def map-subst-Term-empty-id)

```

```

lemma in-dom-Basic-empty-subst-subst:
  assumes  $s \in \text{dom } \mathcal{T}_B(C, D, \emptyset)$ 
  shows  $s \cdot \vartheta \cdot \tau = s \cdot \text{undefined}$ 
  using assms
  by (auto elim!: in-dom-hastypeE hastype-BasicE simp: o-def map-subst-subst-Term-empty)

```

end

theory FinFun-RBT-Impl

imports

FinFun.FinFun

HOL-Library.RBT

begin

fun def-option :: $'a \Rightarrow 'a \text{ option} \Rightarrow 'a$ **where**

def-option d (Some x) = x

| *def-option d None = d*

lift-definition ff-of-rbt :: $'b \times ('a::\text{linorder}, 'b) \text{ rbt} \Rightarrow ('a, 'b) \text{ finfun}$ **is**

$\lambda dt x. \text{def-option (fst dt) (RBT.lookup (snd dt) x)}$

unfolding *finfun-def*

proof (*standard, goal-cases*)

case (*1 dt*)

have $\{a. \text{def-option (fst dt) (RBT.lookup (snd dt) a)} \neq \text{fst dt}\} \subseteq \text{set (RBT.keys (snd dt))}$

apply *clarsimp*

subgoal for *x*

by (*cases RBT.lookup (snd dt) x, auto simp: RBT.keys-entries RBT.lookup-in-tree*)

done

```

from finite-subset[OF this]
show ?case
  by (intro exI[of - fst dt], auto)
qed

```

```

code-datatype ff-of-rbt

```

```

declare [[code drop: finfun-const]]
lemma finfun-const-impl[code]: finfun-const c = ff-of-rbt (c, RBT.empty)
  by transfer auto

```

```

declare [[code drop: finfun-apply]]
lemma finfun-apply-impl[code]: finfun-apply (ff-of-rbt (c, t)) x = def-option c
(RBT.lookup t x)
  by transfer auto

```

```

declare [[code drop: finfun-update]]
lemma finfun-update-impl[code]: finfun-update (ff-of-rbt (c, t)) x y = ff-of-rbt (c,
RBT.insert x y t)
  by transfer auto

```

```

end

```

7 Computing Nonempty and Infinite sorts

This theory provides two algorithms, which both take a description of a set of sorts with their constructors. The first algorithm computes the set of sorts that are nonempty, i.e., those sorts that are inhabited by ground terms; and the second algorithm computes the set of sorts that are infinite, i.e., where one can build arbitrary large ground terms. Furthermore, the cardinalities of finite sorts can be computed (exactly, or up to a certain precision).

```

theory Compute-Nonempty-Infinite-Sorts

```

```

  imports
    Sorted-Terms
    LP-Duality.Minimum-Maximum
    Matrix.Utility
    FinFun-RBT-Impl

```

```

begin

```

```

lemma finite-set-Cons:

```

```

  assumes A: finite A and B: finite B
  shows finite (set-Cons A B)

```

```

proof –

```

```

  have set-Cons A B = case-prod (#) ‘(A × B) by (auto simp: set-Cons-def)

```

```

  then show ?thesis

```

```

    by (simp add: finite-imageI[OF finite-cartesian-product[OF A B], of case-prod

```

(#))
qed

lemma *finite-listset*:
assumes $\forall A \in \text{set } As. \text{finite } A$
shows *finite (listset As)*
using *assms*
by (*induct As*) (*auto simp: finite-set-Cons*)

lemma *listset-conv-nth*:
 $xs \in \text{listset } As = (\text{length } xs = \text{length } As \wedge (\forall i < \text{length } As. xs ! i \in As ! i))$
proof (*induct As arbitrary: xs*)
case (*Cons A As xs*) **then show** *?case*
by (*cases xs*) (*auto simp: set-Cons-def nth-Cons nat.splits*)
qed *auto*

lemma *card-listset*: **assumes** $\bigwedge A. A \in \text{set } As \implies \text{finite } A$
shows $\text{card } (\text{listset } As) = \text{prod-list } (\text{map } \text{card } As)$
using *assms*
proof (*induct As*)
case (*Cons A As*)
have *sC: set-Cons A B = case-prod (#) ‘ (A × B) for B* **by** (*auto simp: set-Cons-def*)
have *IH: prod-list (map card As) = card (listset As)* **using** *Cons* **by** *auto*
have $\text{card } A * \text{card } (\text{listset } As) = \text{card } (A \times \text{listset } As)$
by (*simp add: card-cartesian-product*)
also have $\dots = \text{card } ((\lambda (a,as). \text{Cons } a \text{ as}) ‘ (A \times \text{listset } As))$
by (*subst card-image, auto simp: inj-on-def*)
finally
show *?case* **by** (*simp add: sC IH*)
qed *auto*

7.1 Deciding the nonemptyness of all sorts under consideration

function *compute-nonempty-main* :: $'\tau \text{ set} \Rightarrow ((f \times '\tau \text{ list}) \times '\tau) \text{ list} \Rightarrow '\tau \text{ set}$
where
 $\text{compute-nonempty-main } ne \text{ ls} = (\text{let } \text{rem-ls} = \text{filter } (\lambda f. \text{snd } f \notin ne) \text{ ls in}$
 $\text{case partition } (\lambda ((-,args),-). \text{set } args \subseteq ne) \text{ rem-ls of}$
 $(\text{new}, \text{rem}) \Rightarrow \text{if } \text{new} = [] \text{ then } ne \text{ else } \text{compute-nonempty-main } (ne \cup \text{set } (\text{map } \text{snd } \text{new})) \text{ rem})$
by *pat-completeness auto*

termination

proof (*relation measure (length o snd), goal-cases*)
case $2 \text{ } ne \text{ ls } \text{rem-ls } \text{new } \text{rem}$
have $\text{length } \text{new} + \text{length } \text{rem} = \text{length } \text{rem-ls}$
using $2(2) \text{ sum-length-filter-compl[of - rem-ls]}$ **by** (*auto simp: o-def*)
with $2(3) \text{ have } \text{length } \text{rem} < \text{length } \text{rem-ls}$ **by** (*cases new, auto*)

