PAC Checker

Mathias Fleury and Daniela Kaufmann

July 21, 2020

Abstract

Abstract—Generating and checking proof certificates is important to increase the trust in automated reasoning tools. In recent years formal verification using computer algebra became more important and is heavily used in automated circuit verification. An existing proof format which covers algebraic reasoning and allows efficient proof checking is the practical algebraic calculus. In this development, we present the verified checker Pastèque that is obtained by synthesis via the Refinement Framework.

This is the formalization going with our FMCAD'20 tool presentation [1].

Contents

1

theory *Duplicate-Free-Multiset*
imports *Nested-Multisets-Ordinals.Multiset-More*
begin

1   Duplicate Free Multisets

Duplicate free multisets are isomorphic to finite sets, but it can be useful to reason about duplication to speak about intermediate execution steps in the refinements.

lemma *distinct-mset-remdups-mset-id*: ‹*distinct-mset C $\Longrightarrow$ remdups-mset C = C*›
  ⟨*proof*⟩

lemma *notin-add-mset-remdups-mset*:
  ‹*a $\notin\#$ A $\Longrightarrow$ add-mset a (remdups-mset A) = remdups-mset (add-mset a A)*›
  ⟨*proof*⟩

lemma *distinct-mset-image-mset*:
  ‹*distinct-mset (image-mset f (mset xs)) $\longleftrightarrow$ distinct (map f xs)*›
  ⟨*proof*⟩

lemma *distinct-image-mset-not-equal*:
  assumes

LL': ⟨$L \neq L'$⟩ **and**
$dist$: ⟨*distinct-mset* (*image-mset f M*)⟩ **and**
L: ⟨$L \in\# M$⟩ **and**
L': ⟨$L' \in\# M$⟩ **and**
$fLL'[simp]$: ⟨$f L = f L'$⟩
  shows ⟨*False*⟩
⟨*proof*⟩

lemma *distinct-mset-mono*: ⟨$D' \subseteq\# D \Longrightarrow$ *distinct-mset* $D \Longrightarrow$ *distinct-mset* D'⟩
  ⟨*proof*⟩

lemma *distinct-mset-mono-strict*: ⟨$D' \subset\# D \Longrightarrow$ *distinct-mset* $D \Longrightarrow$ *distinct-mset* D'⟩
  ⟨*proof*⟩

lemma *distinct-set-mset-eq-iff*:
  assumes ⟨*distinct-mset M*⟩ ⟨*distinct-mset N*⟩
  shows ⟨*set-mset* $M = set\text{-}mset$ $N \longleftrightarrow M = N$⟩
  ⟨*proof*⟩

lemma *distinct-mset-union2*:
  ⟨*distinct-mset* $(A + B) \Longrightarrow$ *distinct-mset* B⟩
  ⟨*proof*⟩

lemma *distinct-mset-mset-set*: ⟨*distinct-mset* (*mset-set A*)⟩
  ⟨*proof*⟩

lemma *distinct-mset-inter-remdups-mset*:
  assumes *dist*: ⟨*distinct-mset A*⟩
  shows ⟨$A \cap\#$ *remdups-mset* $B = A \cap\# B$⟩
⟨*proof*⟩

lemma *finite-mset-set-inter*:
  ⟨*finite* $A \Longrightarrow$ *finite* $B \Longrightarrow$ *mset-set* $(A \cap B) =$ *mset-set* $A \cap\#$ *mset-set* B⟩
  ⟨*proof*⟩

lemma *removeAll-notin*: ⟨$a \notin\# A \Longrightarrow$ *removeAll-mset* $a A = A$⟩
  ⟨*proof*⟩

lemma *same-mset-distinct-iff*:
  ⟨*mset* $M = mset$ $M' \Longrightarrow$ *distinct* $M \longleftrightarrow$ *distinct* M'⟩
  ⟨*proof*⟩

1.1   More Lists

lemma *in-set-conv-iff*:
  ⟨$x \in set$ (*take n xs*) $\longleftrightarrow$ ($\exists\, i < n.\ i < length\ xs \wedge xs\ !\ i = x$)⟩
  ⟨*proof*⟩

lemma *in-set-take-conv-nth*:
  ⟨$x \in set$ (*take n xs*) $\longleftrightarrow$ ($\exists\, m < min\ n$ (*length xs*). $xs\ !\ m = x$)⟩
  ⟨*proof*⟩

lemma *in-set-remove1D*:
  ⟨$a \in set$ (*remove1 x xs*) $\Longrightarrow a \in set\ xs$⟩
  ⟨*proof*⟩

1.2 Generic Multiset

lemma *mset-drop-upto*: ‹*mset (drop a N) = {#N!i. i ∈# mset-set {a..<length N}#}*›
‹*proof*›

1.3 Other

I believe this should be activated by default, as the set becomes much easier...

lemma *Collect-eq-comp'*: ‹ *{(x, y). P x y} O {(c, a). c = f a} = {(x, a). P x (f a)}*›
  ‹*proof*›

end

theory *WB-Sort*
  imports *Refine-Imperative-HOL.IICF HOL−Library.Rewrite Duplicate-Free-Multiset*
begin

This a complete copy-paste of the IsaFoL version because sharing is too hard.

Every element between *lo* and *hi* can be chosen as pivot element.

definition *choose-pivot* :: ‹(*'b ⇒ 'b ⇒ bool*) ⇒ (*'a ⇒ 'b*) ⇒ *'a list ⇒ nat ⇒ nat ⇒ nat nres*› **where**
  ‹*choose-pivot - - - lo hi = SPEC(λk. k ≥ lo ∧ k ≤ hi)*›

The element at index *p* partitions the subarray *lo..hi*. This means that every element

definition *isPartition-wrt* :: ‹(*'b ⇒ 'b ⇒ bool*) ⇒ *'b list ⇒ nat ⇒ nat ⇒ nat ⇒ bool*› **where**
  ‹*isPartition-wrt R xs lo hi p ≡ (∀ i. i ≥ lo ∧ i < p ⟶ R (xs!i) (xs!p)) ∧ (∀ j. j > p ∧ j ≤ hi ⟶ R (xs!p) (xs!j))*›

lemma *isPartition-wrtI*:
  ‹(⋀ i. ⟦i ≥ lo; i < p⟧ ⟹ R (xs!i) (xs!p)) ⟹ (⋀ j. ⟦j > p; j ≤ hi⟧ ⟹ R (xs!p) (xs!j)) ⟹ isPartition-wrt R xs lo hi p*›
  ‹*proof*›

definition *isPartition* :: ‹*'a :: order list ⇒ nat ⇒ nat ⇒ nat ⇒ bool*› **where**
  ‹*isPartition xs lo hi p ≡ isPartition-wrt (≤) xs lo hi p*›

abbreviation *isPartition-map* :: ‹(*'b ⇒ 'b ⇒ bool*) ⇒ (*'a ⇒ 'b*) ⇒ *'a list ⇒ nat ⇒ nat ⇒ nat ⇒ bool*› **where**
  ‹*isPartition-map R h xs i j k ≡ isPartition-wrt (λa b. R (h a) (h b)) xs i j k*›

lemma *isPartition-map-def'*:
  ‹*lo ≤ p ⟹ p ≤ hi ⟹ hi < length xs ⟹ isPartition-map R h xs lo hi p = isPartition-wrt R (map h xs) lo hi p*›
  ‹*proof*›

Example: 6 is the pivot element (with index 4); *7::'a* is equal to the *length xs − 1*.

lemma ‹*isPartition [0,5,3,4,6,9,8,10::nat] 0 7 4*›
  ‹*proof*›

definition *sublist* :: ‹*'a list ⇒ nat ⇒ nat ⇒ 'a list*› **where**
  ‹*sublist xs i j ≡ take (Suc j − i) (drop i xs)*›

lemma *take-Suc0*:
  $l \neq [] \implies take\ (Suc\ 0)\ l = [l!0]$
  $0 < length\ l \implies take\ (Suc\ 0)\ l = [l!0]$
  $Suc\ n \leq length\ l \implies take\ (Suc\ 0)\ l = [l!0]$
  ⟨*proof*⟩

lemma *sublist-single*: ⟨$i < length\ xs \implies sublist\ xs\ i\ i = [xs!i]$⟩
  ⟨*proof*⟩

lemma *insert-eq*: ⟨$insert\ a\ b = b \cup \{a\}$⟩
  ⟨*proof*⟩

lemma *sublist-nth*: ⟨$[\![lo \leq hi;\ hi < length\ xs;\ k+lo \leq hi]\!] \implies (sublist\ xs\ lo\ hi)!k = xs!(lo+k)$⟩
  ⟨*proof*⟩

lemma *sublist-length*: ⟨$[\![i \leq j;\ j < length\ xs]\!] \implies length\ (sublist\ xs\ i\ j) = 1 + j - i$⟩
  ⟨*proof*⟩

lemma *sublist-not-empty*: ⟨$[\![i \leq j;\ j < length\ xs;\ xs \neq [\,]]\!] \implies sublist\ xs\ i\ j \neq [\,]$⟩
  ⟨*proof*⟩

lemma *sublist-app*: ⟨$[\![i1 \leq i2;\ i2 \leq i3]\!] \implies sublist\ xs\ i1\ i2\ @\ sublist\ xs\ (Suc\ i2)\ i3 = sublist\ xs\ i1\ i3$⟩
  ⟨*proof*⟩

definition *sorted-sublist-wrt* :: ⟨$('b \Rightarrow 'b \Rightarrow bool) \Rightarrow 'b\ list \Rightarrow nat \Rightarrow nat \Rightarrow bool$⟩ **where**
  ⟨*sorted-sublist-wrt R xs lo hi = sorted-wrt R (sublist xs lo hi)*⟩

definition *sorted-sublist* :: ⟨$'a\ ::\ linorder\ list \Rightarrow nat \Rightarrow nat \Rightarrow bool$⟩ **where**
  ⟨*sorted-sublist xs lo hi = sorted-sublist-wrt* $(\leq)$ *xs lo hi*⟩

abbreviation *sorted-sublist-map* :: ⟨$('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow nat \Rightarrow nat \Rightarrow bool$⟩
where
  ⟨*sorted-sublist-map R h xs lo hi* $\equiv$ *sorted-sublist-wrt* $(\lambda a\ b.\ R\ (h\ a)\ (h\ b))$ *xs lo hi*⟩

lemma *sorted-sublist-map-def′*:
  ⟨$lo < length\ xs \implies sorted\text{-}sublist\text{-}map\ R\ h\ xs\ lo\ hi \equiv sorted\text{-}sublist\text{-}wrt\ R\ (map\ h\ xs)\ lo\ hi$⟩
  ⟨*proof*⟩

lemma *sorted-sublist-wrt-refl*: ⟨$i < length\ xs \implies sorted\text{-}sublist\text{-}wrt\ R\ xs\ i\ i$⟩
  ⟨*proof*⟩

lemma *sorted-sublist-refl*: ⟨$i < length\ xs \implies sorted\text{-}sublist\ xs\ i\ i$⟩
  ⟨*proof*⟩

lemma *sublist-map*: ⟨$sublist\ (map\ f\ xs)\ i\ j = map\ f\ (sublist\ xs\ i\ j)$⟩
  ⟨*proof*⟩

lemma *take-set*: ⟨$j \leq length\ xs \implies x \in set\ (take\ j\ xs) \equiv (\exists\ k.\ k < j \wedge xs!k = x)$⟩
  ⟨*proof*⟩

lemma *drop-set*: ⟨$j \leq length\ xs \implies x \in set\ (drop\ j\ xs) \equiv (\exists k.\ j \leq k \wedge k < length\ xs \wedge xs!k = x)$⟩
  ⟨*proof*⟩

lemma *sublist-el*: ‹$i \leq j \Longrightarrow j < length\ xs \Longrightarrow x \in set\ (sublist\ xs\ i\ j) \equiv (\exists\ k.\ k < Suc\ j{-}i \land xs!(i{+}k){=}x)$›
  ⟨*proof*⟩

lemma *sublist-el'*: ‹$i \leq j \Longrightarrow j < length\ xs \Longrightarrow x \in set\ (sublist\ xs\ i\ j) \equiv (\exists\ k.\ i{\leq}k{\land}k{\leq}j \land xs!k{=}x)$›
  ⟨*proof*⟩

lemma *sublist-lt*: ‹$hi < lo \Longrightarrow sublist\ xs\ lo\ hi = []$›
  ⟨*proof*⟩

lemma *nat-le-eq-or-lt*: ‹$(a :: nat) \leq b = (a = b \lor a < b)$›
  ⟨*proof*⟩

lemma *sorted-sublist-wrt-le*: ‹$hi \leq lo \Longrightarrow hi < length\ xs \Longrightarrow sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ hi$›
  ⟨*proof*⟩

Elements in a sorted sublists are actually sorted

lemma *sorted-sublist-wrt-nth-le*:
  assumes ‹*sorted-sublist-wrt R xs lo hi*› **and** ‹$lo \leq hi$› **and** ‹$hi < length\ xs$› **and**
    ‹$lo \leq i$› **and** ‹$i < j$› **and** ‹$j \leq hi$›
  shows ‹$R\ (xs!i)\ (xs!j)$›
⟨*proof*⟩

We can make the assumption $i < j$ weaker if we have a reflexivie relation.

lemma *sorted-sublist-wrt-nth-le'*:
  assumes *ref*: ‹$\bigwedge\ x.\ R\ x\ x$›
    and ‹*sorted-sublist-wrt R xs lo hi*› **and** ‹$lo \leq hi$› **and** ‹$hi < length\ xs$›
    and ‹$lo \leq i$› **and** ‹$i \leq j$› **and** ‹$j \leq hi$›
  shows ‹$R\ (xs!i)\ (xs!j)$›
⟨*proof*⟩

lemma *sorted-sublist-le*: ‹$hi \leq lo \Longrightarrow hi < length\ xs \Longrightarrow sorted\text{-}sublist\ xs\ lo\ hi$›
  ⟨*proof*⟩

lemma *sorted-sublist-map-le*: ‹$hi \leq lo \Longrightarrow hi < length\ xs \Longrightarrow sorted\text{-}sublist\text{-}map\ R\ h\ xs\ lo\ hi$›
  ⟨*proof*⟩

lemma *sublist-cons*: ‹$lo < hi \Longrightarrow hi < length\ xs \Longrightarrow sublist\ xs\ lo\ hi = xs!lo\ \#\ sublist\ xs\ (Suc\ lo)\ hi$›
  ⟨*proof*⟩

lemma *sorted-sublist-wrt-cons'*:
  ‹$sorted\text{-}sublist\text{-}wrt\ R\ xs\ (lo{+}1)\ hi \Longrightarrow lo \leq hi \Longrightarrow hi < length\ xs \Longrightarrow (\forall j.\ lo{<}j{\land}j{\leq}hi \longrightarrow R\ (xs!lo)$
$(xs!j)) \Longrightarrow sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ hi$›
  ⟨*proof*⟩

lemma *sorted-sublist-wrt-cons*:
  assumes *trans*: ‹$(\bigwedge\ x\ y\ z.\ [\![R\ x\ y;\ R\ y\ z]\!] \Longrightarrow R\ x\ z)$› **and**

‹*sorted-sublist-wrt R xs (lo+1) hi*› **and**
‹*lo ≤ hi*› **and** ‹*hi < length xs* › **and** ‹*R (xs!lo) (xs!(lo+1))*›
shows ‹*sorted-sublist-wrt R xs lo hi*›
⟨*proof*⟩

lemma *sorted-sublist-map-cons*:
‹(⋀ *x y z*. ⟦*R (h x) (h y)*; *R (h y) (h z)*⟧ ⟹ *R (h x) (h z)*) ⟹
sorted-sublist-map R h xs (lo+1) hi ⟹ *lo ≤ hi* ⟹ *hi < length xs* ⟹ *R (h (xs!lo)) (h (xs!(lo+1)))*
⟹ *sorted-sublist-map R h xs lo hi*›
⟨*proof*⟩

lemma *sublist-snoc*: ‹*lo < hi* ⟹ *hi < length xs* ⟹ *sublist xs lo hi = sublist xs lo (hi−1)* @ *[xs!hi]*›
⟨*proof*⟩

lemma *sorted-sublist-wrt-snoc′*:
‹*sorted-sublist-wrt R xs lo (hi−1)* ⟹ *lo ≤ hi* ⟹ *hi < length xs* ⟹ (∀ *j*. *lo≤j∧j<hi* ⟶ *R (xs!j)*
(xs!hi)) ⟹ *sorted-sublist-wrt R xs lo hi*›
⟨*proof*⟩

lemma *sorted-sublist-wrt-snoc*:
  assumes *trans*: ‹(⋀ *x y z*. ⟦*R x y*; *R y z*⟧ ⟹ *R x z*)› **and**
    ‹*sorted-sublist-wrt R xs lo (hi−1)*› **and**
    ‹*lo ≤ hi*› **and** ‹*hi < length xs*› **and** ‹*(R (xs!(hi−1)) (xs!hi))*›
  shows ‹*sorted-sublist-wrt R xs lo hi*›
⟨*proof*⟩

lemma *sublist-split*: ‹*lo ≤ hi* ⟹ *lo < p* ⟹ *p < hi* ⟹ *hi < length xs* ⟹ *sublist xs lo p* @ *sublist xs*
(p+1) hi = sublist xs lo hi›
⟨*proof*⟩

lemma *sublist-split-part*: ‹*lo ≤ hi* ⟹ *lo < p* ⟹ *p < hi* ⟹ *hi < length xs* ⟹ *sublist xs lo (p−1)* @
xs!p # sublist xs (p+1) hi = sublist xs lo hi›
⟨*proof*⟩

A property for partitions (we always assume that *R* is transitive.

lemma *isPartition-wrt-trans*:
‹(⋀ *x y z*. ⟦*R x y*; *R y z*⟧ ⟹ *R x z*) ⟹
isPartition-wrt R xs lo hi p ⟹
(∀ *i j*. *lo ≤ i ∧ i < p ∧ p < j ∧ j ≤ hi* ⟶ *R (xs!i) (xs!j)*)›
⟨*proof*⟩

lemma *isPartition-map-trans*:
‹(⋀ *x y z*. ⟦*R (h x) (h y)*; *R (h y) (h z)*⟧ ⟹ *R (h x) (h z)*) ⟹
hi < length xs ⟹
isPartition-map R h xs lo hi p ⟹
(∀ *i j*. *lo ≤ i ∧ i < p ∧ p < j ∧ j ≤ hi* ⟶ *R (h (xs!i)) (h (xs!j))*)›
⟨*proof*⟩

lemma *merge-sorted-wrt-partitions-between′*:
  ‹*lo ≤ hi* ⟹ *lo < p* ⟹ *p < hi* ⟹ *hi < length xs* ⟹
    isPartition-wrt R xs lo hi p ⟹
    sorted-sublist-wrt R xs lo (p−1) ⟹ *sorted-sublist-wrt R xs (p+1) hi* ⟹

7

$(\forall\, i\ j.\ lo \le i \wedge i < p \wedge p < j \wedge j \le hi \longrightarrow R\ (xs!i)\ (xs!j)) \Longrightarrow$
sorted-sublist-wrt R xs lo hi⟩
⟨*proof*⟩

lemma *merge-sorted-wrt-partitions-between*:
⟨$(\bigwedge\, x\ y\ z.\ [\![R\ x\ y;\ R\ y\ z]\!] \Longrightarrow R\ x\ z) \Longrightarrow$
isPartition-wrt R xs lo hi p $\Longrightarrow$
sorted-sublist-wrt R xs lo (p−1) $\Longrightarrow$ *sorted-sublist-wrt R xs (p+1) hi* $\Longrightarrow$
$lo \le hi \Longrightarrow hi < length\ xs \Longrightarrow lo < p \Longrightarrow p < hi \Longrightarrow hi < length\ xs \Longrightarrow$
sorted-sublist-wrt R xs lo hi⟩
⟨*proof*⟩

The main theorem to merge sorted lists

lemma *merge-sorted-wrt-partitions*:
⟨*isPartition-wrt R xs lo hi p* $\Longrightarrow$
sorted-sublist-wrt R xs lo (p − Suc 0) $\Longrightarrow$ *sorted-sublist-wrt R xs (Suc p) hi* $\Longrightarrow$
$lo \le hi \Longrightarrow lo \le p \Longrightarrow p \le hi \Longrightarrow hi < length\ xs \Longrightarrow$
$(\forall\, i\ j.\ lo \le i \wedge i < p \wedge p < j \wedge j \le hi \longrightarrow R\ (xs!i)\ (xs!j)) \Longrightarrow$
sorted-sublist-wrt R xs lo hi⟩
⟨*proof*⟩

theorem *merge-sorted-map-partitions*:
⟨$(\bigwedge\, x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)) \Longrightarrow$
isPartition-map R h xs lo hi p $\Longrightarrow$
sorted-sublist-map R h xs lo (p − Suc 0) $\Longrightarrow$ *sorted-sublist-map R h xs (Suc p) hi* $\Longrightarrow$
$lo \le hi \Longrightarrow lo \le p \Longrightarrow p \le hi \Longrightarrow hi < length\ xs \Longrightarrow$
sorted-sublist-map R h xs lo hi⟩
⟨*proof*⟩

lemma *partition-wrt-extend*:
⟨*isPartition-wrt R xs lo′ hi′ p* $\Longrightarrow$
$hi < length\ xs \Longrightarrow$
$lo \le lo′ \Longrightarrow lo′ \le hi \Longrightarrow hi′ \le hi \Longrightarrow$
$lo′ \le p \Longrightarrow p \le hi′ \Longrightarrow$
$(\bigwedge\, i.\ lo \le i \Longrightarrow i < lo′ \Longrightarrow R\ (xs!i)\ (xs!p)) \Longrightarrow$
$(\bigwedge\, j.\ hi′ < j \Longrightarrow j \le hi \Longrightarrow R\ (xs!p)\ (xs!j)) \Longrightarrow$
isPartition-wrt R xs lo hi p⟩
⟨*proof*⟩

lemma *partition-map-extend*:
⟨*isPartition-map R h xs lo′ hi′ p* $\Longrightarrow$
$hi < length\ xs \Longrightarrow$
$lo \le lo′ \Longrightarrow lo′ \le hi \Longrightarrow hi′ \le hi \Longrightarrow$
$lo′ \le p \Longrightarrow p \le hi′ \Longrightarrow$
$(\bigwedge\, i.\ lo \le i \Longrightarrow i < lo′ \Longrightarrow R\ (h\ (xs!i))\ (h\ (xs!p))) \Longrightarrow$
$(\bigwedge\, j.\ hi′ < j \Longrightarrow j \le hi \Longrightarrow R\ (h\ (xs!p))\ (h\ (xs!j))) \Longrightarrow$
isPartition-map R h xs lo hi p⟩
⟨*proof*⟩

lemma *isPartition-empty*:
⟨$(\bigwedge\, j.\ [\![lo < j;\ j \le hi]\!] \Longrightarrow R\ (xs\ !\ lo)\ (xs\ !\ j)) \Longrightarrow$
isPartition-wrt R xs lo hi lo⟩
⟨*proof*⟩

lemma *take-ext*:
  ⟨(∀ i<k. xs′!i=xs!i) ⟹
  k < length xs ⟹ k < length xs′ ⟹
  take k xs′ = take k xs⟩
  ⟨proof⟩

lemma *drop-ext′*:
  ⟨(∀ i. i≥k ∧ i<length xs ⟶ xs′!i=xs!i) ⟹
  0<k ⟹ xs≠[] ⟹ — These corner cases will be dealt with in the next lemma
  length xs′=length xs ⟹
  drop k xs′ = drop k xs⟩
  ⟨proof⟩

lemma *drop-ext*:
⟨(∀ i. i≥k ∧ i<length xs ⟶ xs′!i=xs!i) ⟹
  length xs′=length xs ⟹
  drop k xs′ = drop k xs⟩
  ⟨proof⟩

lemma *sublist-ext′*:
  ⟨(∀ i. lo≤i∧i≤hi ⟶ xs′!i=xs!i) ⟹
  length xs′ = length xs ⟹
  lo ≤ hi ⟹ Suc hi < length xs ⟹
  sublist xs′ lo hi = sublist xs lo hi⟩
  ⟨proof⟩

lemma *lt-Suc*: ⟨(a < b) = (Suc a = b ∨ Suc a < b)⟩
  ⟨proof⟩

lemma *sublist-until-end-eq-drop*: ⟨Suc hi = length xs ⟹ sublist xs lo hi = drop lo xs⟩
  ⟨proof⟩

lemma *sublist-ext*:
  ⟨(∀ i. lo≤i∧i≤hi ⟶ xs′!i=xs!i) ⟹
  length xs′ = length xs ⟹
  lo ≤ hi ⟹ hi < length xs ⟹
  sublist xs′ lo hi = sublist xs lo hi⟩
  ⟨proof⟩

lemma *sorted-wrt-lower-sublist-still-sorted*:
  assumes ⟨sorted-sublist-wrt R xs lo (lo′ − Suc 0)⟩ **and**
    ⟨lo ≤ lo′⟩ **and** ⟨lo′ < length xs⟩ **and**
    ⟨(∀ i. lo≤i∧i<lo′ ⟶ xs′!i=xs!i)⟩ **and** ⟨length xs′ = length xs⟩
  shows ⟨sorted-sublist-wrt R xs′ lo (lo′ − Suc 0)⟩
⟨proof⟩

lemma *sorted-map-lower-sublist-still-sorted*:
  assumes ⟨sorted-sublist-map R h xs lo (lo′ − Suc 0)⟩ **and**
    ⟨lo ≤ lo′⟩ **and** ⟨lo′ < length xs⟩ **and**
    ⟨(∀ i. lo≤i∧i<lo′ ⟶ xs′!i=xs!i)⟩ **and** ⟨length xs′ = length xs⟩

shows ⟨*sorted-sublist-map R h xs′ lo (lo′ − Suc 0)*⟩
⟨*proof*⟩

lemma *sorted-wrt-upper-sublist-still-sorted*:
  assumes ⟨*sorted-sublist-wrt R xs (hi′+1) hi*⟩ **and**
    ⟨*lo ≤ lo′*⟩ **and** ⟨*hi < length xs*⟩ **and**
    ⟨∀ *j. hi′<j∧j≤hi ⟶ xs′!j=xs!j*⟩ **and** ⟨*length xs′ = length xs*⟩
  shows ⟨*sorted-sublist-wrt R xs′ (hi′+1) hi*⟩
⟨*proof*⟩

lemma *sorted-map-upper-sublist-still-sorted*:
  assumes ⟨*sorted-sublist-map R h xs (hi′+1) hi*⟩ **and**
    ⟨*lo ≤ lo′*⟩ **and** ⟨*hi < length xs*⟩ **and**
    ⟨∀ *j. hi′<j∧j≤hi ⟶ xs′!j=xs!j*⟩ **and** ⟨*length xs′ = length xs*⟩
  shows ⟨*sorted-sublist-map R h xs′ (hi′+1) hi*⟩
  ⟨*proof*⟩

The specification of the partition function

definition *partition-spec* :: ⟨(′*b* ⇒ ′*b* ⇒ *bool*) ⇒ (′*a* ⇒ ′*b*) ⇒ ′*a list* ⇒ *nat* ⇒ *nat* ⇒ ′*a list* ⇒ *nat* ⇒ *bool*⟩ **where**
  ⟨*partition-spec R h xs lo hi xs′ p* ≡
    mset xs′ = mset xs ∧ — The list is a permutation
    isPartition-map R h xs′ lo hi p ∧ — We have a valid partition on the resulting list
    lo ≤ p ∧ *p ≤ hi* ∧ — The partition index is in bounds
    (∀ *i. i<lo ⟶ xs′!i=xs!i*) ∧ (∀ *i. hi<i∧i<length xs′ ⟶ xs′!i=xs!i*)⟩ — Everything else is unchanged.

lemma *in-set-take-conv-nth*:
  ⟨*x* ∈ *set (take n xs)* ⟷ (∃ *m<min n (length xs). xs ! m = x*)⟩
  ⟨*proof*⟩

lemma *mset-drop-upto*: ⟨*mset (drop a N) = {#N!i. i ∈# mset-set {a..<length N}#}*⟩
⟨*proof*⟩

lemma *mathias*:
  assumes
      Perm: ⟨*mset xs′ = mset xs*⟩
    and *I*: ⟨*lo≤i*⟩ ⟨*i≤hi*⟩ ⟨*xs′!i=x*⟩
    and *Bounds*: ⟨*hi < length xs*⟩
    and *Fix*: ⟨⋀ *i. i<lo ⟹ xs′!i = xs!i*⟩ ⟨⋀ *j.* ⟦*hi<j; j<length xs*⟧ ⟹ *xs′!j = xs!j*⟩
  shows ⟨∃ *j. lo≤j∧j≤hi ∧ xs!j = x*⟩
⟨*proof*⟩

If we fix the left and right rest of two permutated lists, then the sublists are also permutations.

