

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	5
3.1	Maps between Sorted Sets	10
3.1.1	Sorted bijection	12
3.2	Sorted Images	13

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

4	Sorted Terms	15
4.1	Overloaded Notations	16
4.2	Sorted Signatures and Sorted Sets of Terms	16
4.3	Sorted Algebras	19
4.3.1	Term Algebras	20
4.3.2	Homomorphisms	20
4.4	Lifting Sorts	22
4.5	Collecting Variables via Evaluation	25
4.6	Ground Terms	26
4.6.1	Cardinality of Sorts	27
4.6.2	Enumerating Ground Terms	28
4.7	Subsignatures	29
5	Sorted Contexts	31
6	Basic Terms	33
7	Computing Nonempty and Infinite sorts	36
7.1	Deciding the nonemptiness of all sorts under consideration . .	36
7.2	Deciding infiniteness of a sort and computing cardinalities . .	37

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:
   $(\exists x \in \text{set } xs. P x) = (\exists i. i < \text{length } xs \wedge P (xs ! i))$ 
  <proof>

```

```

lemma Ball-Pair-conv:  $(\forall (x,y) \in R. P x y) \longleftrightarrow (\forall x y. (x,y) \in R \longrightarrow P x y)$  <proof>

```

lemma *Some-eq-bind-conv*: $(\text{Some } x = f \ggg g) = (\exists y. f = \text{Some } y \wedge g y = \text{Some } x)$

<proof>

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

<proof>

lemma *list-all2-same-left*:

$\forall a' \in \text{set } as. a' = a \implies \text{list-all2 } r \text{ as } bs \longleftrightarrow \text{length } as = \text{length } bs \wedge (\forall b \in \text{set } bs. r \ a \ b)$

<proof>

lemma *list-all2-same-leftI*:

$\forall a' \in \text{set } as. a' = a \implies \text{length } as = \text{length } bs \implies \forall b \in \text{set } bs. r \ a \ b \implies \text{list-all2 } r \text{ as } bs$

<proof>

lemma *list-all2-same-right*:

$\forall b' \in \text{set } bs. b' = b \implies \text{list-all2 } r \text{ as } bs \longleftrightarrow \text{length } as = \text{length } bs \wedge (\forall a \in \text{set } as. r \ a \ b)$

<proof>

lemma *list-all2-same-rightI*:

$\forall b' \in \text{set } bs. b' = b \implies \text{length } as = \text{length } bs \implies \forall a \in \text{set } as. r \ a \ b \implies \text{list-all2 } r \text{ as } bs$

<proof>

lemma *list-all2-all-all*:

$\forall a \in \text{set } as. \forall b \in \text{set } bs. r \ a \ b \implies \text{list-all2 } r \text{ as } bs \longleftrightarrow \text{length } as = \text{length } bs$

<proof>

lemma *list-all2-indep1*:

$\text{list-all2 } r \ (\lambda a \ b. P \ b) \text{ as } bs \longleftrightarrow \text{length } as = \text{length } bs \wedge (\forall b \in \text{set } bs. P \ b)$

<proof>

lemma *list-all2-indep2*:

$\text{list-all2 } r \ (\lambda a \ b. P \ a) \text{ as } bs \longleftrightarrow \text{length } as = \text{length } bs \wedge (\forall a \in \text{set } as. P \ a)$

<proof>

lemma *list-all2-replicate[simp]*:

$\text{list-all2 } r \ (\text{replicate } n \ x) \ ys \longleftrightarrow \text{length } ys = n \wedge (\forall y \in \text{set } ys. r \ x \ y)$

$\text{list-all2 } r \ xs \ (\text{replicate } n \ y) \longleftrightarrow \text{length } xs = n \wedge (\forall x \in \text{set } xs. r \ x \ y)$

<proof>

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>

lemma *list-all2-choice*: $\forall x \in \text{set } xs. \exists y. r \ x \ y \implies \exists ys. \text{list-all2 } r \ xs \ ys$

<proof>

lemma *list-all2-concat*:

list-all2 (list-all2 r) ass bss \implies list-all2 r (concat ass) (concat bss)

<proof>

lemma *those-eq-None[simp]*: *those as = None \longleftrightarrow None \in set as* *<proof>*

lemma *those-eq-Some[simp]*: *those xos = Some xs \longleftrightarrow xos = map Some xs*

<proof>

lemma *those-map-Some[simp]*: *those (map Some xs) = Some xs* *<proof>*

lemma *those-append*:

those (as @ bs) = do {xs \leftarrow those as; ys \leftarrow those bs; Some (xs@ys)}

<proof>

lemma *those-Cons*:

those (a#as) = do {x \leftarrow a; xs \leftarrow those as; Some (x # xs)}

<proof>

lemma *map-singleton-o[simp]*: *($\lambda x. [x]$) \circ f = ($\lambda x. [f x]$)* *<proof>*

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

lemma *in-set-updateD*: *x \in set (xs[n := y]) \implies x \in set xs \vee x = y*

<proof>

lemma *map-nth'*: *length xs = n \implies map (nth xs) [0.. n] = xs*

<proof>

lemma *product-lists-map-map*: *product-lists (map (map f) xss) = map (map f) (product-lists xss)*

<proof>

lemma (*in monoid-add*) *sum-list-concat*: *sum-list (concat xs) = sum-list (map sum-list xs)*

<proof>

context *semiring-1* **begin**

lemma *prod-list-map-sum-list-distrib*:

shows *prod-list (map sum-list xss) = sum-list (map prod-list (product-lists xss))*

<proof>

lemma *prod-list-sum-list-distrib*:

(\prod xs \leftarrow xss. \sum x \leftarrow xs. f x) = (\sum xs \leftarrow product-lists xss. \prod x \leftarrow xs. f x)

<proof>

end

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

$(\forall xs \in \text{set } xss. \exists x \in \text{set } xs. f x) \longleftrightarrow (\exists xs \in \text{set } (\text{product-lists } xss). \forall x \in \text{set } xs. f x)$
<proof>

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

$(\exists xs \in \text{set } xss. \forall x \in \text{set } xs. f x) \longleftrightarrow (\forall xs \in \text{set } (\text{product-lists } xss). \exists x \in \text{set } xs. f x)$
<proof>

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

lemma *map-nth-Cons*: $\text{map } (\text{nth } (x\#xs)) [0..<n] = (\text{case } n \text{ of } 0 \Rightarrow [] \mid \text{Suc } n \Rightarrow x \# \text{map } (\text{nth } xs) [0..<n])$
<proof>

lemma *upt-0-Suc-Cons*: $[0..<\text{Suc } i] = 0 \# \text{map } \text{Suc } [0..<i]$
<proof>

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

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* $xs @ \text{map } (\text{nth } ys) [0..<n - \text{length } xs]$ *)*
<proof>

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

lemma *trancl-Collect*: $\{(x,y). r x y\}^+ = \{(x,y). \text{tranclp } r x y\}$
<proof>

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

lemma *restrict-map-mono-left*: $A \subseteq_m A' \implies A \mid' S \subseteq_m A' \mid' S$
and *restrict-map-mono-right*: $S \subseteq S' \implies A \mid' S \subseteq_m A \mid' S'$
<proof>

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

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

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

syntax

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

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

syntax-consts

$all\text{-}hastype \equiv all\text{-}hastype\ \mathbf{and}$

$ex\text{-}hastype \equiv ex\text{-}hastype$

translations

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

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

lemmas $hastypeI = hastype\text{-}def[un\text{-}folded\ atomize\text{-}eq, THEN\ iffD2]$

lemmas $hastypeD[dest] = hastype\text{-}def[un\text{-}folded\ atomize\text{-}eq, THEN\ iffD1]$

lemmas $eq\text{-}Some\text{-}iff\text{-}hastype = hastype\text{-}def[symmetric]$

lemma $has\text{-}same\text{-}type$: **assumes** $a : \sigma\ in\ A$ **shows** $a : \sigma'\ in\ A \longleftrightarrow \sigma' = \sigma$
 $\langle proof \rangle$

lemma $sset\text{-}eqI$: **assumes** $(\bigwedge a. a : \sigma\ in\ A \longleftrightarrow a : \sigma\ in\ B)$ **shows** $A = B$
 $\langle proof \rangle$

lemma $in\text{-}dom\text{-}iff\text{-}ex\text{-}type$: $a \in dom\ A \longleftrightarrow (\exists \sigma. a : \sigma\ in\ A)\ \langle proof \rangle$

lemma $in\text{-}dom\text{-}hastypeE$: $a \in dom\ A \implies (\bigwedge \sigma. a : \sigma\ in\ A \implies thesis) \implies thesis$
 $\langle proof \rangle$

lemma $hastype\text{-}imp\text{-}dom[simp]$: $a : \sigma\ in\ A \implies a \in dom\ A\ \langle proof \rangle$

lemma $untyped\text{-}imp\text{-}not\text{-}hastype$: $A\ a = None \implies \neg a : \sigma\ in\ A\ \langle proof \rangle$

lemma $nex\text{-}hastype\text{-}iff$: $(\nexists \sigma. a : \sigma\ in\ A) \longleftrightarrow A\ a = None\ \langle proof \rangle$

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

Explicitly sorted sets:

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

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

Listwise type judgement:

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

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

lemma *has-same-type-list*:

$as :_l \sigma s \text{ in } A \implies as :_l \sigma s' \text{ in } A \iff \sigma s' = \sigma s$
<proof>

lemma *hastype-list-iff-those*: $as :_l \sigma s \text{ in } A \iff those (map A as) = Some \sigma s$
<proof>

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

lemma *hastype-list-imp-lists-dom*: $xs :_l \sigma s \text{ in } A \implies xs \in lists (dom A)$
<proof>

lemma *subssel*: $A \subseteq_m A' \iff (\forall a \sigma. a : \sigma \text{ in } A \implies a : \sigma \text{ in } A')$
<proof>

lemmas *subsselI* = *subssel*[THEN iffD2, rule-format]

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

lemma *subssel-hastype-listD*: $A \subseteq_m A' \implies as :_l \sigma s \text{ in } A \implies as :_l \sigma s \text{ in } A'$
<proof>

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

$a : \sigma \text{ in } A' \implies A \subseteq_m A' \implies a : \sigma' \text{ in } A \implies \sigma' = \sigma$
<proof>

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

$a : \sigma \text{ in } A' \implies A \subseteq_m A' \implies a \in dom A \iff a : \sigma \text{ in } A$
<proof>

Restriction of partial map, also depending on the value.