```

also have ...  $\leq$  length ls using 2(1) by auto
finally show ?case by simp
qed simp

declare compute-nonempty-main.simps[simp del]

definition compute-nonempty-sorts :: (('f × 'τ list) × 'τ) list ⇒ 'τ set where
  compute-nonempty-sorts Cs = compute-nonempty-main {} Cs

lemma compute-nonempty-sorts:
  assumes distinct (map fst Cs)
shows compute-nonempty-sorts Cs = {τ. ¬ empty-sort (map-of Cs) τ} (is - = ?NE)
proof -
  let ?TC =  $\mathcal{T}(\text{map-of } Cs)$ 
  have  $ne \subseteq ?NE \implies \text{set } ls \subseteq \text{set } Cs \implies \text{snd } '(\text{set } Cs - \text{set } ls) \subseteq ne \implies$ 
    compute-nonempty-main ne ls = ?NE for ne ls
  proof (induct ne ls rule: compute-nonempty-main.induct)
    case (1 ne ls)
    note  $ne = 1(2)$ 
    define rem-ls where rem-ls = filter (λ f. snd f ∉ ne) ls
    have rem-ls:  $\text{set } rem-ls \subseteq \text{set } Cs$ 
       $\text{snd } '(\text{set } Cs - \text{set } rem-ls) \subseteq ne$ 
    using 1(2-) by (auto simp: rem-ls-def)
    obtain new rem where part: partition (λ((f, args), target). set args ⊆ ne)
rem-ls = (new,rem) by force
    have [simp]: compute-nonempty-main ne ls = (if new = [] then ne else compute-nonempty-main (ne ∪ set (map snd new)) rem)
    unfolding compute-nonempty-main.simps[of ne ls] Let-def rem-ls-def[symmetric]
  part by auto
    have new:  $\text{set } (\text{map } \text{snd } new) \subseteq ?NE$ 
  proof
    fix τ
    assume τ ∈  $\text{set } (\text{map } \text{snd } new)$ 
    then obtain f args where ((f,args),τ) ∈ set rem-ls and args:  $\text{set } args \subseteq ne$ 
  using part by auto
    with rem-ls have ((f,args),τ) ∈ set Cs by auto
    with assms have map-of Cs (f,args) = Some τ by auto
    hence fC:  $f : args \rightarrow \tau$  in map-of Cs by (simp add: fun-hastype-def)
    from args ne empty-sortI have  $\forall \text{tau. } \exists t. \text{tau} \in \text{set } args \longrightarrow t : \text{tau}$  in ?TC
  by force
    from choice[OF this] obtain ts where  $\bigwedge \text{tau. } \text{tau} \in \text{set } args \implies \text{ts } \text{tau} : \text{tau}$ 
in ?TC by auto
    hence  $\text{Fun } f (\text{map } \text{ts } args) : \tau$  in ?TC
    apply (intro Fun-hastypeI[OF fC])
    by (simp add: list-all2-conv-all-nth)
    thus τ ∈ ?NE by auto
  qed
show ?case

```

```

proof (cases new = [])
  case False
    note IH = 1(1)[OF rem-ls-def part[symmetric] False]
    have compute-nonempty-main ne ls = compute-nonempty-main (ne ∪ set
(map snd new)) rem using False by simp
    also have ... = ?NE
    proof (rule IH)
      show ne ∪ set (map snd new) ⊆ ?NE using new ne by auto
      show set rem ⊆ set Cs using rem-ls part by auto
      show snd ' (set Cs - set rem) ⊆ ne ∪ set (map snd new)
      proof
        fix τ
        assume τ ∈ snd ' (set Cs - set rem)
        then obtain f args where in-ls: ((f,args),τ) ∈ set Cs and nrem: ((f,args),τ)
∉ set rem by force
        thus τ ∈ ne ∪ set (map snd new) using new part rem-ls by force
      qed
    qed
    finally show ?thesis .
  next
  case True
    have compute-nonempty-main ne ls = ne using True by simp
    also have ... = ?NE
    proof (rule ccontr)
      assume ¬ ?thesis
      with ne empty-sortI obtain τ t where counter: t : τ in ?TC τ ∉ ne by
force
      thus False
      proof (induct t τ)
        case (Fun f ts τs τ)
          from Fun(1) have map-of Cs (f,τs) = Some τ by (simp add: fun-hastype-def)
          then have mem: ((f,τs),τ) ∈ set Cs by (meson map-of-SomeD)
          from Fun(3) have τs: set τs ⊆ ne by (induct, auto)
          from rem-ls mem Fun(4) have ((f,τs),τ) ∈ set rem-ls by auto
          with τs have ((f,τs),τ) ∈ set new using part by auto
          with True show ?case by auto
        qed auto
      qed
    finally show ?thesis .
  qed
qed
from this[of {} Cs] show ?thesis unfolding compute-nonempty-sorts-def by
auto
qed

```

definition decide-nonempty-sorts :: 't list ⇒ (('f × 't list) × 't)list ⇒ 't option
where

```

decide-nonempty-sorts τs Cs = (let ne = compute-nonempty-sorts Cs in
find (λ τ. τ ∉ ne) τs)

```

lemma *decide-nonempty-sorts*:
assumes *distinct (map fst Cs)*
shows *decide-nonempty-sorts τs Cs = None $\implies \forall \tau \in \text{set } \tau s. \neg \text{empty-sort (map-of Cs) } \tau$*
decide-nonempty-sorts τs Cs = Some $\tau \implies \tau \in \text{set } \tau s \wedge \text{empty-sort (map-of Cs) } \tau$
unfolding *decide-nonempty-sorts-def Let-def compute-nonempty-sorts[OF assms]*
find-None-iff find-Some-iff **by** *auto*

7.2 Deciding infiniteness of a sort and computing cardinalities

We provide an algorithm, that given a list of sorts with constructors, computes the set of those sorts that are infinite. Here a sort is defined as infinite iff there is no upper bound on the size of the ground terms of that sort. Moreover, we also compute for each sort the cardinality of the set of constructor ground terms of that sort.

context

includes *finfun-syntax*

begin

fun *finfun-update-all* :: *'a list \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow f 'b) \Rightarrow ('a \Rightarrow f 'b)* **where**
finfun-update-all [] g f = f
| *finfun-update-all (x # xs) g f = (finfun-update-all xs g f)(x $\$$:= g x)*

lemma *finfun-update-all[simp]*: *finfun-update-all xs g f $\$$ x = (if x \in set xs then g x else f $\$$ x)*

proof (*induct xs*)

case (*Cons y xs*)

thus *?case* **by** (*cases x = y, auto*)

qed *auto*

definition *update-card-of-sort* :: *' $\tau \Rightarrow$ ('f \times ' τ list)list \Rightarrow (' $\tau \Rightarrow$ f nat) \Rightarrow nat*
where

update-card-of-sort τ cs cards = ($\sum f \sigma s \leftarrow \text{remdups cs. prod-list (map (($\$$) cards) (snd f \sigma s))$)

function *compute-inf-card-main* :: *' τ set \Rightarrow (' $\tau \Rightarrow$ f nat) \Rightarrow (' $\tau \times$ ('f \times ' τ list)list) list \Rightarrow ' τ set \times (' $\tau \Rightarrow$ nat)* **where**

compute-inf-card-main m-inf cards ls = (

let (fin, ls') =

partition ($\lambda (\tau, fs). \forall \tau s \in \text{set (map snd fs). } \forall \tau \in \text{set } \tau s. \tau \notin \text{m-inf) ls$

in if fin = [] then (m-inf, $\lambda \tau. \text{cards } \$ \tau$) else

let new = map fst fin;

cards' = finfun-update-all new ($\lambda \tau. \text{update-card-of-sort } \tau$ (the (map-of ls τ)))

cards) cards in
 compute-inf-card-main (m-inf - set new) cards' ls'
 by pat-completeness auto

termination

proof (relation measure (length o snd o snd), goal-cases)
 case (2 m-inf cards ls pair fin ls')
 have length fin + length ls' = length ls
 using 2 sum-length-filter-compl[of - ls] by (auto simp: o-def)
 with 2(3) have length ls' < length ls by (cases fin, auto)
 thus ?case by auto
qed simp

lemma compute-inf-card-main: fixes C :: ('f,'t)ssig
 assumes C-Cs: C = map-of Cs'
 and Cs': set Cs' = set (concat (map ((λ (τ, fs). map (λ f. (f,τ)) fs)) Cs))
 and arg-types-nonempty: ∀ f τ s τ'. f : τ s → τ in C → τ' ∈ set τ s → ¬
 empty-sort C τ'
 and dist: distinct (map fst Cs) distinct (map fst Cs')
 and inhabitet: ∀ τ fs. (τ,fs) ∈ set Cs → set fs ≠ {}
 and ∀ τ. τ ∉ m-inf → bdd-above (size ' {t. t : τ in T(C)})
 and set ls ⊆ set Cs
 and fst ' (set Cs - set ls) ∩ m-inf = {}
 and m-inf ⊆ fst ' set ls
 and ∀ τ. τ ∉ m-inf → cards \$ τ = card-of-sort C τ ∧ finite-sort C τ
 and ∀ τ. τ ∈ m-inf → cards \$ τ = 0
shows compute-inf-card-main m-inf cards ls = ({τ. ¬ bdd-above (size ' {t. t : τ in
 T(C)})},
 λ τ. card-of-sort C τ)
 using assms(7-)
proof (induct m-inf cards ls rule: compute-inf-card-main.induct)
 case (1 m-inf cards ls)
 let ?terms = λ τ. {t. t : τ in T(C)}
 let ?fin = λ τ. bdd-above (size ' ?terms τ)
 define crit where crit = (λ (τ :: 't,fs :: ('f × 't list) list). ∀ τ s ∈ set (map snd
 fs). ∀ τ ∈ set τ s. τ ∉ m-inf)
 define S where S τ' = size ' {t. t : τ' in T(C)} for τ'
 define M where M τ' = Maximum (S τ') for τ'
 define M' where M' σ s = sum-list (map M σ s) + (1 + length σ s) for σ s
 define L where L = [σ s . (τ,cs) <- Cs, (f,σ s) <- cs]
 define N where N = max-list (map M' L)
 obtain fin ls' where part: partition crit ls = (fin, ls') by force
 {
 fix τ cs
 assume inCs: (τ,cs) ∈ set Cs
 have nonempty: ∃ t. t : τ in T(C)
proof -
 from inhabitet[rule-format, OF inCs] obtain f σ s where (f,σ s) ∈ set cs by
 (cases cs,auto)