But we only need that the sets are equal.

lemma *mset-sublist-incl*:
  assumes *Perm*: ⟨*mset xs′ = mset xs*⟩
    and *Fix*: ⟨⋀ *i. i<lo ⟹ xs′!i = xs!i*⟩ ⟨⋀ *j.* ⟦*hi<j; j<length xs*⟧ ⟹ *xs′!j = xs!j*⟩
    and *bounds*: ⟨*lo ≤ hi*⟩ ⟨*hi < length xs*⟩
  shows ⟨*set (sublist xs′ lo hi) ⊆ set (sublist xs lo hi)*⟩
⟨*proof*⟩

lemma *mset-sublist-eq*:

assumes ⟨*mset xs′ = mset xs*⟩
  and ⟨⋀ *i. i<lo* ⟹ *xs′!i = xs!i*⟩
  and ⟨⋀ *j.* ⟦*hi<j; j<length xs*⟧ ⟹ *xs′!j = xs!j*⟩
  and *bounds*: ⟨*lo ≤ hi*⟩ ⟨*hi < length xs*⟩
 shows ⟨*set (sublist xs′ lo hi) = set (sublist xs lo hi)*⟩
⟨*proof*⟩

Our abstract recursive quicksort procedure. We abstract over a partition procedure.

definition *quicksort* :: ⟨($'b$ ⟹ $'b$ ⟹ *bool*) ⟹ ($'a$ ⟹ $'b$) ⟹ *nat × nat × $'a$ list* ⟹ $'a$ *list nres*⟩ **where**
⟨*quicksort R h = (λ(lo,hi,xs0). do {*
  RECT (λf (lo,hi,xs). do {
    ASSERT(lo ≤ hi ∧ hi < length xs ∧ mset xs = mset xs0); — Premise for a partition function
    (xs, p) ← SPEC(uncurry (partition-spec R h xs lo hi)); — Abstract partition function
    ASSERT(mset xs = mset xs0);
    xs ← (if p−1≤lo then RETURN xs else f (lo, p−1, xs));
    ASSERT(mset xs = mset xs0);
    if hi≤p+1 then RETURN xs else f (p+1, hi, xs)
  }) (lo,hi,xs0)
 })⟩

As premise for quicksor, we only need that the indices are ok.

definition *quicksort-pre* :: ⟨($'b$ ⟹ $'b$ ⟹ *bool*) ⟹ ($'a$ ⟹ $'b$) ⟹ $'a$ *list* ⟹ *nat* ⟹ *nat* ⟹ $'a$ *list* ⟹ *bool*⟩
where
 ⟨*quicksort-pre R h xs0 lo hi xs ≡ lo ≤ hi ∧ hi < length xs ∧ mset xs = mset xs0*⟩

definition *quicksort-post* :: ⟨($'b$ ⟹ $'b$ ⟹ *bool*) ⟹ ($'a$ ⟹ $'b$) ⟹ *nat* ⟹ *nat* ⟹ $'a$ *list* ⟹ $'a$ *list* ⟹ *bool*⟩
where
 ⟨*quicksort-post R h lo hi xs xs′ ≡*
  mset xs′ = mset xs ∧
  sorted-sublist-map R h xs′ lo hi ∧
  (∀ *i. i<lo* ⟶ *xs′!i = xs!i*) ∧
  (∀ *j. hi<j∧j<length xs* ⟶ *xs′!j = xs!j*)⟩

Convert Pure to HOL

lemma *quicksort-postI*:
 ⟨⟦*mset xs′ = mset xs; sorted-sublist-map R h xs′ lo hi;* (⋀ *i.* ⟦*i<lo*⟧ ⟹ *xs′!i = xs!i*); (⋀ *j.* ⟦*hi<j; j<length xs*⟧ ⟹ *xs′!j = xs!j*)⟧ ⟹ *quicksort-post R h lo hi xs xs′*⟩
 ⟨*proof*⟩

The first case for the correctness proof of (abstract) quicksort: We assume that we called the partition function, and we have $p - (1::'a) \le lo$ and $hi \le p + (1::'a)$.

lemma *quicksort-correct-case1*:
 assumes *trans*: ⟨⋀ *x y z.* ⟦*R (h x) (h y); R (h y) (h z)*⟧ ⟹ *R (h x) (h z)*⟩ **and** *lin*: ⟨⋀*x y. x ≠ y* ⟹ *R (h x) (h y) ∨ R (h y) (h x)*⟩
  and *pre*: ⟨*quicksort-pre R h xs0 lo hi xs*⟩
  and *part*: ⟨*partition-spec R h xs lo hi xs′ p*⟩
  and *ifs*: ⟨*p−1 ≤ lo*⟩ ⟨*hi ≤ p+1*⟩
 shows ⟨*quicksort-post R h lo hi xs xs′*⟩
⟨*proof*⟩

In the second case, we have to show that the precondition still holds for (p+1, hi, x′) after the partition.

lemma *quicksort-correct-case2*:
 assumes

      pre: ‹*quicksort-pre R h xs0 lo hi xs*›
  and *part*: ‹*partition-spec R h xs lo hi xs′ p*›
  and *ifs*: ‹¬ *hi* ≤ *p* + *1*›
 shows ‹*quicksort-pre R h xs0 (Suc p) hi xs′*›
⟨*proof*⟩

lemma *quicksort-post-set*:
 assumes ‹*quicksort-post R h lo hi xs xs′*›
  and *bounds*: ‹*lo* ≤ *hi*› ‹*hi* < *length xs*›
 shows ‹*set (sublist xs′ lo hi) = set (sublist xs lo hi)*›
⟨*proof*⟩

In the third case, we have run quicksort recursively on (p+1, hi, xs') after the partition, with hi<=p+1 and p-1<=lo.

lemma *quicksort-correct-case3*:
 assumes *trans*: ‹⋀ *x y z*. ⟦*R (h x) (h y)*; *R (h y) (h z)*⟧ ⟹ *R (h x) (h z)*› **and** *lin*: ‹⋀*x y*. *x* ≠ *y* ⟹ *R (h x) (h y)* ∨ *R (h y) (h x)*›
  and *pre*: ‹*quicksort-pre R h xs0 lo hi xs*›
  and *part*: ‹*partition-spec R h xs lo hi xs′ p*›
  and *ifs*: ‹*p* − *Suc 0* ≤ *lo*› ‹¬ *hi* ≤ *Suc p*›
  and *IH1′*: ‹*quicksort-post R h (Suc p) hi xs′ xs′′*›
 shows ‹*quicksort-post R h lo hi xs xs′′*›
⟨*proof*⟩

In the 4th case, we have to show that the premise holds for (*lo, p* − (*1*::′*b*), *xs′*), in case ¬ *p* − (*1*::′*a*) ≤ *lo*

Analogous to case 2.

lemma *quicksort-correct-case4*:
 assumes
    pre: ‹*quicksort-pre R h xs0 lo hi xs*›
  and *part*: ‹*partition-spec R h xs lo hi xs′ p*›
  and *ifs*: ‹¬ *p* − *Suc 0* ≤ *lo* ›
 shows ‹*quicksort-pre R h xs0 lo (p−Suc 0) xs′*›
⟨*proof*⟩

In the 5th case, we have run quicksort recursively on (lo, p-1, xs').

lemma *quicksort-correct-case5*:
 assumes *trans*: ‹⋀ *x y z*. ⟦*R (h x) (h y)*; *R (h y) (h z)*⟧ ⟹ *R (h x) (h z)*› **and** *lin*: ‹⋀*x y*. *x* ≠ *y* ⟹ *R (h x) (h y)* ∨ *R (h y) (h x)*›
  and *pre*: ‹*quicksort-pre R h xs0 lo hi xs*›
  and *part*: ‹*partition-spec R h xs lo hi xs′ p*›
  and *ifs*: ‹¬ *p* − *Suc 0* ≤ *lo*› ‹*hi* ≤ *Suc p*›
  and *IH1′*: ‹*quicksort-post R h lo (p* − *Suc 0) xs′ xs′′*›
 shows ‹*quicksort-post R h lo hi xs xs′′*›
⟨*proof*⟩

In the 6th case, we have run quicksort recursively on (lo, p-1, xs'). We show the precondition on the second call on (p+1, hi, xs")

lemma *quicksort-correct-case6*:
 assumes
    pre: ‹*quicksort-pre R h xs0 lo hi xs*›

    and *part*: ‹*partition-spec R h xs lo hi xs′ p*›
    and *ifs*: ‹¬ *p* − *Suc 0* ≤ *lo*› ‹¬ *hi* ≤ *Suc p*›
    and *IH1*: ‹*quicksort-post R h lo (p* − *Suc 0) xs′ xs″*›
  shows ‹*quicksort-pre R h xs0 (Suc p) hi xs″*›
⟨*proof*⟩

In the 7th (and last) case, we have run quicksort recursively on (lo, p-1, xs'). We show the postcondition on the second call on (p+1, hi, xs")

lemma *quicksort-correct-case7*:
  assumes *trans*: ‹⋀ *x y z*. ⟦*R (h x) (h y)*; *R (h y) (h z)*⟧ ⟹ *R (h x) (h z)*› **and** *lin*: ‹⋀*x y*. *x* ≠ *y* ⟹ *R (h x) (h y)* ∨ *R (h y) (h x)*›
    and *pre*: ‹*quicksort-pre R h xs0 lo hi xs*›
    and *part*: ‹*partition-spec R h xs lo hi xs′ p*›
    and *ifs*: ‹¬ *p* − *Suc 0* ≤ *lo*› ‹¬ *hi* ≤ *Suc p*›
    and *IH1′*: ‹*quicksort-post R h lo (p* − *Suc 0) xs′ xs″*›
    and *IH2′*: ‹*quicksort-post R h (Suc p) hi xs″ xs‴*›
  shows ‹*quicksort-post R h lo hi xs xs‴*›
⟨*proof*⟩

We can now show the correctness of the abstract quicksort procedure, using the refinement framework and the above case lemmas.

lemma *quicksort-correct*:
  assumes *trans*: ‹⋀ *x y z*. ⟦*R (h x) (h y)*; *R (h y) (h z)*⟧ ⟹ *R (h x) (h z)*› **and** *lin*: ‹⋀*x y*. *x* ≠ *y* ⟹ *R (h x) (h y)* ∨ *R (h y) (h x)*›
    and *Pre*: ‹*lo0* ≤ *hi0*› ‹*hi0* < *length xs0*›
  shows ‹*quicksort R h (lo0,hi0,xs0)* ≤ ⇓ *Id (SPEC(λxs. quicksort-post R h lo0 hi0 xs0 xs))*›
⟨*proof*⟩

definition *partition-main-inv* :: ‹(′*b* ⇒ ′*b* ⇒ *bool*) ⇒ (′*a* ⇒ ′*b*) ⇒ *nat* ⇒ *nat* ⇒ ′*a list* ⇒ (*nat*×*nat*×′*a list*) ⇒ *bool*› **where**
  ‹*partition-main-inv R h lo hi xs0 p* ≡
    case p of (i,j,xs) ⇒
    j < *length xs* ∧ *j* ≤ *hi* ∧ *i* < *length xs* ∧ *lo* ≤ *i* ∧ *i* ≤ *j* ∧ *mset xs* = *mset xs0* ∧
    (∀ *k*. *k* ≥ *lo* ∧ *k* < *i* ⟶ *R (h (xs!k)) (h (xs!hi))*) ∧ — All elements from *lo* to *i* − (*1*::′*c*) are smaller than the pivot
    (∀ *k*. *k* ≥ *i* ∧ *k* < *j* ⟶ *R (h (xs!hi)) (h (xs!k))*) ∧ — All elements from *i* to *j* − (*1*::′*c*) are greater than the pivot
    (∀ *k*. *k* < *lo* ⟶ *xs!k* = *xs0!k*) ∧ — Everything below *lo* is unchanged
    (∀ *k*. *k* ≥ *j* ∧ *k* < *length xs* ⟶ *xs!k* = *xs0!k*) — All elements from *j* are unchanged (including everyting above *hi*)
  ›

The main part of the partition function. The pivot is assumed to be the last element. This is exactly the "Lomuto partition scheme" partition function from Wikipedia.

definition *partition-main* :: ‹(′*b* ⇒ ′*b* ⇒ *bool*) ⇒ (′*a* ⇒ ′*b*) ⇒ *nat* ⇒ *nat* ⇒ ′*a list* ⇒ (′*a list* × *nat*) *nres*› **where**
  ‹*partition-main R h lo hi xs0* = *do* {
    ASSERT(*hi* < *length xs0*);
    pivot ← *RETURN (h (xs0 ! hi))*;
    (*i,j,xs*) ← *WHILE$_T$$^{partition-main-inv R h lo hi xs0}$* — We loop from *j* = *lo* to *j* = *hi* − (*1*::′*c*).

$(\lambda(i,j,xs).\ j\ <\ hi)$
$(\lambda(i,j,xs).\ do\ \{$
 $ASSERT(i\ <\ length\ xs\ \wedge\ j\ <\ length\ xs);$
 $if\ R\ (h\ (xs!j))\ pivot$
 $then\ RETURN\ (i{+}1,\ j{+}1,\ swap\ xs\ i\ j)$
 $else\ RETURN\ (i,\quad j{+}1,\ xs)$
$\})$
$(lo,\ lo,\ xs0);$ — i and j are both initialized to lo
$ASSERT(i\ <\ length\ xs\ \wedge\ j\ =\ hi\ \wedge\ lo\ \leq\ i\ \wedge\ hi\ <\ length\ xs\ \wedge\ mset\ xs\ =\ mset\ xs0);$
$RETURN\ (swap\ xs\ i\ hi,\ i)$
$\}\rangle$

lemma *partition-main-correct*:
 assumes *bounds*: ⟨*hi < length xs*⟩ ⟨*lo ≤ hi*⟩ **and**
 trans: ⟨$\bigwedge x\ y\ z.$ ⟦$R\ (h\ x)\ (h\ y); R\ (h\ y)\ (h\ z)$⟧ $\Longrightarrow R\ (h\ x)\ (h\ z)$⟩ **and** *lin*: ⟨$\bigwedge x\ y.\ R\ (h\ x)$ $(h\ y)\ \vee\ R$ $(h\ y)\ (h\ x)$⟩
 shows ⟨*partition-main R h lo hi xs ≤ SPEC*$(\lambda(xs',\ p).\ mset\ xs\ =\ mset\ xs'\ \wedge$
 $lo\ \leq\ p\ \wedge\ p\ \leq\ hi\ \wedge\ isPartition\text{-}map\ R\ h\ xs'\ lo\ hi\ p\ \wedge\ (\forall\ i.\ i{<}lo\ \longrightarrow\ xs'!i{=}xs!i)\ \wedge\ (\forall\ i.\ hi{<}i\wedge i{<}length$
$xs'\ \longrightarrow\ xs'!i{=}xs!i))$⟩
⟨*proof*⟩

definition *partition-between* :: ⟨$('b\ \Rightarrow\ 'b\ \Rightarrow\ bool)\ \Rightarrow\ ('a\ \Rightarrow\ 'b)\ \Rightarrow\ nat\ \Rightarrow\ nat\ \Rightarrow\ 'a\ list\ \Rightarrow\ ('a\ list\ \times\ nat)$
nres⟩ **where**
 ⟨*partition-between R h lo hi xs0 = do* {
 $ASSERT(hi\ <\ length\ xs0\ \wedge\ lo\ \leq\ hi);$
 $k\ \leftarrow\ choose\text{-}pivot\ R\ h\ xs0\ lo\ hi;$ — choice of pivot
 $ASSERT(k\ <\ length\ xs0);$
 $xs\ \leftarrow\ RETURN\ (swap\ xs0\ k\ hi);$ — move the pivot to the last position, before we start the actual
loop
 $ASSERT(length\ xs\ =\ length\ xs0);$
 partition-main R h lo hi xs
 $\}\rangle$

lemma *partition-between-correct*:
 assumes ⟨*hi < length xs*⟩ **and** ⟨*lo ≤ hi*⟩ **and**
 ⟨$\bigwedge x\ y\ z.$ ⟦$R\ (h\ x)\ (h\ y); R\ (h\ y)\ (h\ z)$⟧ $\Longrightarrow R\ (h\ x)\ (h\ z)$⟩ **and** ⟨$\bigwedge x\ y.\ R\ (h\ x)\ (h\ y)\ \vee\ R\ (h\ y)\ (h\ x)$⟩
 shows ⟨*partition-between R h lo hi xs ≤ SPEC(uncurry (partition-spec R h xs lo hi))*⟩
⟨*proof*⟩

We use the median of the first, the middle, and the last element.

definition *choose-pivot3* **where**
 ⟨*choose-pivot3 R h xs lo* $(hi::nat)\ =\ do$ {
 $ASSERT(lo\ <\ length\ xs);$
 $ASSERT(hi\ <\ length\ xs);$
 $let\ k'\ =\ (hi\ -\ lo)\ div\ 2;$
 $let\ k\ =\ lo\ +\ k';$
 $ASSERT(k\ <\ length\ xs);$
 $let\ start\ =\ h\ (xs\ !\ lo);$
 $let\ mid\ =\ h\ (xs\ !\ k);$
 $let\ end\ =\ h\ (xs\ !\ hi);$
 $if\ (R\ start\ mid\ \wedge\ R\ mid\ end)\ \vee\ (R\ end\ mid\ \wedge\ R\ mid\ start)\ then\ RETURN\ k$

  else if $(R\ start\ end \wedge R\ end\ mid) \vee (R\ mid\ end \wedge R\ end\ start)$ *then RETURN hi*

  else RETURN lo

$\}\rangle$

— We only have to show that this procedure yields a valid index between *lo* and *hi*.

lemma *choose-pivot3-choose-pivot*:

 assumes ‹*lo < length xs*› ‹*hi < length xs*› ‹*hi ≥ lo*›

 shows ‹*choose-pivot3 R h xs lo hi* $\leq \Downarrow$ *Id* (*choose-pivot R h xs lo hi*)›

 ⟨*proof*⟩

The refined partion function: We use the above pivot function and fold instead of non-deterministic iteration.

definition *partition-between-ref*

 :: ‹$('b \Rightarrow {}'b \Rightarrow bool) \Rightarrow ('a \Rightarrow {}'b) \Rightarrow nat \Rightarrow nat \Rightarrow {}'a\ list \Rightarrow ('a\ list \times nat)\ nres$›

where

 ‹*partition-between-ref R h lo hi xs0* = *do* {

  ASSERT(*hi < length xs0* $\wedge$ *hi < length xs0* $\wedge$ *lo* $\leq$ *hi*);

  $k \leftarrow$ *choose-pivot3 R h xs0 lo hi*; — choice of pivot

  ASSERT(*k < length xs0*);

  $xs \leftarrow$ *RETURN* (*swap xs0 k hi*); — move the pivot to the last position, before we start the actual loop

  ASSERT(*length xs = length xs0*);

  partition-main R h lo hi xs

 }›

lemma *partition-main-ref′*:

 ‹*partition-main R h lo hi xs*

  $\leq \Downarrow ((\lambda\ a\ b\ c\ d.\ Id)\ a\ b\ c\ d)$ (*partition-main R h lo hi xs*)›

 ⟨*proof*⟩

lemma *Down-id-eq*:

 ‹$\Downarrow Id\ x = x$›

 ⟨*proof*⟩

lemma *partition-between-ref-partition-between*:

 ‹*partition-between-ref R h lo hi xs* $\leq$ (*partition-between R h lo hi xs*)›

⟨*proof*⟩

Technical lemma for sepref

lemma *partition-between-ref-partition-between′*:

 ‹(*uncurry2* (*partition-between-ref R h*), *uncurry2* (*partition-between R h*)) $\in$

  (*nat-rel* $\times_r$ *nat-rel*) $\times_r$ ⟨*Id*⟩*list-rel* $\rightarrow_f$ ⟨⟨*Id*⟩*list-rel* $\times_r$ *nat-rel*⟩*nres-rel*›

 ⟨*proof*⟩

Example instantiation for pivot

definition *choose-pivot3-impl* **where**

 ‹*choose-pivot3-impl* = *choose-pivot3* ($\leq$) *id*›

lemma *partition-between-ref-correct*:

 assumes *trans*: ‹$\bigwedge x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \implies R\ (h\ x)\ (h\ z)$› **and** *lin*: ‹$\bigwedge x\ y.\ R\ (h\ x)\ (h\ y) \vee R\ (h\ y)\ (h\ x)$›

and *bounds*: ⟨*hi* < *length xs*⟩ ⟨*lo* ≤ *hi*⟩
  shows ⟨*partition-between-ref R h lo hi xs* ≤ *SPEC* (*uncurry* (*partition-spec R h xs lo hi*))⟩
⟨*proof*⟩

Refined quicksort algorithm: We use the refined partition function.

definition *quicksort-ref* :: ⟨- ⇒ - ⇒ *nat* × *nat* × *'a list* ⇒ *'a list nres*⟩ **where**
⟨*quicksort-ref R h* = (λ(*lo,hi,xs0*).
  do {
  RECT (λ*f* (*lo,hi,xs*). *do* {
    ASSERT(*lo* ≤ *hi* ∧ *hi* < *length xs0* ∧ *mset xs* = *mset xs0*);
    (*xs*, *p*) ← *partition-between-ref R h lo hi xs*; — This is the refined partition function. Note that we
need the premises (trans,lin,bounds) here.
    ASSERT(*mset xs* = *mset xs0* ∧ *p* ≥ *lo* ∧ *p* < *length xs0*);
    xs ← (*if p−1≤lo then RETURN xs else f* (*lo, p−1, xs*));
    ASSERT(*mset xs* = *mset xs0*);
    if hi≤p+1 then RETURN xs else f (*p+1, hi, xs*)
  }) (*lo,hi,xs0*)
  })⟩

lemma *fref-to-Down-curry2*:
  ⟨(*uncurry2 f*, *uncurry2 g*) ∈ [*P*]$_f$ *A* → ⟨*B*⟩*nres-rel* ⟹
    (⋀*x x' y y' z z'*. *P* ((*x', y'*), *z'*) ⟹ (((*x, y*), *z*), ((*x', y'*), *z'*)) ∈ *A*⟹
      f x y z ≤ ⇓ *B* (*g x' y' z'*))⟩
⟨*proof*⟩

lemma *fref-to-Down-curry*:
  ⟨(*f*, *g*) ∈ [*P*]$_f$ *A* → ⟨*B*⟩*nres-rel* ⟹
    (⋀*x x'*. *P x'* ⟹ (*x, x'*) ∈ *A*⟹
      f x ≤ ⇓ *B* (*g x'*))⟩
⟨*proof*⟩

lemma *quicksort-ref-quicksort*:
  assumes *bounds*: ⟨*hi* < *length xs*⟩ ⟨*lo* ≤ *hi*⟩ **and**
    trans: ⟨⋀ *x y z*. ⟦*R* (*h x*) (*h y*); *R* (*h y*) (*h z*)⟧ ⟹ *R* (*h x*) (*h z*)⟩ **and** *lin*: ⟨⋀*x y*. *R* (*h x*) (*h y*) ∨ *R*
(*h y*) (*h x*)⟩
  shows ⟨*quicksort-ref R h x0* ≤ ⇓ *Id* (*quicksort R h x0*)⟩
⟨*proof*⟩
definition *full-quicksort* **where**
  ⟨*full-quicksort R h xs* ≡ *if xs* = [] *then RETURN xs else quicksort R h* (*0, length xs* − *1, xs*)⟩

definition *full-quicksort-ref* **where**
  ⟨*full-quicksort-ref R h xs* ≡
    if List.null xs then RETURN xs
    else quicksort-ref R h (*0, length xs* − *1, xs*)⟩

definition *full-quicksort-impl* :: ⟨*nat list* ⇒ *nat list nres*⟩ **where**
  ⟨*full-quicksort-impl xs* = *full-quicksort-ref* (≤) *id xs*⟩

lemma *full-quicksort-ref-full-quicksort*:
  assumes *trans*: ⟨⋀ *x y z*. ⟦*R* (*h x*) (*h y*); *R* (*h y*) (*h z*)⟧ ⟹ *R* (*h x*) (*h z*)⟩ **and** *lin*: ⟨⋀*x y*. *R* (*h x*) (*h
y*) ∨ *R* (*h y*) (*h x*)⟩

shows ⟨*(full-quicksort-ref R h, full-quicksort R h)* ∈
    ⟨*Id*⟩*list-rel* →$_f$ ⟨ ⟨*Id*⟩*list-rel*⟩*nres-rel*⟩
⟨*proof*⟩

lemma *sublist-entire*:
  ⟨*sublist xs 0 (length xs − 1) = xs*⟩
  ⟨*proof*⟩

lemma *sorted-sublist-wrt-entire*:
  assumes ⟨*sorted-sublist-wrt R xs 0 (length xs − 1)*⟩
  shows ⟨*sorted-wrt R xs*⟩
⟨*proof*⟩

lemma *sorted-sublist-map-entire*:
  assumes ⟨*sorted-sublist-map R h xs 0 (length xs − 1)*⟩
  shows ⟨*sorted-wrt (λ x y. R (h x) (h y)) xs*⟩
⟨*proof*⟩

Final correctness lemma

theorem *full-quicksort-correct-sorted*:
  assumes
    trans: ⟨⋀x y z. ⟦*R (h x) (h y)*; *R (h y) (h z)*⟧ ⟹ *R (h x) (h z)*⟩ **and** *lin*: ⟨⋀x y. $x \neq y$ ⟹ *R (h x)*
(h y) ∨ R (h y) (h x)⟩
  shows ⟨*full-quicksort R h xs* ≤ ⇓ *Id (SPEC(λxs'. mset xs' = mset xs ∧ sorted-wrt (λ x y. R (h x) (h*
y)) xs'))⟩
⟨*proof*⟩

lemma *full-quicksort-correct*:
  assumes
    trans: ⟨⋀x y z. ⟦*R (h x) (h y)*; *R (h y) (h z)*⟧ ⟹ *R (h x) (h z)*⟩ **and**
    lin: ⟨⋀x y. *R (h x) (h y) ∨ R (h y) (h x)*⟩
  shows ⟨*full-quicksort R h xs* ≤ ⇓ *Id (SPEC(λxs'. mset xs' = mset xs))*⟩
  ⟨*proof*⟩

end
theory *More-Loops*
imports
  Refine-Monadic.Refine-While
  Refine-Monadic.Refine-Foreach
  HOL−Library.Rewrite
begin