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

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

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

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

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

$a : \sigma \text{ in } \{a : \sigma \text{ in } A. P a \sigma\} \iff a : \sigma \text{ in } A \wedge P a \sigma$
<proof>

lemma *restrict-sset-cong*:

assumes $A = A'$

and $\bigwedge a \sigma. a : \sigma \text{ in } A \implies P a \sigma \iff P' a \sigma$

shows $\{a : \sigma \text{ in } A. P a \sigma\} = \{a : \sigma \text{ in } A'. P' a \sigma\}$

<proof>

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

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

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

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

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

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

lemma *all-set-hastype-iff-those*: $\forall a \in \text{set } as. a : \sigma \text{ in } A \implies \text{those } (\text{map } A as) = \text{Some } (\text{replicate } (\text{length } as) \sigma)$
 ⟨proof⟩

The partial version of list nth:

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

lemma *safe-nth-simp*[simp]: $i < \text{length } as \implies \text{safe-nth } as i = \text{Some } (as ! i)$
 ⟨proof⟩

lemma *safe-nth-None*[simp]:
 $\text{length } as \leq i \implies \text{safe-nth } as i = \text{None}$
 ⟨proof⟩

lemma *safe-nth*: $\text{safe-nth } as i = (\text{if } i < \text{length } as \text{ then } \text{Some } (as ! i) \text{ else } \text{None})$
 ⟨proof⟩

lemma *safe-nth-eq-SomeE*:
 $\text{safe-nth } as i = \text{Some } a \implies (i < \text{length } as \implies as ! i = a \implies \text{thesis}) \implies \text{thesis}$
 ⟨proof⟩

lemma *dom-safe-nth*[simp]: $\text{dom } (\text{safe-nth } as) = \{0..<\text{length } as\}$
 ⟨proof⟩

lemma *safe-nth-replicate*[simp]:
 $\text{safe-nth } (\text{replicate } n a) i = (\text{if } i < n \text{ then } \text{Some } a \text{ else } \text{None})$
 ⟨proof⟩

lemma *safe-nth-append*:

safe-nth (ls@rs) i = (if i < length ls then Some (ls!i) else safe-nth rs (i - length ls))
<proof>

lemma *hastype-in-safe-nth[simp]*: $i : \sigma$ in *safe-nth* $\sigma s \longleftrightarrow i < \text{length } \sigma s \wedge \sigma = \sigma s!i$

<proof>

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

lemma *hastype-in-o[simp]*: $a : \sigma$ in $A \circ f \longleftrightarrow f a : \sigma$ in A *<proof>*

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*: $a : \sigma'$ in $f \circ s A \longleftrightarrow (\exists \sigma. a : \sigma$ in $A \wedge \sigma' = f \sigma)$
<proof>

lemma *hastype-in-o-ssetI*: $a : \sigma$ in $A \implies f \sigma = \sigma' \implies a : \sigma'$ in $f \circ s A$
<proof>

lemma *hastype-in-o-ssetD*: $a : \tau$ in $f \circ s A \implies \exists \sigma. a : \sigma$ in $A \wedge \tau = f \sigma$
<proof>

lemma *hastype-in-o-ssetE*: $a : \tau$ in $f \circ s A \implies (\bigwedge \sigma. a : \sigma$ in $A \implies \tau = f \sigma \implies \text{thesis}) \implies \text{thesis}$
<proof>

lemma *o-sset-restrict-sset-assoc[simp]*: $f \circ s (A \upharpoonright' X) = (f \circ s A) \upharpoonright' X$
<proof>

lemma *id-o-sset[simp]*: $\text{id} \circ s A = A$
and *identity-o-sset[simp]*: $(\lambda x. x) \circ s A = A$
<proof>

lemma *o-ssetI*: $A x = \text{Some } y \implies z = f y \implies (f \circ s A) x = \text{Some } z$ *<proof>*

lemma *o-ssetE*: $(f \circ s A) x = \text{Some } z \implies (\bigwedge y. A x = \text{Some } y \implies z = f y \implies \text{thesis}) \implies \text{thesis}$
<proof>

lemma *dom-o-sset[simp]*: $\text{dom } (f \circ s A) = \text{dom } A$
<proof>

lemma *safe-nth-map*: *safe-nth* $(\text{map } f \text{ as}) = f \circ s \text{ safe-nth as}$
<proof>

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

lemma *safe-nth-Nil[simp]*: *safe-nth* [] = \emptyset *<proof>*

lemma *o-sset-empty[simp]*: *f* \circ *s* \emptyset = \emptyset *<proof>*

lemma *hastype-in-empty[simp]*: $\neg x : \sigma$ *in* \emptyset *<proof>*

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* : σ *in* *A* \Longrightarrow *b* : τ *in* *A* \Longrightarrow *f a* = *f b* \Longrightarrow
 $\sigma = \tau$
begin

lemma *same-value-imp-in-dom-iff*:
 assumes *fafa'*: *f a* = *f a'* **and** *a*: *a* : σ *in* *A* **shows** *a'*: *a' ∈ dom A* \longleftrightarrow *a' : σ*
 in A
 <proof>

lemma *sort-preserving-subset*:
 assumes *A' ⊆_m A*
 shows *sort-preserving f A'*
 <proof>

end

lemma *sort-preserving-cong*:
 $A = A' \Longrightarrow (\bigwedge a \sigma. a : \sigma \text{ in } A \Longrightarrow f a = f' a) \Longrightarrow \text{sort-preserving } f A \longleftrightarrow$
 $\text{sort-preserving } f' A'$
 <proof>

lemma *inj-on-dom-imp-sort-preserving*:
 assumes *inj-on f (dom A)* **shows** *sort-preserving f A*
 <proof>

lemma *inj-imp-sort-preserving*:
 assumes *inj f* **shows** *sort-preserving f A*
 <proof>

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

lemma *target-has-same-type*: *a* : σ *in* *A* \Longrightarrow *f a* : τ *in* *B* \longleftrightarrow $\sigma = \tau$
 <proof>

lemma *target-dom-iff-hastype*:
 $a : \sigma \text{ in } A \Longrightarrow f a \in \text{dom } B \longleftrightarrow f a : \sigma \text{ in } B$
 <proof>

lemma *source-dom-iff-hastype*:

$f a : \sigma \text{ in } B \implies a \in \text{dom } A \longleftrightarrow a : \sigma \text{ in } A$
<proof>

lemma *elim*:

assumes $a: (\bigwedge \sigma. a : \sigma \text{ in } A \implies f a : \sigma \text{ in } B) \implies P$
shows P
<proof>

sublocale *sort-preserving*

<proof>

lemma *funcset-dom*: $f : \text{dom } A \rightarrow \text{dom } B$

<proof>

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

<proof>

lemma *in-dom*: $a \in \text{dom } A \implies f a \in \text{dom } B$ *<proof>*

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

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

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

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

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

<proof>

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*: $B \subseteq_m B' \implies f :_s A \rightarrow B'$

<proof>

lemma *sorted-map-cong*:

$(\bigwedge a \sigma. a : \sigma \text{ in } A \implies f a = f' a) \implies$
 $A = A' \implies$
 $(\bigwedge a \sigma. a : \sigma \text{ in } A \implies f a : \sigma \text{ in } B \iff f a : \sigma \text{ in } B') \implies$
 $f :_s A \rightarrow B \iff f' :_s A' \rightarrow B'$
<proof>

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>

lemma *sorted-map-empty[simp]*: $f :_s \emptyset \rightarrow A$
<proof>

lemma *sorted-map-comp-nth*:

$\alpha :_s (f \circ_s \text{safe-nth } (a \# as)) \rightarrow A \iff \alpha 0 : f a \text{ in } A \wedge (\alpha \circ \text{Suc} :_s (f \circ_s \text{safe-nth}$
 $as) \rightarrow A)$
(is $?l \iff ?r$
<proof>

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*: $b : \sigma \text{ in } B \iff (\exists a : \sigma \text{ in } A. b = f a)$
<proof>

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

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

end

locale *sorted-bijection = sorted-map +*
assumes *bij*: *bij-betw* f $(\text{dom } A)$ $(\text{dom } B)$
begin

lemma *inj*: *inj-on* f $(\text{dom } A)$
<proof>

sublocale *sorted-surjection*
<proof>

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

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

end

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

lemma *some-hastype*:
 (*SOME* $a. a : \sigma \text{ in } A$) : $\sigma \text{ in } A$
 ⟨*proof*⟩

lemma *ex-sorted-map*: $\exists \alpha. \alpha :_s V \rightarrow A$
 ⟨*proof*⟩

end

3.2 Sorted Images

The partial version of *The* operator.