```

with inCs have ((f,σs),τ) ∈ set Cs' unfolding Cs' by auto
hence fC: f : σs → τ in C using dist(2) unfolding C-Cs
  by (meson fun-hastype-def map-of-is-SomeI)
hence ∀σ. ∃ t. σ ∈ set σs → t : σ in T(C)
  by (auto dest!: arg-types-nonempty[rule-format] elim!: not-empty-sortE)
from choice[OF this] obtain t where σ ∈ set σs ⇒ t σ : σ in T(C) for σ
by auto
  hence Fun f (map t σs) : τ in T(C) using list-all2-conv-all-nth
  apply (intro Fun-hastypeI[OF fC]) by (simp add: list-all2-conv-all-nth)
  then show ?thesis by auto
qed
} note inhabited = this

define cards' where cards' = finfun-update-all (map fst fin) (λ τ. update-card-of-sort
τ (the (map-of ls τ)) cards) cards
{
  fix τ
  assume asm: τ ∈ fst ' set fin
  let ?TT = ?terms τ
  from asm obtain cs where tau-cs-fin: (τ,cs) ∈ set fin by auto
  hence tau-ls: (τ,cs) ∈ set ls using part by auto
  with dist(1) ⟨set ls ⊆ set Cs⟩
  have map: map-of Cs τ = Some cs map-of ls τ = Some cs
  by (metis (no-types, opaque-lifting) eq-key-imp-eq-value map-of-SomeD subsetD
weak-map-of-SomeI)+
  from asm have cards': cards' $ τ = update-card-of-sort τ cs cards unfolding
cards'-def by (auto simp: map)
  from part asm have tau-fin: τ ∈ set (map fst fin) by auto
  {
    fix f σs
    have f : σs → τ in C ⟷ ((f,σs),τ) ∈ set Cs'
    proof
      assume f : σs → τ in C
      hence map-of Cs' (f,σs) = Some τ unfolding C-Cs by (rule fun-hastypeD)
      thus ((f,σs),τ) ∈ set Cs' by (rule map-of-SomeD)
    next
      assume ((f, σs), τ) ∈ set Cs'
      hence map-of Cs' (f, σs) = Some τ using dist(2) by simp
      thus f : σs → τ in C unfolding C-Cs by (rule fun-hastypeI)
    qed
  }
  also have ... ⟷ (∃ cs. (τ, cs) ∈ set Cs ∧ (f,σs) ∈ set cs)
  unfolding Cs' by auto
  also have ... ⟷ (∃ cs. map-of Cs τ = Some cs ∧ (f,σs) ∈ set cs)
  using dist(1) by simp
  also have ... ⟷ (f,σs) ∈ set cs unfolding map by auto
  finally have (f : σs → τ in C) = ((f, σs) ∈ set cs) by auto
} note C-to-cs = this

define T where T σ = ?terms σ for σ

```

```

have to-ls: {ts. ts :l σs in T(C)} = listset (map T σs) for σs
by (intro set-eqI, unfold listset-conv-nth, auto simp: T-def list-all2-conv-all-nth)
{
  fix f σs σ
  assume in-cs: (f, σs) ∈ set cs σ ∈ set σs
  from tau-cs-fin part have crit (τ,cs) by auto
  from this[unfolded crit-def split] in-cs have σ ∉ m-inf by auto
  with 1(6) have cards $ σ = card (T σ) and finite (T σ)
    by (auto simp: T-def card-of-sort finite-sort)
} note σs-infos = this

have ?TT = { Fun f ts | f ts σs. f : σs → τ in C ∧ ts :l σs in T(C)} (is - =
?FunApps)
proof (intro set-eqI)
  fix t
  {
    assume t : τ in T(C)
    hence t ∈ ?FunApps by (induct, auto)
  }
  moreover
  {
    assume t ∈ ?FunApps
    hence t : τ in T(C) by (auto intro: Fun-hastypeI)
  }
  ultimately show t ∈ ?TT ⟷ t ∈ ?FunApps by auto
qed
also have ... = { Fun f ts | f ts σs. (f, σs) ∈ set cs ∧ ts :l σs in T(C)}
unfolding C-to-cs ..
also have ... = (λ (f, ts). Fun f ts) ‘ (⋃ (f, σs) ∈ set cs. Pair f ‘ { ts. ts :l σs
in T(C)}) (is - = ?f ‘ ?A) by auto
finally have TTfA: ?TT = ?f ‘ ?A .
have finPair: finite (Pair f ‘ A) = finite A for f :: 'f and A :: ('f, 'v) Term.term
list set
  by (intro finite-image-iff inj-onI, auto)
have inj: inj ?f by (intro injI, auto)
from inj have card: card ?TT = card ?A
  unfolding TTfA by (meson UNIV-I card-image inj-on-def)
also have ... = (∑ i ∈ set cs. card (case i of (f, σs) ⇒ Pair f ‘ listset (map T
σs))) unfolding to-ls
proof (rule card-UN-disjoint[OF finite-set ballI ballI[OF ballI[OF impI]]],
goal-cases)
  case *: (1 fσs)
  obtain f σs where fσs: fσs = (f,σs) by force
  thus ?case using * σs-infos(2) by (cases fσs, auto intro!: finite-imageI
finite-listset)
next
  case *: (2 fσs gτs)
  obtain f σs where fσs: fσs = (f,σs) by force
  obtain g τs where gτs: gτs = (g,τs) by force

```