1.4 More Theorem about Loops

Most theorem below have a counterpart in the Refinement Framework that is weaker (by missing
assertions for example that are critical for code generation).

lemma *Down-id-eq*:
  ⟨⇓*Id x = x*⟩
  ⟨*proof*⟩

lemma *while-upt-while-direct1*:
  $b \geq a$ ⟹
  do {

```
   (-,σ) ← WHILE_T (FOREACH-cond c) (λx. do {ASSERT (FOREACH-cond c x); FOREACH-body
f x})
     ([a..<b],σ);
   RETURN σ
 } ≤ do {
 (-,σ) ← WHILE_T (λ(i, x). i < b ∧ c x) (λ(i, x). do {ASSERT (i < b); σ'←f i x; RETURN (i+1,σ')
}) (a,σ);
   RETURN σ
 }
 ⟨proof⟩
```

lemma *while-upt-while-direct2*:
```
 b ≥ a ⟹
 do {
 (-,σ) ← WHILE_T (FOREACH-cond c) (λx. do {ASSERT (FOREACH-cond c x); FOREACH-body
f x})
     ([a..<b],σ);
   RETURN σ
 } ≥ do {
 (-,σ) ← WHILE_T (λ(i, x). i < b ∧ c x) (λ(i, x). do {ASSERT (i < b); σ'←f i x; RETURN (i+1,σ')
}) (a,σ);
   RETURN σ
 }
 ⟨proof⟩
```

lemma *while-upt-while-direct*:
```
 b ≥ a ⟹
 do {
 (-,σ) ← WHILE_T (FOREACH-cond c) (λx. do {ASSERT (FOREACH-cond c x); FOREACH-body
f x})
     ([a..<b],σ);
   RETURN σ
 } = do {
 (-,σ) ← WHILE_T (λ(i, x). i < b ∧ c x) (λ(i, x). do {ASSERT (i < b); σ'←f i x; RETURN (i+1,σ')
}) (a,σ);
   RETURN σ
 }
 ⟨proof⟩
```

lemma *while-nfoldli*:
```
 do {
 (-,σ) ← WHILE_T (FOREACH-cond c) (λx. do {ASSERT (FOREACH-cond c x); FOREACH-body
f x}) (l,σ);
   RETURN σ
 } ≤ nfoldli l c f σ
 ⟨proof⟩
```
lemma *nfoldli-while*: nfoldli l c f σ
```
       ≤
     (WHILE_T^I
        (FOREACH-cond c) (λx. do {ASSERT (FOREACH-cond c x); FOREACH-body f x}) (l, σ)
≫=
       (λ(-, σ). RETURN σ))
⟨proof⟩
```

lemma *while-eq-nfoldli*: do {

```
    (-,σ) ← WHILE_T (FOREACH-cond c) (λx. do {ASSERT (FOREACH-cond c x); FOREACH-body
f x}) (l,σ);
    RETURN σ
  } = nfoldli l c f σ
  ⟨proof⟩

end
theory PAC-More-Poly
  imports HOL−Library.Poly-Mapping HOL−Algebra.Polynomials Polynomials.MPoly-Type-Class
  HOL−Algebra.Module
  HOL−Library.Countable-Set
begin
```

2 Libraries

2.1 More Polynomials

Here are more theorems on polynomials. Most of these facts are extremely trivial and should probably be generalised and moved to the Isabelle distribution.

lemma $Const_0$-add:
‹$Const_0$ (a + b) = $Const_0$ a + $Const_0$ b›
⟨proof⟩

lemma Const-mult:
‹Const (a * b) = Const a * Const b›
⟨proof⟩

lemma $Const_0$-mult:
‹$Const_0$ (a * b) = $Const_0$ a * $Const_0$ b›
⟨proof⟩

lemma Const0[simp]:
‹Const 0 = 0›
⟨proof⟩

lemma (**in** −) Const-uminus[simp]:
‹Const (−n) = − Const n›
⟨proof⟩

lemma [simp]: ‹$Const_0$ 0 = 0›
‹MPoly 0 = 0›
⟨proof⟩

lemma Const-add:
‹Const (a + b) = Const a + Const b›
⟨proof⟩

instance mpoly :: (comm-semiring-1) comm-semiring-1
⟨proof⟩

lemma degree-uminus[simp]:
‹degree (−A) x′ = degree A x′›
⟨proof⟩

lemma *degree-sum-notin*:
  ‹$x' \notin$ vars $B \implies$ degree $(A + B)$ $x' =$ degree A x'›
  ⟨*proof*⟩

lemma *degree-notin-vars*:
  ‹$x \notin$ (vars B) $\implies$ degree $(B :: {}'a :: \{monoid\text{-}add\}$ mpoly) $x = 0$›
  ⟨*proof*⟩

lemma *not-in-vars-coeff0*:
  ‹$x \notin$ vars $p \implies$ MPoly-Type.coeff p (monomial (Suc 0) x) $= 0$›
  ⟨*proof*⟩

lemma *keys-mapping-sum-add*:
  ‹finite $A \implies$ keys (mapping-of $(\sum v \in A.\ f\ v)) \subseteq \bigcup$ (keys ' mapping-of ' f ' UNIV)›
  ⟨*proof*⟩

lemma *vars-sum-vars-union*:
  fixes $f ::$ ‹int mpoly $\Rightarrow$ int mpoly›
  assumes ‹finite $\{v.\ f\ v \neq 0\}$›
  shows ‹vars $(\sum v \mid f\ v \neq 0.\ f\ v * v) \subseteq \bigcup$ (vars ' $\{v.\ f\ v \neq 0\}) \cup \bigcup$ (vars ' f ' $\{v.\ f\ v \neq 0\}$)›
    (**is** ‹?$A \subseteq$?B›)
⟨*proof*⟩

lemma *vars-in-right-only*:
  $x \in$ vars $q \implies x \notin$ vars $p \implies x \in$ vars $(p+q)$
  ⟨*proof*⟩

lemma [*simp*]:
  ‹vars $0 = \{\}$›
  ⟨*proof*⟩

lemma *vars-Un-nointer*:
  ‹keys (mapping-of p) $\cap$ keys (mapping-of q) $= \{\} \implies$ vars $(p + q) =$ vars $p \cup$ vars q›
  ⟨*proof*⟩

lemmas [*simp*] $=$ *zero-mpoly.rep-eq*

lemma *polynomial-sum-monoms*:
  fixes $p ::$ ‹${}'a :: \{comm\text{-}monoid\text{-}add,cancel\text{-}comm\text{-}monoid\text{-}add\}$ mpoly›
  shows
    ‹$p = (\sum x \in$ keys (mapping-of p). MPoly-Type.monom x (MPoly-Type.coeff p x))›
    ‹keys (mapping-of p) $\subseteq I \implies$ finite $I \implies p = (\sum x \in I.$ MPoly-Type.monom x (MPoly-Type.coeff p
x))›
⟨*proof*⟩

lemma *vars-mult-monom*:
  fixes $p ::$ ‹int mpoly›
  shows ‹vars $(p * ($monom (monomial (Suc 0) x') 1)) = ($if $p = 0$ then $\{\}$ else insert x' (vars p))›
⟨*proof*⟩

lemma *in-mapping-mult-single*:
  ‹$x \in (\lambda x.$ lookup x x') ' keys $(A * (Var_0$ $x' :: ($nat $\Rightarrow_0$ nat$) \Rightarrow_0 {}'b :: \{monoid\text{-}mult,zero\text{-}neq\text{-}one,semiring\text{-}0\}))$

$\longleftrightarrow$
$x > 0 \land x - 1 \in (\lambda x.\ lookup\ x\ x')\ `\ keys\ (A)$⟩
⟨*proof*⟩

lemma *Max-Suc-Suc-Max*:
⟨*finite* $A \implies A \neq \{\} \implies Max\ (insert\ 0\ (Suc\ `\ A)) =$
Suc $(Max\ (insert\ 0\ A))$⟩
⟨*proof*⟩

lemma [*simp*]:
⟨*keys* $(Var_0\ x' :: ('a \Rightarrow_0 nat) \Rightarrow_0 'b :: \{zero\text{-}neq\text{-}one\}) = \{Poly\text{-}Mapping.single\ x'\ 1\}$⟩
⟨*proof*⟩

lemma *degree-mult-Var*:
⟨*degree* $(A * Var\ x')\ x' = (if\ A = 0\ then\ 0\ else\ Suc\ (degree\ A\ x'))$⟩ **for** $A ::$ ⟨*int mpoly*⟩
⟨*proof*⟩

lemma *degree-mult-Var'*:
⟨*degree* $(Var\ x' * A)\ x' = (if\ A = 0\ then\ 0\ else\ Suc\ (degree\ A\ x'))$⟩ **for** $A ::$ ⟨*int mpoly*⟩
⟨*proof*⟩

lemma *degree-add-max*:
⟨*degree* $(A + B)\ x \leq max\ (degree\ A\ x)\ (degree\ B\ x)$⟩
⟨*proof*⟩

lemma *degree-times-le*:
⟨*degree* $(A * B)\ x \leq degree\ A\ x + degree\ B\ x$⟩
⟨*proof*⟩

lemma *monomial-inj*:
monomial $c\ s = monomial\ (d::'b::zero\text{-}neq\text{-}one)\ t \longleftrightarrow (c = 0 \land d = 0) \lor (c = d \land s = t)$
⟨*proof*⟩

lemma *MPoly-monomial-power'*:
⟨*MPoly* $(monomial\ 1\ x')\ \widehat{\ }\ (n+1) = MPoly\ (monomial\ (1)\ (((\lambda x.\ x + x')\ \overset{\frown}{\ }\ n)\ x'))$⟩
⟨*proof*⟩

lemma *MPoly-monomial-power*:
⟨$n > 0 \implies MPoly\ (monomial\ 1\ x')\ \widehat{\ }\ (n) = MPoly\ (monomial\ (1)\ (((\lambda x.\ x + x')\ \overset{\frown}{\ }\ (n - 1))\ x'))$⟩
⟨*proof*⟩

lemma *vars-uminus*[*simp*]:
⟨*vars* $(-p) = vars\ p$⟩
⟨*proof*⟩

lemma *coeff-uminus*[*simp*]:
⟨*MPoly-Type.coeff* $(-p)\ x = -MPoly\text{-}Type.coeff\ p\ x$⟩
⟨*proof*⟩

definition *decrease-key*::$'a \Rightarrow ('a \Rightarrow_0 'b::\{monoid\text{-}add,\ minus, one\}) \Rightarrow ('a \Rightarrow_0 'b)$ **where**
decrease-key $k0\ f = Abs\text{-}poly\text{-}mapping\ (\lambda k.\ if\ k = k0 \land lookup\ f\ k \neq 0\ then\ lookup\ f\ k - 1\ else\ lookup$
$f\ k)$

lemma *remove-key-lookup*:
 lookup (*decrease-key k0 f*) *k* = (*if k* = *k0* ∧ *lookup f k* ≠ *0 then lookup f k* − *1 else lookup f k*)
 ⟨*proof*⟩

lemma *polynomial-split-on-var*:
  fixes *p* :: ⟨′*a* :: {*comm-monoid-add*,*cancel-comm-monoid-add*,*semiring-0*,*comm-semiring-1*} *mpoly*⟩
  obtains *q r* **where**
    ⟨*p* = *monom* (*monomial* (*Suc 0*) *x*′) *1* ∗ *q* + *r*⟩ **and**
    ⟨*x*′ ∉ *vars r*⟩
⟨*proof*⟩

lemma *polynomial-split-on-var2*:
  fixes *p* :: ⟨*int mpoly*⟩
  assumes ⟨*x*′ ∉ *vars s*⟩
  obtains *q r* **where**
    ⟨*p* = (*monom* (*monomial* (*Suc 0*) *x*′) *1* − *s*) ∗ *q* + *r*⟩ **and**
    ⟨*x*′ ∉ *vars r*⟩
⟨*proof*⟩

lemma *polynomial-split-on-var-diff-sq2*:
 fixes *p* :: ⟨*int mpoly*⟩
  obtains *q r s* **where**
    ⟨*p* = *monom* (*monomial* (*Suc 0*) *x*′) *1* ∗ *q* + *r* + *s* ∗ (*monom* (*monomial* (*Suc 0*) *x*′) *1*^*2* − *monom*
(*monomial* (*Suc 0*) *x*′) *1*)⟩ **and**
    ⟨*x*′ ∉ *vars r*⟩ **and**
    ⟨*x*′ ∉ *vars q*⟩
⟨*proof*⟩

lemma *polynomial-decomp-alien-var*:
  fixes *q A b* :: ⟨*int mpoly*⟩
  assumes
    q: ⟨*q* = *A* ∗ (*monom* (*monomial* (*Suc 0*) *x*′) *1*) + *b*⟩ **and**
    x: ⟨*x*′ ∉ *vars q*⟩ ⟨*x*′ ∉ *vars b*⟩
  shows
    ⟨*A* = *0*⟩ **and**
    ⟨*q* = *b*⟩
⟨*proof*⟩

lemma *polynomial-decomp-alien-var2*:
  fixes *q A b* :: ⟨*int mpoly*⟩
  assumes
    q: ⟨*q* = *A* ∗ (*monom* (*monomial* (*Suc 0*) *x*′) *1* + *p*) + *b*⟩ **and**
    x: ⟨*x*′ ∉ *vars q*⟩ ⟨*x*′ ∉ *vars b*⟩ ⟨*x*′ ∉ *vars p*⟩
  shows
    ⟨*A* = *0*⟩ **and**
    ⟨*q* = *b*⟩
⟨*proof*⟩

lemma *vars-unE*: ⟨*x* ∈ *vars* (*a* ∗ *b*) ⟹ (*x* ∈ *vars a* ⟹ *thesis*) ⟹ (*x* ∈ *vars b* ⟹ *thesis*) ⟹ *thesis*⟩
  ⟨*proof*⟩

lemma *in-keys-minusI1*:

assumes $t \in keys\ p$ **and** $t \notin keys\ q$
shows $t \in keys\ (p - q)$
⟨*proof*⟩

lemma *in-keys-minusI2*:
  fixes $t :: \langle'a\rangle$ **and** $q :: \langle'a \Rightarrow_0 {'}b :: \{cancel\text{-}comm\text{-}monoid\text{-}add, group\text{-}add\}\rangle$
  assumes $t \in keys\ q$ **and** $t \notin keys\ p$
  shows $t \in keys\ (p - q)$
  ⟨*proof*⟩

lemma *in-vars-addE*:
  ⟨$x \in vars\ (p + q) \Longrightarrow (x \in vars\ p \Longrightarrow thesis) \Longrightarrow (x \in vars\ q \Longrightarrow thesis) \Longrightarrow thesis$⟩
  ⟨*proof*⟩

lemma *lookup-monomial-If*:
  ⟨$lookup\ (monomial\ v\ k) = (\lambda k'.\ if\ k = k'\ then\ v\ else\ 0)$⟩
  ⟨*proof*⟩

lemma *vars-mult-Var*:
  ⟨$vars\ (Var\ x * p) = (if\ p = 0\ then\ \{\}\ else\ insert\ x\ (vars\ p))$⟩ **for** $p :: \langle int\ mpoly\rangle$
  ⟨*proof*⟩

lemma *keys-mult-monomial*:
  ⟨$keys\ (monomial\ (n :: int)\ k * mapping\text{-}of\ a) = (if\ n = 0\ then\ \{\}\ else\ ((+)\ k)\ `\ keys\ (mapping\text{-}of\ a))$⟩
⟨*proof*⟩

lemma *vars-mult-Const*:
  ⟨$vars\ (Const\ n * a) = (if\ n = 0\ then\ \{\}\ else\ vars\ a)$⟩ **for** $a :: \langle int\ mpoly\rangle$
  ⟨*proof*⟩

lemma *coeff-minus*: *coeff p m* $-$ *coeff q m* $=$ *coeff* $(p-q)$ *m*
  ⟨*proof*⟩

lemma *Const-1-eq-1*: ⟨$Const\ (1 :: int) = (1 :: int\ mpoly)$⟩
  ⟨*proof*⟩

lemma [*simp*]:
  ⟨$vars\ (1 :: int\ mpoly) = \{\}$⟩
  ⟨*proof*⟩

2.2   More Ideals

lemma
  fixes $A :: \langle(('x \Rightarrow_0 nat) \Rightarrow_0 {'}a::comm\text{-}ring\text{-}1)\ set\rangle$
  assumes ⟨$p \in ideal\ A$⟩
  shows ⟨$p * q \in ideal\ A$⟩
  ⟨*proof*⟩

The following theorem is very close to *More-Modules.ideal* (*insert ?a ?S*) $= \{x.\ \exists k.\ x - k *$
?a $\in$ *More-Modules.ideal ?S*}, except that it is more useful if we need to take an element of
More-Modules.ideal (*insert a S*).

lemma *ideal-insert′*:
  ⟨*More-Modules.ideal* (*insert a S*) $= \{y.\ \exists x\ k.\ y = x + k * a \land x \in$ *More-Modules.ideal S*}⟩

$\langle proof \rangle$

lemma *ideal-mult-right-in*:
  $\langle a \in ideal\ A \implies a * b \in More\text{-}Modules.ideal\ A \rangle$
  $\langle proof \rangle$

lemma *ideal-mult-right-in2*:
  $\langle a \in ideal\ A \implies b * a \in More\text{-}Modules.ideal\ A \rangle$
  $\langle proof \rangle$

lemma [*simp*]: $\langle vars\ (Var\ x :: 'a :: \{zero\text{-}neq\text{-}one\}\ mpoly) = \{x\} \rangle$
  $\langle proof \rangle$

lemma *vars-minus-Var-subset*:
  $\langle vars\ (p' - Var\ x :: 'a :: \{ab\text{-}group\text{-}add,one,zero\text{-}neq\text{-}one\}\ mpoly) \subseteq \mathcal{V} \implies vars\ p' \subseteq insert\ x\ \mathcal{V} \rangle$
  $\langle proof \rangle$

lemma *vars-add-Var-subset*:
  $\langle vars\ (p' + Var\ x :: 'a :: \{ab\text{-}group\text{-}add,one,zero\text{-}neq\text{-}one\}\ mpoly) \subseteq \mathcal{V} \implies vars\ p' \subseteq insert\ x\ \mathcal{V} \rangle$
  $\langle proof \rangle$

lemma *coeff-monomila-in-varsD*:
  $\langle coeff\ p\ (monomial\ (Suc\ 0)\ x) \neq 0 \implies x \in vars\ (p :: int\ mpoly) \rangle$
  $\langle proof \rangle$

lemma (**in** −)*coeff-MPoly-monomila*[*simp*]:
  $\langle Const\ (MPoly\text{-}Type.coeff\ (MPoly\ (monomial\ a\ m))\ m) = Const\ a \rangle$
  $\langle proof \rangle$

end
theory *PAC-Specification*
  imports *PAC-More-Poly*
begin

3 Specification of the PAC checker

3.1 Ideals

type-synonym *int-poly* = $\langle int\ mpoly \rangle$
definition *polynomial-bool* :: $\langle int\text{-}poly\ set \rangle$ **where**
  $\langle polynomial\text{-}bool = (\lambda c.\ Var\ c\ \hat{}\ 2 - Var\ c)\ `\ UNIV \rangle$

definition *pac-ideal* **where**
  $\langle pac\text{-}ideal\ A \equiv ideal\ (A \cup polynomial\text{-}bool) \rangle$

lemma *X2-X-in-pac-ideal*:
  $\langle Var\ c\ \hat{}\ 2 - Var\ c \in pac\text{-}ideal\ A \rangle$
  $\langle proof \rangle$

lemma *pac-idealI1*[*intro*]:
  $\langle p \in A \implies p \in pac\text{-}ideal\ A \rangle$
  $\langle proof \rangle$

lemma *pac-idealI2*[*intro*]:

‹p ∈ ideal A ⟹ p ∈ pac-ideal A›
⟨*proof*⟩

lemma *pac-idealI3*[*intro*]:
‹p ∈ ideal A ⟹ p∗q ∈ pac-ideal A›
⟨*proof*⟩

lemma *pac-ideal-Xsq2-iff*:
‹ Var c ^ 2 ∈ pac-ideal A ⟷ Var c ∈ pac-ideal A›
⟨*proof*⟩

lemma *diff-in-polynomial-bool-pac-idealI*:
  assumes *a1*: *p ∈ pac-ideal A*
  assumes *a2*: *p − p′ ∈ More-Modules.ideal polynomial-bool*
  shows *‹p′ ∈ pac-ideal A›*
⟨*proof*⟩

lemma *diff-in-polynomial-bool-pac-idealI2*:
  assumes *a1*: *p ∈ A*
  assumes *a2*: *p − p′ ∈ More-Modules.ideal polynomial-bool*
  shows *‹p′ ∈ pac-ideal A›*
  ⟨*proof*⟩

lemma *pac-ideal-alt-def*:
‹pac-ideal A = ideal (A ∪ ideal polynomial-bool)›
⟨*proof*⟩

The equality on ideals is restricted to polynomials whose variable appear in the set of ideals. The function restrict sets:

definition *restricted-ideal-to* **where**
‹restricted-ideal-to B A = {p ∈ A. vars p ⊆ B}›

abbreviation *restricted-ideal-to_I* **where**
‹restricted-ideal-to_I B A ≡ restricted-ideal-to B (pac-ideal (set-mset A))›

abbreviation *restricted-ideal-to_V* **where**
‹restricted-ideal-to_V B ≡ restricted-ideal-to (⋃(vars ' set-mset B))›

abbreviation *restricted-ideal-to_{V I}* **where**
‹restricted-ideal-to_{V I} B A ≡ restricted-ideal-to (⋃(vars ' set-mset B)) (pac-ideal (set-mset A))›

lemma *restricted-idealI*:
‹p ∈ pac-ideal (set-mset A) ⟹ vars p ⊆ C ⟹ p ∈ restricted-ideal-to_I C A›
⟨*proof*⟩

lemma *pac-ideal-insert-already-in*:
‹pq ∈ pac-ideal (set-mset A) ⟹ pac-ideal (insert pq (set-mset A)) = pac-ideal (set-mset A)›
⟨*proof*⟩

lemma *pac-ideal-add*:
‹p ∈# A ⟹ q ∈# A ⟹ p + q ∈ pac-ideal (set-mset A)›
⟨*proof*⟩
lemma *pac-ideal-mult*:
‹p ∈# A ⟹ p ∗ q ∈ pac-ideal (set-mset A)›

⟨*proof*⟩

lemma *pac-ideal-mono*:
  ⟨*A ⊆ B ⟹ pac-ideal A ⊆ pac-ideal B*⟩
  ⟨*proof*⟩

3.2 PAC Format

The PAC format contains three kind of steps:

- add that adds up two polynomials that are known.

- mult that multiply a known polynomial with another one.

- del that removes a polynomial that cannot be reused anymore.

To model the simplification that happens, we add the $p - p' \in$ *polynomial-bool* stating that p and p' are equivalent.

type-synonym *pac-st =* ⟨*(nat set × int-poly multiset)*⟩

inductive *PAC-Format ::* ⟨*pac-st ⇒ pac-st ⇒ bool*⟩ **where**
add:
  ⟨*PAC-Format (𝒱, A) (𝒱, add-mset p' A)*⟩
if
  ⟨*p ∈# A*⟩ ⟨*q ∈# A*⟩
  ⟨*p+q − p' ∈ ideal polynomial-bool*⟩
  ⟨*vars p' ⊆ 𝒱*⟩ |
mult:
  ⟨*PAC-Format (𝒱, A) (𝒱, add-mset p' A)*⟩
if
  ⟨*p ∈# A*⟩
  ⟨*p∗q − p' ∈ ideal polynomial-bool*⟩
  ⟨*vars p' ⊆ 𝒱*⟩
  ⟨*vars q ⊆ 𝒱*⟩ |
del:
  ⟨*p ∈# A ⟹ PAC-Format (𝒱, A) (𝒱, A − {#p#})*⟩ |
extend-pos:
  ⟨*PAC-Format (𝒱, A) (𝒱 ∪ {x' ∈ vars (− Var x + p'). x' ∉ 𝒱}, add-mset (− Var x + p') A)*⟩
  if
    ⟨*(p')² − p' ∈ ideal polynomial-bool*⟩
    ⟨*vars p' ⊆ 𝒱*⟩
    ⟨*x ∉ 𝒱*⟩

In the PAC format above, we have a technical condition on the normalisation: *vars p' ⊆ vars (p + q)* is here to ensure that we don't normalise *0* to *(Var x)² − Var x* for a new variable *x*. This is completely obvious for the normalisation processe we have in mind when we write the specification, but we must add it explicitly because we are too general.

lemmas *PAC-Format-induct-split =*
  PAC-Format.induct[split-format(complete), of V A V′ A′ **for** *V A V′ A′*]

lemma *PAC-Format-induct[consumes 1, case-names add mult del ext]*:
  assumes
    ⟨*PAC-Format (𝒱, A) (𝒱′, A′)*⟩ **and**
    cases:

26

⟨⋀p q p' A V. p ∈# A ⟹ q ∈# A ⟹ p+q − p' ∈ ideal polynomial-bool ⟹ vars p' ⊆ V ⟹ P
V A V (add-mset p' A)⟩
    ⟨⋀p q p' A V. p ∈# A ⟹ p∗q − p' ∈ ideal polynomial-bool ⟹ vars p' ⊆ V ⟹ vars q ⊆ V ⟹
      P V A V (add-mset p' A)⟩
    ⟨⋀p A V. p ∈# A ⟹ P V A V (A − {#p#})⟩
    ⟨⋀p' x r.
      (p')^2 − (p') ∈ ideal polynomial-bool ⟹ vars p' ⊆ V ⟹
      x ∉ V ⟹ P V A (V ∪ {x' ∈ vars (p' − Var x). x' ∉ V}) (add-mset (p' − Var x) A)⟩
  shows
    ⟨P V A V' A'⟩
⟨*proof*⟩

The theorem below (based on the proof ideal by Manuel Kauers) is the correctness theorem of
extensions. Remark that the assumption *vars q ⊆ V* is only used to show that $x' \notin vars\ q$.

lemma *extensions-are-safe*:
  assumes ⟨x' ∈ vars p⟩ **and**
    x': ⟨x' ∉ V⟩ **and**
    ⟨⋃ (vars ' set-mset A) ⊆ V⟩ **and**
    p-x-coeff: ⟨coeff p (monomial (Suc 0) x') = 1⟩ **and**
    vars-q: ⟨vars q ⊆ V⟩ **and**
    q: ⟨q ∈ More-Modules.ideal (insert p (set-mset A ∪ polynomial-bool))⟩ **and**
    leading: ⟨x' ∉ vars (p − Var x')⟩ **and**
    diff: ⟨(Var x' − p)² − (Var x' − p) ∈ More-Modules.ideal polynomial-bool⟩
  shows
    ⟨q ∈ More-Modules.ideal (set-mset A ∪ polynomial-bool)⟩
⟨*proof*⟩

lemma *extensions-are-safe-uminus*:
  assumes ⟨x' ∈ vars p⟩ **and**
    x': ⟨x' ∉ V⟩ **and**
    ⟨⋃ (vars ' set-mset A) ⊆ V⟩ **and**
    p-x-coeff: ⟨coeff p (monomial (Suc 0) x') = −1⟩ **and**
    vars-q: ⟨vars q ⊆ V⟩ **and**
    q: ⟨q ∈ More-Modules.ideal (insert p (set-mset A ∪ polynomial-bool))⟩ **and**
    leading: ⟨x' ∉ vars (p + Var x')⟩ **and**
    diff: ⟨(Var x' + p)^2 − (Var x' + p) ∈ More-Modules.ideal polynomial-bool⟩
  shows
    ⟨q ∈ More-Modules.ideal (set-mset A ∪ polynomial-bool)⟩
⟨*proof*⟩