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

lemma *safe-The-cong*[*cong*]:
assumes *eq*: $\bigwedge x. P x \longleftrightarrow Q x$
shows *safe-The* $P =$ *safe-The* Q
 ⟨*proof*⟩

lemma *safe-The-eq-Some*: *safe-The* $P =$ *Some* $x \longleftrightarrow P x \wedge (\forall x'. P x' \longrightarrow x' = x)$
 ⟨*proof*⟩

lemma *safe-The-eq-None*: *safe-The* $P =$ *None* $\longleftrightarrow \neg(\exists !x. P x)$
 ⟨*proof*⟩

lemma *safe-The-False*[*simp*]: *safe-The* $(\lambda x. \text{False}) =$ *None*
 ⟨*proof*⟩

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

lemma *hastype-in-imageE*:
assumes *fx* : $\sigma \text{ in } f \text{ } ^{s} X$
and $\bigwedge x. x : \sigma \text{ in } X \Longrightarrow fx = f x \Longrightarrow$ *thesis*
shows *thesis*
 ⟨*proof*⟩

lemma *in-dom-image-hastypeE*:

$b \in \text{dom } (f^{\text{is}} A) \implies (\bigwedge a \sigma. a : \sigma \text{ in } A \implies b = f a \implies \text{thesis}) \implies \text{thesis}$
 ⟨proof⟩

lemma *in-dom-imageE*:

assumes $x: x \in \text{dom } (f^{\text{is}} A)$ **and** *main*: $\bigwedge a. a \in \text{dom } A \implies x = f a \implies \text{thesis}$
shows *thesis*

⟨proof⟩

context *sort-preserving* **begin**

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

⟨proof⟩

lemma *hastype-in-imageI2*: $a : \sigma \text{ in } A \implies f a : \sigma \text{ in } f^{\text{is}} A$

⟨proof⟩

lemma *hastype-in-image*: $b : \sigma \text{ in } f^{\text{is}} A \longleftrightarrow (\exists a : \sigma \text{ in } A. f a = b)$

⟨proof⟩

lemma *in-dom-imageI*: $a \in \text{dom } A \implies b = f a \implies b \in \text{dom } (f^{\text{is}} A)$

⟨proof⟩

lemma *in-dom-imageI2*: $a \in \text{dom } A \implies f a \in \text{dom } (f^{\text{is}} A)$

⟨proof⟩

lemma *hastype-list-in-image*: $bs :_l \sigma s \text{ in } f^{\text{is}} A \longleftrightarrow (\exists as. as :_l \sigma s \text{ in } A \wedge \text{map } f as = bs)$

⟨proof⟩

lemma *dom-image[simp]*: $\text{dom } (f^{\text{is}} A) = f^{\text{is}} \text{ dom } A$

⟨proof⟩

sublocale *to-image*: *sorted-map* $f A f^{\text{is}} A$

⟨proof⟩

lemma *sorted-map-iff-image-subset*:

$f :_s A \rightarrow B \longleftrightarrow f^{\text{is}} A \subseteq_m B$

⟨proof⟩

end

lemma *sort-preserving-o*:

assumes f : *sort-preserving* $f A$ **and** g : *sort-preserving* $g (f^{\text{is}} A)$

shows *sort-preserving* $(g \circ f) A$

⟨proof⟩

lemma *sorted-image-image*:

assumes f : *sort-preserving* $f A$ **and** g : *sort-preserving* $g (f^{\text{is}} A)$

shows $g^{\text{is}} f^{\text{is}} A = (g \circ f)^{\text{is}} A$

<proof>

context *sorted-map* **begin**

lemma *image-subset[intro!]*: $f \text{ }^{\text{is}}$ $A \subseteq_m B$
<proof>

lemma *dom-image-subset[intro!]*: $f \text{ }^{\text{c}}$ $\text{dom } A \subseteq \text{dom } B$
<proof>

end

lemma *sorted-image-cong*: $(\bigwedge a \sigma. a : \sigma \text{ in } A \implies f a = f' a) \implies f \text{ }^{\text{is}} A = f' \text{ }^{\text{is}} A$
<proof>

lemma *inj-on-dom-imp-sort-preserving-inv-into*:
assumes *inj*: *inj-on* f (*dom* A) **shows** *sort-preserving* (*inv-into* (*dom* A) f) ($f \text{ }^{\text{is}}$ A)
<proof>

lemma *inj-imp-sort-preserving-inv*:
assumes *inj*: *inj* f **shows** *sort-preserving* (*inv* f) ($f \text{ }^{\text{is}}$ A)
<proof>

lemma *inj-on-dom-imp-inv-into-image-cancel*:
assumes *inj*: *inj-on* f (*dom* A)
shows *inv-into* (*dom* A) $f \text{ }^{\text{is}}$ $f \text{ }^{\text{is}}$ $A = A$
<proof>

lemma *inj-imp-inv-image-cancel*:
assumes *inj*: *inj* f
shows *inv* $f \text{ }^{\text{is}}$ $f \text{ }^{\text{is}}$ $A = A$
<proof>

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

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

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 $vars \equiv vars\text{-term}$

consts $map\text{-vars} :: ('a \Rightarrow 'b) \Rightarrow 'c \Rightarrow 'd$

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

lemma $map\text{-term-eq-Var}$: $map\text{-term } F V s = Var y \longleftrightarrow (\exists x. s = Var x \wedge y = V x)$
 $\langle proof \rangle$

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

lemma $map\text{-var-term-id[simp]}$: $map\text{-term } (\lambda x. x) id = id \langle proof \rangle$

lemma $map\text{-term-eq-Fun}$:

$map\text{-term } F V s = Fun g ts \longleftrightarrow (\exists f ss. s = Fun f ss \wedge g = F f \wedge ts = map (map\text{-term } F V) ss)$
 $\langle proof \rangle$

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 $('f, 's) \text{ sig} = 'f \times 's \text{ list} \rightarrow 's$

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

lemmas $fun\text{-hastypeI} = fun\text{-hastype-def}[unfolding atomize-eq, THEN iffD2]$

lemmas $fun\text{-hastypeD} = fun\text{-hastype-def}[unfolding atomize-eq, THEN iffD1]$

lemma $fun\text{-hastype-imp-dom[simp]}$:

assumes $f : \sigma \rightarrow \tau \text{ in } F$ **shows** $(f, \sigma) \in dom F$
 $\langle proof \rangle$

lemma $in\text{-dom-fun-hastypeE}$:

assumes $(f, \sigma) \in dom F$ **and** $\bigwedge \tau. f : \sigma \rightarrow \tau \text{ in } F \implies thesis$ **shows** $thesis$
 $\langle proof \rangle$

lemma $fun\text{-has-same-type}$:

assumes $f : \sigma \rightarrow \tau \text{ in } F$ **and** $f : \sigma \rightarrow \tau' \text{ in } F$ **shows** $\tau = \tau'$

$\langle \text{proof} \rangle$

lemma *fun-hastype-empty[simp]*: $\neg f : \sigma \rightarrow \tau$ in \emptyset
 $\langle \text{proof} \rangle$