```

show ?case
proof (cases g = f)
  case False
  thus ?thesis unfolding fσs gτs split by auto
next
case True
note fτs = gτs[unfolded True]
show ?thesis
proof (rule ccontr)
  assume ¬ ?thesis
  from this[unfolded fσs fτs split]
  obtain ts where ts: ts ∈ listset (map T σs) ts ∈ listset (map T τs) by
auto
  hence len: length σs = length ts length τs = length ts unfolding list-
set-conv-nth by auto
  from *(3)[unfolded fσs fτs] have σs ≠ τs by auto
  with len obtain i where i: i < length ts and diff: σs ! i ≠ τs ! i
  by (metis nth-equalityI)
  define ti where ti = ts ! i
  define σi where σi = σs ! i
  define τi where τi = τs ! i
  note diff = diff[folded σi-def τi-def]
  from ts i have ti ∈ T σi ti ∈ T τi
  unfolding ti-def σi-def τi-def listset-conv-nth by auto
  hence ti: ti : σi in T(C) ti : τi in T(C) unfolding T-def by auto
  hence σi = τi by fastforce
  with diff show False ..
qed
qed
qed
also have ... = (∑ fσs ∈ set cs. card (listset (map T (snd fσs))))
proof (rule sum.cong[OF refl], goal-cases)
  case (1 fσs)
  obtain f σs where id: fσs = (f,σs) by force
  show ?case unfolding id split snd-conv
  by (rule card-image, auto simp: inj-on-def)
qed
also have ... = (∑ fσs ∈ set cs. prod-list (map card (map T (snd fσs))))
  by (rule sum.cong[OF refl], rule card-listset, insert σs-infos, auto)
also have ... = (∑ fσs ∈ set cs. prod-list (map ((\$) cards) (snd fσs)))
  unfolding map-map o-def using σs-infos
  by (intro sum.cong[OF refl] arg-cong[of - - prod-list], auto)
also have ... = sum-list (map (λ fσs. prod-list (map ((\$) cards) (snd fσs)))
(remdups cs))
  by (rule sum.set-conv-list)
also have ... = cards' \$ τ unfolding cards' update-card-of-sort-def ..
finally have cards': card ?TT = cards' \$ τ by auto

```

```

from inj have finite ?TT = finite ?A
  by (metis (no-types, lifting) TTfA finite-imageD finite-imageI subset-UNIV
inj-on-subset)
  also have ... = ( $\forall f \sigma s. (f, \sigma s) \in \text{set } cs \longrightarrow \text{finite } (\text{Pair } f \text{ ' } \{ts. ts :_l \sigma s \text{ in } \mathcal{T}(C)\})$ )
  by auto
  finally have finite ?TT = ( $\forall f \sigma s. (f, \sigma s) \in \text{set } cs \longrightarrow \text{finite } \{ts. ts :_l \sigma s \text{ in } \mathcal{T}(C)\}$ )
    unfolding finPair by auto
    also have ... = True unfolding to-ls using  $\sigma s\text{-infos}(2)$  by (auto intro!:
finite-listset)
    finally have fin: finite ?TT by simp

from fin cards'
have cards' $  $\tau = \text{card } (?terms \tau)$  finite (?terms  $\tau$ ) ?fin  $\tau$  by auto
} note fin = this

show ?case
proof (cases fin = [])
  case False
    hence compute-inf-card-main m-inf cards ls = compute-inf-card-main (m-inf
- set (map fst fin)) cards' ls'
    unfolding compute-inf-card-main.simps[of m-inf] part[unfolded crit-def]
cards'-def Let-def by auto
    also have ... = ( $\{\tau. \neg ?fin \tau\}, \lambda \tau. \text{card-of-sort } C \tau$ )
    proof (rule 1(1)[OF refl part[unfolded crit-def, symmetric] False])
      show set ls'  $\subseteq$  set Cs using 1(3) part by auto
      show fst ' (set Cs - set ls')  $\cap$  (m-inf - set (map fst fin)) = {} using 1(3-4)
part by force
      show m-inf - set (map fst fin)  $\subseteq$  fst ' set ls' using 1(5) part by force
      show  $\forall \tau. \tau \notin m\text{-inf} - \text{set } (\text{map } \text{fst } \text{fin}) \longrightarrow \text{cards}' \$ \tau = \text{card-of-sort } C \tau \wedge$ 
finite-sort C  $\tau$ 
    proof (intro allI impI)
      fix  $\tau$ 
      assume nmem:  $\tau \notin m\text{-inf} - \text{set } (\text{map } \text{fst } \text{fin})$ 
      show cards' $  $\tau = \text{card-of-sort } C \tau \wedge \text{finite-sort } C \tau$ 
      proof (cases  $\tau \in \text{set } (\text{map } \text{fst } \text{fin})$ )
        case False
          with nmem have tau:  $\tau \notin m\text{-inf}$  by auto
          with False 1(6)[rule-format, OF this] show ?thesis
          unfolding cards'-def by auto
        next
          case True
            with fin show ?thesis by (auto simp: card-of-sort finite-sort)
      qed
    qed
  thus  $\forall \tau. \tau \notin m\text{-inf} - \text{set } (\text{map } \text{fst } \text{fin}) \longrightarrow ?fin \tau$ 
  by (force simp: 1(2) intro: fin(3))
  show  $\forall \tau. \tau \in m\text{-inf} - \text{set } (\text{map } \text{fst } \text{fin}) \longrightarrow \text{cards}' \$ \tau = 0$  using 1(7)

```