This is the correctness theorem of a PAC step: no polynomials are added to the ideal.

lemma *vars-subst-in-left-only*:
  ⟨x ∉ vars p ⟹ x ∈ vars (p − Var x)⟩ **for** p :: ⟨int mpoly⟩
  ⟨*proof*⟩

lemma *vars-subst-in-left-only-diff-iff*:
  ⟨x ∉ vars p ⟹ vars (p − Var x) = insert x (vars p)⟩ **for** p :: ⟨int mpoly⟩
  ⟨*proof*⟩

lemma *vars-subst-in-left-only-iff*:
  ⟨x ∉ vars p ⟹ vars (p + Var x) = insert x (vars p)⟩ **for** p :: ⟨int mpoly⟩
  ⟨*proof*⟩

lemma *coeff-add-right-notin*:
  ⟨x ∉ vars p ⟹ MPoly-Type.coeff (Var x − p) (monomial (Suc 0) x) = 1⟩

⟨*proof*⟩

lemma *coeff-add-left-notin*:
⟨$x \notin vars\ p \implies$ *MPoly-Type.coeff* $(p - Var\ x)$ *(monomial (Suc 0) x)* $= -1$⟩ **for** p :: ⟨*int mpoly*⟩
⟨*proof*⟩

lemma *ideal-insert-polynomial-bool-swap*: ⟨$r - s \in$ *ideal polynomial-bool* $\implies$
More-Modules.ideal (*insert r* $(A \cup$ *polynomial-bool*)) = *More-Modules.ideal* (*insert s* $(A \cup$ *polynomial-bool*))⟩
⟨*proof*⟩

lemma *PAC-Format-subset-ideal*:
⟨*PAC-Format* $(\mathcal{V}, A)$ $(\mathcal{V}', B) \implies \bigcup$ (*vars ' set-mset A*) $\subseteq \mathcal{V} \implies$
restricted-ideal-to$_I$ $\mathcal{V}\ B \subseteq$ *restricted-ideal-to$_I$* $\mathcal{V}\ A \wedge \mathcal{V} \subseteq \mathcal{V}' \wedge \bigcup$ (*vars ' set-mset B*) $\subseteq \mathcal{V}'$⟩
⟨*proof*⟩

In general, if deletions are disallowed, then the stronger $B = $ *pac-ideal A* holds.

lemma *restricted-ideal-to-restricted-ideal-to$_I$D*:
⟨*restricted-ideal-to* $\mathcal{V}$ (*set-mset A*) $\subseteq$ *restricted-ideal-to$_I$* $\mathcal{V}\ A$⟩
⟨*proof*⟩

lemma *rtranclp-PAC-Format-subset-ideal*:
⟨*rtranclp PAC-Format* $(\mathcal{V}, A)$ $(\mathcal{V}', B) \implies \bigcup$ (*vars ' set-mset A*) $\subseteq \mathcal{V} \implies$
restricted-ideal-to$_I$ $\mathcal{V}\ B \subseteq$ *restricted-ideal-to$_I$* $\mathcal{V}\ A \wedge \mathcal{V} \subseteq \mathcal{V}' \wedge \bigcup$ (*vars ' set-mset B*) $\subseteq \mathcal{V}'$⟩
⟨*proof*⟩

end
theory *Finite-Map-Multiset*
imports *HOL−Library.Finite-Map Duplicate-Free-Multiset*
begin

notation *image-mset* (**infixr** '# 90)

4 Finite maps and multisets

4.1 Finite sets and multisets

abbreviation *mset-fset* :: ⟨$'a\ fset \Rightarrow\ 'a\ multiset$⟩ **where**
⟨*mset-fset N* $\equiv$ *mset-set* (*fset N*)⟩

definition *fset-mset* :: ⟨$'a\ multiset \Rightarrow\ 'a\ fset$⟩ **where**
⟨*fset-mset N* $\equiv$ *Abs-fset* (*set-mset N*)⟩

lemma *fset-mset-mset-fset*: ⟨*fset-mset* (*mset-fset N*) $= N$⟩
⟨*proof*⟩

lemma *mset-fset-fset-mset*[*simp*]:
⟨*mset-fset* (*fset-mset N*) $=$ *remdups-mset N*⟩
⟨*proof*⟩

lemma *in-mset-fset-fmember*[*simp*]: ⟨$x \in\#$ *mset-fset N* $\longleftrightarrow x \mid\in\mid N$⟩
⟨*proof*⟩

lemma *in-fset-mset-mset*[*simp*]: ⟨$x \mid\in\mid$ *fset-mset N* $\longleftrightarrow x \in\# N$⟩

⟨*proof*⟩

4.2 Finite map and multisets

Roughly the same as *ran* and *dom*, but with duplication in the content (unlike their finite sets counterpart) while still working on finite domains (unlike a function mapping). Remark that *dom-m* (the keys) does not contain duplicates, but we keep for symmetry (and for easier use of multiset operators as in the definition of *ran-m*).

definition *dom-m* **where**
  ‹*dom-m N = mset-fset (fmdom N)*›

definition *ran-m* **where**
  ‹*ran-m N = the '# fmlookup N '# dom-m N*›

lemma *dom-m-fmdrop*[*simp*]: ‹*dom-m (fmdrop C N) = remove1-mset C (dom-m N)*›
  ⟨*proof*⟩

lemma *dom-m-fmdrop-All*: ‹*dom-m (fmdrop C N) = removeAll-mset C (dom-m N)*›
  ⟨*proof*⟩

lemma *dom-m-fmupd*[*simp*]: ‹*dom-m (fmupd k C N) = add-mset k (remove1-mset k (dom-m N))*›
  ⟨*proof*⟩

lemma *distinct-mset-dom*: ‹*distinct-mset (dom-m N)*›
  ⟨*proof*⟩

lemma *in-dom-m-lookup-iff*: ‹*C ∈# dom-m N' ⟷ fmlookup N' C ≠ None*›
  ⟨*proof*⟩

lemma *in-dom-in-ran-m*[*simp*]: ‹*i ∈# dom-m N ⟹ the (fmlookup N i) ∈# ran-m N*›
  ⟨*proof*⟩

lemma *fmupd-same*[*simp*]:
  ‹*x1 ∈# dom-m x1aa ⟹ fmupd x1 (the (fmlookup x1aa x1)) x1aa = x1aa*›
  ⟨*proof*⟩

lemma *ran-m-fmempty*[*simp*]: ‹*ran-m fmempty = {#}*› **and**
   dom-m-fmempty[*simp*]: ‹*dom-m fmempty = {#}*›
  ⟨*proof*⟩

lemma *fmrestrict-set-fmupd*:
  ‹*a ∈ xs ⟹ fmrestrict-set xs (fmupd a C N) = fmupd a C (fmrestrict-set xs N)*›
  ‹*a ∉ xs ⟹ fmrestrict-set xs (fmupd a C N) = fmrestrict-set xs N*›
  ⟨*proof*⟩

lemma *fset-fmdom-fmrestrict-set*:
  ‹*fset (fmdom (fmrestrict-set xs N)) = fset (fmdom N) ∩ xs*›
  ⟨*proof*⟩

lemma *dom-m-fmrestrict-set*: ‹*dom-m (fmrestrict-set (set xs) N) = mset xs ∩# dom-m N*›
  ⟨*proof*⟩

lemma *dom-m-fmrestrict-set'*: ‹*dom-m (fmrestrict-set xs N) = mset-set (xs ∩ set-mset (dom-m N))*›
  ⟨*proof*⟩

lemma *indom-mI*: ‹*fmlookup m x = Some y* ⟹ *x* ∈# *dom-m m*›
  ⟨*proof*⟩

lemma *fmupd-fmdrop-id*:
  assumes ‹*k* |∈| *fmdom N′*›
  shows ‹*fmupd k (the (fmlookup N′ k)) (fmdrop k N′) = N′*›
⟨*proof*⟩

lemma *fm-member-split*: ‹*k* |∈| *fmdom N′* ⟹ ∃ *N′′ v. N′ = fmupd k v N′′* ∧ *the (fmlookup N′ k) = v*
∧
    k |∉| *fmdom N′′*›
  ⟨*proof*⟩

lemma ‹*fmdrop k (fmupd k va N′′) = fmdrop k N′′*›
  ⟨*proof*⟩

lemma *fmap-ext-fmdom*:
  ‹(*fmdom N = fmdom N′*) ⟹ (⋀ *x. x* |∈| *fmdom N* ⟹ *fmlookup N x = fmlookup N′ x*) ⟹
    N = N′›
  ⟨*proof*⟩

lemma *fmrestrict-set-insert-in*:
  ‹*xa* ∈ *fset (fmdom N)* ⟹
    fmrestrict-set (insert xa l1) N = fmupd xa (the (fmlookup N xa)) (fmrestrict-set l1 N)›
  ⟨*proof*⟩

lemma *fmrestrict-set-insert-notin*:
  ‹*xa* ∉ *fset (fmdom N)* ⟹
    fmrestrict-set (insert xa l1) N = fmrestrict-set l1 N›
  ⟨*proof*⟩

lemma *fmrestrict-set-insert-in-dom-m*[*simp*]:
  ‹*xa* ∈# *dom-m N* ⟹
    fmrestrict-set (insert xa l1) N = fmupd xa (the (fmlookup N xa)) (fmrestrict-set l1 N)›
  ⟨*proof*⟩

lemma *fmrestrict-set-insert-notin-dom-m*[*simp*]:
  ‹*xa* ∉# *dom-m N* ⟹
    fmrestrict-set (insert xa l1) N = fmrestrict-set l1 N›
  ⟨*proof*⟩

lemma *fmlookup-restrict-set-id*: ‹*fset (fmdom N)* ⊆ *A* ⟹ *fmrestrict-set A N = N*›
  ⟨*proof*⟩

lemma *fmlookup-restrict-set-id′*: ‹*set-mset (dom-m N)* ⊆ *A* ⟹ *fmrestrict-set A N = N*›
  ⟨*proof*⟩

lemma *ran-m-mapsto-upd*:
  assumes
    NC: ‹*C* ∈# *dom-m N*›
  shows ‹*ran-m (fmupd C C′ N) =*
      add-mset C′ (remove1-mset (the (fmlookup N C)) (ran-m N))›
⟨*proof*⟩

lemma *ran-m-mapsto-upd-notin*:

30

assumes *NC*: ‹*C* ∉# *dom-m N*›
  shows ‹*ran-m (fmupd C C′ N) = add-mset C′ (ran-m N)*›
  ⟨*proof*⟩

lemma *image-mset-If-eq-notin*:
  ‹*C* ∉# *A* ⟹ {#*f (if x = C then a x else b x). x* ∈# *A*#} = {# *f(b x). x* ∈# *A* #}›
  ⟨*proof*⟩

lemma *filter-mset-cong2*:
  (⋀*x. x* ∈# *M* ⟹ *f x = g x*) ⟹ *M = N* ⟹ *filter-mset f M = filter-mset g N*
  ⟨*proof*⟩

lemma *ran-m-fmdrop*:
  ‹*C* ∈# *dom-m N* ⟹ *ran-m (fmdrop C N) = remove1-mset (the (fmlookup N C)) (ran-m N)*›
  ⟨*proof*⟩

lemma *ran-m-fmdrop-notin*:
  ‹*C* ∉# *dom-m N* ⟹ *ran-m (fmdrop C N) = ran-m N*›
  ⟨*proof*⟩

lemma *ran-m-fmdrop-If*:
  ‹*ran-m (fmdrop C N) = (if C* ∈# *dom-m N then remove1-mset (the (fmlookup N C)) (ran-m N) else
ran-m N)*›
  ⟨*proof*⟩

lemma *dom-m-empty-iff* [*iff*]:
  ‹*dom-m NU* = {#} ⟷ *NU = fmempty*›
  ⟨*proof*⟩

end
theory *PAC-Map-Rel*
  imports
    Refine-Imperative-HOL.IICF Finite-Map-Multiset
begin

5   Hash-Map for finite mappings

This function declares hash-maps for (′*a*, ′*b*) *fmap*, that are nicer to use especially here where
everything is finite.

definition *fmap-rel* **where**
  [*to-relAPP*]:
  fmap-rel K V ≡ {(*m1, m2*).
    (∀ *i j. i* |∈| *fmdom m2* ⟶ (*j, i*) ∈ *K* ⟶ (*the (fmlookup m1 j), the (fmlookup m2 i)*) ∈ *V*) ∧
    fset (fmdom m1) ⊆ *Domain K* ∧ *fset (fmdom m2)* ⊆ *Range K* ∧
    (∀ *i j. (i, j)* ∈ *K* ⟶ *j* |∈| *fmdom m2* ⟷ *i* |∈| *fmdom m1*)}

lemma *fmap-rel-alt-def*:
  ‹⟨*K, V*⟩*fmap-rel* ≡
    {(*m1, m2*).
      (∀ *i j. i* ∈# *dom-m m2* ⟶
          (*j, i*) ∈ *K* ⟶ (*the (fmlookup m1 j), the (fmlookup m2 i)*) ∈ *V*) ∧
      fset (fmdom m1) ⊆ *Domain K* ∧
      fset (fmdom m2) ⊆ *Range K* ∧

$$(\forall\ i\ j.\ (i,\ j)\ \in\ K\ \longrightarrow\ (j\ \in\#\ dom\text{-}m\ m2)\ =\ (i\ \in\#\ dom\text{-}m\ m1))\}$$

$\rangle$

⟨*proof*⟩

lemma *fmap-rel-empty1-simp*[*simp*]:
  $(fmempty,m)\in\langle K,V\rangle fmap\text{-}rel\ \longleftrightarrow\ m{=}fmempty$
  ⟨*proof*⟩

lemma *fmap-rel-empty2-simp*[*simp*]:
  $(m,fmempty)\in\langle K,V\rangle fmap\text{-}rel\ \longleftrightarrow\ m{=}fmempty$
  ⟨*proof*⟩

sepref-decl-intf $('k,'v)$ *f-map* **is** $('k,\ 'v)$ *fmap*

lemma [*synth-rules*]: $\llbracket INTF\text{-}OF\text{-}REL\ K\ TYPE('k);\ INTF\text{-}OF\text{-}REL\ V\ TYPE('v)\rrbracket$
  $\implies INTF\text{-}OF\text{-}REL\ (\langle K,V\rangle fmap\text{-}rel)\ TYPE(('k,'v)\ f\text{-}map)$ ⟨*proof*⟩

5.1   Operations

  sepref-decl-op *fmap-empty*: $fmempty :: \langle K,V\rangle fmap\text{-}rel$ ⟨*proof*⟩

  sepref-decl-op *fmap-is-empty*: $(=)\ fmempty :: \langle K,V\rangle fmap\text{-}rel\ \to\ bool\text{-}rel$
    ⟨*proof*⟩

lemma *fmap-rel-fmupd-fmap-rel*:
  $\langle (A,\ B)\ \in\ \langle K,\ R\rangle fmap\text{-}rel\ \implies\ (p,\ p')\ \in\ K\ \implies\ (q,\ q')\ \in\ R\ \implies$
  $(fmupd\ p\ q\ A,\ fmupd\ p'\ q'\ B)\ \in\ \langle K,\ R\rangle fmap\text{-}rel\rangle$
  if *single-valued K single-valued* (K^{-1})
  ⟨*proof*⟩

  sepref-decl-op *fmap-update*: $fmupd :: K\ \to\ V\ \to\ \langle K,V\rangle fmap\text{-}rel\ \to\ \langle K,V\rangle fmap\text{-}rel$
    where *single-valued K single-valued* (K^{-1})
    ⟨*proof*⟩

lemma *fmap-rel-fmdrop-fmap-rel*:
  $\langle (A,\ B)\ \in\ \langle K,\ R\rangle fmap\text{-}rel\ \implies\ (p,\ p')\ \in\ K\ \implies$
  $(fmdrop\ p\ A,\ fmdrop\ p'\ B)\ \in\ \langle K,\ R\rangle fmap\text{-}rel\rangle$
  if *single-valued K single-valued* (K^{-1})
  ⟨*proof*⟩

  sepref-decl-op *fmap-delete*: $fmdrop :: K\ \to\ \langle K,V\rangle fmap\text{-}rel\ \to\ \langle K,V\rangle fmap\text{-}rel$
    where *single-valued K single-valued* (K^{-1})
    ⟨*proof*⟩

  lemma *fmap-rel-nat-the-fmlookup*[*intro*]:
    $\langle (A,\ B)\ \in\ \langle S,\ R\rangle fmap\text{-}rel\ \implies\ (p,\ p')\ \in\ S\ \implies\ p'\ \in\#\ dom\text{-}m\ B\ \implies$
    $(the\ (fmlookup\ A\ p),\ the\ (fmlookup\ B\ p'))\ \in\ R\rangle$
    ⟨*proof*⟩

  lemma *fmap-rel-in-dom-iff*:
    $\langle (aa,\ a'a)\ \in\ \langle K,\ V\rangle fmap\text{-}rel\ \implies$
    $(a,\ a')\ \in\ K\ \implies$
    $a'\ \in\#\ dom\text{-}m\ a'a\ \longleftrightarrow$

a ∈# *dom-m aa*⟩
⟨*proof*⟩

lemma *fmap-rel-fmlookup-rel*:
⟨(*a*, *a*′) ∈ *K* ⟹ (*aa*, *a*′*a*) ∈ ⟨*K*, *V*⟩*fmap-rel* ⟹
   (*fmlookup aa a*, *fmlookup a*′*a a*′) ∈ ⟨*V*⟩*option-rel*⟩
⟨*proof*⟩

sepref-decl-op *fmap-lookup*: *fmlookup* :: ⟨*K*,*V*⟩*fmap-rel* → *K* → ⟨*V*⟩*option-rel*
⟨*proof*⟩

lemma *in-fdom-alt*: *k*∈#*dom-m m* ⟷ ¬*is-None* (*fmlookup m k*)
⟨*proof*⟩

sepref-decl-op *fmap-contains-key*: λ*k m*. *k*∈#*dom-m m* :: *K* → ⟨*K*,*V*⟩*fmap-rel* → *bool-rel*
⟨*proof*⟩

5.2   Patterns

lemma *pat-fmap-empty*[*pat-rules*]: *fmempty* ≡ *op-fmap-empty* ⟨*proof*⟩

lemma *pat-map-is-empty*[*pat-rules*]:
 (=) $*m*$*fmempty* ≡ *op-fmap-is-empty*$*m*
 (=) $*fmempty*$*m* ≡ *op-fmap-is-empty*$*m*
 (=) $(*dom-m*$*m*)${#} ≡ *op-fmap-is-empty*$*m*
 (=) ${#}$(*dom-m*$*m*) ≡ *op-fmap-is-empty*$*m*
 ⟨*proof*⟩

lemma *op-map-contains-key*[*pat-rules*]:
 (∈#) $ *k* $ (*dom-m*$*m*) ≡ *op-fmap-contains-key*$′*k*$′*m*
  ⟨*proof*⟩

5.3   Mapping to Normal Hashmaps

abbreviation *map-of-fmap* :: ⟨(′*k* ⟹ ′*v option*) ⟹ (′*k*, ′*v*) *fmap*⟩ **where**
⟨*map-of-fmap h* ≡ *Abs-fmap h*⟩

definition *map-fmap-rel* **where**
 ⟨*map-fmap-rel* = *br map-of-fmap* (λ*a*. *finite* (*dom a*))⟩

lemma *fmdrop-set-None*:
 ⟨(*op-map-delete*, *fmdrop*) ∈ *Id* → *map-fmap-rel* → *map-fmap-rel*⟩
 ⟨*proof*⟩

lemma *map-upd-fmupd*:
 ⟨(*op-map-update*, *fmupd*) ∈ *Id* → *Id* → *map-fmap-rel* → *map-fmap-rel*⟩
 ⟨*proof*⟩

Technically *op-map-lookup* has the arguments in the wrong direction.

definition *fmlookup*′ **where**
 [*simp*]: ⟨*fmlookup*′ *A k* = *fmlookup k A*⟩

lemma [*def-pat-rules*]:
 ⟨((∈#)$*k*$(*dom-m*$*A*)) ≡ *Not*$(*is-None*$(*fmlookup*′$*k*$*A*))⟩

⟨*proof*⟩

lemma *op-map-lookup-fmlookup*:
⟨(*op-map-lookup*, *fmlookup′*) ∈ *Id* → *map-fmap-rel* → ⟨*Id*⟩*option-rel*⟩
⟨*proof*⟩

abbreviation *hm-fmap-assn* **where**
⟨*hm-fmap-assn K V* ≡ *hr-comp* (*hm.assn K V*) *map-fmap-rel*⟩

lemmas *fmap-delete-hnr* [*sepref-fr-rules*] =
  hm.delete-hnr[*FCOMP fmdrop-set-None*]

lemmas *fmap-update-hnr* [*sepref-fr-rules*] =
  hm.update-hnr[*FCOMP map-upd-fmupd*]

lemmas *fmap-lookup-hnr* [*sepref-fr-rules*] =
  hm.lookup-hnr[*FCOMP op-map-lookup-fmlookup*]

lemma *fmempty-empty*:
⟨(*uncurry0* (*RETURN op-map-empty*), *uncurry0* (*RETURN fmempty*)) ∈ *unit-rel* →$_f$ ⟨*map-fmap-rel*⟩*nres-rel*⟩
⟨*proof*⟩

lemmas [*sepref-fr-rules*] =
  hm.empty-hnr[*FCOMP fmempty-empty*, *unfolded op-fmap-empty-def*[*symmetric*]]

abbreviation *iam-fmap-assn* **where**
⟨*iam-fmap-assn K V* ≡ *hr-comp* (*iam.assn K V*) *map-fmap-rel*⟩

lemmas *iam-fmap-delete-hnr* [*sepref-fr-rules*] =
  iam.delete-hnr[*FCOMP fmdrop-set-None*]

lemmas *iam-ffmap-update-hnr* [*sepref-fr-rules*] =
  iam.update-hnr[*FCOMP map-upd-fmupd*]

lemmas *iam-ffmap-lookup-hnr* [*sepref-fr-rules*] =
  iam.lookup-hnr[*FCOMP op-map-lookup-fmlookup*]

definition *op-iam-fmap-empty* **where**
⟨*op-iam-fmap-empty* = *fmempty*⟩

lemma *iam-fmempty-empty*:
⟨(*uncurry0* (*RETURN op-map-empty*), *uncurry0* (*RETURN op-iam-fmap-empty*)) ∈ *unit-rel* →$_f$
⟨*map-fmap-rel*⟩*nres-rel*⟩
⟨*proof*⟩

lemmas [*sepref-fr-rules*] =
  iam.empty-hnr[*FCOMP fmempty-empty*, *unfolded op-iam-fmap-empty-def*[*symmetric*]]

definition *upper-bound-on-dom* **where**
⟨*upper-bound-on-dom A* = *SPEC*(λ*n*. ∀ *i* ∈#(*dom-m A*). *i* < *n*)⟩

lemma [*sepref-fr-rules*]:
  ‹((*Array.len*), *upper-bound-on-dom*) ∈ (*iam-fmap-assn nat-assn V*)^k →$_a$ *nat-assn*›
⟨*proof*⟩

lemma *fmap-rel-nat-rel-dom-m*[*simp*]:
  ‹(*A*, *B*) ∈ ⟨*nat-rel*, *R*⟩*fmap-rel* ⟹ *dom-m A* = *dom-m B*›
  ⟨*proof*⟩

lemma *ref-two-step′*:
  ‹*A* ≤ *B* ⟹ ⇓ *R A* ≤ ⇓ *R B*›
  ⟨*proof*⟩

end
theory *PAC-Checker-Specification*
  imports *PAC-Specification*
    Refine-Imperative-HOL.IICF
    Finite-Map-Multiset
begin

6   Checker Algorithm

In this level of refinement, we define the first level of the implementation of the checker, both
with the specification as on ideals and the first version of the loop.