lemma *fun-hastype-upd*: $f : \sigma \rightarrow \tau$ in $F((f', \sigma') \mapsto \tau') \longleftrightarrow$
(if $f = f' \wedge \sigma = \sigma'$ then $\tau = \tau'$ else $f : \sigma \rightarrow \tau$ in F)
 $\langle \text{proof} \rangle$

lemma *fun-hastype-restrict*: $f : \sigma \rightarrow \tau$ in $F \mid S \longleftrightarrow (f, \sigma) \in S \wedge f : \sigma \rightarrow \tau$ in F
 $\langle \text{proof} \rangle$

lemma *subssigI*: **assumes** $\bigwedge f \sigma \tau. f : \sigma \rightarrow \tau$ in $F \implies f : \sigma \rightarrow \tau$ in F'
shows $F \subseteq_m F'$
 $\langle \text{proof} \rangle$

lemma *subssigD*: **assumes** $FF: F \subseteq_m F'$ **and** $f : \sigma \rightarrow \tau$ in F **shows** $f : \sigma \rightarrow \tau$
in F'
 $\langle \text{proof} \rangle$

The sorted set of terms:

primrec *Term* ($\langle \mathcal{T}'(-, -) \rangle$) **where**
 $\mathcal{T}(F, V) (\text{Var } v) = V v$
 $\mid \mathcal{T}(F, V) (\text{Fun } f \text{ ss}) =$
(case those (map $\mathcal{T}(F, V)$ ss) of None \implies None \mid Some $\sigma s \implies F (f, \sigma s)$)

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

lemma *Fun-hastype*:
 $\text{Fun } f \text{ ss} : \tau$ in $\mathcal{T}(F, V) \longleftrightarrow (\exists \sigma s. f : \sigma s \rightarrow \tau$ in $F \wedge \text{ss} :_l \sigma s$ in $\mathcal{T}(F, V))$
 $\langle \text{proof} \rangle$

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

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

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

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 $F : \bigwedge f \text{ ss } \sigma s \tau.$
 $f : \sigma s \rightarrow \tau \text{ in } F \implies \text{ss} ;_l \sigma s \text{ in } \mathcal{T}(F, V) \implies \forall s \in \text{set ss}. P s \implies P (F \text{un } f \text{ ss})$
shows $P s$
 $\langle \text{proof} \rangle$

lemma *Term-mono-left*: **assumes** $FF : F \subseteq_m F'$ **shows** $\mathcal{T}(F, V) \subseteq_m \mathcal{T}(F', V)$
 $\langle \text{proof} \rangle$

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** $\mathcal{T}(F, V) \subseteq_m \mathcal{T}(F, V')$
 $\langle \text{proof} \rangle$

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*: $s : \sigma \text{ in } \mathcal{T}(F, V \mid \text{' vars } s) \longleftrightarrow s : \sigma \text{ in } \mathcal{T}(F, V)$
(is $?l s \longleftrightarrow ?r s$
 $\langle \text{proof} \rangle$

lemma *hastype-in-Term-imp-vars*: $s : \sigma \text{ in } \mathcal{T}(F, V) \implies v \in \text{vars } s \implies v \in \text{dom } V$
 $\langle \text{proof} \rangle$

lemma *hastype-in-Term-imp-vars-subset*: $s : \sigma \text{ in } \mathcal{T}(F, V) \implies \text{vars } s \subseteq \text{dom } V$
 $\langle \text{proof} \rangle$

lemma *in-dom-Term-imp-vars*: $s \in \text{dom } \mathcal{T}(F, V) \implies v \in \text{vars } s \implies v \in \text{dom } V$
 $\langle \text{proof} \rangle$

lemma *in-dom-Term-vars-subset*: $s \in \text{dom } \mathcal{T}(F, V) \implies \text{vars } s \subseteq \text{dom } V$
 $\langle \text{proof} \rangle$

interpretation *Var*: *sorted-map Var V* $\mathcal{T}(F, V)$ **for** $F V$ $\langle \text{proof} \rangle$

4.3 Sorted Algebras

```

locale sorted-algebra-syntax =
  fixes  $F :: ('f, 's)$  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  $F \Longrightarrow as :_l \sigma s$  in  $A \Longrightarrow I f as : \tau$  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  $A$ 
  <proof>

interpretation eval: sorted-map  $\lambda s. I[s]\alpha$   $\mathcal{T}(F, V)$   $A$ 
  <proof>

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>

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

```

shows *sorted-algebra* $F A I = \text{sorted-algebra } F' A' I'$
 ⟨*proof*⟩

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 $F V$
 ⟨*proof*⟩

Sorted substitution preserves type:

lemma *subst-hastype*: $\vartheta :_s X \rightarrow \mathcal{T}(F, V) \implies s : \sigma \text{ in } \mathcal{T}(F, X) \implies s \cdot \vartheta : \sigma \text{ in } \mathcal{T}(F, V)$
 ⟨*proof*⟩

lemma *subst-compose-sorted-map*:

$\vartheta :_s X \rightarrow \mathcal{T}(F, Y) \implies \varrho :_s Y \rightarrow \mathcal{T}(F, Z) \implies \vartheta \circ_s \varrho :_s X \rightarrow \mathcal{T}(F, Z)$
 ⟨*proof*⟩

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 $\forall x \in \text{vars } s. \forall \sigma. \vartheta x : \sigma \text{ in } \mathcal{T}(F, W) \iff x : \sigma \text{ in } V$

shows $s \cdot \vartheta : \sigma \text{ in } \mathcal{T}(F, W) \iff s : \sigma \text{ in } \mathcal{T}(F, V)$

⟨*proof*⟩

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** $\vartheta x \in \text{dom } \mathcal{T}(F, V)$

⟨*proof*⟩

lemma *subst-sorted-map-restrict-vars*:

assumes $\vartheta : \vartheta :_s X \rightarrow \mathcal{T}(F, V)$ **and** $WV : W \subseteq_m V$ **and** $s\vartheta : s \cdot \vartheta \in \text{dom } \mathcal{T}(F, W)$

shows $\vartheta :_s X \upharpoonright' \text{vars } s \rightarrow \mathcal{T}(F, W)$

⟨*proof*⟩

4.3.2 Homomorphisms

locale *sorted-distributive* =

sort-preserving $\varphi A +$ *source: sorted-algebra* $F A I$ **for** $F \varphi A I J +$

assumes *distrib*: $f : \sigma s \rightarrow \tau \text{ in } F \implies \text{as} :_l \sigma s \text{ in } A \implies \varphi (I f \text{as}) = J f (\text{map } \varphi \text{as})$

begin

lemma *distrib-eval*:

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

shows $\varphi (I[[s]]\alpha) = J[[s]](\varphi \circ \alpha)$

<proof>

The image of a distributive map forms a sorted algebra.