```

unfolding cards'-def
  by auto
  qed (auto simp: cards'-def)
  finally show ?thesis .
next
  case True
  let ?cards = λτ. cards $ τ
  have m-inf: m-inf = {τ. ¬ ?fin τ}
  proof
    show  $\{\tau. \neg ?fin \tau\} \subseteq m-inf$  using fin 1(2) by auto
    {
      fix  $\tau$ 
      assume  $\tau \in m-inf$ 
      with 1(5) obtain cs where mem: (τ,cs) ∈ set ls by auto
      from part True have ls': ls' = ls by (induct ls arbitrary: ls', auto)
      from partition-P[OF part, unfolded ls']
      have  $\bigwedge e. e \in set\ ls \implies \neg crit\ e$  by auto
      from this[OF mem, unfolded crit-def split]
      obtain  $c\ \tau s\ \tau'$  where  $*(c, \tau s) \in set\ cs\ \tau' \in set\ \tau s\ \tau' \in m-inf$  by auto
      from mem 1(2-) have  $(\tau, cs) \in set\ Cs$  by auto
      with  $*$  have  $((c, \tau s), \tau) \in set\ Cs'$  unfolding Cs' by force
      with dist(2) have map-of Cs' ((c, τ s)) = Some τ by simp
      from this[folded C-Cs] have  $c : c : \tau s \rightarrow \tau$  in C unfolding fun-hastype-def
      .
      have  $\forall \sigma. \exists t. \sigma \in set\ \tau s \implies t : \sigma$  in  $\mathcal{T}(C)$ 
      by (auto dest!: arg-types-nonempty[rule-format, OF c] elim!: not-empty-sortE)
      from choice[OF this] obtain t where  $\bigwedge \sigma. \sigma \in set\ \tau s \implies t\ \sigma : \sigma$  in  $\mathcal{T}(C)$ 
by auto
      hence list: map t τ s :l τ s in  $\mathcal{T}(C)$  by (simp add: list-all2-conv-all-nth)
      with c have Fun c (map t τ s) : τ in  $\mathcal{T}(C)$  by (intro Fun-hastypeI)
      with  $*$  c list have  $\exists c\ \tau s\ \tau' ts. Fun\ c\ ts : \tau$  in  $\mathcal{T}(C) \wedge ts :l \tau s$  in  $\mathcal{T}(C) \wedge$ 
       $c : \tau s \rightarrow \tau$  in C  $\wedge \tau' \in set\ \tau s \wedge \tau' \in m-inf$ 
      by blast
    }
    note m-invD = this
    {
      fix  $n :: nat$ 
      have  $\tau \in m-inf \implies \exists t. t : \tau$  in  $\mathcal{T}(C) \wedge size\ t \geq n$  for  $\tau$ 
      proof (induct n arbitrary: τ)
        case  $(0\ \tau)$ 
        from m-invD[OF 0] show ?case by blast
      next
        case  $(Suc\ n\ \tau)$ 
        from m-invD[OF Suc(2)] obtain  $c\ \tau s\ \tau' ts$ 
          where  $*: ts :l \tau s$  in  $\mathcal{T}(C)$   $c : \tau s \rightarrow \tau$  in C  $\tau' \in set\ \tau s\ \tau' \in m-inf$ 
          by auto
        from  $*(1)[unfolded\ list-all2-conv-all-nth]\ *(3)[unfolded\ set-conv-nth]$ 
        obtain i where  $i < length\ \tau s$  and  $ts[i] : \tau'$  in  $\mathcal{T}(C)$  and len: length
         $ts = length\ \tau s$  by auto
        from Suc(1)[OF *(4)] obtain t where  $t : \tau'$  in  $\mathcal{T}(C)$  and  $ns : n \leq size$ 

```

```

t by auto
  define ts' where ts' = ts[i := t]
  have ts' :1 τs in  $\mathcal{T}(C)$  using list-all2-conv-all-nth unfolding ts'-def
  by (metis *(1) tsi has-same-type i list-all2-update-cong list-update-same-conv
t(1))
  hence **:Fun c ts' : τ in  $\mathcal{T}(C)$  apply (intro Fun-hastypeI[OF *(2)]) by
fastforce
  have t ∈ set ts' unfolding ts'-def using t
  by (simp add: i len set-update-memI)
  hence size (Fun c ts') ≥ Suc n using *
  by (simp add: size-list-estimation' ns)
  thus ?case using ** by blast
qed
} note main = this
show m-inf ⊆ {τ. ¬ ?fin τ}
proof (standard, standard)
  fix τ
  assume asm: τ ∈ m-inf
  have ∃ t. t : τ in  $\mathcal{T}(C)$  ∧ n < size t for n using main[OF asm, of Suc n]
by auto
  thus ¬ ?fin τ
  by (metis bdd-above-Maximum-nat imageI mem-Collect-eq order.strict-iff)
qed
qed
from True have compute-inf-card-main m-inf cards ls = (m-inf, ?cards)
  unfolding compute-inf-card-main.simps[of m-inf] part[unfolded crit-def] by
auto
also have ?cards = (λ τ. card-of-sort C τ)
proof (intro ext)
  fix τ
  show cards $ τ = card-of-sort C τ
proof (cases τ ∈ m-inf)
  case False
  thus ?thesis using 1(6) by auto
next
  case True
  define TT where TT = ?terms τ
  from True m-inf have ¬ bdd-above (size ' TT) unfolding TT-def by auto
  hence infinite TT by auto
  hence card TT = 0 by auto
  thus ?thesis unfolding TT-def using True 1(7) by (auto simp: card-of-sort)
qed
qed
finally show ?thesis using m-inf by auto
qed
qed

```

definition *compute-inf-card-sorts* :: ((f × t list) × t)list ⇒ t set × (t ⇒ nat)
where