6.1   Specification

datatype *status* =
  is-failed: *FAILED* |
  is-success: *SUCCESS* |
  is-found: *FOUND*

lemma *is-success-alt-def*:
  ‹*is-success a* ⟷ *a* = *SUCCESS*›
  ⟨*proof*⟩

datatype (′*a*, ′*b*, ′*lbls*) *pac-step* =
  Add (*pac-src1*: ′*lbls*) (*pac-src2*: ′*lbls*) (*new-id*: ′*lbls*) (*pac-res*: ′*a*) |
  Mult (*pac-src1*: ′*lbls*) (*pac-mult*: ′*a*) (*new-id*: ′*lbls*) (*pac-res*: ′*a*) |
  Extension (*new-id*: ′*lbls*) (*new-var*: ′*b*) (*pac-res*: ′*a*) |
  Del (*pac-src1*: ′*lbls*)

type-synonym *pac-state* = ‹(*nat set* × *int-poly multiset*)›

definition *PAC-checker-specification*
  :: ‹*int-poly* ⟹ *int-poly multiset* ⟹ (*status* × *nat set* × *int-poly multiset*) *nres*›
where
  ‹*PAC-checker-specification spec A* = *SPEC*(λ(*b*, 𝒱, *B*).
      (¬*is-failed b* ⟶ *restricted-ideal-to*$_I$ (⋃(*vars* ' *set-mset A*) ∪ *vars spec*) *B* ⊆ *restricted-ideal-to*$_I$
(⋃(*vars* ' *set-mset A*) ∪ *vars spec*) *A*) ∧
      (*is-found b* ⟶ *spec* ∈ *pac-ideal* (*set-mset A*)))›

definition *PAC-checker-specification-spec*

:: ⟨*int-poly* ⇒ *pac-state* ⇒ (*status* × *pac-state*) ⇒ *bool*⟩
where
⟨*PAC-checker-specification-spec spec* = (λ(𝒱, *A*) (*b*, *B*). (¬*is-failed b* ⟶ ⋃(*vars ' set-mset A*) ⊆ 𝒱) ∧
  (*is-success b* ⟶ *PAC-Format*** (𝒱, *A*) *B*) ∧
  (*is-found b* ⟶ *PAC-Format*** (𝒱, *A*) *B* ∧ *spec* ∈ *pac-ideal* (*set-mset A*)))⟩

abbreviation *PAC-checker-specification2*
:: ⟨*int-poly* ⇒ (*nat set* × *int-poly multiset*) ⇒ (*status* × (*nat set* × *int-poly multiset*)) *nres*⟩
where
⟨*PAC-checker-specification2 spec A* ≡ *SPEC*(*PAC-checker-specification-spec spec A*)⟩

definition *PAC-checker-specification-step-spec*
:: ⟨*pac-state* ⇒ *int-poly* ⇒ *pac-state* ⇒ (*status* × *pac-state*) ⇒ *bool*⟩
where
⟨*PAC-checker-specification-step-spec* = (λ($\mathcal{V}_0$, A_0) *spec* (𝒱, *A*) (*b*, *B*).
  (*is-success b* ⟶
    ⋃(*vars ' set-mset* A_0) ⊆ $\mathcal{V}_0$ ∧
    ⋃(*vars ' set-mset A*) ⊆ 𝒱 ∧ *PAC-Format*** ($\mathcal{V}_0$, A_0) (𝒱, *A*) ∧ *PAC-Format*** (𝒱, *A*) *B*) ∧
  (*is-found b* ⟶
    ⋃(*vars ' set-mset* A_0) ⊆ $\mathcal{V}_0$ ∧
    ⋃(*vars ' set-mset A*) ⊆ 𝒱 ∧ *PAC-Format*** ($\mathcal{V}_0$, A_0) (𝒱, *A*) ∧ *PAC-Format*** (𝒱, *A*) *B* ∧
    spec ∈ *pac-ideal* (*set-mset* A_0)))⟩

abbreviation *PAC-checker-specification-step2*
:: ⟨*pac-state* ⇒ *int-poly* ⇒ *pac-state* ⇒ (*status* × *pac-state*) *nres*⟩
where
⟨*PAC-checker-specification-step2* A_0 *spec A* ≡ *SPEC*(*PAC-checker-specification-step-spec* A_0 *spec A*)⟩

definition *normalize-poly-spec* :: ⟨-⟩ **where**
⟨*normalize-poly-spec p* = *SPEC* (λ*r*. *p* − *r* ∈ *ideal polynomial-bool* ∧ *vars r* ⊆ *vars p*)⟩

lemma *normalize-poly-spec-alt-def*:
⟨*normalize-poly-spec p* = *SPEC* (λ*r*. *r* − *p* ∈ *ideal polynomial-bool* ∧ *vars r* ⊆ *vars p*)⟩
⟨*proof*⟩

definition *mult-poly-spec* :: ⟨*int mpoly* ⇒ *int mpoly* ⇒ *int mpoly nres*⟩ **where**
⟨*mult-poly-spec p q* = *SPEC* (λ*r*. *p* ∗ *q* − *r* ∈ *ideal polynomial-bool*)⟩

definition *check-add* :: ⟨(*nat*, *int mpoly*) *fmap* ⇒ *nat set* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *int mpoly* ⇒ *bool nres*⟩ **where**
⟨*check-add A* 𝒱 *p q i r* =
  SPEC(λ*b*. *b* ⟶ *p* ∈# *dom-m A* ∧ *q* ∈# *dom-m A* ∧ *i* ∉# *dom-m A* ∧ *vars r* ⊆ 𝒱 ∧
    the (*fmlookup A p*) + *the* (*fmlookup A q*) − *r* ∈ *ideal polynomial-bool*)⟩

definition *check-mult* :: ⟨(*nat*, *int mpoly*) *fmap* ⇒ *nat set* ⇒ *nat* ⇒ *int mpoly* ⇒ *nat* ⇒ *int mpoly* ⇒ *bool nres*⟩ **where**
⟨*check-mult A* 𝒱 *p q i r* =
  SPEC(λ*b*. *b* ⟶ *p* ∈# *dom-m A* ∧*i* ∉# *dom-m A* ∧ *vars q* ⊆ 𝒱 ∧ *vars r* ⊆ 𝒱 ∧
    the (*fmlookup A p*) ∗ *q* − *r* ∈ *ideal polynomial-bool*)⟩

definition *check-extension* :: ⟨(*nat*, *int mpoly*) *fmap* ⇒ *nat set* ⇒ *nat* ⇒ *nat* ⇒ *int mpoly* ⇒ (*bool*) *nres*⟩ **where**
⟨*check-extension A* 𝒱 *i v p* =

$SPEC(\lambda b.\ b \longrightarrow (i \notin\#\ dom\text{-}m\ A\ \wedge$
$(v \notin \mathcal{V}\ \wedge$
   $(p+Var\ v)^2 - (p+Var\ v) \in ideal\ polynomial\text{-}bool\ \wedge$
     $vars\ (p+Var\ v) \subseteq \mathcal{V})))\rangle$

fun *merge-status* **where**
⟨*merge-status* (*FAILED*) *-* = *FAILED*⟩ |
⟨*merge-status* *-* (*FAILED*) = *FAILED*⟩ |
⟨*merge-status* *FOUND* *-* = *FOUND*⟩ |
⟨*merge-status* *-* *FOUND* = *FOUND*⟩ |
⟨*merge-status* *-* *-* = *SUCCESS*⟩

type-synonym *fpac-step* = ⟨*nat set* × (*nat*, *int-poly*) *fmap*⟩

definition *check-del* :: ⟨(*nat*, *int mpoly*) *fmap* ⇒ *nat* ⇒ *bool nres*⟩ **where**
⟨*check-del* *A* *p* =
   $SPEC(\lambda b.\ b \longrightarrow True)\rangle$

6.2   Algorithm

definition *PAC-checker-step*
  :: ⟨*int-poly* ⇒ (*status* × *fpac-step*) ⇒ (*int-poly*, *nat*, *nat*) *pac-step* ⇒
    (*status* × *fpac-step*) *nres*⟩
where
⟨*PAC-checker-step* = (λ*spec* (*stat*, ($\mathcal{V}$, *A*)) *st*. *case st of*
   Add *- - - -* ⇒
     do {
       r ← *normalize-poly-spec* (*pac-res st*);
       eq ← *check-add* *A* $\mathcal{V}$ (*pac-src1 st*) (*pac-src2 st*) (*new-id st*) *r*;
       st′ ← $SPEC(\lambda st'.\ (\neg is\text{-}failed\ st' \wedge is\text{-}found\ st' \longrightarrow r - spec \in ideal\ polynomial\text{-}bool))$;
       if eq
       then RETURN (*merge-status stat st′*,
        $\mathcal{V}$, *fmupd* (*new-id st*) *r* *A*)
       else RETURN (*FAILED*, ($\mathcal{V}$, *A*))
   }
  | *Del* *-* ⇒
     do {
       eq ← *check-del* *A* (*pac-src1 st*);
       if eq
       then RETURN (*stat*, ($\mathcal{V}$, *fmdrop* (*pac-src1 st*) *A*))
       else RETURN (*FAILED*, ($\mathcal{V}$, *A*))
   }
  | *Mult* *- - - -* ⇒
     do {
       r ← *normalize-poly-spec* (*pac-res st*);
       q ← *normalize-poly-spec* (*pac-mult st*);
       eq ← *check-mult* *A* $\mathcal{V}$ (*pac-src1 st*) *q* (*new-id st*) *r*;
       st′ ← $SPEC(\lambda st'.\ (\neg is\text{-}failed\ st' \wedge is\text{-}found\ st' \longrightarrow r - spec \in ideal\ polynomial\text{-}bool))$;
       if eq
       then RETURN (*merge-status stat st′*,
        $\mathcal{V}$, *fmupd* (*new-id st*) *r* *A*)
       else RETURN (*FAILED*, ($\mathcal{V}$, *A*))
   }
  | *Extension* *- - -* ⇒
     do {
       r ← *normalize-poly-spec* (*pac-res st* − *Var* (*new-var st*));

37

```
        (eq) ← check-extension A V (new-id st) (new-var st) r;
        if eq
        then do {
         RETURN (stat,
          insert (new-var st) V, fmupd (new-id st) (r) A)}
        else RETURN (FAILED, (V, A))
   }
)›
```

definition *polys-rel* :: ‹*((nat, int mpoly)fmap × -) set*› **where**
‹*polys-rel = {(A, B). B = (ran-m A)}*›

definition *polys-rel-full* :: ‹*((nat set × (nat, int mpoly)fmap) × -) set*› **where**
‹*polys-rel-full = {((V, A), (V′, B)). (A, B) ∈ polys-rel ∧ V = V′}*›

lemma *polys-rel-update-remove*:
‹*x13 ∉# dom-m A ⟹ x11 ∈# dom-m A ⟹ x12 ∈# dom-m A ⟹ x11 ≠ x12 ⟹ (A,B) ∈ polys-rel
⟹*
  (fmupd x13 r (fmdrop x11 (fmdrop x12 A)),
      add-mset r B − {#the (fmlookup A x11), the (fmlookup A x12)#})
      ∈ polys-rel›
‹*x13 ∉# dom-m A ⟹ x11 ∈# dom-m A ⟹ (A,B) ∈ polys-rel ⟹*
  (fmupd x13 r (fmdrop x11 A),add-mset r B − {#the (fmlookup A x11)#})
      ∈ polys-rel›
‹*x13 ∉# dom-m A ⟹ (A,B) ∈ polys-rel ⟹*
  (fmupd x13 r A, add-mset r B) ∈ polys-rel›
‹*x13 ∈# dom-m A ⟹ (A,B) ∈ polys-rel ⟹*
  (fmdrop x13 A, remove1-mset (the (fmlookup A x13)) B) ∈ polys-rel›
⟨*proof*⟩

lemma *polys-rel-in-dom-inD*:
‹*(A, B) ∈ polys-rel ⟹*
  x12 ∈# dom-m A ⟹
  the (fmlookup A x12) ∈# B›
⟨*proof*⟩

lemma *PAC-Format-add-and-remove*:
‹*r − x14 ∈ More-Modules.ideal polynomial-bool ⟹*
    (A, B) ∈ polys-rel ⟹
    x12 ∈# dom-m A ⟹
    x13 ∉# dom-m A ⟹
    vars r ⊆ V ⟹
    2 ∗ the (fmlookup A x12) − r ∈ More-Modules.ideal polynomial-bool ⟹
    PAC-Format** (V, B) (V, remove1-mset (the (fmlookup A x12)) (add-mset r B))›
‹*r − x14 ∈ More-Modules.ideal polynomial-bool ⟹*
    (A, B) ∈ polys-rel ⟹
    the (fmlookup A x11) + the (fmlookup A x12) − r ∈ More-Modules.ideal polynomial-bool ⟹
    x11 ∈# dom-m A ⟹
    x12 ∈# dom-m A ⟹
    vars r ⊆ V ⟹
    PAC-Format** (V, B) (V, add-mset r B)›
‹*r − x14 ∈ More-Modules.ideal polynomial-bool ⟹*
    (A, B) ∈ polys-rel ⟹
    x11 ∈# dom-m A ⟹
    x12 ∈# dom-m A ⟹