sublocale *image: sorted-algebra* $F \varphi \text{ } ^s A J$

<proof>

end

lemma *sorted-distributive-cong*:

fixes $A A' :: 'a \rightarrow 's$ **and** $\varphi :: 'a \Rightarrow 'b$ **and** $I :: 'f \Rightarrow 'a \text{ list} \Rightarrow 'a$

assumes $\varphi: \bigwedge a \sigma. a : \sigma$ in $A \implies \varphi a = \varphi' a$

and $A: A = A'$

and $I: \bigwedge f \sigma s \tau as. f : \sigma s \rightarrow \tau$ in $F \implies as :_l \sigma s$ in $A \implies I f as = I' f as$

and $J: \bigwedge f \sigma s \tau as. f : \sigma s \rightarrow \tau$ in $F \implies as :_l \sigma s$ in $A \implies J f (map \varphi as) = J' f (map \varphi as)$

shows *sorted-distributive* $F \varphi A I J = \text{sorted-distributive } F \varphi' A' I' J'$

<proof>

lemma *sorted-distributive-o*:

assumes *sorted-distributive* $F \varphi A I J$ **and** *sorted-distributive* $F \psi (\varphi \text{ } ^s A) J K$

shows *sorted-distributive* $F (\psi \circ \varphi) A I K$

<proof>

locale *sorted-homomorphism = sorted-distributive* $F \varphi A I J + \text{sorted-map } \varphi A B +$

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>

context *sorted-algebra* **begin**

context **fixes** αV **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* $\alpha V A$ *<proof>*

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

interpretation *eval*: *sorted-homomorphism* $F \langle \lambda s. I[s]\alpha \rangle \langle \mathcal{T}(F, V) \rangle \text{Fun } A \ I$
 $\langle \text{proof} \rangle$

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

and $A: A = A'$

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

and $B: B = B'$

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

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

$\langle \text{proof} \rangle$

context *sort-preserving* **begin**

lemma *sort-preserving-map-vars*: *sort-preserving* (*map-vars* f) $\mathcal{T}(F, A)$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

end

context *sorted-map* **begin**

lemma *sorted-map-map-vars*: *map-vars* $f \text{ }_s \mathcal{T}(F, A) \rightarrow \mathcal{T}(F, B)$

$\langle \text{proof} \rangle$

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* $A \ a \equiv \text{if } a \in A \text{ then } \text{Some } () \text{ else } \text{None}$

lemma *unisorted-eq-Some[simp]*: *unisorted* $A \ a = \text{Some } \sigma \longleftrightarrow a \in A$

and *unisorted-eq-None[simp]*: *unisorted* $A \ a = \text{None} \longleftrightarrow a \notin A$

and *hastype-in-unisorted[simp]*: $a : \sigma \text{ in } \text{unisorted } A \longleftrightarrow a \in A$

$\langle \text{proof} \rangle$

lemma *hastype-list-in-unsorted*[simp]: $as :_l \sigma s$ in *unsorted* $A \iff \text{length } as = \text{length } \sigma s \wedge \text{set } as \subseteq A$

<proof>

lemma *dom-unsorted*[simp]: $\text{dom } (\text{unsorted } A) = A$

<proof>

lemma *unsorted-map*[simp]:

$f :_s \text{unsorted } A \rightarrow \tau \iff f : A \rightarrow \text{dom } \tau$

$f :_s \sigma \rightarrow \text{unsorted } B \iff f : \text{dom } \sigma \rightarrow B$

<proof>

lemma *image-unsorted*[simp]: $f \text{ ` } s \text{ unsorted } A = \text{unsorted } (f \text{ ` } A)$

<proof>

definition *unsorted-sig* :: $(f \times \text{nat}) \text{ set} \Rightarrow (f, \text{unit}) \text{ sig}$

where *unsorted-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-unsorted-sig*[simp]: $f : \sigma s \rightarrow \tau$ in *unsorted-sig* $F \iff (f, \text{length } \sigma s) \in F$

<proof>

inductive-set *uTerm* $(\langle \mathfrak{T}'(-, -) \rangle [1, 1] 1000)$ **for** $F V$ **where**

$\text{Var } v \in \mathfrak{T}(F, V)$ **if** $v \in V$

$|\forall s \in \text{set } ss. s \in \mathfrak{T}(F, V) \implies \text{Fun } f \text{ } ss \in \mathfrak{T}(F, V)$ **if** $(f, \text{length } ss) \in F$

lemma *Var-in-Term*[simp]: $\text{Var } x \in \mathfrak{T}(F, V) \iff x \in V$

<proof>

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

<proof>

lemma *hastype-in-unsorted-Term*[simp]:

$s : \sigma$ in $\mathcal{T}(\text{unsorted-sig } F, \text{unsorted } V) \iff s \in \mathfrak{T}(F, V)$

<proof>

lemma *unsorted-Term*: $\mathcal{T}(\text{unsorted-sig } F, \text{unsorted } V) = \text{unsorted } \mathfrak{T}(F, V)$

<proof>

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 *unsorted-algebra*: *sorted-algebra* $(\text{unsorted-sig } F)$ $(\text{unsorted } A)$ $I \iff \text{algebra } F A I$

(is $?l \iff ?r$ **)**

<proof>

context algebra begin

interpretation *unsorted: sorted-algebra* $\langle \text{unsorted-sig } F \rangle \langle \text{unsorted } A \rangle I$
<proof>

lemma *eval-closed*: $\alpha : V \rightarrow A \implies s \in \mathfrak{T}(F, V) \implies I[s]\alpha \in A$
<proof>

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 $(\text{unsorted-sig } F) \varphi (\text{unsorted } A) I J \longleftrightarrow$
distributive $F \varphi A I J$ (**is** $?l \longleftrightarrow ?r$)
<proof>

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 $(\text{unsorted-sig } F) \varphi (\text{unsorted } A) I (\text{unsorted } B) J \longleftrightarrow$
homomorphism $F \varphi A I B J$ (**is** $?l \longleftrightarrow ?r$)
<proof>

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>

context algebra begin

interpretation *unsorted: sorted-algebra* $\langle \text{unsorted-sig } F \rangle \langle \text{unsorted } A \rangle I$
<proof>

lemma *eval-homomorphism*: $\alpha : V \rightarrow A \implies \text{homomorphism } F (\lambda s. I[s]\alpha) \mathfrak{T}(F, V)$
Fun $A I$
<proof>

end

context *homomorphism* **begin**

interpretation *unsorted*: *sorted-homomorphism* $\langle \text{unsorted-sig } F \rangle \varphi \langle \text{unsorted } A \rangle I \langle \text{unsorted } B \rangle J$
 $\langle \text{proof} \rangle$

lemma *distrib-eval*: $\alpha : V \rightarrow A \implies s \in \mathfrak{T}(F, V) \implies \varphi (I[s]\alpha) = J[s](\varphi \circ \alpha)$
 $\langle \text{proof} \rangle$

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 = -)
 $\langle \text{proof} \rangle$

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

interpretation *unsorted*: *algebra* $UNIV$ $UNIV$ I
rewrites $\bigwedge a. a \in UNIV \longleftrightarrow True$
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 F I
 $\langle \text{proof} \rangle$

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
 $\langle \text{proof} \rangle$

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

lemma *subst-eval*: $I[s \cdot \vartheta]\alpha = I[s](\lambda x. I[\vartheta x]\alpha)$
 $\langle \text{proof} \rangle$

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* $(Fun f ss) = concat (map \text{var-list-term } ss)$
and *var-list-Var[simp]*: *var-list-term* $(Var x) = [x]$
 $\langle \text{proof} \rangle$

lemma *set-var-list[simp]*: $set (var-list-term s) = vars s$
 ⟨proof⟩

lemma *eval-subset-Un-vars*:
 assumes $\forall f as. foo (I f as) \subseteq \bigcup (foo \text{ ` } set as)$
 shows $foo (I[s]\alpha) \subseteq (\bigcup_{x \in vars-term s. foo (\alpha x))$
 ⟨proof⟩

4.6 Ground Terms

lemma *Term-empty-vars*: $s : \sigma$ in $\mathcal{T}(F, \emptyset) \implies vars s = \{\}$
 ⟨proof⟩

lemma *Term-empty-vars-subst*: $s : \sigma$ in $\mathcal{T}(F, \emptyset) \implies vars (s \cdot \vartheta) = \{\}$
 ⟨proof⟩

lemma *Term-empty-iff*: $s : \sigma$ in $\mathcal{T}(F, V) \wedge ground s \iff s : \sigma$ in $\mathcal{T}(F, \emptyset)$
 ⟨proof⟩

lemma *Term-empty-imp-ground*: $s : \sigma$ in $\mathcal{T}(F, \emptyset) \implies ground s$
 ⟨proof⟩

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

lemma *in-dom-Term-empty-imp-subst*:
 $s \in dom \mathcal{T}(F, \emptyset) \implies s \cdot \vartheta \in dom \mathcal{T}(F, V)$
 ⟨proof⟩

lemma *eval-ground-eq*: $ground s \implies I[s]\alpha = I[s]\alpha'$
 ⟨proof⟩

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

lemmas *subst-Term-empty-eq = eval-Term-empty-eq[where I=Fun]*

lemma *subst-Term-empty-id*:
 assumes $s : s : \sigma$ in $\mathcal{T}(F, \emptyset)$ shows $s \cdot \vartheta = s$
 ⟨proof⟩

lemma *subst-subst-Term-empty*:
 $s : \sigma$ in $\mathcal{T}(F, \emptyset) \implies s \cdot \vartheta \cdot \rho = s \cdot undefined$
 ⟨proof⟩

lemma *in-dom-Term-empty-subst-id*:
 $s \in dom \mathcal{T}(F, \emptyset) \implies s \cdot \vartheta = s$
 ⟨proof⟩

lemma *in-dom-Term-empty-subst-subst*:

$s \in \text{dom } \mathcal{T}(F, \emptyset) \implies s \cdot \vartheta \cdot \varrho = s \cdot \text{undefined}$
 ⟨proof⟩

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

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

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

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

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

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

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*:
 $(\forall s : \sigma \text{ in } \mathcal{T}(F, \emptyset). P (s \cdot \vartheta)) \longleftrightarrow (\forall s : \sigma \text{ in } \mathcal{T}(F, \emptyset). P s)$
 ⟨proof⟩

end

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

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

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**
 $empty_sort F \sigma \longleftrightarrow \{s. s : \sigma \text{ in } \mathcal{T}(F)\} = \{\}$

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

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

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

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

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

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

lemma *empty-sort-imp-finite[simp]*: $\text{empty-sort } F \sigma \implies \text{finite-sort } F \sigma$
<proof>

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

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

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>

4.6.2 Enumerating Ground Terms

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

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

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

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

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

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

$\langle \text{proof} \rangle$

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

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

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

$\langle \text{proof} \rangle$

lemma *term-of-index-bij*:

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

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

$\langle \text{proof} \rangle$

4.7 Subsignatures

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

lemmas $\text{Term-subset} = \text{Term-mono-left}[OF \ \text{subssig}]$

lemmas $\text{hastype-in-Term-sub} = \text{Term-subset}[THEN \ \text{subsetD}]$

lemma *subsignature*: $f : \sigma s \rightarrow \tau$ in $F \implies f : \sigma s \rightarrow \tau$ in G

$\langle \text{proof} \rangle$

end

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

begin

sublocale *sorted-algebra* $F \ A \ I$

$\langle \text{proof} \rangle$

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 $\text{subcar} : A \subseteq_m B$

assumes $\text{subintp} : f : \sigma s \rightarrow \tau$ in $F \implies \text{as} : \iota \ \sigma s$ in $A \implies I \ f \ \text{as} = J \ f \ \text{as}$

begin

lemma *subcarrier*: $a : \sigma$ in $A \implies a : \sigma$ in B

$\langle \text{proof} \rangle$

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>

end

lemma *term-subalgebra*:

assumes $FG: F \subseteq_m G$ **and** $VW: V \subseteq_m W$
shows *subalgebra* $F \mathcal{T}(F, V)$ *Fun* $G \mathcal{T}(G, W)$ *Fun*
<proof>

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

definition *constant-at f* $\sigma s i \equiv$

$\forall as b. as :_i \sigma s \text{ in } A \longrightarrow A b = A (as!i) \longrightarrow If (as[i:=b]) = If as$

lemma *constant-atI[intro]*:

assumes $\bigwedge as b. as :_i \sigma s \text{ in } A \implies A b = A (as!i) \implies If (as[i:=b]) = If as$
shows *constant-at f* $\sigma s i$ *<proof>*

lemma *constant-atD*:

constant-at f $\sigma s i \implies as :_i \sigma s \text{ in } A \implies A b = A (as!i) \implies If (as[i:=b]) = If as$
<proof>

lemma *constant-atE[elim]*:

assumes *constant-at f* $\sigma s i$
and $(\bigwedge as b. as :_i \sigma s \text{ in } A \implies A b = A (as!i) \implies If (as[i:=b]) = If as) \implies$
thesis
shows *thesis* *<proof>*

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*
<proof>

lemma *constant-term-onD*:

assumes *constant-term-on s x* **shows** $I[s]\alpha(x:=a) = I[s]\alpha$
<proof>

lemma *constant-term-onE*:

assumes *constant-term-on s x* **and** $(\bigwedge \alpha a. I[s]\alpha(x:=a) = I[s]\alpha) \implies$ *thesis*
shows *thesis* *<proof>*

lemma *constant-term-on-extra-var*: $x \notin \text{vars } s \implies$ *constant-term-on s x*
<proof>

lemma *constant-term-on-eq*:

assumes $st: I[s] = I[t]$ **and** $s: \text{constant-term-on } s x$ **shows** *constant-term-on t x*

<proof>

definition *constant-term* $s \equiv \forall x. \text{constant-term-on } s \ x$

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

<proof>

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

<proof>

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 $F \ A \ (\text{Hole}, \sigma) = \text{Some } \sigma$

| *aContext* $F \ A \ (\text{More } f \ ls \ C \ rs, \sigma) = \text{do } \{$

$\varrho s \leftarrow \text{those } (\text{map } A \ ls);$

$\mu \leftarrow \text{aContext } F \ A \ (C, \sigma);$

$\nu s \leftarrow \text{those } (\text{map } A \ rs);$

$F \ (f, \varrho s @ \mu \# \nu s)\}$

Term contexts are abstract contexts in the term algebra.

abbreviation *Context* $(\langle \mathcal{C}'(-, / -') \rangle [1, 1] 1000)$ **where**

$\mathcal{C}(F, V) \equiv \text{aContext } F \ \mathcal{T}(F, V)$

lemma *Hole-hastype[simp]*: $\text{Hole} : \sigma \rightarrow \tau$ *in* *aContext* $F \ A \longleftrightarrow \sigma = \tau$

and *More-hastype*: $\text{More } f \ ls \ C \ rs : \sigma \rightarrow \tau$ *in* *aContext* $F \ A \longleftrightarrow (\exists \varrho s \ \mu \ \nu s.$

$f : \varrho s @ \mu \# \nu s \rightarrow \tau$ *in* $F \wedge$

$ls :_1 \varrho s$ *in* $A \wedge$

$C : \sigma \rightarrow \mu$ *in* *aContext* $F \ A \wedge$

$rs :_1 \nu s$ *in* $A)$

<proof>

lemma *More-hastypeI*:

assumes $f : \varrho s @ \mu \# \nu s \rightarrow \tau$ *in* F

and $ls :_1 \varrho s$ *in* A

and $C : \sigma \rightarrow \mu$ *in* *aContext* $F \ A$

and $rs :_l \nu s$ in A
shows $\text{More } f \text{ } ls \text{ } C \text{ } rs : \sigma \rightarrow \tau$ in $\text{aContext } F \text{ } A$
 $\langle \text{proof} \rangle$

lemma $\text{hastype-aContext-induct}$ [*consumes 1, case-names Hole More*]:

assumes $C: C : \sigma \rightarrow \tau$ in $\text{aContext } F \text{ } A$
and $\text{hole}: P \square \sigma$
and $\text{more}: \bigwedge f \mu s \varrho \nu s \tau \text{ } ls \text{ } C \text{ } rs.$
 $f : \mu s @ \varrho \# \nu s \rightarrow \tau$ in $F \implies$
 $ls :_l \mu s$ in $A \implies$
 $C : \sigma \rightarrow \varrho$ in $\text{aContext } F \text{ } A \implies$