```

compute-inf-card-sorts Cs = (let
  Cs' = map (λ τ. (τ, map fst (filter(λf. snd f = τ) Cs))) (remdups (map snd
Cs))
  in compute-inf-card-main (set (map fst Cs')) (K$ 0) Cs')

```

lemma *finite-imp-size-bdd-above*: **assumes** *finite T*

shows *bdd-above (size ' T)*

proof –

from *assms* **have** *finite (size ' T)* **by** *auto*

thus *?thesis* **by** *simp*

qed

lemma *finite-sig-imp-finite-terms-of-bounded-size*: **assumes** *finite (dom F)* **and** *finite (dom V)*

shows *finite {t. ∃ τ. size t ≤ n ∧ t : τ in T(F,V)}* (**is** *finite (?terms n)*)

proof (*induct n*)

case (*0*)

have $t \notin ?terms\ 0$ **for** t **by** (*cases t, auto*)

hence *id: ?terms 0 = {}* **by** *auto*

show *?case unfolding id by simp*

next

case (*Suc n*)

let *?funsInter = (λ (f, τs). (f, listset (map (λ -. (?terms n)) τs)))* ' *dom F*

define *funsI* **where** *funsI = ?funsInter*

let *?funs = ∪ ((λ (f, tss). Fun f ' tss) ' funsI)*

{

fix t

assume $t \in ?terms\ (Suc\ n)$

then obtain τ **where** $t\tau: t : \tau$ **in** $\mathcal{T}(F,V)$ **and** *size: size t ≤ Suc n* **by** *auto*

have $t \in Var\ ' dom\ V \cup ?funs$

proof (*cases t*)

case (*Var x*)

thus *?thesis* **using** $t\tau$ **by** *auto*

next

case $t: (Fun\ f\ ts)$

from $t\tau[unfolded\ t\ Fun\ hastype]$ **obtain** τs **where** $ts: ts :_l \tau s$ **in** $\mathcal{T}(F,V)$

and $f: (f, \tau s) \in dom\ F$ **by** *auto*

hence $(f, listset (map (\lambda -. (?terms\ n)) \tau s)) \in funsI$ **unfolding** *funsI-def* **by**

auto

moreover **have** $ts \in listset (map (\lambda -. (?terms\ n)) \tau s)$

unfolding *listset-conv-nth length-map*

proof (*intro conjI allI impI*)

show *len: length ts = length τs* **using** ts **by** (*metis list-all2-lengthD*)

fix i

assume $i: i < length\ \tau s$

with ts **have** $i': i < length\ ts$ **and** *type: ts ! i : τs ! i* **in** $\mathcal{T}(F,V)$

using *list-all2-nthD2[OF ts] len* **by** *auto*

from i' **have** $ts\ !\ i \in set\ ts$ **by** *auto*

from *split-list*[*OF this*] **obtain** *bef aft* **where** $ts = bef @ ts ! i \# aft$ **by**
auto
from *size*[*unfolded t*] **this have** $size (Fun f (bef @ ts ! i \# aft)) \leq Suc n$
by *simp*
hence $size (ts ! i) \leq n$ **by** *simp*
with *type* **have** $ts ! i \in ?terms n$ **by** *auto*
with *i* **show** $ts ! i \in map (\lambda-. ?terms n) \tau s ! i$ **by** *auto*
qed
ultimately show *?thesis unfolding t by blast*
qed
}
hence $?terms (Suc n) \subseteq Var ' dom V \cup ?funs$ **by** *blast*
moreover have *finite* ($Var ' dom V \cup ?funs$)
proof (*intro finite-UnI finite-imageI assms finite-Union*)
show *finite* (*funsI*) **unfolding** *funsI-def*
by (*intro finite-imageI assms*)
fix *M*
assume $M \in \{Fun f ' tss \mid (f, tss) \in funsI\}$
from *this* **obtain** *f tss* **where** $tss: (f, tss) \in funsI$ **and** $M: M = Fun f ' tss$
by *auto*
from *tss*[*unfolded funsI-def*] **obtain** τs **where**
 $tss: tss = listset (map (\lambda-. \{t. size t \leq n \wedge (\exists \tau. t : \tau \text{ in } \mathcal{T}(F, V))\}) \tau s)$ **and**
 $\tau s \in snd ' dom F$
by *force*
have *finite tss unfolding tss*
by (*intro finite-listset, insert Suc, auto*)
thus *finite M unfolding M*
by (*intro finite-imageI*)
qed
ultimately show *?case by (rule finite-subset)*
qed

lemma *finite-sig-bdd-above-imp-finite*: **assumes** *finite (dom F)* **and** *finite (dom V)*

and *bdd-above* ($size ' \{t. t : \tau \text{ in } \mathcal{T}(F, V)\}$)

shows *finite* $\{t. t : \tau \text{ in } \mathcal{T}(F, V)\}$

proof –

from *assms*(3)[*unfolded bdd-above-def*] **obtain** *n* **where**

$size: \forall s \in size ' \{t. t : \tau \text{ in } \mathcal{T}(F, V)\}. s \leq n$ **by** *auto*

from *finite-sig-imp-finite-terms-of-bounded-size*[*OF assms(1-2)*]

have *fin*: *finite* $\{t. \exists \tau. size t \leq n \wedge t : \tau \text{ in } \mathcal{T}(F, V)\}$ **by** *auto*

have *finite* $\{t. size t \leq n \wedge t : \tau \text{ in } \mathcal{T}(F, V)\}$

by (*rule finite-subset*[*OF - fin*], *auto*)

also have $\{t. size t \leq n \wedge t : \tau \text{ in } \mathcal{T}(F, V)\} = \{t. t : \tau \text{ in } \mathcal{T}(F, V)\}$

using *size by blast*

finally show *?thesis by auto*

qed

lemma *finite-sig-bdd-above-iff-finite*: **assumes** *finite (dom F)* **and** *finite (dom V)*

shows $\text{bdd-above } (\text{size } \{t. t : \tau \text{ in } \mathcal{T}(F, V)\}) = \text{finite } \{t. t : \tau \text{ in } \mathcal{T}(F, V)\}$
using $\text{finite-sig-bdd-above-imp-finite}[OF \text{ assms}] \text{ finite-imp-size-bdd-above}$
by metis

lemma $\text{compute-inf-card-sorts}$:

fixes $C :: (f, t)\text{ssig}$

assumes $C\text{-Cs}: C = \text{map-of } Cs$

and $\text{arg-types-nonempty}: \forall f \tau s \tau \tau'. f : \tau s \rightarrow \tau \text{ in } C \longrightarrow \tau' \in \text{set } \tau s \longrightarrow \neg \text{empty-sort } C \tau'$

and $\text{dist}: \text{distinct } (\text{map fst } Cs)$

and $\text{result}: \text{compute-inf-card-sorts } Cs = (\text{unb}, \text{cards})$

shows $\text{unb} = \{\tau. \neg \text{bdd-above } (\text{size } \{t. t : \tau \text{ in } \mathcal{T}(C)\})\}$ (**is** $- = ?\text{unb}$)

$\text{cards} = \text{card-of-sort } C$ (**is** $- = ?\text{cards}$)

$\text{unb} = \{\tau. \neg \text{finite-sort } C \tau\}$ (**is** $- = ?\text{inf}$)

proof $-$

let $?terms = \lambda \tau. \{t. t : \tau \text{ in } \mathcal{T}(C)\}$

define taus **where** $\text{taus} = \text{remdups } (\text{map snd } Cs)$

define Cs' **where** $Cs' = \text{map } (\lambda \tau. (\tau, \text{map fst } (\text{filter}(\lambda f. \text{snd } f = \tau) Cs))) \text{taus}$

have $\text{compute-inf-card-sorts } Cs = \text{compute-inf-card-main } (\text{set } (\text{map fst } Cs')) (K\$ 0) Cs'$

unfolding $\text{compute-inf-card-sorts-def } \text{taus-def } Cs'\text{-def } \text{Let-def}$ **by** auto

also have $\dots = (?unb, ?cards)$

proof ($\text{rule } \text{compute-inf-card-main}[OF C\text{-Cs} - \text{arg-types-nonempty} - \text{dist} - - \text{subset-refl}]$)

have $\text{distinct } \text{taus}$ **unfolding** taus-def **by** auto

thus $\text{distinct } (\text{map fst } Cs')$ **unfolding** $Cs'\text{-def } \text{map-map } o\text{-def } \text{fst-conv}$ **by** auto

show $\text{set } Cs = \text{set } (\text{concat } (\text{map } (\lambda(\tau, fs). \text{map } (\lambda f. (f, \tau)) fs) Cs'))$

unfolding $Cs'\text{-def } \text{taus-def}$ **by** force

show $\forall \tau fs. (\tau, fs) \in \text{set } Cs' \longrightarrow \text{set } fs \neq \{\}$

unfolding $Cs'\text{-def } \text{taus-def}$ **by** ($\text{force simp: filter-empty-conv}$)

show $\text{fst } \{(\text{set } Cs' - \text{set } Cs') \cap \text{set } (\text{map fst } Cs') = \{\}\}$ **by** auto

show $\text{set } (\text{map fst } Cs') \subseteq \text{fst } \{ \text{set } Cs' \}$ **by** auto

{ fix τ

assume $\tau \notin \text{set } (\text{map fst } Cs')$

hence $\tau \notin \text{snd } \{ \text{set } Cs \}$ **unfolding** $Cs'\text{-def } \text{taus-def}$ **by** auto

hence $\text{diff}: C f \neq \text{Some } \tau$ **for** f **unfolding** $C\text{-Cs}$

by ($\text{metis } \text{Some-eq-map-of-iff } \text{dist } \text{imageI } \text{snd-conv}$)

have $\text{emp}: \text{empty-sort } C \tau$

proof ($\text{intro } \text{empty-sortI } \text{notI}$)

fix t

assume $t : \tau \text{ in } \mathcal{T}(C)$

thus False **using** diff

proof induct

case ($\text{Fun } f \text{ ss } \sigma s \tau$)

from $\text{Fun}(1,4)$ **show** False **unfolding** fun-hastype-def **by** auto

qed auto

qed