38
```

      *the* (*fmlookup A x11*) + *the* (*fmlookup A x12*) − *r* ∈ *More-Modules.ideal polynomial-bool* ⟹
      *vars r* ⊆ 𝒱 ⟹
      *x11* ≠ *x12* ⟹
      *PAC-Format** (𝒱, B)*
      (𝒱, *add-mset r B* − {#*the* (*fmlookup A x11*), *the* (*fmlookup A x12*)#})⟩
   ⟨(*A, B*) ∈ *polys-rel* ⟹
      *r* − *x34* ∈ *More-Modules.ideal polynomial-bool* ⟹
      *x11* ∈# *dom-m A* ⟹
      *the* (*fmlookup A x11*) ∗ *x32* − *r* ∈ *More-Modules.ideal polynomial-bool* ⟹
      *vars x32* ⊆ 𝒱 ⟹
      *vars r* ⊆ 𝒱 ⟹
      *PAC-Format** (𝒱, B) (𝒱, add-mset r B*)⟩
   ⟨(*A, B*) ∈ *polys-rel* ⟹
      *r* − *x34* ∈ *More-Modules.ideal polynomial-bool* ⟹
      *x11* ∈# *dom-m A* ⟹
      *the* (*fmlookup A x11*) ∗ *x32* − *r* ∈ *More-Modules.ideal polynomial-bool* ⟹
      *vars x32* ⊆ 𝒱 ⟹
      *vars r* ⊆ 𝒱 ⟹
      *PAC-Format** (𝒱, B) (𝒱, remove1-mset* (*the* (*fmlookup A x11*)) (*add-mset r B*))⟩
   ⟨(*A, B*) ∈ *polys-rel* ⟹
      *x12* ∈# *dom-m A* ⟹
      *PAC-Format** (𝒱, B) (𝒱, remove1-mset* (*the* (*fmlookup A x12*)) *B*)⟩
   ⟨(*A, B*) ∈ *polys-rel* ⟹
      $(p' + Var\ x)^2 - (p' + Var\ x)$ ∈ *ideal polynomial-bool* ⟹
      *x* ∉ 𝒱 ⟹
      *x* ∉ *vars*(*p' + Var x*) ⟹
      *vars*(*p' + Var x*) ⊆ 𝒱 ⟹
      *PAC-Format** (𝒱, B)*
        (*insert x 𝒱, add-mset p' B*)⟩
  ⟨*proof*⟩


**abbreviation** *status-rel* :: ⟨(*status* × *status*) *set*⟩ **where**
  ⟨*status-rel* ≡ *Id*⟩

**lemma** *is-merge-status*[*simp*]:
  ⟨*is-failed* (*merge-status a st'*) ⟷ *is-failed a* ∨ *is-failed st'*⟩
  ⟨*is-found* (*merge-status a st'*) ⟷ ¬*is-failed a* ∧ ¬*is-failed st'* ∧ (*is-found a* ∨ *is-found st'*)⟩
  ⟨*is-success* (*merge-status a st'*) ⟷ (*is-success a* ∧ *is-success st'*)⟩
  ⟨*proof*⟩

**lemma** *status-rel-merge-status*:
  ⟨(*merge-status a b, SUCCESS*) ∉ *status-rel* ⟷
   (*a* = *FAILED*) ∨ (*b* = *FAILED*) ∨
   *a* = *FOUND* ∨ (*b* = *FOUND*)⟩
  ⟨*proof*⟩

**lemma** *Ex-status-iff*:
  ⟨(∃ *a. P a*) ⟷ *P SUCCESS* ∨ *P FOUND* ∨ (*P* (*FAILED*))⟩
  ⟨*proof*⟩

**lemma** *is-failed-alt-def*:
  ⟨*is-failed st'* ⟷ ¬*is-success st'* ∧ ¬*is-found st'*⟩
  ⟨*proof*⟩

**lemma** *merge-status-eq-iff* [*simp*]:
 ‹*merge-status a SUCCESS = SUCCESS* $\longleftrightarrow$ *a = SUCCESS*›
 ‹*merge-status a SUCCESS = FOUND* $\longleftrightarrow$ *a = FOUND*›
 ‹*merge-status SUCCESS a = SUCCESS* $\longleftrightarrow$ *a = SUCCESS*›
 ‹*merge-status SUCCESS a = FOUND* $\longleftrightarrow$ *a = FOUND*›
 ‹*merge-status SUCCESS a = FAILED* $\longleftrightarrow$ *a = FAILED*›
 ‹*merge-status a SUCCESS = FAILED* $\longleftrightarrow$ *a = FAILED*›
 ‹*merge-status FOUND a = FAILED* $\longleftrightarrow$ *a = FAILED*›
 ‹*merge-status a FOUND = FAILED* $\longleftrightarrow$ *a = FAILED*›
 ‹*merge-status a FOUND = SUCCESS* $\longleftrightarrow$ *False*›
 ‹*merge-status a b = FOUND* $\longleftrightarrow$ *(a = FOUND* $\lor$ *b = FOUND)* $\land$ *(a* $\neq$ *FAILED* $\land$ *b* $\neq$ *FAILED)*›
 ‹*proof*›

**lemma** *fmdrop-irrelevant*: ‹*x11* $\notin\#$ *dom-m A* $\implies$ *fmdrop x11 A = A*›
 ‹*proof*›

**lemma** *PAC-checker-step-PAC-checker-specification2*:
 **fixes** *a* :: ‹*status*›
 **assumes** *AB*: ‹$((\mathcal{V}, A),(\mathcal{V}_B, B)) \in$ *polys-rel-full*› **and**
  ‹$\neg$*is-failed a*› **and**
  [*simp,intro*]: ‹*a = FOUND* $\implies$ *spec* $\in$ *pac-ideal (set-mset $A_0$)*› **and**
  $A_0B$: ‹*PAC-Format**$^{**}$ $(\mathcal{V}_0, A_0)$ $(\mathcal{V}, B)$*› **and**
  *spec$_0$*: ‹*vars spec* $\subseteq \mathcal{V}_0$› **and**
  *vars-$A_0$*: ‹$\bigcup$ *(vars ' set-mset $A_0$)* $\subseteq \mathcal{V}_0$›
 **shows** ‹*PAC-checker-step spec $(a, (\mathcal{V}, A))$ st* $\leq \Downarrow$ *(status-rel* $\times_r$ *polys-rel-full) (PAC-checker-specification-step2*
$(\mathcal{V}_0, A_0)$ *spec $(\mathcal{V}, B))$*›
‹*proof*›


**definition** *PAC-checker*
 :: ‹*int-poly* $\Rightarrow$ *fpac-step* $\Rightarrow$ *status* $\Rightarrow$ *(int-poly, nat, nat) pac-step list* $\Rightarrow$
   *(status* $\times$ *fpac-step) nres*›
**where**
 ‹*PAC-checker spec A b st = do {*
   *(S, -)* $\leftarrow$ *WHILE$_T$*
    *($\lambda((b$ :: status, A :: fpac-step), st). $\neg$is-failed b* $\land$ *st* $\neq$ *[])*
    *($\lambda((bA), st). do {*
       *ASSERT(st* $\neq$ *[]);*
       *S* $\leftarrow$ *PAC-checker-step spec (bA) (hd st);*
       *RETURN (S, tl st)*
     *})*
    *((b, A), st);*
   *RETURN S*
 *}*›


**lemma** *PAC-checker-specification-spec-trans*:
 ‹*PAC-checker-specification-spec spec A (st, x2)* $\implies$
   *PAC-checker-specification-step-spec A spec x2 (st', x1a)* $\implies$
   *PAC-checker-specification-spec spec A (st', x1a)*›
 ‹*proof*›

**lemma** *RES-SPEC-eq*:
 ‹*RES* $\Phi$ = *SPEC($\lambda P. P \in \Phi$)*›
 ‹*proof*›

**lemma** *is-failed-is-success-completeD*:
  ⟨¬ *is-failed x* ⟹ ¬*is-success x* ⟹ *is-found x*⟩
  ⟨*proof*⟩

**lemma** *PAC-checker-PAC-checker-specification2*:
  ⟨(A, B) ∈ *polys-rel-full* ⟹
    ¬*is-failed a* ⟹
    (a = FOUND ⟹ *spec* ∈ *pac-ideal* (*set-mset* (*snd B*))) ⟹
    ⋃ (*vars* ' *set-mset* (*ran-m* (*snd A*))) ⊆ *fst B* ⟹
    *vars spec* ⊆ *fst B* ⟹
  PAC-checker *spec A a st* ≤ ⇓ (*status-rel* ×ᵣ *polys-rel-full*) (*PAC-checker-specification2 spec B*)⟩
  ⟨*proof*⟩


**definition** *remap-polys-polynomial-bool* :: ⟨*int mpoly* ⇒ *nat set* ⇒ (*nat, int-poly*) *fmap* ⇒ (*status* ×
*fpac-step*) *nres*⟩ **where**
⟨*remap-polys-polynomial-bool spec* = (λ𝒱 A.
    SPEC(λ(*st, 𝒱′, A′*). (¬*is-failed st* ⟶
      *dom-m A* = *dom-m A′* ∧
      (∀ i ∈# *dom-m A. the* (*fmlookup A i*) − *the* (*fmlookup A′ i*) ∈ *ideal polynomial-bool*) ∧
      ⋃ (*vars* ' *set-mset* (*ran-m A*)) ⊆ 𝒱′ ∧
      ⋃ (*vars* ' *set-mset* (*ran-m A′*)) ⊆ 𝒱′) ∧
      (*st* = FOUND ⟶ *spec* ∈# *ran-m A′*)))⟩


**definition** *remap-polys-change-all* :: ⟨*int mpoly* ⇒ *nat set* ⇒ (*nat, int-poly*) *fmap* ⇒ (*status* × *fpac-step*)
*nres*⟩ **where**
⟨*remap-polys-change-all spec* = (λ𝒱 A. SPEC (λ(*st, 𝒱′, A′*).
    (¬*is-failed st* ⟶
      *pac-ideal* (*set-mset* (*ran-m A*)) = *pac-ideal* (*set-mset* (*ran-m A′*)) ∧
      ⋃ (*vars* ' *set-mset* (*ran-m A*)) ⊆ 𝒱′ ∧
      ⋃ (*vars* ' *set-mset* (*ran-m A′*)) ⊆ 𝒱′) ∧
      (*st* = FOUND ⟶ *spec* ∈# *ran-m A′*)))⟩


**lemma** *fmap-eq-dom-iff*:
  ⟨A = A′ ⟷ *dom-m A* = *dom-m A′* ∧ (∀ i ∈# *dom-m A. the* (*fmlookup A i*) = *the* (*fmlookup A′ i*))⟩
  ⟨*proof*⟩


**lemma** *ideal-remap-incl*:
  ⟨*finite A′* ⟹ (∀ a′∈A′. ∃ a∈A. a−a′ ∈ B) ⟹ *ideal* (A′ ∪ B) ⊆ *ideal* (A ∪ B)⟩
  ⟨*proof*⟩


**lemma** *pac-ideal-remap-eq*:
  ⟨*dom-m b* = *dom-m ba* ⟹
      ∀ i∈#*dom-m ba*.
        *the* (*fmlookup b i*) − *the* (*fmlookup ba i*)
        ∈ *More-Modules.ideal polynomial-bool* ⟹
    *pac-ideal* ((λx. *the* (*fmlookup b x*)) ' *set-mset* (*dom-m ba*)) = *pac-ideal* ((λx. *the* (*fmlookup ba x*)) '
*set-mset* (*dom-m ba*))⟩
  ⟨*proof*⟩


**lemma** *remap-polys-polynomial-bool-remap-polys-change-all*:
  ⟨*remap-polys-polynomial-bool spec 𝒱 A* ≤ *remap-polys-change-all spec 𝒱 A*⟩
  ⟨*proof*⟩

41

**definition** *remap-polys* :: ‹*int mpoly* ⇒ *nat set* ⇒ (*nat, int-poly*) *fmap* ⇒ (*status* × *fpac-step*) *nres*›
**where**
  ‹*remap-polys spec* = (λ𝒱 *A. do*{
  *dom* ← *SPEC*(λ*dom. set-mset* (*dom-m A*) ⊆ *dom* ∧ *finite dom*);

  *failed* ← *SPEC*(λ-::*bool. True*);
  *if failed*
  *then do* {
      *RETURN* (*FAILED*, 𝒱, *fmempty*)
  }
  *else do* {
    (*b, N*) ← *FOREACH dom*
      (λ*i* (*b*, 𝒱, *A′*).
          *if i* ∈# *dom-m A*
          *then do* {
            *p* ← *SPEC*(λ*p. the* (*fmlookup A i*) − *p* ∈ *ideal polynomial-bool* ∧ *vars p* ⊆ *vars* (*the* (*fmlookup*
*A i*)));
            *eq* ← *SPEC*(λ*eq. eq* ⟶ *p* = *spec*);
            𝒱 ← *SPEC*(λ𝒱′. 𝒱 ∪ *vars* (*the* (*fmlookup A i*)) ⊆ 𝒱′);
            *RETURN*(*b* ∨ *eq*, 𝒱, *fmupd i p A′*)
          } *else RETURN* (*b*, 𝒱, *A′*))
      (*False*, 𝒱, *fmempty*);
      *RETURN* (*if b then FOUND else SUCCESS, N*)
  }
  })›


**lemma** *remap-polys-spec*:
  ‹*remap-polys spec* 𝒱 *A* ≤ *remap-polys-polynomial-bool spec* 𝒱 *A*›
  ⟨*proof*⟩


## 6.3 Full Checker

**definition** *full-checker*
  :: ‹*int-poly* ⇒ (*nat, int-poly*) *fmap* ⇒ (*int-poly, nat,nat*) *pac-step list* ⇒ (*status* × -) *nres*›
  **where**
  ‹*full-checker spec0 A pac* = *do* {
      *spec* ← *normalize-poly-spec spec0*;
      (*st*, 𝒱, *A*) ← *remap-polys-change-all spec* {} *A*;
      *if is-failed st then*
      *RETURN* (*st*, 𝒱, *A*)
      *else do* {
        𝒱 ← *SPEC*(λ𝒱′. 𝒱 ∪ *vars spec0* ⊆ 𝒱′);
        *PAC-checker spec* (𝒱, *A*) *st pac*
    }
}›


**lemma** *restricted-ideal-to-mono*:
  ‹*restricted-ideal-to$_I$* 𝒱 *I* ⊆ *restricted-ideal-to$_I$* 𝒱′ *J* ⟹
  𝒰 ⊆ 𝒱 ⟹
  *restricted-ideal-to$_I$* 𝒰 *I* ⊆ *restricted-ideal-to$_I$* 𝒰  *J*›
  ⟨*proof*⟩


**lemma** *full-checker-spec*:
  **assumes** ‹(*A, A′*) ∈ *polys-rel*›
  **shows**
      ‹*full-checker spec A pac* ≤ ⇓{((*st, G*), (*st′, G′*)). (*st, st′*) ∈ *status-rel* ∧

42

$$(st \neq \mathit{FAILED} \longrightarrow (G, G') \in \mathit{polys\text{-}rel\text{-}full})\}$$
$$(\mathit{PAC\text{-}checker\text{-}specification\ spec}\ (A'))\rangle$$
⟨*proof*⟩

**lemma** *full-checker-spec′*:
  **shows**
    ⟨(*uncurry2 full-checker*, *uncurry2* (λ*spec A* -. *PAC-checker-specification spec A*)) ∈
      (*Id* ×$_r$ *polys-rel*) ×$_r$ *Id* →$_f$ ⟨{(((*st*, *G*), (*st′*, *G′*)). (*st*, *st′*) ∈ *status-rel* ∧
        (*st* ≠ *FAILED* ⟶ (*G*, *G′*) ∈ *polys-rel-full*)}⟩*nres-rel*⟩
  ⟨*proof*⟩

**end**
**theory** *PAC-Polynomials*
  **imports** *PAC-Specification Finite-Map-Multiset*
**begin**

# 7 Polynomials of strings

Isabelle's definition of polynomials only work with variables of type *nat*. Therefore, we introduce a version that uses strings.

## 7.1 Polynomials and Variables

**lemma** *poly-embed-EX*:
  ⟨∃ φ. *bij* (φ :: *string* ⇒ *nat*)⟩
  ⟨*proof*⟩

Using a multiset instead of a list has some advantage from an abstract point of view. First, we can have monomials that appear several times and the coefficient can also be zero. Basically, we can represent un-normalised polynomials, which is very useful to talk about intermediate states in our program.

**type-synonym** *term-poly* = ⟨*string multiset*⟩
**type-synonym** *mset-polynomial* =
  ⟨(*term-poly* ∗ *int*) *multiset*⟩

**definition** *normalized-poly* :: ⟨*mset-polynomial* ⇒ *bool*⟩ **where**
  ⟨*normalized-poly p* ⟷
    *distinct-mset* (*fst* '# *p*) ∧
    0 ∉# *snd* '# *p*⟩

**lemma** *normalized-poly-simps*[*simp*]:
  ⟨*normalized-poly* {#}⟩
  ⟨*normalized-poly* (*add-mset t p*) ⟷ *snd t* ≠ *0* ∧
    *fst t* ∉# *fst* '# *p* ∧ *normalized-poly p*⟩
  ⟨*proof*⟩

**lemma** *normalized-poly-mono*:
  ⟨*normalized-poly B* ⟹ *A* ⊆# *B* ⟹ *normalized-poly A*⟩
  ⟨*proof*⟩

**definition** *mult-poly-by-monom* :: ⟨*term-poly* ∗ *int* ⇒ *mset-polynomial* ⇒ *mset-polynomial*⟩ **where**
  ⟨*mult-poly-by-monom* = (λ*ys q*. *image-mset* (λ*xs*. (*fst xs* + *fst ys*, *snd ys* ∗ *snd xs*)) *q*)⟩

**definition** *mult-poly-raw* :: ‹*mset-polynomial* $\Rightarrow$ *mset-polynomial* $\Rightarrow$ *mset-polynomial*› **where**
‹*mult-poly-raw p q =*
  *(sum-mset (($\lambda$y. mult-poly-by-monom y q) '# p))*›


**definition** *remove-powers* :: ‹*mset-polynomial* $\Rightarrow$ *mset-polynomial*› **where**
‹*remove-powers xs = image-mset (apfst remdups-mset) xs*›


**definition** *all-vars-mset* :: ‹*mset-polynomial* $\Rightarrow$ *string multiset*› **where**
‹*all-vars-mset p* = $\bigcup$*# (fst '# p)*›

**abbreviation** *all-vars* :: ‹*mset-polynomial* $\Rightarrow$ *string set*› **where**
‹*all-vars p* $\equiv$ *set-mset (all-vars-mset p)*›

**definition** *add-to-coefficient* :: ‹*-* $\Rightarrow$ *mset-polynomial* $\Rightarrow$ *mset-polynomial*›  **where**
‹*add-to-coefficient* = ($\lambda$(*a, n*) *b.* {#(*a', -*) $\in$# *b. a'* $\neq$ *a*#} +
    (*if n + sum-mset (snd '#* {#(*a', -*) $\in$# *b. a'* = *a*#}) = *0 then* {#}
      *else* {#(*a, n + sum-mset (snd '#* {#(*a', -*) $\in$# *b. a'* = *a*#}))#}))›


**definition** *normalize-poly* :: ‹*mset-polynomial* $\Rightarrow$ *mset-polynomial*› **where**
‹*normalize-poly p = fold-mset add-to-coefficient* {#} *p*›

**lemma** *add-to-coefficient-simps*:
‹*n + sum-mset (snd '#* {#(*a', -*) $\in$# *b. a'* = *a*#}) $\neq$ *0* $\Longrightarrow$
  *add-to-coefficient* (*a, n*) *b* = {#(*a', -*) $\in$# *b. a'* $\neq$ *a*#} +
      {#(*a, n + sum-mset (snd '#* {#(*a', -*) $\in$# *b. a'* = *a*#}))#}›
‹*n + sum-mset (snd '#* {#(*a', -*) $\in$# *b. a'* = *a*#}) = *0* $\Longrightarrow$
  *add-to-coefficient* (*a, n*) *b* = {#(*a', -*) $\in$# *b. a'* $\neq$ *a*#}› **and**
*add-to-coefficient-simps-If*:
‹*add-to-coefficient* (*a, n*) *b* = {#(*a', -*) $\in$# *b. a'* $\neq$ *a*#} +
      (*if n + sum-mset (snd '#* {#(*a', -*) $\in$# *b. a'* = *a*#}) = *0 then* {#}
        *else* {#(*a, n + sum-mset (snd '#* {#(*a', -*) $\in$# *b. a'* = *a*#}))#})›
⟨*proof*⟩


**interpretation** *comp-fun-commute* ‹*add-to-coefficient*›
⟨*proof*⟩

**lemma** *normalized-poly-normalize-poly*[*simp*]:
‹*normalized-poly (normalize-poly p)*›
⟨*proof*⟩

## 7.2   Addition

**inductive** *add-poly-p* :: ‹*mset-polynomial* $\times$ *mset-polynomial* $\times$ *mset-polynomial* $\Rightarrow$ *mset-polynomial* $\times$ *mset-polynomial* $\times$ *mset-polynomial* $\Rightarrow$ *bool*› **where**
*add-new-coeff-r*:
  ‹*add-poly-p* (*p, add-mset x q, r*) (*p, q, add-mset x r*)› |
*add-new-coeff-l*:
  ‹*add-poly-p* (*add-mset x p, q, r*) (*p, q, add-mset x r*)› |
*add-same-coeff-l*:
  ‹*add-poly-p* (*add-mset* (*x, n*) *p, q, add-mset* (*x, m*) *r*) (*p, q, add-mset* (*x, n + m*) *r*)› |
*add-same-coeff-r*:
  ‹*add-poly-p* (*p, add-mset* (*x, n*) *q, add-mset* (*x, m*) *r*) (*p, q, add-mset* (*x, n + m*) *r*)› |
*rem-0-coeff*:

‹*add-poly-p* (*p*, *q*, *add-mset* (*x*, *0*) *r*) (*p*, *q*, *r*)›

**inductive-cases** *add-poly-pE*: ‹*add-poly-p S T*›

**lemmas** *add-poly-p-induct* =
  *add-poly-p.induct*[*split-format*(*complete*)]

**lemma** *add-poly-p-empty-l*:
  ‹*add-poly-p**\*\*** (*p*, *q*, *r*) ({#}, *q*, *p* + *r*)›
  ⟨*proof*⟩

**lemma** *add-poly-p-empty-r*:
  ‹*add-poly-p**\*\*** (*p*, *q*, *r*) (*p*, {#}, *q* + *r*)›
  ⟨*proof*⟩

**lemma** *add-poly-p-sym*:
  ‹*add-poly-p* (*p*, *q*, *r*) (*p'*, *q'*, *r'*) ⟷ *add-poly-p* (*q*, *p*, *r*) (*q'*, *p'*, *r'*)›
  ⟨*proof*⟩

**lemma** *wf-if-measure-in-wf*:
  ‹*wf R* ⟹ (⋀*a b*. (*a*, *b*) ∈ *S* ⟹ (*ν a*, *ν b*)∈*R*) ⟹ *wf S*›
  ⟨*proof*⟩

**lemma** *lexn-n*:
  ‹*n* > *0* ⟹ (*x* # *xs*, *y* # *ys*) ∈ *lexn r n* ⟷
  (*length xs* = *n−1* ∧ *length ys* = *n−1*) ∧ ((*x*, *y*) ∈ *r* ∨ (*x* = *y* ∧ (*xs*, *ys*) ∈ *lexn r* (*n* − *1*)))›
  ⟨*proof*⟩

**lemma** *wf-add-poly-p*:
  ‹*wf* {(*x*, *y*). *add-poly-p y x*}›
  ⟨*proof*⟩

**lemma** *mult-poly-by-monom-simps*[*simp*]:
  ‹*mult-poly-by-monom t* {#} = {#}›
  ‹*mult-poly-by-monom t* (*ps* + *qs*) = *mult-poly-by-monom t ps* + *mult-poly-by-monom t qs*›
  ‹*mult-poly-by-monom a* (*add-mset p ps*) = *add-mset* (*fst a* + *fst p*, *snd a* ∗ *snd p*) (*mult-poly-by-monom a ps*)›
⟨*proof*⟩

**inductive** *mult-poly-p* :: ‹*mset-polynomial* ⟹ *mset-polynomial* × *mset-polynomial* ⟹ *mset-polynomial* × *mset-polynomial* ⟹ *bool*›
  **for** *q* :: *mset-polynomial* **where**
*mult-step*:
  ‹*mult-poly-p q* (*add-mset* (*xs*, *n*) *p*, *r*) (*p*, (λ(*ys*, *m*). (*remdups-mset* (*xs* + *ys*), *n* ∗ *m*)) '# *q* + *r*)›

**lemmas** *mult-poly-p-induct* = *mult-poly-p.induct*[*split-format*(*complete*)]

## 7.3 Normalisation

**inductive** *normalize-poly-p* :: ‹*mset-polynomial* ⟹ *mset-polynomial* ⟹ *bool*›**where**
*rem-0-coeff*[*simp*, *intro*]:
  ‹*normalize-poly-p p q* ⟹ *normalize-poly-p* (*add-mset* (*xs*, *0*) *p*) *q*› |
*merge-dup-coeff*[*simp*, *intro*]:
  ‹*normalize-poly-p p q* ⟹ *normalize-poly-p* (*add-mset* (*xs*, *m*) (*add-mset* (*xs*, *n*) *p*)) (*add-mset* (*xs*, *m* + *n*) *q*)› |

*same*[*simp, intro*]:
    ‹*normalize-poly-p p p*› |
*keep-coeff* [*simp, intro*]:
    ‹*normalize-poly-p p q* ⟹ *normalize-poly-p* (*add-mset x p*) (*add-mset x q*)›

## 7.4   Correctness

This locales maps string polynomials to real polynomials.

**locale** *poly-embed* =
  **fixes** $\varphi$ :: ‹*string* ⟹ *nat*›
  **assumes** $\varphi$-*inj*: ‹*inj* $\varphi$›
**begin**

**definition** *poly-of-vars* :: *term-poly* ⟹ (′*a* :: {*comm-semiring-1*}) *mpoly* **where**
  ‹*poly-of-vars xs* = *fold-mset* ($\lambda a\ b.\ Var$ ($\varphi\ a$) ∗ *b*) (*1* :: ′*a mpoly*) *xs*›

**lemma** *poly-of-vars-simps*[*simp*]:
  **shows**
    ‹*poly-of-vars* (*add-mset x xs*) = *Var* ($\varphi\ x$) ∗ (*poly-of-vars xs* :: (′*a* :: {*comm-semiring-1*}) *mpoly*)› (**is
?A**) **and**
    ‹*poly-of-vars* (*xs* + *ys*) = *poly-of-vars xs* ∗ (*poly-of-vars ys* :: (′*a* :: {*comm-semiring-1*}) *mpoly*)› (**is
?B**)
⟨*proof*⟩


**definition** *mononom-of-vars* **where**
  ‹*mononom-of-vars* ≡ ($\lambda(xs,\ n).$ (+) (*Const n* ∗ *poly-of-vars xs*))›

**interpretation** *comp-fun-commute* ‹*mononom-of-vars*›
  ⟨*proof*⟩

**lemma** [*simp*]:
  ‹*poly-of-vars* {#} = *1*›
  ⟨*proof*⟩

**lemma** *mononom-of-vars-add*[*simp*]:
  ‹*NO-MATCH 0 b* ⟹ *mononom-of-vars xs b* = *Const* (*snd xs*) ∗ *poly-of-vars* (*fst xs*) + *b*›
  ⟨*proof*⟩

**definition** *polynomial-of-mset* :: ‹*mset-polynomial* ⟹ -› **where**
  ‹*polynomial-of-mset p* = *sum-mset* (*mononom-of-vars* '# *p*) *0*›

**lemma** *polynomial-of-mset-append*[*simp*]:
  ‹*polynomial-of-mset* (*xs* + *ys*) = *polynomial-of-mset xs* + *polynomial-of-mset ys*›
  ⟨*proof*⟩

**lemma** *polynomial-of-mset-Cons*[*simp*]:
  ‹*polynomial-of-mset* (*add-mset x ys*) = *Const* (*snd x*) ∗ *poly-of-vars* (*fst x*) + *polynomial-of-mset ys*›
  ⟨*proof*⟩

**lemma** *polynomial-of-mset-empty*[*simp*]:
  ‹*polynomial-of-mset* {#} = *0*›
  ⟨*proof*⟩

**lemma** *polynomial-of-mset-mult-poly-by-monom*[*simp*]:

‹*polynomial-of-mset* (*mult-poly-by-monom* *x* *ys*) =
    (*Const* (*snd* *x*) ∗ *poly-of-vars* (*fst* *x*) ∗ *polynomial-of-mset* *ys*)›
⟨*proof*⟩

**lemma** *polynomial-of-mset-mult-poly-raw*[*simp*]:
‹*polynomial-of-mset* (*mult-poly-raw* *xs* *ys*) = *polynomial-of-mset* *xs* ∗ *polynomial-of-mset* *ys*›
⟨*proof*⟩

**lemma** *polynomial-of-mset-uminus*:
‹*polynomial-of-mset* {#*case* *x* *of* (*a*, *b*) ⇒ (*a*, − *b*). *x* ∈# *za*#} =
    − *polynomial-of-mset* *za*›
⟨*proof*⟩

**lemma** *X2-X-polynomial-bool-mult-in*:
‹*Var* (*x1*) ∗ (*Var* (*x1*) ∗ *p*) −   *Var* (*x1*) ∗ *p* ∈ *More-Modules.ideal* *polynomial-bool*›
⟨*proof*⟩

**lemma** *polynomial-of-list-remove-powers-polynomial-bool*:
‹(*polynomial-of-mset* *xs*) − *polynomial-of-mset* (*remove-powers* *xs*) ∈ *ideal* *polynomial-bool*›
⟨*proof*⟩

**lemma** *add-poly-p-polynomial-of-mset*:
‹*add-poly-p* (*p*, *q*, *r*) (*p′*, *q′*, *r′*) ⟹
    *polynomial-of-mset* *r* + (*polynomial-of-mset* *p* + *polynomial-of-mset* *q*) =
    *polynomial-of-mset* *r′* + (*polynomial-of-mset* *p′* + *polynomial-of-mset* *q′*)›
⟨*proof*⟩

**lemma** *rtranclp-add-poly-p-polynomial-of-mset*:
‹*add-poly-p*∗∗ (*p*, *q*, *r*) (*p′*, *q′*, *r′*) ⟹
    *polynomial-of-mset* *r* + (*polynomial-of-mset* *p* + *polynomial-of-mset* *q*) =
    *polynomial-of-mset* *r′* + (*polynomial-of-mset* *p′* + *polynomial-of-mset* *q′*)›
⟨*proof*⟩

**lemma** *rtranclp-add-poly-p-polynomial-of-mset-full*:
‹*add-poly-p*∗∗ (*p*, *q*, {#}) ({#}, {#}, *r′*) ⟹
    *polynomial-of-mset* *r′* = (*polynomial-of-mset* *p* + *polynomial-of-mset* *q*)›
⟨*proof*⟩

**lemma** *poly-of-vars-remdups-mset*:
‹*poly-of-vars* (*remdups-mset* (*xs*)) − (*poly-of-vars* *xs*)
    ∈ *More-Modules.ideal* *polynomial-bool*›
⟨*proof*⟩

**lemma** *polynomial-of-mset-mult-map*:
‹*polynomial-of-mset*
    {#*case* *x* *of* (*ys*, *n*) ⇒ (*remdups-mset* (*ys* + *xs*), *n* ∗ *m*). *x* ∈# *q*#} −
    *Const* *m* ∗ (*poly-of-vars* *xs* ∗ *polynomial-of-mset* *q*)
    ∈ *More-Modules.ideal* *polynomial-bool*›
    (**is** ‹*?P* *q* ∈ -›)
⟨*proof*⟩

**lemma** *mult-poly-p-mult-ideal*:

⟨*mult-poly-p q* (*p, r*) (*p′, r′*) ⟹
   (*polynomial-of-mset p′* ∗ *polynomial-of-mset q* + *polynomial-of-mset r′*) − (*polynomial-of-mset p* ∗
*polynomial-of-mset q* + *polynomial-of-mset r*)
      ∈ *ideal polynomial-bool*⟩
⟨*proof*⟩

**lemma** *rtranclp-mult-poly-p-mult-ideal*:
 ⟨(*mult-poly-p q*)\*\* (*p, r*) (*p′, r′*) ⟹
   (*polynomial-of-mset p′* ∗ *polynomial-of-mset q* + *polynomial-of-mset r′*) − (*polynomial-of-mset p* ∗
*polynomial-of-mset q* + *polynomial-of-mset r*)
      ∈ *ideal polynomial-bool*⟩
  ⟨*proof*⟩

**lemma** *rtranclp-mult-poly-p-mult-ideal-final*:
 ⟨(*mult-poly-p q*)\*\* (*p*, {#}) ({#}, *r*) ⟹
  (*polynomial-of-mset r*) − (*polynomial-of-mset p* ∗ *polynomial-of-mset q*)
      ∈ *ideal polynomial-bool*⟩
  ⟨*proof*⟩

**lemma** *normalize-poly-p-poly-of-mset*:
 ⟨*normalize-poly-p p q* ⟹ *polynomial-of-mset p* = *polynomial-of-mset q*⟩
  ⟨*proof*⟩


**lemma** *rtranclp-normalize-poly-p-poly-of-mset*:
 ⟨*normalize-poly-p*\*\* *p q* ⟹ *polynomial-of-mset p* = *polynomial-of-mset q*⟩
  ⟨*proof*⟩

**end**

It would be nice to have the property in the other direction too, but this requires a deep dive
into the definitions of polynomials.

**locale** *poly-embed-bij* = *poly-embed* +
  **fixes** *V N*
  **assumes** *φ-bij*: ⟨*bij-betw φ V N*⟩
**begin**

**definition** *φ′* :: ⟨*nat* ⟹ *string*⟩ **where**
 ⟨*φ′* = *the-inv-into V φ*⟩

**lemma** *φ′-φ*[*simp*]:
 ⟨*x* ∈ *V* ⟹ *φ′* (*φ x*) = *x*⟩
 ⟨*proof*⟩

**lemma** *φ-φ′*[*simp*]:
 ⟨*x* ∈ *N* ⟹ *φ* (*φ′ x*) = *x*⟩
 ⟨*proof*⟩

**end**

**end**
**theory** *PAC-Polynomials-Term*
  **imports** *PAC-Polynomials*
    *Refine-Imperative-HOL.IICF*
**begin**

# 8 Terms

We define some helper functions.

## 8.1 Ordering

**lemma** *fref-to-Down-curry-left*:
  **fixes** $f$:: ‹$'a \Rightarrow {}'b \Rightarrow {}'c$ *nres*› **and**
    $A$::‹$(('a \times {}'b) \times {}'d)$ *set*›
  **shows**
    ‹$(uncurry\ f,\ g) \in [P]_f\ A \rightarrow \langle B \rangle nres\text{-}rel \Longrightarrow$
    $(\bigwedge a\ b\ x'.\ P\ x' \Longrightarrow ((a,\ b),\ x') \in A \Longrightarrow f\ a\ b \leq \Downarrow B\ (g\ x'))$›
⟨*proof*⟩

**lemma** *fref-to-Down-curry-right*:
  **fixes** $g$ :: ‹$'a \Rightarrow {}'b \Rightarrow {}'c$ *nres*› **and** $f$ :: ‹$'d \Rightarrow$ - *nres*› **and**
    $A$::‹$('d \times ('a \times {}'b))$ *set*›
  **shows**
    ‹$(f,\ uncurry\ g) \in [P]_f\ A \rightarrow \langle B \rangle nres\text{-}rel \Longrightarrow$
    $(\bigwedge a\ b\ x'.\ P\ (a,b) \Longrightarrow (x',\ (a,\ b)) \in A \Longrightarrow f\ x' \leq \Downarrow B\ (g\ a\ b))$›
⟨*proof*⟩

**type-synonym** *term-poly-list* = ‹*string list*›
**type-synonym** *llist-polynomial* = ‹(*term-poly-list* × *int*) *list*›

We instantiate the characters with typeclass linorder to be able to talk abourt sorted and so on.

**definition** *less-eq-char* :: ‹*char* $\Rightarrow$ *char* $\Rightarrow$ *bool*› **where**
  ‹*less-eq-char* $c\ d$ = $(((of\text{-}char\ c) :: nat) \leq of\text{-}char\ d)$›

**definition** *less-char* :: ‹*char* $\Rightarrow$ *char* $\Rightarrow$ *bool*› **where**
  ‹*less-char* $c\ d$ = $(((of\text{-}char\ c) :: nat) < of\text{-}char\ d)$›

**global-interpretation** *char*: *linorder less-eq-char less-char*
  ⟨*proof*⟩

**abbreviation** *less-than-char* :: ‹(*char* × *char*) *set*› **where**
  ‹*less-than-char* ≡ *p2rel less-char*›

**lemma** *less-than-char-def*:
  ‹$(x,y) \in less\text{-}than\text{-}char \longleftrightarrow less\text{-}char\ x\ y$›
  ⟨*proof*⟩

**lemma** *trans-less-than-char*[*simp*]:
    ‹*trans less-than-char*› **and**
  *irrefl-less-than-char*:
    ‹*irrefl less-than-char*› **and**
  *antisym-less-than-char*:
    ‹*antisym less-than-char*›
  ⟨*proof*⟩

## 8.2 Polynomials

**definition** *var-order-rel* :: ‹(*string* × *string*) *set*› **where**
  ‹*var-order-rel* ≡ *lexord less-than-char*›

**abbreviation** *var-order* :: ‹*string* ⇒ *string* ⇒ *bool*› **where**
‹*var-order* ≡ *rel2p var-order-rel*›

**abbreviation** *term-order-rel* :: ‹(*term-poly-list* × *term-poly-list*) *set*› **where**
‹*term-order-rel* ≡ *lexord var-order-rel*›

**abbreviation** *term-order* :: ‹*term-poly-list* ⇒ *term-poly-list* ⇒ *bool*› **where**
‹*term-order* ≡ *rel2p term-order-rel*›

**definition** *term-poly-list-rel* :: ‹(*term-poly-list* × *term-poly*) *set*› **where**
‹*term-poly-list-rel* = {(*xs*, *ys*).
    *ys* = *mset xs* ∧
    *distinct xs* ∧
    *sorted-wrt* (*rel2p var-order-rel*) *xs*}›

**definition** *unsorted-term-poly-list-rel* :: ‹(*term-poly-list* × *term-poly*) *set*› **where**
‹*unsorted-term-poly-list-rel* = {(*xs*, *ys*).
    *ys* = *mset xs* ∧ *distinct xs*}›

**definition** *poly-list-rel* :: ‹- ⇒ ((′*a* × *int*) *list* × *mset-polynomial*) *set*› **where**
‹*poly-list-rel* *R* = {(*xs*, *ys*).
    (*xs*, *ys*) ∈ ⟨*R* ×ᵣ *int-rel*⟩*list-rel* O *list-mset-rel* ∧
    0 ∉# *snd* '# *ys*}›

**definition** *sorted-poly-list-rel-wrt* :: ‹(′*a* ⇒ ′*a* ⇒ *bool*)
    ⇒ (′*a* × *string multiset*) *set* ⇒ ((′*a* × *int*) *list* × *mset-polynomial*) *set*› **where**
‹*sorted-poly-list-rel-wrt* *S* *R* = {(*xs*, *ys*).
    (*xs*, *ys*) ∈ ⟨*R* ×ᵣ *int-rel*⟩*list-rel* O *list-mset-rel* ∧
    *sorted-wrt* *S* (*map fst xs*) ∧
    *distinct* (*map fst xs*) ∧
    0 ∉# *snd* '# *ys*}›

**abbreviation** *sorted-poly-list-rel* **where**
‹*sorted-poly-list-rel* *R* ≡ *sorted-poly-list-rel-wrt* *R* *term-poly-list-rel*›

**abbreviation** *sorted-poly-rel* **where**
‹*sorted-poly-rel* ≡ *sorted-poly-list-rel* *term-order*›


**definition** *sorted-repeat-poly-list-rel-wrt* :: ‹(′*a* ⇒ ′*a* ⇒ *bool*)
    ⇒ (′*a* × *string multiset*) *set* ⇒ ((′*a* × *int*) *list* × *mset-polynomial*) *set*› **where**
‹*sorted-repeat-poly-list-rel-wrt* *S* *R* = {(*xs*, *ys*).
    (*xs*, *ys*) ∈ ⟨*R* ×ᵣ *int-rel*⟩*list-rel* O *list-mset-rel* ∧
    *sorted-wrt* *S* (*map fst xs*) ∧
    0 ∉# *snd* '# *ys*}›

**abbreviation** *sorted-repeat-poly-list-rel* **where**
‹*sorted-repeat-poly-list-rel* *R* ≡ *sorted-repeat-poly-list-rel-wrt* *R* *term-poly-list-rel*›

**abbreviation** *sorted-repeat-poly-rel* **where**
‹*sorted-repeat-poly-rel* ≡ *sorted-repeat-poly-list-rel* (*rel2p* (*Id* ∪ *lexord var-order-rel*))›


**abbreviation** *unsorted-poly-rel* **where**

⟨*unsorted-poly-rel* ≡ *poly-list-rel term-poly-list-rel*⟩

**lemma** *sorted-poly-list-rel-empty-l*[*simp*]:
⟨([], *s′*) ∈ *sorted-poly-list-rel-wrt S T* ⟷ *s′* = {#}⟩
⟨*proof*⟩

**definition** *fully-unsorted-poly-list-rel* :: ⟨- ⇒ (('*a* × *int*) *list* × *mset-polynomial*) *set*⟩ **where**
⟨*fully-unsorted-poly-list-rel R* = {(*xs, ys*).
  (*xs, ys*) ∈ ⟨*R* ×$_r$ *int-rel*⟩*list-rel O list-mset-rel*}⟩

**abbreviation** *fully-unsorted-poly-rel* **where**
⟨*fully-unsorted-poly-rel* ≡ *fully-unsorted-poly-list-rel unsorted-term-poly-list-rel*⟩

**lemma** *fully-unsorted-poly-list-rel-empty-iff*[*simp*]:
⟨(*p*, {#}) ∈ *fully-unsorted-poly-list-rel R* ⟷ *p* = []⟩
⟨([], *p′*) ∈ *fully-unsorted-poly-list-rel R* ⟷ *p′* = {#}⟩
⟨*proof*⟩

**definition** *poly-list-rel-with0* :: ⟨- ⇒ (('*a* × *int*) *list* × *mset-polynomial*) *set*⟩ **where**
⟨*poly-list-rel-with0 R* = {(*xs, ys*).
  (*xs, ys*) ∈ ⟨*R* ×$_r$ *int-rel*⟩*list-rel O list-mset-rel*}⟩

**abbreviation** *unsorted-poly-rel-with0* **where**
⟨*unsorted-poly-rel-with0* ≡ *fully-unsorted-poly-list-rel term-poly-list-rel*⟩

**lemma** *poly-list-rel-with0-empty-iff*[*simp*]:
⟨(*p*, {#}) ∈ *poly-list-rel-with0 R* ⟷ *p* = []⟩
⟨([], *p′*) ∈ *poly-list-rel-with0 R* ⟷ *p′* = {#}⟩
⟨*proof*⟩

**definition** *sorted-repeat-poly-list-rel-with0-wrt* :: ⟨('*a* ⇒ '*a* ⇒ *bool*)
    ⇒ ('*a* × *string multiset*) *set* ⇒ (('*a* × *int*) *list* × *mset-polynomial*) *set*⟩ **where**
⟨*sorted-repeat-poly-list-rel-with0-wrt S R* = {(*xs, ys*).
  (*xs, ys*) ∈ ⟨*R* ×$_r$ *int-rel*⟩*list-rel O list-mset-rel* ∧
  *sorted-wrt S* (*map fst xs*)}⟩

**abbreviation** *sorted-repeat-poly-list-rel-with0* **where**
⟨*sorted-repeat-poly-list-rel-with0 R* ≡ *sorted-repeat-poly-list-rel-with0-wrt R term-poly-list-rel*⟩

**abbreviation** *sorted-repeat-poly-rel-with0* **where**
⟨*sorted-repeat-poly-rel-with0* ≡ *sorted-repeat-poly-list-rel-with0* (*rel2p* (*Id* ∪ *lexord var-order-rel*))⟩

**lemma** *term-poly-list-relD*:
⟨(*xs, ys*) ∈ *term-poly-list-rel* ⟹ *distinct xs*⟩
⟨(*xs, ys*) ∈ *term-poly-list-rel* ⟹ *ys* = *mset xs*⟩
⟨(*xs, ys*) ∈ *term-poly-list-rel* ⟹ *sorted-wrt* (*rel2p var-order-rel*) *xs*⟩
⟨(*xs, ys*) ∈ *term-poly-list-rel* ⟹ *sorted-wrt* (*rel2p* (*Id* ∪ *var-order-rel*)) *xs*⟩
⟨*proof*⟩

**end**
**theory** *PAC-Polynomials-Operations*
  **imports** *PAC-Polynomials-Term PAC-Checker-Specification*

**begin**

# 9 Polynomialss as Lists

## 9.1 Addition

In this section, we refine the polynomials to list. These lists will be used in our checker to represent the polynomials and execute operations.

There is one *key* difference between the list representation and the usual representation: in the former, coefficients can be zero and monomials can appear several times. This makes it easier to reason on intermediate representation where this has not yet been sanitized.

**fun** *add-poly-l′* :: ⟨*llist-polynomial* × *llist-polynomial* ⇒ *llist-polynomial*⟩ **where**
⟨*add-poly-l′* (*p*, []) = *p*⟩ |
⟨*add-poly-l′* ([], *q*) = *q*⟩ |
⟨*add-poly-l′* ((*xs*, *n*) # *p*, (*ys*, *m*) # *q*) =
        (*if xs* = *ys then if n* + *m* = *0 then add-poly-l′* (*p*, *q*) *else*
            *let pq* = *add-poly-l′* (*p*, *q*) *in*
            ((*xs*, *n* + *m*) # *pq*)
        *else if* (*xs*, *ys*) ∈ *term-order-rel*
          *then*
            *let pq* = *add-poly-l′* (*p*, (*ys*, *m*) # *q*) *in*
            ((*xs*, *n*) # *pq*)
        *else*
            *let pq* = *add-poly-l′* ((*xs*, *n*) # *p*, *q*) *in*
            ((*ys*, *m*) # *pq*)
        )⟩

**definition** *add-poly-l* :: ⟨*llist-polynomial* × *llist-polynomial* ⇒ *llist-polynomial nres*⟩ **where**
⟨*add-poly-l* = *REC*$_T$
    (λ*add-poly-l* (*p*, *q*).
      *case* (*p*,*q*) *of*
        (*p*, []) ⇒ *RETURN p*
      | ([], *q*) ⇒ *RETURN q*
      | ((*xs*, *n*) # *p*, (*ys*, *m*) # *q*) ⇒
          (*if xs* = *ys then if n* + *m* = *0 then add-poly-l* (*p*, *q*) *else*
            *do* {
              *pq* ← *add-poly-l* (*p*, *q*);
              *RETURN* ((*xs*, *n* + *m*) # *pq*)
          }
          *else if* (*xs*, *ys*) ∈ *term-order-rel*
            *then do* {
              *pq* ← *add-poly-l* (*p*, (*ys*, *m*) # *q*);
              *RETURN* ((*xs*, *n*) # *pq*)
          }
          *else do* {
              *pq* ← *add-poly-l* ((*xs*, *n*) # *p*, *q*);
              *RETURN* ((*ys*, *m*) # *pq*)
          }))⟩

**definition** *nonzero-coeffs* **where**
⟨*nonzero-coeffs a* ⟷ *0* ∉# *snd* '# *a*⟩

**lemma** *nonzero-coeffs-simps*[*simp*]:
⟨*nonzero-coeffs* {#}⟩

‹*nonzero-coeffs* (*add-mset* (*xs*, *n*) *a*) ⟷ *nonzero-coeffs* *a* ∧ *n* ≠ *0*›
⟨*proof*⟩

**lemma** *nonzero-coeffsD*:
‹*nonzero-coeffs* *a* ⟹ (*x*, *n*) ∈# *a* ⟹ *n* ≠ *0*›
⟨*proof*⟩

**lemma** *sorted-poly-list-rel-ConsD*:
‹((*ys*, *n*) # *p*, *a*) ∈ *sorted-poly-list-rel* *S* ⟹ (*p*, *remove1-mset* (*mset* *ys*, *n*) *a*) ∈ *sorted-poly-list-rel* *S*
∧
  (*mset* *ys*, *n*) ∈# *a* ∧ (∀ *x* ∈ *set* *p*. *S* *ys* (*fst* *x*)) ∧ *sorted-wrt* (*rel2p* *var-order-rel*) *ys* ∧
  *distinct* *ys* ∧ *ys* ∉ *set* (*map* *fst* *p*) ∧ *n* ≠ *0* ∧ *nonzero-coeffs* *a*›
⟨*proof*⟩

**lemma** *sorted-poly-list-rel-Cons-iff*:
‹((*ys*, *n*) # *p*, *a*) ∈ *sorted-poly-list-rel* *S* ⟷ (*p*, *remove1-mset* (*mset* *ys*, *n*) *a*) ∈ *sorted-poly-list-rel* *S*
∧
  (*mset* *ys*, *n*) ∈# *a* ∧ (∀ *x* ∈ *set* *p*. *S* *ys* (*fst* *x*)) ∧ *sorted-wrt* (*rel2p* *var-order-rel*) *ys* ∧
  *distinct* *ys* ∧ *ys* ∉ *set* (*map* *fst* *p*) ∧ *n* ≠ *0* ∧ *nonzero-coeffs* *a*›
⟨*proof*⟩

**lemma** *sorted-repeat-poly-list-rel-ConsD*:
‹((*ys*, *n*) # *p*, *a*) ∈ *sorted-repeat-poly-list-rel* *S* ⟹ (*p*, *remove1-mset* (*mset* *ys*, *n*) *a*) ∈ *sorted-repeat-poly-list-rel*
*S* ∧
  (*mset* *ys*, *n*) ∈# *a* ∧ (∀ *x* ∈ *set* *p*. *S* *ys* (*fst* *x*)) ∧ *sorted-wrt* (*rel2p* *var-order-rel*) *ys* ∧
  *distinct* *ys* ∧ *n* ≠ *0* ∧ *nonzero-coeffs* *a*›
⟨*proof*⟩

**lemma** *sorted-repeat-poly-list-rel-Cons-iff*:
‹((*ys*, *n*) # *p*, *a*) ∈ *sorted-repeat-poly-list-rel* *S* ⟷ (*p*, *remove1-mset* (*mset* *ys*, *n*) *a*) ∈ *sorted-repeat-poly-list-rel*
*S* ∧
  (*mset* *ys*, *n*) ∈# *a* ∧ (∀ *x* ∈ *set* *p*. *S* *ys* (*fst* *x*)) ∧ *sorted-wrt* (*rel2p* *var-order-rel*) *ys* ∧
  *distinct* *ys* ∧ *n* ≠ *0* ∧ *nonzero-coeffs* *a*›
⟨*proof*⟩

**lemma** *add-poly-p-add-mset-sum-0*:
‹*n* + *m* = *0* ⟹*add-poly-p*∗∗ (*A*, *Aa*, {#}) ({#}, {#}, *r*) ⟹
    *add-poly-p*∗∗
    (*add-mset* (*mset* *ys*, *n*) *A*, *add-mset* (*mset* *ys*, *m*) *Aa*, {#})
    ({#}, {#}, *r*)›
⟨*proof*⟩

**lemma** *monoms-add-poly-l′D*:
‹(*aa*, *ba*) ∈ *set* (*add-poly-l′* *x*) ⟹ *aa* ∈ *fst* ' *set* (*fst* *x*) ∨ *aa* ∈ *fst* ' *set* (*snd* *x*)›
⟨*proof*⟩

**lemma** *add-poly-p-add-to-result*:
‹*add-poly-p*∗∗ (*A*, *B*, *r*) (*A′*, *B′*, *r′*) ⟹
    *add-poly-p*∗∗
    (*A*, *B*, *p* + *r*) (*A′*, *B′*, *p* + *r′*)›
⟨*proof*⟩

**lemma** *add-poly-p-add-mset-comb*:
 ⟨*add-poly-p*$^{**}$ *(A, Aa, {#}) ({#}, {#}, r)* $\Longrightarrow$
     *add-poly-p*$^{**}$
      *(add-mset (xs, n) A, Aa, {#})*
      *({#}, {#}, add-mset (xs, n) r)*⟩
 ⟨*proof*⟩

**lemma** *add-poly-p-add-mset-comb2*:
 ⟨*add-poly-p*$^{**}$ *(A, Aa, {#}) ({#}, {#}, r)* $\Longrightarrow$
     *add-poly-p*$^{**}$
      *(add-mset (ys, n) A, add-mset (ys, m) Aa, {#})*
      *({#}, {#}, add-mset (ys, n + m) r)*⟩
 ⟨*proof*⟩


**lemma** *add-poly-p-add-mset-comb3*:
 ⟨*add-poly-p*$^{**}$ *(A, Aa, {#}) ({#}, {#}, r)* $\Longrightarrow$
     *add-poly-p*$^{**}$
      *(A, add-mset (ys, m) Aa, {#})*
      *({#}, {#}, add-mset (ys, m) r)*⟩
 ⟨*proof*⟩

**lemma** *total-on-lexord*:
 ⟨*Relation.total-on UNIV R* $\Longrightarrow$ *Relation.total-on UNIV (lexord R)*⟩
 ⟨*proof*⟩

**lemma** *antisym-lexord*:
 ⟨*antisym R* $\Longrightarrow$ *irrefl R* $\Longrightarrow$ *antisym (lexord R)*⟩
 ⟨*proof*⟩

**lemma** *less-than-char-linear*:
 ⟨*(a, b)* $\in$ *less-than-char* $\lor$
       *a = b* $\lor$ *(b, a)* $\in$ *less-than-char*⟩
 ⟨*proof*⟩

**lemma** *total-on-lexord-less-than-char-linear*:
 ⟨*xs* $\neq$ *ys* $\Longrightarrow$ *(xs, ys)* $\notin$ *lexord (lexord less-than-char)* $\longleftrightarrow$
     *(ys, xs)* $\in$ *lexord (lexord less-than-char)*⟩
  ⟨*proof*⟩

**lemma** *sorted-poly-list-rel-nonzeroD*:
 ⟨*(p, r)* $\in$ *sorted-poly-list-rel term-order* $\Longrightarrow$
     *nonzero-coeffs (r)*⟩
 ⟨*(p, r)* $\in$ *sorted-poly-list-rel (rel2p (lexord (lexord less-than-char)))* $\Longrightarrow$
     *nonzero-coeffs (r)*⟩
 ⟨*proof*⟩


**lemma** *add-poly-l'-add-poly-p*:
  **assumes** ⟨*(pq, pq')* $\in$ *sorted-poly-rel* $\times_r$ *sorted-poly-rel*⟩
  **shows** ⟨$\exists$ *r. (add-poly-l' pq, r)* $\in$ *sorted-poly-rel* $\land$
                    *add-poly-p*$^{**}$ *(fst pq', snd pq', {#}) ({#}, {#}, r)*⟩
 ⟨*proof*⟩

**lemma** *add-poly-l-add-poly*:
  ⟨*add-poly-l x = RETURN (add-poly-l' x)*⟩
  ⟨*proof*⟩

**lemma** *add-poly-l-spec*:
  ⟨(*add-poly-l, uncurry (λp q. SPEC(λr. add-poly-p** (p, q, {#}) ({#}, {#}, r)))) ∈*
    *sorted-poly-rel ×_r sorted-poly-rel →_f ⟨sorted-poly-rel⟩nres-rel*⟩
  ⟨*proof*⟩

**definition** *sort-poly-spec* :: ⟨*llist-polynomial ⇒ llist-polynomial nres*⟩ **where**
⟨*sort-poly-spec p =*
  *SPEC(λp'. mset p = mset p' ∧ sorted-wrt (rel2p (Id ∪ term-order-rel)) (map fst p'))*⟩

**lemma** *sort-poly-spec-id*:
  **assumes** ⟨(*p, p'*) ∈ *unsorted-poly-rel*⟩
  **shows** ⟨*sort-poly-spec p ≤ ⇓ (sorted-repeat-poly-rel) (RETURN p')*⟩
⟨*proof*⟩

## 9.2 Multiplication

**fun** *mult-monoms* :: ⟨*term-poly-list ⇒ term-poly-list ⇒ term-poly-list*⟩ **where**
  ⟨*mult-monoms p [] = p*⟩ |
  ⟨*mult-monoms [] p = p*⟩ |
  ⟨*mult-monoms (x # p) (y # q) =*
    (*if x = y then x # mult-monoms p q*
      *else if (x, y) ∈ var-order-rel then x # mult-monoms p (y # q)*
        *else y # mult-monoms (x # p) q*)⟩

**lemma** *term-poly-list-rel-empty-iff*[*simp*]:
  ⟨([], *q'*) ∈ *term-poly-list-rel* ⟷ *q' = {#}*⟩
  ⟨*proof*⟩

**lemma** *term-poly-list-rel-Cons-iff*:
  ⟨(*y # p, p'*) ∈ *term-poly-list-rel* ⟷
    (*p, remove1-mset y p'*) ∈ *term-poly-list-rel* ∧
    *y ∈# p' ∧ y ∉ set p ∧ y ∉# remove1-mset y p'* ∧
    (∀ *x∈#mset p. (y, x) ∈ var-order-rel*)⟩
  ⟨*proof*⟩

**lemma** *var-order-rel-antisym*[*simp*]:
  ⟨(*y, y*) ∉ *var-order-rel*⟩
  ⟨*proof*⟩

**lemma** *term-poly-list-rel-remdups-mset*:
  ⟨(*p, p'*) ∈ *term-poly-list-rel* ⟹
    (*p, remdups-mset p'*) ∈ *term-poly-list-rel*⟩
  ⟨*proof*⟩

**lemma** *var-notin-notin-mult-monomsD*:
  ⟨*y ∈ set (mult-monoms p q) ⟹ y ∈ set p ∨ y ∈ set q*⟩
  ⟨*proof*⟩

**lemma** *term-poly-list-rel-set-mset*:
  ⟨(*p, q*) ∈ *term-poly-list-rel* ⟹ *set p = set-mset q*⟩
  ⟨*proof*⟩

**lemma** *mult-monoms-spec*:
⟨(*mult-monoms*, (λ*a b. remdups-mset* (*a* + *b*))) ∈ *term-poly-list-rel* → *term-poly-list-rel* → *term-poly-list-rel*⟩
⟨*proof*⟩

**definition** *mult-monomials* :: ⟨*term-poly-list* × *int* ⇒ *term-poly-list* × *int* ⇒ *term-poly-list* × *int*⟩ **where**
⟨*mult-monomials* = (λ(*x*, *a*) (*y*, *b*). (*mult-monoms x y*, *a* ∗ *b*))⟩

**definition** *mult-poly-raw* :: ⟨*llist-polynomial* ⇒ *llist-polynomial* ⇒ *llist-polynomial*⟩ **where**
⟨*mult-poly-raw p q* = *foldl* (λ*b x. map* (*mult-monomials x*) *q* @ *b*) [] *p*⟩

**fun** *map-append* **where**
⟨*map-append f b* [] = *b*⟩ |
⟨*map-append f b* (*x* # *xs*) = *f x* # *map-append f b xs*⟩

**lemma** *map-append-alt-def*:
⟨*map-append f b xs* = *map f xs* @ *b*⟩
⟨*proof*⟩

**lemma** *foldl-append-empty*:
⟨*NO-MATCH* [] *xs* ⟹ *foldl* (λ*b x. f x* @ *b*) *xs p* = *foldl* (λ*b x. f x* @ *b*) [] *p* @ *xs*⟩
⟨*proof*⟩

**lemma** *poly-list-rel-empty-iff*[*simp*]:
⟨([], *r*) ∈ *poly-list-rel R* ⟷ *r* = {#}⟩
⟨*proof*⟩

**lemma** *mult-poly-raw-simp*[*simp*]:
⟨*mult-poly-raw* [] *q* = []⟩
⟨*mult-poly-raw* (*x* # *p*) *q* = *mult-poly-raw p q* @ *map* (*mult-monomials x*) *q*⟩
⟨*proof*⟩

**lemma** *sorted-poly-list-relD*:
⟨(*q*, *q'*) ∈ *sorted-poly-list-rel R* ⟹ *q'* = (λ(*a*, *b*). (*mset a*, *b*)) '# *mset q*⟩
⟨*proof*⟩

**lemma** *list-all2-in-set-ExD*:
⟨*list-all2 R p q* ⟹ *x* ∈ *set p* ⟹ ∃ *y* ∈ *set q. R x y*⟩
⟨*proof*⟩

**inductive-cases** *mult-poly-p-elim*: ⟨*mult-poly-p q* (*A*, *r*) (*B*, *r'*)⟩

**lemma** *mult-poly-p-add-mset-same*:
⟨(*mult-poly-p q'*)∗∗ (*A*, *r*) (*B*, *r'*) ⟹ (*mult-poly-p q'*)∗∗ (*add-mset x A*, *r*) (*add-mset x B*, *r'*)⟩
⟨*proof*⟩

**lemma** *mult-poly-raw-mult-poly-p*:
  **assumes** ⟨(*p*, *p'*) ∈ *sorted-poly-rel*⟩ **and** ⟨(*q*, *q'*) ∈ *sorted-poly-rel*⟩
  **shows** ⟨∃ *r*. (*mult-poly-raw p q*, *r*) ∈ *unsorted-poly-rel* ∧ (*mult-poly-p q'*)∗∗ (*p'*, {#}) ({#}, *r*)⟩
⟨*proof*⟩

**fun** *merge-coeffs* :: ⟨*llist-polynomial* ⇒ *llist-polynomial*⟩ **where**
⟨*merge-coeffs* [] = []⟩ |
⟨*merge-coeffs* [(*xs*, *n*)] = [(*xs*, *n*)]⟩ |

‹merge-coeffs ((xs, n) # (ys, m) # p) =
 (if xs = ys
  then if n + m ≠ 0 then merge-coeffs ((xs, n + m) # p) else merge-coeffs p
  else (xs, n) # merge-coeffs ((ys, m) # p))›

**abbreviation** (**in** −)*mononoms* :: ‹*llist-polynomial* ⇒ *term-poly-list set*› **where**
 ‹*mononoms p ≡ fst ' set p*›

**lemma** *fst-normalize-polynomial-subset*:
 ‹*mononoms (merge-coeffs p) ⊆ mononoms p*›
 ⟨*proof*⟩

**lemma** *fst-normalize-polynomial-subsetD*:
 ‹(a, b) ∈ set (merge-coeffs p) ⟹ a ∈ mononoms p›
 ⟨*proof*⟩

**lemma** *distinct-merge-coeffs*:
 **assumes** ‹*sorted-wrt R (map fst xs)*› **and** ‹*transp R*› ‹*antisymp R*›
 **shows** ‹*distinct (map fst (merge-coeffs xs))*›
 ⟨*proof*⟩

**lemma** *in-set-merge-coeffsD*:
 ‹(a, b) ∈ set (merge-coeffs p) ⟹ ∃ b. (a, b) ∈ set p›
 ⟨*proof*⟩

**lemma** *rtranclp-normalize-poly-add-mset*:
 ‹*normalize-poly-p*** A r ⟹ normalize-poly-p*** (add-mset x A) (add-mset x r)›
 ⟨*proof*⟩

**lemma** *nonzero-coeffs-diff*:
 ‹*nonzero-coeffs A ⟹ nonzero-coeffs (A − B)*›
 ⟨*proof*⟩

**lemma** *merge-coeffs-is-normalize-poly-p*:
 ‹(xs, ys) ∈ sorted-repeat-poly-rel ⟹ ∃ r. (merge-coeffs xs, r) ∈ sorted-poly-rel ∧ normalize-poly-p*** ys
r›
 ⟨*proof*⟩

## 9.3 Normalisation

**definition** *normalize-poly* **where**
 ‹*normalize-poly p = do* {
   *p ← sort-poly-spec p*;
   *RETURN (merge-coeffs p)*
}›
**definition** *sort-coeff* :: ‹*string list* ⇒ *string list nres*› **where**
‹*sort-coeff ys = SPEC(λxs. mset xs = mset ys ∧ sorted-wrt (rel2p (Id ∪ var-order-rel)) xs)*›

**lemma** *distinct-var-order-Id-var-order*:
 ‹*distinct a ⟹ sorted-wrt (rel2p (Id ∪ var-order-rel)) a ⟹*
      *sorted-wrt var-order a*›
 ⟨*proof*⟩

**definition** *sort-all-coeffs* :: ‹*llist-polynomial* ⇒ *llist-polynomial nres*› **where**
‹*sort-all-coeffs xs = monadic-nfoldli xs* (λ-. *RETURN True*) (λ(*a, n*) *b. do* {*a* ← *sort-coeff a*; *RETURN*
((*a, n*) # *b*)}) []›

**lemma** *sort-all-coeffs-gen*:
  **assumes** ‹(∀ *xs* ∈ *mononoms xs′. sorted-wrt* (*rel2p* (*var-order-rel*)) *xs*)› **and**
    ‹∀ *x* ∈ *mononoms* (*xs @ xs′*). *distinct x*›
  **shows** ‹*monadic-nfoldli xs* (λ-. *RETURN True*) (λ(*a, n*) *b. do* {*a* ← *sort-coeff a*; *RETURN* ((*a, n*)
# *b*)}) *xs′* ≤
    ⇓*Id* (*SPEC*(λ*ys. map* (λ(*a,b*). (*mset a, b*)) (*rev xs @ xs′*) = *map* (λ(*a,b*). (*mset a, b*)) (*ys*) ∧
    (∀ *xs* ∈ *mononoms ys. sorted-wrt* (*rel2p* (*var-order-rel*)) *xs*)))›
  ⟨*proof*⟩

**definition** *shuffle-coefficients* **where**
  ‹*shuffle-coefficients xs* = (*SPEC*(λ*ys. map* (λ(*a,b*). (*mset a, b*)) (*rev xs*) = *map* (λ(*a,b*). (*mset a, b*))
*ys* ∧
    (∀ *xs* ∈ *mononoms ys. sorted-wrt* (*rel2p* (*var-order-rel*)) *xs*)))›

**lemma** *sort-all-coeffs*:
  ‹∀ *x* ∈ *mononoms xs. distinct x* ⟹
  *sort-all-coeffs xs* ≤ ⇓ *Id* (*shuffle-coefficients xs*)›
  ⟨*proof*⟩

**lemma** *unsorted-term-poly-list-rel-mset*:
  ‹(*ys, aa*) ∈ *unsorted-term-poly-list-rel* ⟹ *mset ys = aa*›
  ⟨*proof*⟩

**lemma** *RETURN-map-alt-def*:
  ‹*RETURN o* (*map f*) =
    *REC_T* (λ*g xs*.
      *case xs of*
        [] ⟹ *RETURN* []
      | *x* # *xs* ⟹ *do* {*xs* ← *g xs*; *RETURN* (*f x* # *xs*)})›
  ⟨*proof*⟩

**lemma** *fully-unsorted-poly-rel-Cons-iff*:
  ‹((*ys, n*) # *p, a*) ∈ *fully-unsorted-poly-rel* ⟷
    (*p, remove1-mset* (*mset ys, n*) *a*) ∈ *fully-unsorted-poly-rel* ∧
    (*mset ys, n*) ∈# *a* ∧ *distinct ys*›
  ⟨*proof*⟩

**lemma** *map-mset-unsorted-term-poly-list-rel*:
  ‹(⋀*a. a* ∈ *mononoms s* ⟹ *distinct a*) ⟹ ∀ *x* ∈ *mononoms s. distinct x* ⟹
    (∀ *xs* ∈ *mononoms s. sorted-wrt* (*rel2p* (*Id* ∪ *var-order-rel*)) *xs*) ⟹
    (*s, map* (λ(*a, y*). (*mset a, y*)) *s*)
      ∈ ⟨*term-poly-list-rel* ×_r *int-rel*⟩*list-rel*›
  ⟨*proof*⟩

**lemma** *list-rel-unsorted-term-poly-list-relD*:
  ‹(*p, y*) ∈ ⟨*unsorted-term-poly-list-rel* ×_r *int-rel*⟩*list-rel* ⟹
  *mset y* = (λ(*a, y*). (*mset a, y*)) '# *mset p* ∧ (∀ *x* ∈ *mononoms p. distinct x*)›
  ⟨*proof*⟩

**lemma** *shuffle-terms-distinct-iff*:

58

**assumes** ‹*map* (λ(*a*, *y*). (*mset a*, *y*)) *p* = *map* (λ(*a*, *y*). (*mset a*, *y*)) *s*›
  **shows** ‹(∀ *x*∈*set p*. *distinct* (*fst x*)) ⟷ (∀ *x*∈*set s*. *distinct* (*fst x*))›
⟨*proof*⟩

**lemma**
  ‹(*p*, *y*) ∈ ⟨*unsorted-term-poly-list-rel* ×$_r$ *int-rel*⟩*list-rel* ⟹
    (*a*, *b*) ∈ *set p* ⟹ *distinct a*›
  ⟨*proof*⟩

**lemma** *sort-all-coeffs-unsorted-poly-rel-with0*:
  **assumes** ‹(*p*, *p′*) ∈ *fully-unsorted-poly-rel*›
  **shows** ‹*sort-all-coeffs p* ≤ ⇓ (*unsorted-poly-rel-with0*) (*RETURN p′*)›
⟨*proof*⟩

**lemma** *sort-poly-spec-id′*:
  **assumes** ‹(*p*, *p′*) ∈ *unsorted-poly-rel-with0*›
  **shows** ‹*sort-poly-spec p* ≤ ⇓ (*sorted-repeat-poly-rel-with0*) (*RETURN p′*)›
⟨*proof*⟩

**fun** *merge-coeffs0* :: ‹*llist-polynomial* ⟹ *llist-polynomial*› **where**
  ‹*merge-coeffs0* [] = []› |
  ‹*merge-coeffs0* [(*xs*, *n*)] = (*if n* = *0 then* [] *else* [(*xs*, *n*)])› |
  ‹*merge-coeffs0* ((*xs*, *n*) # (*ys*, *m*) # *p*) =
    (*if xs* = *ys*
    *then if n* + *m* ≠ *0 then merge-coeffs0* ((*xs*, *n* + *m*) # *p*) *else merge-coeffs0 p*
    *else if n* = *0 then merge-coeffs0* ((*ys*, *m*) # *p*)
      *else*(*xs*, *n*) # *merge-coeffs0* ((*ys*, *m*) # *p*))›

**lemma** *sorted-repeat-poly-list-rel-with0-wrt-ConsD*:
  ‹((*ys*, *n*) # *p*, *a*) ∈ *sorted-repeat-poly-list-rel-with0-wrt S term-poly-list-rel* ⟹
    (*p*, *remove1-mset* (*mset ys*, *n*) *a*) ∈ *sorted-repeat-poly-list-rel-with0-wrt S term-poly-list-rel* ∧
    (*mset ys*, *n*) ∈# *a* ∧ (∀ *x* ∈ *set p*. *S ys* (*fst x*)) ∧ *sorted-wrt* (*rel2p var-order-rel*) *ys* ∧
    *distinct ys*›
  ⟨*proof*⟩

**lemma** *sorted-repeat-poly-list-rel-with0-wrtl-Cons-iff*:
  ‹((*ys*, *n*) # *p*, *a*) ∈ *sorted-repeat-poly-list-rel-with0-wrt S term-poly-list-rel* ⟷
    (*p*, *remove1-mset* (*mset ys*, *n*) *a*) ∈ *sorted-repeat-poly-list-rel-with0-wrt S term-poly-list-rel* ∧
    (*mset ys*, *n*) ∈# *a* ∧ (∀ *x* ∈ *set p*. *S ys* (*fst x*)) ∧ *sorted-wrt* (*rel2p var-order-rel*) *ys* ∧
    *distinct ys*›
  ⟨*proof*⟩

**lemma** *fst-normalize0-polynomial-subsetD*:
  ‹(*a*, *b*) ∈ *set* (*merge-coeffs0 p*) ⟹ *a* ∈ *mononoms p*›
  ⟨*proof*⟩

**lemma** *in-set-merge-coeffs0D*:
  ‹(*a*, *b*) ∈ *set* (*merge-coeffs0 p*) ⟹∃ *b*. (*a*, *b*) ∈ *set p*›
  ⟨*proof*⟩

**lemma** *merge-coeffs0-is-normalize-poly-p*:
  ‹(*xs*, *ys*) ∈ *sorted-repeat-poly-rel-with0* ⟹ ∃ *r*. (*merge-coeffs0 xs*, *r*) ∈ *sorted-poly-rel* ∧ *normalize-poly-p*$^{**}$

*ys r⟩*
  ⟨*proof*⟩

**definition** *full-normalize-poly* **where**
  ⟨*full-normalize-poly p = do {*
     *p ← sort-all-coeffs p;*
     *p ← sort-poly-spec p;*
     *RETURN (merge-coeffs0 p)*
  *}*⟩

**fun** *sorted-remdups* **where**
  ⟨*sorted-remdups (x # y # zs) =*
    *(if x = y then sorted-remdups (y # zs) else x # sorted-remdups (y # zs))*⟩ |
  ⟨*sorted-remdups zs = zs*⟩

**lemma** *set-sorted-remdups*[*simp*]:
  ⟨*set (sorted-remdups xs) = set xs*⟩
  ⟨*proof*⟩

**lemma** *distinct-sorted-remdups*:
  ⟨*sorted-wrt R xs ⟹ transp R ⟹ Restricted-Predicates.total-on R UNIV ⟹*
    *antisymp R ⟹ distinct (sorted-remdups xs)*⟩
  ⟨*proof*⟩

**lemma** *full-normalize-poly-normalize-poly-p*:
  **assumes** ⟨$(p, p') ∈$ *fully-unsorted-poly-rel*⟩
  **shows** ⟨*full-normalize-poly p ≤ ⇓ (sorted-poly-rel) (SPEC (λr. normalize-poly-p*** $p'$ *r))*⟩
  (**is** ⟨$?A ≤ ⇓ ?R ?B$⟩)
⟨*proof*⟩

**definition** *mult-poly-full* :: ⟨*-*⟩ **where**
⟨*mult-poly-full p q = do {*
  *let pq = mult-poly-raw p q;*
  *normalize-poly pq*
*}*⟩

**lemma** *normalize-poly-normalize-poly-p*:
  **assumes** ⟨$(p, p') ∈$ *unsorted-poly-rel*⟩
  **shows** ⟨*normalize-poly p ≤ ⇓ (sorted-poly-rel) (SPEC (λr. normalize-poly-p*** $p'$ *r))*⟩
⟨*proof*⟩

## 9.4 Multiplication and normalisation

**definition** *mult-poly-p′* :: ⟨*-*⟩ **where**
⟨*mult-poly-p′ p′ q′ = do {*
  *pq ← SPEC(λr. (mult-poly-p q′)*** (p′, {#}) ({#}, r));*
  *SPEC (λr. normalize-poly-p*** pq r)*
*}*⟩

**lemma** *unsorted-poly-rel-fully-unsorted-poly-rel*:
  ⟨*unsorted-poly-rel ⊆ fully-unsorted-poly-rel*⟩
⟨*proof*⟩

**lemma** *mult-poly-full-mult-poly-p′*:
  **assumes** ⟨$(p, p') ∈$ *sorted-poly-rel*⟩ ⟨$(q, q') ∈$ *sorted-poly-rel*⟩
  **shows** ⟨*mult-poly-full p q ≤ ⇓ (sorted-poly-rel) (mult-poly-p′ p′ q′)*⟩

⟨*proof*⟩

**definition** *add-poly-spec* :: ⟨*-*⟩ **where**
⟨*add-poly-spec p q = SPEC (λr. p + q − r ∈ ideal polynomial-bool)*⟩

**definition** *add-poly-p′* :: ⟨*-*⟩ **where**
⟨*add-poly-p′ p q = SPEC(λr. add-poly-p\*\* (p, q, {#}) ({#}, {#}, r))*⟩

**lemma** *add-poly-l-add-poly-p′*:
  **assumes** ⟨*(p, p′) ∈ sorted-poly-rel*⟩ ⟨*(q, q′) ∈ sorted-poly-rel*⟩
  **shows** ⟨*add-poly-l (p, q) ≤ ⇓ (sorted-poly-rel) (add-poly-p′ p′ q′)*⟩
  ⟨*proof*⟩

## 9.5 Correctness

**context** *poly-embed*
**begin**

**definition** *mset-poly-rel* **where**
  ⟨*mset-poly-rel = {(a, b). b = polynomial-of-mset a}*⟩

**definition** *var-rel* **where**
  ⟨*var-rel = br φ (λ-. True)*⟩

**lemma** *normalize-poly-p-normalize-poly-spec*:
  ⟨*(p, p′) ∈ mset-poly-rel ⟹*
    *SPEC (λr. normalize-poly-p\*\* p r) ≤ ⇓mset-poly-rel (normalize-poly-spec p′)*⟩
  ⟨*proof*⟩

**lemma** *mult-poly-p′-mult-poly-spec*:
  ⟨*(p, p′) ∈ mset-poly-rel ⟹ (q, q′) ∈ mset-poly-rel ⟹*
  *mult-poly-p′ p q ≤ ⇓mset-poly-rel (mult-poly-spec p′ q′)*⟩
  ⟨*proof*⟩

**lemma** *add-poly-p′-add-poly-spec*:
  ⟨*(p, p′) ∈ mset-poly-rel ⟹ (q, q′) ∈ mset-poly-rel ⟹*
  *add-poly-p′ p q ≤ ⇓mset-poly-rel (add-poly-spec p′ q′)*⟩
  ⟨*proof*⟩

**end**

**definition** *weak-equality-l* :: ⟨*llist-polynomial ⇒ llist-polynomial ⇒ bool nres*⟩ **where**
  ⟨*weak-equality-l p q = RETURN (p = q)*⟩

**definition** *weak-equality* :: ⟨*int mpoly ⇒ int mpoly ⇒ bool nres*⟩ **where**
  ⟨*weak-equality p q = SPEC (λr. r ⟶ p = q)*⟩

**definition** *weak-equality-spec* :: ⟨*mset-polynomial ⇒ mset-polynomial ⇒ bool nres*⟩ **where**
  ⟨*weak-equality-spec p q = SPEC (λr. r ⟶ p = q)*⟩

**lemma** *term-poly-list-rel-same-rightD*:
  ⟨*(a, aa) ∈ term-poly-list-rel ⟹ (a, ab) ∈ term-poly-list-rel ⟹ aa = ab*⟩
    ⟨*proof*⟩

**lemma** *list-rel-term-poly-list-rel-same-rightD*:
⟨(xa, y) ∈ ⟨term-poly-list-rel ×ᵣ int-rel⟩list-rel ⟹
(xa, ya) ∈ ⟨term-poly-list-rel ×ᵣ int-rel⟩list-rel ⟹
y = ya⟩
⟨proof⟩

**lemma** *weak-equality-l-weak-equality-spec*:
⟨(uncurry weak-equality-l, uncurry weak-equality-spec) ∈
sorted-poly-rel ×ᵣ sorted-poly-rel →_f ⟨bool-rel⟩nres-rel⟩
⟨proof⟩

**end**
**theory** *PAC-Checker*
  **imports** *PAC-Polynomials-Operations*
    *PAC-Checker-Specification*
    *PAC-Map-Rel*
    *Show.Show*
    *Show.Show-Instances*
**begin**

# 10 Executable Checker

In this layer we finally refine the checker to executable code.

## 10.1 Definitions

Compared to the previous layer, we add an error message when an error is discovered. We do not attempt to prove anything on the error message (neither that there really is an error, nor that the error message is correct).

**Extended error message** **datatype** $'a$ *code-status* =
*is-cfailed*: *CFAILED* (*the-error*: $'a$) |
*CSUCCESS* |
*is-cfound*: *CFOUND*

In the following function, we merge errors. We will never merge an error message with an another error message; hence we do not attempt to concatenate error messages.

**fun** *merge-cstatus* **where**
⟨merge-cstatus (CFAILED a) - = CFAILED a⟩ |
⟨merge-cstatus - (CFAILED a) = CFAILED a⟩ |
⟨merge-cstatus CFOUND - = CFOUND⟩ |
⟨merge-cstatus - CFOUND = CFOUND⟩ |
⟨merge-cstatus - - = CSUCCESS⟩

**definition** *code-status-status-rel* :: ⟨($'a$ code-status × status) set⟩ **where**
⟨code-status-status-rel =
{(CFOUND, FOUND), (CSUCCESS, SUCCESS)} ∪
{(CFAILED a, FAILED)| a. True}⟩

**lemma** *in-code-status-status-rel-iff* [*simp*]:
⟨(CFOUND, b) ∈ code-status-status-rel ⟷ b = FOUND⟩
⟨(a, FOUND) ∈ code-status-status-rel ⟷ a = CFOUND⟩

⟨(*CSUCCESS*, b) ∈ *code-status-status-rel* ⟷ b = *SUCCESS*⟩
⟨(a, *SUCCESS*) ∈ *code-status-status-rel* ⟷ a = *CSUCCESS*⟩
⟨(a, *FAILED*) ∈ *code-status-status-rel* ⟷ *is-cfailed a*⟩
⟨(*CFAILED C*, b) ∈ *code-status-status-rel* ⟷ b = *FAILED*⟩
⟨*proof*⟩

**Refinement relation** **fun** *pac-step-rel-raw* :: ⟨(′*olbl* × ′*lbl*) *set* ⇒ (′*a* × ′*b*) *set* ⇒ (′*c* × ′*d*) *set* ⇒ (′*a*, ′*c*, ′*olbl*) *pac-step* ⇒ (′*b*, ′*d*, ′*lbl*) *pac-step* ⇒ *bool*⟩ **where**
⟨*pac-step-rel-raw R1 R2 R3* (*Add p1 p2 i r*) (*Add p1′ p2′ i′ r′*) ⟷
  (*p1*, *p1′*) ∈ *R1* ∧ (*p2*, *p2′*) ∈ *R1* ∧ (*i*, *i′*) ∈ *R1* ∧
  (*r*, *r′*) ∈ *R2*⟩ |
⟨*pac-step-rel-raw R1 R2 R3* (*Mult p1 p2 i r*) (*Mult p1′ p2′ i′ r′*) ⟷
  (*p1*, *p1′*) ∈ *R1* ∧ (*p2*, *p2′*) ∈ *R2* ∧ (*i*, *i′*) ∈ *R1* ∧
  (*r*, *r′*) ∈ *R2*⟩ |
⟨*pac-step-rel-raw R1 R2 R3* (*Del p1*) (*Del p1′*) ⟷
  (*p1*, *p1′*) ∈ *R1*⟩ |
⟨*pac-step-rel-raw R1 R2 R3* (*Extension i x p1*) (*Extension j x′ p1′*) ⟷
  (*i*, *j*) ∈ *R1* ∧ (*x*, *x′*) ∈ *R3* ∧ (*p1*, *p1′*) ∈ *R2*⟩ |
⟨*pac-step-rel-raw R1 R2 R3* - - ⟷ *False*⟩

**fun** *pac-step-rel-assn* :: ⟨(′*olbl* ⇒ ′*lbl* ⇒ *assn*) ⇒ (′*a* ⇒ ′*b* ⇒ *assn*) ⇒ (′*c* ⇒ ′*d* ⇒ *assn*) ⇒ (′*a*, ′*c*, ′*olbl*) *pac-step* ⇒ (′*b*, ′*d*, ′*lbl*) *pac-step* ⇒ *assn*⟩ **where**
⟨*pac-step-rel-assn R1 R2 R3* (*Add p1 p2 i r*) (*Add p1′ p2′ i′ r′*) =
  *R1 p1 p1′* ∗ *R1 p2 p2′* ∗ *R1 i i′* ∗
  *R2 r r′*⟩ |
⟨*pac-step-rel-assn R1 R2 R3* (*Mult p1 p2 i r*) (*Mult p1′ p2′ i′ r′*) =
  *R1 p1 p1′* ∗ *R2 p2 p2′* ∗ *R1 i i′* ∗
  *R2 r r′*⟩ |
⟨*pac-step-rel-assn R1 R2 R3* (*Del p1*) (*Del p1′*) =
  *R1 p1 p1′*⟩ |
⟨*pac-step-rel-assn R1 R2 R3* (*Extension i x p1*) (*Extension i′ x′ p1′*) =
  *R1 i i′* ∗ *R3 x x′* ∗ *R2 p1 p1′*⟩ |
⟨*pac-step-rel-assn R1 R2* - - - = *false*⟩

**lemma** *pac-step-rel-assn-alt-def*:
 ⟨*pac-step-rel-assn R1 R2 R3 x y* = (
 *case* (*x*, *y*) *of*
   (*Add p1 p2 i r*, *Add p1′ p2′ i′ r′*) ⇒
    *R1 p1 p1′* ∗ *R1 p2 p2′* ∗ *R1 i i′* ∗ *R2 r r′*
  | (*Mult p1 p2 i r*, *Mult p1′ p2′ i′ r′*) ⇒
    *R1 p1 p1′* ∗ *R2 p2 p2′* ∗ *R1 i i′* ∗ *R2 r r′*
  | (*Del p1*, *Del p1′*) ⇒ *R1 p1 p1′*
  | (*Extension i x p1*, *Extension i′ x′ p1′*) ⇒ *R1 i i′* ∗ *R3 x x′* ∗ *R2 p1 p1′*
  | - ⇒ *false*
  )⟩
 ⟨*proof*⟩

**Addition checking** **definition** *error-msg* **where**
 ⟨*error-msg i msg* = *CFAILED* (″*s CHECKING failed at line* ″ @ *show i* @ ″ *with error* ″ @ *msg*)⟩

**definition** *error-msg-notin-dom-err* **where**
 ⟨*error-msg-notin-dom-err* = ″ *notin domain*″⟩

**definition** *error-msg-notin-dom* :: ⟨*nat* ⇒ *string*⟩ **where**
 ⟨*error-msg-notin-dom i* = *show i* @ *error-msg-notin-dom-err*⟩

**definition** *error-msg-reused-dom* **where**
⟨*error-msg-reused-dom i = show i @ '' already in domain''*⟩

**definition** *error-msg-not-equal-dom* **where**
⟨*error-msg-not-equal-dom p q pq r = show p @ '' + '' @ show q @ '' = '' @ show pq @ '' not equal''*
@ *show r*⟩

**definition** *check-not-equal-dom-err* :: ⟨*llist-polynomial ⇒ llist-polynomial ⇒ llist-polynomial ⇒ llist-polynomial*
⇒ *string nres*⟩ **where**
⟨*check-not-equal-dom-err p q pq r = SPEC (λ-. True)*⟩

**definition** *vars-llist* :: ⟨*llist-polynomial ⇒ string set*⟩ **where**
⟨*vars-llist  xs = ⋃(set ' fst ' set xs)*⟩

**definition** *check-addition-l* :: ⟨*- ⇒ - ⇒ string set ⇒ nat ⇒ nat ⇒ nat ⇒ llist-polynomial ⇒ string*
*code-status nres*⟩ **where**
⟨*check-addition-l spec A 𝒱 p q i r = do {*
  *let b = p ∈# dom-m A ∧ q ∈# dom-m A ∧ i ∉# dom-m A ∧ vars-llist r ⊆ 𝒱;*
  *if ¬b*
  *then RETURN (error-msg i ((if p ∉# dom-m A then error-msg-notin-dom p else []) @ (if q ∉#*
*dom-m A then error-msg-notin-dom p else []) @*
    *(if i ∈# dom-m A then error-msg-reused-dom p else [])))*
  *else do {*
    *ASSERT (p ∈# dom-m A);*
    *let p = the (fmlookup A p);*
    *ASSERT (q ∈# dom-m A);*
    *let q = the (fmlookup A q);*
    *pq ← add-poly-l (p, q);*
    *b ← weak-equality-l pq r;*
    *b' ← weak-equality-l r spec;*
    *if b then (if b' then RETURN CFOUND else RETURN CSUCCESS)*
    *else do {*
      *c ← check-not-equal-dom-err p q pq r;*
      *RETURN (error-msg i c)}*
  *}*
*}*⟩

**Multiplication checking**   **definition** *check-mult-l-dom-err* :: ⟨*bool ⇒ nat ⇒ bool ⇒ nat ⇒ string*
*nres*⟩ **where**
⟨*check-mult-l-dom-err p-notin p i-already i = SPEC (λ-. True)*⟩

**definition** *check-mult-l-mult-err* :: ⟨*llist-polynomial ⇒ llist-polynomial ⇒ llist-polynomial ⇒ llist-polynomial*
⇒ *string nres*⟩ **where**
⟨*check-mult-l-mult-err p q pq r = SPEC (λ-. True)*⟩

**definition** *check-mult-l* :: ⟨*- ⇒ - ⇒ - ⇒ nat ⇒llist-polynomial ⇒  nat ⇒ llist-polynomial ⇒ string*
*code-status nres*⟩ **where**
⟨*check-mult-l spec A 𝒱 p q i r = do {*