 $P \text{ } C \text{ } \varrho \implies$
 $rs :_l \nu s$ in $A \implies$
 $P (\text{More } f \text{ } ls \text{ } C \text{ } rs) \tau$
shows $P \text{ } C \text{ } \tau$
 $\langle \text{proof} \rangle$

lemma $\text{hastype-aContext-cases}$ [*consumes 1, case-names Hole More*]:

assumes $C: C : \sigma \rightarrow \tau$ in $\text{aContext } F \text{ } A$
and $\text{hole}: C = \square \implies \text{thesis}$
and $\text{more}: \bigwedge f \mu s \varrho \nu s \text{ } ls \text{ } D \text{ } rs.$
 $C = \text{More } f \text{ } ls \text{ } D \text{ } rs \implies$
 $f : \mu s @ \varrho \# \nu s \rightarrow \tau$ in $F \implies$
 $ls :_l \mu s$ in $A \implies$
 $D : \sigma \rightarrow \varrho$ in $\text{aContext } F \text{ } A \implies$
 $rs :_l \nu s$ in $A \implies$
 thesis
shows thesis
 $\langle \text{proof} \rangle$

lemma (*in sorted-map*) $\text{map-args-actxt-hastype}$:

assumes $C : \sigma \rightarrow \tau$ in $\text{aContext } F \text{ } A$
shows $\text{map-args-actxt } f \text{ } C : \sigma \rightarrow \tau$ in $\text{aContext } F \text{ } B$
 $\langle \text{proof} \rangle$

context sorted-algebra **begin**

lemma intp-ctxt-hastype :

assumes $C: C : \sigma \rightarrow \tau$ in $\text{aContext } F \text{ } A$ **and** $a: a : \sigma$ in A
shows $I\langle C; a \rangle : \tau$ in A
 $\langle \text{proof} \rangle$

lemma $\text{ctxt-has-same-type}$:

assumes $C: C : \sigma \rightarrow \tau$ in $\text{aContext } F \text{ } A$ **and** $a : \sigma$ in A
shows $I\langle C; a \rangle : \tau' \text{ in } A \iff \tau' = \tau$
 $\langle \text{proof} \rangle$

lemma eval-ctxt-hastype :

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

shows $I[C]_c \alpha : \sigma \rightarrow \tau$ in aContext $F A$
 ⟨proof⟩

end

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

lemma *subt-in-dom*:

assumes $s : s \in \text{dom } \mathcal{T}(F, V)$ **and** $st : s \supseteq t$ **shows** $t \in \text{dom } \mathcal{T}(F, V)$
 ⟨proof⟩

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 $\mathcal{T}(F, V)$
 ⟨proof⟩

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

assumes $C\langle t \rangle \in \text{dom } \mathcal{T}(F, V)$
shows $t \in \text{dom } \mathcal{T}(F, V)$
 ⟨proof⟩

lemma *apply-ctx-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 $\mathcal{C}(F, V)$
 ⟨proof⟩

end

theory *Basic-Terms*

imports *Sorted-Contexts*

begin

lemma *map-le-map-add-left*:

assumes *disj*: $\text{dom } m \cap \text{dom } n = \{\}$
shows $m \subseteq_m m ++ n$
 ⟨proof⟩

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* \Rightarrow $(f, 's)$ *ssig* \Rightarrow $(v \rightarrow 's) \Rightarrow (f, 'v)$ *term* \rightarrow $'s$
 ($\mathcal{T}_B'(-, -, -)$)
where $\mathcal{T}_B(C, D, X)$ (*Var* x) = *None*
 | $\mathcal{T}_B(C, D, X)$ (*Fun* f *ss*) = (*case those* (*map* $\mathcal{T}(C, X)$ *ss*) *of Some* $\sigma s \Rightarrow D$ ($f, \sigma s$)
 | - \Rightarrow *None*)

abbreviation *ground-Basic-Term* ($\mathcal{T}_B'(\cdot, \cdot)$) **where**
 $\mathcal{T}_B(C, D) \equiv \mathcal{T}_B(C, D, \lambda x. \text{None}) :: (\text{f}, \text{unit}) \text{ term} \rightarrow 's$

lemma *Var-hastype-Basic*:
 $\text{Var } x : \sigma \text{ in } \mathcal{T}_B(C, D, X) \longleftrightarrow \text{False} \langle \text{proof} \rangle$

lemma *Fun-hastype-Basic*:
 $\text{Fun } f \text{ ss} : \tau \text{ in } \mathcal{T}_B(C, D, X) \longleftrightarrow (\exists \sigma s. f : \sigma s \rightarrow \tau \text{ in } D \wedge \text{ss} :_l \sigma s \text{ in } \mathcal{T}(C, X))$
 $\langle \text{proof} \rangle$

lemma *hastype-Basic*:
 $s : \tau \text{ in } \mathcal{T}_B(C, D, X) \longleftrightarrow (\exists f \text{ ss } \sigma s. s = \text{Fun } f \text{ ss} \wedge f : \sigma s \rightarrow \tau \text{ in } D \wedge \text{ss} :_l \sigma s \text{ in } \mathcal{T}(C, X))$
 $\langle \text{proof} \rangle$

lemma *in-dom-Basic*:
 $s \in \text{dom } \mathcal{T}_B(C, D, X) \longleftrightarrow (\exists f \text{ ss } \sigma s \tau. s = \text{Fun } f \text{ ss} \wedge f : \sigma s \rightarrow \tau \text{ in } D \wedge \text{ss} :_l \sigma s \text{ in } \mathcal{T}(C, X))$
 $\langle \text{proof} \rangle$

lemma *hastype-BasicE*:
assumes $s : \tau \text{ in } \mathcal{T}_B(C, D, X)$
and $\bigwedge f \text{ ss } \sigma s. s = \text{Fun } f \text{ ss} \implies f : \sigma s \rightarrow \tau \text{ in } D \implies \text{ss} :_l \sigma s \text{ in } \mathcal{T}(C, X) \implies$
thesis
shows *thesis*
 $\langle \text{proof} \rangle$

lemma *hastype-BasicI*:
 $s = \text{Fun } f \text{ ss} \implies f : \sigma s \rightarrow \tau \text{ in } D \implies \text{ss} :_l \sigma s \text{ in } \mathcal{T}(C, X) \implies s : \tau \text{ in } \mathcal{T}_B(C, D, X)$
 $\langle \text{proof} \rangle$

lemma *Basic-mono*:
assumes $D : D \subseteq_m D'$ **and** $C : C \subseteq_m C'$ **and** $X : X \subseteq_m X'$
shows $\mathcal{T}_B(C, D, X) \subseteq_m \mathcal{T}_B(C', D', X')$
 $\langle \text{proof} \rangle$

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

lemma *Basic-Term*:
assumes $\text{disj} : \text{dom } C \cap \text{dom } D = \{\}$
shows $\mathcal{T}_B(C, D, X) \subseteq_m \mathcal{T}(C++D, X)$
 $\langle \text{proof} \rangle$

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

lemma *in-dom-Basic-Term-imp-vars*: $s \in \text{dom } \mathcal{T}_B(C,D,V) \implies x \in \text{vars } s \implies x \in \text{dom } V$

<proof>

lemma *in-dom-Basic-empty-subst-id*:

assumes $s \in \text{dom } \mathcal{T}_B(C,D,\emptyset)$

shows $s \cdot \vartheta = s$

<proof>

lemma *in-dom-Basic-empty-subst-subst*:

assumes $s \in \text{dom } \mathcal{T}_B(C,D,\emptyset)$

shows $s \cdot \vartheta \cdot \tau = s \cdot \text{undefined}$

<proof>

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**

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

<proof>

code-datatype *ff-of-rbt*

declare *[[code drop: finfun-const]]*

lemma *finfun-const-impl**[code]*: *finfun-const* $c = \text{ff-of-rbt } (c, \text{RBT.empty})$

<proof>

declare *[[code drop: finfun-apply]]*

lemma *finfun-apply-impl**[code]*: *finfun-apply* (*ff-of-rbt* (c, t)) $x = \text{def-option } c$
(*RBT.lookup* $t x$)

<proof>

declare *[[code drop: finfun-update]]*

lemma *finfun-update-impl**[code]*: *finfun-update* (*ff-of-rbt* (c, t)) $x y = \text{ff-of-rbt } (c,$
RBT.insert $x y t)$

<proof>

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>

lemma *finite-listset*:

assumes $\forall A \in \text{set } As. \text{finite } A$

shows *finite (listset As)*

<proof>

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>

lemma *card-listset*: **assumes** $\bigwedge A. A \in \text{set } As \implies \text{finite } A$

shows $\text{card } (\text{listset } As) = \text{prod-list } (\text{map } \text{card } As)$

<proof>

7.1 Deciding the nonemptiness 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

compute-nonempty-main *ne* *ls* = (let *rem-ls* = *filter* ($\lambda f. \text{snd } f \notin \text{ne}$) *ls* in

case partition ($\lambda ((-, \text{args}), -). \text{set } \text{args} \subseteq \text{ne}$) *rem-ls* of

(*new*, *rem*) \Rightarrow if *new* = [] then *ne* else *compute-nonempty-main* (*ne* \cup *set*

(*map snd new*)) *rem*)

<proof>

termination

<proof>

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

definition *compute-nonempty-sorts* :: $((f \times 't \text{ list}) \times 't) \text{ list} \Rightarrow 't \text{ set}$ **where**
compute-nonempty-sorts *Cs* = *compute-nonempty-main* {} *Cs*

lemma *compute-nonempty-sorts*:

assumes *distinct* (*map fst Cs*)