```

}
note * = this
show  $\forall \tau. \tau \notin \text{set } (\text{map fst } Cs') \longrightarrow \text{bdd-above } (\text{size } \text{' } ?\text{terms } \tau)$ 
 $\forall \tau. \tau \notin \text{set } (\text{map fst } Cs') \longrightarrow (K\$ 0) \$ \tau = \text{card-of-sort } C \tau \wedge \text{finite-sort } C$ 
 $\tau$ 
by (auto simp del: set-map dest!: *)
qed auto
finally show unb: unb = ?unb and cards: cards = ?cards unfolding result by
auto
show unb = ?inf unfolding unb
proof (subst finite-sig-bdd-above-iff-finite)
show finite (dom C) unfolding C-Cs by (rule finite-dom-map-of)
show finite (dom  $\emptyset$ ) by auto
qed (auto simp: finite-sort)
qed

```

```

lemma min-sumlist-cong: assumes  $\bigwedge x. x \in \text{set } xs \implies \text{min } b (f x :: \text{nat}) = \text{min } b (g x)$ 
shows  $\text{min } b (\text{sum-list } (\text{map } f \text{ } xs)) = \text{min } b (\text{sum-list } (\text{map } g \text{ } xs))$ 
using assms
proof (induct xs)
case (Cons x xs)
hence IH:  $\text{min } b (\text{sum-list } (\text{map } f \text{ } xs)) = \text{min } b (\text{sum-list } (\text{map } g \text{ } xs))$ 
and  $\text{min } b (f x) = \text{min } b (g x)$  by auto
thus ?case by simp
qed simp

```

```

lemma min-prod-min-left:  $\text{min } b (\text{min } b x * y :: \text{nat}) = \text{min } b (x * y)$ 
by (metis One-nat-def Suc-leI bot-nat-0.not-eq-extremum min.cobounded1 min-def
mult.right-neutral
mult-is-0 mult-le-mono)

```

```

lemma min-prod-min-right:  $\text{min } b (x * \text{min } b y :: \text{nat}) = \text{min } b (x * y)$ 
using min-prod-min-left[of b y x] by (simp add: ac-simps)

```

```

lemma min-prod-min:  $\text{min } b (\text{min } b x * \text{min } b y :: \text{nat}) = \text{min } b (x * y)$ 
unfolding min-prod-min-left min-prod-min-right ..

```

```

lemma min-prod-cong: assumes  $\text{min } b x1 = (\text{min } b x2 :: \text{nat}) \text{min } b y1 = \text{min } b y2$ 
shows  $\text{min } b (x1 * y1) = \text{min } b (x2 * y2)$ 
using min-prod-min[of b x1 y1] min-prod-min[of b x2 y2] assms by metis

```

```

lemma min-prodlist-cong: assumes  $\bigwedge x. x \in \text{set } xs \implies \text{min } b (f x :: \text{nat}) = \text{min } b (g x)$ 
shows  $\text{min } b (\text{prod-list } (\text{map } f \text{ } xs)) = \text{min } b (\text{prod-list } (\text{map } g \text{ } xs))$ 
using assms
proof (induct xs)
case (Cons x xs)

```

hence $IH: \min b (\text{prod-list } (\text{map } f \text{ } xs)) = \min b (\text{prod-list } (\text{map } g \text{ } xs))$
and $\min b (f \text{ } x) = \min b (g \text{ } x)$ **by** *auto*
thus *?case* **by** (*auto intro: min-prod-cong*)
qed *simp*

context

fixes $k :: \text{nat}$

begin

abbreviation (*input*) minBnd **where** $\text{minBnd} \equiv \text{min } k$

abbreviation minBndFFun **where** $\text{minBndFFun} \equiv ((o\$) \text{minBnd})$

definition $\text{update-card-of-sort-bnd} :: 'a \Rightarrow ('f \times 'a \text{ list}) \text{ list} \Rightarrow ('a \Rightarrow f \text{ nat}) \Rightarrow \text{nat}$
where

$\text{update-card-of-sort-bnd } \tau \text{ } cs \text{ } cards = \text{minBnd } (\sum f \sigma s \leftarrow \text{remdups } cs. \text{minBnd } (\text{prod-list } (\text{map } ((\$) \text{ cards}) (\text{snd } f \sigma s))))$

partial-function (*tailrec*) $\text{compute-inf-card-main-bnd}$ **where**

[*code*]: $\text{compute-inf-card-main-bnd } m\text{-inf } cards \text{ } ls =$

$\text{let } (fin, ls') =$

$\text{partition } (\lambda (\tau, fs). \forall \tau s \in \text{set } (\text{map } \text{snd } fs). \forall \tau \in \text{set } \tau s. \tau \notin m\text{-inf}) \text{ } ls$

$\text{in if } fin = [] \text{ then } (m\text{-inf}, \lambda \tau. \text{ cards } \$ \tau) \text{ else}$

$\text{let } new = \text{map } \text{fst } fin;$

$\text{cards}' = \text{finfun-update-all } new \text{ } (\lambda \tau. \text{ update-card-of-sort-bnd } \tau \text{ } (\text{the } (\text{map-of } ls \text{ } \tau)) \text{ } cards) \text{ } cards \text{ in}$

$\text{compute-inf-card-main-bnd } (m\text{-inf} - \text{set } new) \text{ } cards' \text{ } ls'$

definition $\text{compute-inf-card-sorts-bnd} :: (('f \times 'a \text{ list}) \times 'a) \text{ list} \Rightarrow 'a \text{ set} \times ('a \Rightarrow \text{nat})$ **where**

$\text{compute-inf-card-sorts-bnd } Cs =$ (*let*

$Cs' = \text{map } (\lambda \tau. (\tau, \text{map } \text{fst } (\text{filter}(\lambda f. \text{snd } f = \tau) \text{ } Cs))) (\text{remdups } (\text{map } \text{snd } Cs))$

$\text{in } \text{compute-inf-card-main-bnd } (\text{set } (\text{map } \text{fst } Cs')) (K\$ 0) \text{ } Cs'$)

lemma $\text{compute-inf-card-sorts-bnd-equiv: compute-inf-card-sorts-bnd } Cs = \text{map-prod id } ((o) \text{minBnd}) (\text{compute-inf-card-sorts } Cs)$

proof –

have $(K\$ 0) = (\text{minBndFFun } (K\$ 0) :: 'a \Rightarrow f \text{ nat})$ **by** *simp*

define Cs' **where** $Cs' = \text{map } (\lambda \tau. (\tau, \text{map } \text{fst } (\text{filter}(\lambda f. \text{snd } f = \tau) \text{ } Cs))) (\text{remdups } (\text{map } \text{snd } Cs))$

define ls **where** $ls = \text{set } (\text{map } \text{fst } Cs')$

define $cards :: 'a \Rightarrow f \text{ nat}$ **where** $cards = (K\$ 0)$

have *?thesis* $\longleftrightarrow \text{compute-inf-card-main-bnd } ls (\text{minBndFFun } cards) \text{ } Cs'$

$= \text{map-prod id } ((o) \text{minBnd}) (\text{compute-inf-card-main } ls \text{ } cards)$

unfolding $\text{compute-inf-card-sorts-bnd-def } ls\text{-def } Cs'\text{-def}$ [*symmetric*]

$\text{compute-inf-card-sorts-def } cards\text{-def}$ **by** *simp*

also have ...

proof (*induct* $ls \text{ } cards \text{ } Cs'$ *rule: compute-inf-card-main.induct*)

case (*1* $m\text{-inf } cards \text{ } ls$ *)*