```
    let b = p ∈# dom-m A ∧ i ∉# dom-m A ∧ vars-llist q ⊆ V∧ vars-llist r ⊆ V;
    if ¬b
    then do {
      c ← check-mult-l-dom-err (p ∉# dom-m A) p (i ∈# dom-m A) i;
      RETURN (error-msg i c)}
    else do {
        ASSERT (p ∈# dom-m A);
        let p = the (fmlookup A p);
        pq ← mult-poly-full p q;
        b ← weak-equality-l pq r;
        b' ← weak-equality-l r spec;
        if b then (if b' then RETURN CFOUND else RETURN CSUCCESS) else do {
          c ← check-mult-l-mult-err p q pq r;
          RETURN (error-msg i c)
        }
      }
    }⟩
```

Deletion checking   **definition** *check-del-l* :: ⟨- ⇒ - ⇒ *nat* ⇒ *string code-status nres*⟩ **where**
⟨*check-del-l spec A p = RETURN CSUCCESS*⟩

Extension checking   **definition** *check-extension-l-dom-err* :: ⟨*nat* ⇒ *string nres*⟩ **where**
  ⟨*check-extension-l-dom-err p = SPEC (λ-. True)*⟩

definition *check-extension-l-no-new-var-err* :: ⟨*llist-polynomial* ⇒ *string nres*⟩ **where**
  ⟨*check-extension-l-no-new-var-err p = SPEC (λ-. True)*⟩

definition *check-extension-l-new-var-multiple-err* :: ⟨*string* ⇒ *llist-polynomial* ⇒ *string nres*⟩ **where**
  ⟨*check-extension-l-new-var-multiple-err v p = SPEC (λ-. True)*⟩

definition *check-extension-l-side-cond-err*
  :: ⟨*string* ⇒ *llist-polynomial* ⇒ *llist-polynomial* ⇒ *llist-polynomial* ⇒ *string nres*⟩
where
  ⟨*check-extension-l-side-cond-err v p p' q = SPEC (λ-. True)*⟩

definition *check-extension-l*
  :: ⟨- ⇒ - ⇒ *string set* ⇒ *nat* ⇒ *string* ⇒ *llist-polynomial* ⇒ (*string code-status*) *nres*⟩
where
⟨*check-extension-l spec A V i v p = do {*
  let b = i ∉# dom-m A ∧ v ∉ V ∧ ([v], −1) ∈ set p;
  if ¬b
  then do {
    c ← check-extension-l-dom-err i;
    RETURN (error-msg i c)
  } else do {
      let p' = remove1 ([v], −1) p;
      let b = vars-llist p' ⊆ V;
      if ¬b
      then do {
        c ← check-extension-l-new-var-multiple-err v p';
        RETURN (error-msg i c)
      }
      else do {
        p2 ← mult-poly-full p' p';
```

```
      let p' = map (λ(a,b). (a, −b)) p';
      q ← add-poly-l (p2, p');
      eq ← weak-equality-l q [];
      if eq then do {
        RETURN (CSUCCESS)
      } else do {
        c ← check-extension-l-side-cond-err v p p' q;
        RETURN (error-msg i c)
      }
    }
  }
}⟩
```