shows *compute-nonempty-sorts Cs* = $\{\tau. \neg \text{empty-sort } (\text{map-of } Cs) \tau\}$ (**is - =**
?NE)

<proof>

definition *decide-nonempty-sorts* :: $'t \text{ list} \Rightarrow ((f \times 't \text{ list}) \times 't) \text{ list} \Rightarrow 't \text{ option}$
where

decide-nonempty-sorts τs *Cs* = (*let ne* = *compute-nonempty-sorts Cs in*
find ($\lambda \tau. \tau \notin ne$) τs)

lemma *decide-nonempty-sorts*:

assumes *distinct* (*map fst Cs*)

shows *decide-nonempty-sorts* τs *Cs* = *None* $\implies \forall \tau \in \text{set } \tau s. \neg \text{empty-sort}$
 $(\text{map-of } Cs) \tau$

decide-nonempty-sorts τs *Cs* = *Some* $\tau \implies \tau \in \text{set } \tau s \wedge \text{empty-sort } (\text{map-of } Cs)$
 τ

<proof>

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* :: $'a \text{ list} \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow_f 'b) \Rightarrow ('a \Rightarrow_f 'b)$ **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* $\in \text{set } xs$ *then g*
x *else f* $\$ x$)

<proof>

definition *update-card-of-sort* :: $'\tau \Rightarrow (f \times 't \text{ list}) \text{ list} \Rightarrow ('t \Rightarrow_f \text{nat}) \Rightarrow \text{nat}$
where

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

function $compute\text{-}inf\text{-}card\text{-}main :: 't\ set \Rightarrow ('\tau \Rightarrow f\ nat) \Rightarrow ('t \times ('f \times 't\ list)\ list)$
 $list \Rightarrow 't\ set \times ('t \Rightarrow nat)$ **where**
 $compute\text{-}inf\text{-}card\text{-}main\ m\text{-}inf\ cards\ ls =$
 $let\ (fin,\ ls') =$
 $partition\ (\lambda\ (\tau,fs).\ \forall\ \tau s \in set\ (map\ snd\ fs).\ \forall\ \tau \in set\ \tau s.\ \tau \notin m\text{-}inf)\ ls$
 $in\ if\ fin = []\ then\ (m\text{-}inf,\ \lambda\ \tau.\ cards\ \$\ \tau)\ else$
 $let\ new = map\ fst\ fin;$
 $cards' = finfun\text{-}update\text{-}all\ new\ (\lambda\ \tau.\ update\text{-}card\text{-}of\text{-}sort\ \tau\ (the\ (map\ of\ ls\ \tau)))$
 $cards\ cards\ in$
 $compute\text{-}inf\text{-}card\text{-}main\ (m\text{-}inf - set\ new)\ cards'\ ls'$
 $\langle proof \rangle$

termination

$\langle proof \rangle$

lemma $compute\text{-}inf\text{-}card\text{-}main$: **fixes** $C :: ('f, 't)\ sig$

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

and Cs' : $set\ Cs' = set\ (concat\ (map\ ((\lambda\ (\tau, fs).\ map\ (\lambda\ f.\ (f, \tau))\ fs))\ Cs))$

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

and $dist$: $distinct\ (map\ fst\ Cs)\ distinct\ (map\ fst\ Cs')$

and $inhabited$: $\forall\ \tau\ fs.\ (\tau, fs) \in set\ Cs \longrightarrow set\ fs \neq \{\}$

and $\forall\ \tau.\ \tau \notin m\text{-}inf \longrightarrow bdd\text{-}above\ (size\ ' \{t.\ t : \tau\ in\ \mathcal{T}(C)\})$

and $set\ ls \subseteq set\ Cs$

and $fst\ ' (set\ Cs - set\ ls) \cap m\text{-}inf = \{\}$

and $m\text{-}inf \subseteq fst\ ' set\ ls$

and $\forall\ \tau.\ \tau \notin m\text{-}inf \longrightarrow cards\ \$\ \tau = card\text{-}of\text{-}sort\ C\ \tau \wedge finite\text{-}sort\ C\ \tau$

and $\forall\ \tau.\ \tau \in m\text{-}inf \longrightarrow cards\ \$\ \tau = 0$

shows $compute\text{-}inf\text{-}card\text{-}main\ m\text{-}inf\ cards\ ls = (\{\tau.\ \neg\ bdd\text{-}above\ (size\ ' \{t.\ t : \tau\ in\ \mathcal{T}(C)\})\},$

$\lambda\ \tau.\ card\text{-}of\text{-}sort\ C\ \tau)$

$\langle proof \rangle$

definition $compute\text{-}inf\text{-}card\text{-}sorts :: (('f \times 't\ list) \times 't)\ list \Rightarrow 't\ set \times ('t \Rightarrow nat)$

where

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

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

$Cs))$

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

lemma $finite\text{-}imp\text{-}size\text{-}bdd\text{-}above$: **assumes** $finite\ T$

shows $bdd\text{-}above\ (size\ ' T)$

$\langle proof \rangle$

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

shows *finite {t. $\exists \tau. \text{size } t \leq n \wedge t : \tau \text{ in } \mathcal{T}(F, V)$ }* (**is** *finite (?terms n)*)
 ⟨*proof*⟩

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

and *bdd-above (size ‘ {t. t : τ in $\mathcal{T}(F, V)$ })*
shows *finite {t. t : τ in $\mathcal{T}(F, V)$ }*
 ⟨*proof*⟩

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

shows *bdd-above (size ‘ {t. t : τ in $\mathcal{T}(F, V)$ }) = finite {t. t : τ in $\mathcal{T}(F, V)$ }*
 ⟨*proof*⟩

lemma *compute-inf-card-sorts*:

fixes *C :: ('f, 't)ssig*

assumes *C-Cs: C = map-of Cs*

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

and *dist: distinct (map fst Cs)*

and *result: compute-inf-card-sorts Cs = (unb, cards)*

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

cards = card-of-sort C (is - = ?cards)

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

⟨*proof*⟩

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 *min b (sum-list (map f xs)) = min b (sum-list (map g xs))*

⟨*proof*⟩

lemma *min-prod-min-left*: *min b (min b x * y :: nat) = min b (x * y)*

⟨*proof*⟩

lemma *min-prod-min-right*: *min b (x * min b y :: nat) = min b (x * y)*

⟨*proof*⟩

lemma *min-prod-min*: *min b (min b x * min b y :: nat) = min b (x * y)*

⟨*proof*⟩

lemma *min-prod-cong*: **assumes** *min b x1 = (min b x2 :: nat) min b y1 = min b y2*

shows *min b (x1 * y1) = min b (x2 * y2)*

⟨*proof*⟩

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 $\min b (\text{prod-list } (\text{map } f \text{ } xs)) = \min b (\text{prod-list } (\text{map } g \text{ } xs))$
 $\langle \text{proof} \rangle$

context

fixes $k :: \text{nat}$

begin

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

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

definition $\text{update-card-of-sort-bnd} :: 't \Rightarrow ('f \times 't \text{ list}) \text{ list} \Rightarrow ('t \Rightarrow f \text{ nat}) \Rightarrow \text{nat}$
where

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

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

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

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

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

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

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

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

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

definition $\text{compute-inf-card-sorts-bnd} :: (('f \times 't \text{ list}) \times 't) \text{ list} \Rightarrow 't \text{ set} \times ('t \Rightarrow \text{nat})$ **where**

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

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

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

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

$\langle \text{proof} \rangle$

end

end

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

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

shows $\text{unb} = \{\tau. \neg \text{bdd-above } (\text{size } ' \{t. t : \tau \text{ in } \mathcal{T}(C)\})\}$ **(is ?A)**

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

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

<proof>

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

lemma *compute-inf-sorts*:

assumes *arg-types-nonempty*: $\forall f \tau s \tau \tau'. f : \tau s \rightarrow \tau$ in *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

compute-inf-sorts Cs $= \{\tau. \neg \text{bdd-above } (\text{size } \{t. t : \tau \text{ in } \mathcal{T}(\text{map-of } Cs)\})\}$

compute-inf-sorts Cs $= \{\tau. \neg \text{finite-sort } (\text{map-of } Cs) \tau\}$

<proof>

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.