```

obtain fin ls' where part: partition ( $\lambda(\tau, fs). \forall \tau s \in \text{set } (\text{map } \text{snd } fs). \forall \tau \in \text{set } \tau s. \tau \notin m\text{-inf})$  ls = (fin, ls')
  by force
  note IH = 1[OF refl part[symmetric]]
  note simps1 = compute-inf-card-main.simps[of m-inf cards ls, unfolded part
Let-def split]
  note simps2 = compute-inf-card-main-bnd.simps[of m-inf minBndFFun cards
ls, unfolded part Let-def split]
  show ?case
  proof (cases fin = [])
    case True
      show ?thesis unfolding simps1 simps2 True by auto
    next
      case False
        define minf2 where minf2 = m-inf - set (map fst fin)
        define new where new = map fst fin
        define cards2 where cards2 = (finfun-update-all new ( $\lambda\tau. \text{update-card-of-sort } \tau$ 
the (map-of ls  $\tau$ ) cards) cards)
        note IH = IH[OF False refl, folded minf2-def, folded new-def, OF cards2-def]
        from False have False: (fin = []) = False by auto
        note simps1 = simps1[folded minf2-def, unfolded False if-False, folded new-def,
folded cards2-def]
        note simps2 = simps2[folded minf2-def, unfolded False if-False, folded new-def]
        show ?thesis unfolding simps1 simps2 IH[symmetric]
        proof (rule arg-cong[of - -  $\lambda c. \text{compute-inf-card-main-bnd minf2 c ls'}$ , unfold
cards2-def])
          show finfun-update-all new ( $\lambda\tau. \text{update-card-of-sort-bnd } \tau$ 
the (map-of ls
 $\tau$ ) (minBndFFun cards))
            (minBndFFun cards) =
              minBndFFun (finfun-update-all new ( $\lambda\tau. \text{update-card-of-sort } \tau$ 
the (map-of
ls  $\tau$ ) cards) cards)
            apply (rule finfun-ext)
            apply (unfold finfun-comp-apply)
            apply (unfold finfun-update-all o-def)
            apply (unfold if-distrib[of minBnd])
            apply (unfold finfun-comp-apply o-def)
            apply (rule if-cong[OF refl - refl])
            apply (unfold update-card-of-sort-bnd-def update-card-of-sort-def)
            apply (unfold finfun-comp-apply o-def)
            apply (rule min-sumlist-cong)
            apply (unfold min.left-idem map-map)
            by (rule min-prodlist-cong, auto)
          qed
        qed
      qed
    finally show ?thesis by blast
  qed
end
end

```

lemma *compute-inf-card-sorts-bnd*:
fixes $C :: ('f, 't)ssig$
assumes $C\text{-Cs}: C = \text{map-of } Cs$
and *arg-types-nonempty*: $\forall f \tau s \tau \tau'. f : \tau s \rightarrow \tau \text{ in } C \longrightarrow \tau' \in \text{set } \tau s \longrightarrow \neg$
empty-sort $C \tau'$
and *dist*: *distinct* (*map fst* Cs)
and *result*: *compute-inf-card-sorts-bnd* $k Cs = (\text{unb}, \text{cards})$
shows $\text{unb} = \{\tau. \neg \text{bdd-above } (\text{size } \{t. t : \tau \text{ in } \mathcal{T}(C)\})\}$ (**is** $?A$)
 $\text{cards} = \text{min } k \text{ o } \text{card-of-sort } C$ (**is** $?B$)
 $\text{unb} = \{\tau. \neg \text{finite-sort } C \tau\}$ (**is** $?C$)
proof –
obtain $\text{un } \text{cds}$ **where** *res*: *compute-inf-card-sorts* $Cs = (\text{un}, \text{cds})$ **by** *force*
note $\text{main} = \text{compute-inf-card-sorts}[OF \text{ assms}(1-3) \text{ this}]$
from *compute-inf-card-sorts-bnd-equiv*[*of* $k Cs$, *unfolded res result*]
have $\text{unb} = \text{un } \text{cards} = \text{min } k \text{ o } \text{cds}$ **by** *auto*
thus $?A ?B ?C$ **using** *main* **by** *auto*
qed

abbreviation *compute-inf-sorts* :: $((f \times 't \text{ list}) \times 't) \text{ list} \Rightarrow 't \text{ set}$ **where**
compute-inf-sorts $Cs \equiv \text{fst } (\text{compute-inf-card-sorts-bnd } 0 Cs)$

lemma *compute-inf-sorts*:
assumes *arg-types-nonempty*: $\forall f \tau s \tau \tau'. f : \tau s \rightarrow \tau \text{ in } \text{map-of } Cs \longrightarrow \tau' \in \text{set}$
 $\tau s \longrightarrow \neg \text{empty-sort } (\text{map-of } Cs) \tau'$
and *dist*: *distinct* (*map fst* Cs)
shows
 $\text{compute-inf-sorts } Cs = \{\tau. \neg \text{bdd-above } (\text{size } \{t. t : \tau \text{ in } \mathcal{T}(\text{map-of } Cs)\})\}$
 $\text{compute-inf-sorts } Cs = \{\tau. \neg \text{finite-sort } (\text{map-of } Cs) \tau\}$
using *compute-inf-card-sorts-bnd*[*OF refl assms*, **where** $k = 0$]
by (*cases compute-inf-card-sorts-bnd* $0 Cs$, *auto*)**+**

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.