lemma *check-extension-alt-def*:
  ⟨*check-extension A 𝒱 i v p ≥ do {*
    b ← SPEC(λb. b ⟶ i ∉# dom-m A ∧ v ∉ 𝒱);
    if ¬b
    then RETURN (False)
    else do {
      p' ← RETURN (p + Var v);
      b ← SPEC(λb. b ⟶ vars p' ⊆ 𝒱);
      if ¬b
      then RETURN (False)
      else do {
        pq ← mult-poly-spec p' p';
        let p' = − p';
        p ← add-poly-spec pq p';
        eq ← weak-equality p 0;
        if eq then RETURN(True)
        else RETURN (False)
      }
    }
  }⟩
⟨*proof*⟩

lemma *RES-RES-RETURN-RES*: ⟨*RES A ≫= (λT. RES (f T)) = RES (⋃(f ' A))*⟩
  ⟨*proof*⟩

lemma *check-add-alt-def*:
  ⟨*check-add A 𝒱 p q i r ≥*
    do {
      b ← SPEC(λb. b ⟶ p ∈# dom-m A ∧ q ∈# dom-m A ∧ i ∉# dom-m A ∧ vars r ⊆ 𝒱);
      if ¬b
      then RETURN False
      else do {
        ASSERT (p ∈# dom-m A);
        let p = the (fmlookup A p);
        ASSERT (q ∈# dom-m A);
        let q = the (fmlookup A q);
        pq ← add-poly-spec p q;
        eq ← weak-equality pq r;
        RETURN eq
```

```
        }
      }⟩ (is ⟨- ≥ ?A⟩)
⟨proof⟩


lemma check-mult-alt-def:
  ⟨check-mult A 𝒱 p q i r ≥
    do {
      b ← SPEC(λb. b ⟶ p ∈# dom-m A ∧ i ∉# dom-m A ∧ vars q ⊆ 𝒱 ∧ vars r ⊆ 𝒱);
      if ¬b
      then RETURN False
      else do {
        ASSERT (p ∈# dom-m A);
        let p = the (fmlookup A p);
        pq ← mult-poly-spec p q;
        p ← weak-equality pq r;
        RETURN p
      }
    }⟩
  ⟨proof⟩


primrec insort-key-rel :: (′b ⇒ ′b ⇒ bool) ⇒ ′b ⇒ ′b list ⇒ ′b list where
insort-key-rel f x [] = [x] |
insort-key-rel f x (y#ys) =
  (if f x y then (x#y#ys) else y#(insort-key-rel f x ys))


lemma set-insort-key-rel[simp]: ⟨set (insort-key-rel R x xs) = insert x (set xs)⟩
  ⟨proof⟩


lemma sorted-wrt-insort-key-rel:
  ⟨total-on R (insert x (set xs)) ⟹ transp R ⟹ reflp R ⟹
   sorted-wrt R xs ⟹ sorted-wrt R (insort-key-rel R x xs)⟩
  ⟨proof⟩


lemma sorted-wrt-insort-key-rel2:
  ⟨total-on R (insert x (set xs)) ⟹ transp R ⟹ x ∉ set xs ⟹
   sorted-wrt R xs ⟹ sorted-wrt R (insort-key-rel R x xs)⟩
  ⟨proof⟩


**Step checking**  definition PAC-checker-l-step :: ⟨- ⇒ string code-status × string set × - ⇒ (llist-polynomial,
string, nat) pac-step ⇒ -⟩ where
  ⟨PAC-checker-l-step = (λspec (st′, 𝒱, A) st. case st of
    Add - - - - ⇒
      do {
        r ← full-normalize-poly (pac-res st);
        eq ← check-addition-l spec A 𝒱 (pac-src1 st) (pac-src2 st) (new-id st) r;
        let - = eq;
        if ¬is-cfailed eq
        then RETURN (merge-cstatus st′ eq,
          𝒱, fmupd (new-id st) r A)
        else RETURN (eq, 𝒱, A)
    }
  | Del - ⇒
      do {
        eq ← check-del-l spec A (pac-src1 st);
        let - = eq;
```

   if ¬*is-cfailed eq*
   then RETURN (*merge-cstatus st' eq*, 𝒱, *fmdrop* (*pac-src1 st*) *A*)
   else RETURN (*eq*, 𝒱, *A*)
  }
  | Mult - - - - ⇒
   do {
    r ← full-normalize-poly (*pac-res st*);
    q ← full-normalize-poly (*pac-mult st*);
    eq ← check-mult-l spec A 𝒱 (*pac-src1 st*) *q* (*new-id st*) *r*;
    let - = eq;
    if ¬*is-cfailed eq*
    then RETURN (*merge-cstatus st' eq*,
     𝒱, *fmupd* (*new-id st*) *r A*)
    else RETURN (*eq*, 𝒱, *A*)
  }
  | Extension - - - ⇒
   do {
    r ← full-normalize-poly (([*new-var st*], −1) # (*pac-res st*));
    (*eq*) ← *check-extension-l spec A* 𝒱 (*new-id st*) (*new-var st*) *r*;
    if ¬*is-cfailed eq*
    then do {
     RETURN (*st'*,
      insert (*new-var st*) 𝒱, *fmupd* (*new-id st*) *r A*)*}*
    else RETURN (*eq*, 𝒱, *A*)
  }
)⟩

lemma *pac-step-rel-raw-def*:
 ⟨⟨*K*, *V*, *R*⟩ *pac-step-rel-raw = pac-step-rel-raw K V R*⟩
 ⟨*proof*⟩

definition *mononoms-equal-up-to-reorder* **where**
 ⟨*mononoms-equal-up-to-reorder xs ys* ⟷
  map (λ(*a*, *b*). (*mset a*, *b*)) *xs = map* (λ(*a*, *b*). (*mset a*, *b*)) *ys*⟩

definition *normalize-poly-l* **where**
 ⟨*normalize-poly-l p = SPEC* (λ*p'*.
  normalize-poly-p** ((λ(*a*, *b*). (*mset a*, *b*)) '# *mset p*) ((λ(*a*, *b*). (*mset a*, *b*)) '# *mset p'*) ∧
  0 ∉# *snd* '# *mset p'* ∧
  sorted-wrt (*rel2p* (*term-order-rel* ×$_r$ *int-rel*)) *p'* ∧
  (∀ *x* ∈ *mononoms p'*. *sorted-wrt* (*rel2p var-order-rel*) *x*))⟩

definition *remap-polys-l-dom-err* :: ⟨*string nres*⟩ **where**
 ⟨*remap-polys-l-dom-err = SPEC* (λ-. *True*)⟩

definition *remap-polys-l* :: ⟨*llist-polynomial* ⇒ *string set* ⇒ (*nat*, *llist-polynomial*) *fmap* ⇒
 (- *code-status* × *string set* × (*nat*, *llist-polynomial*) *fmap*) *nres*⟩ **where**
 ⟨*remap-polys-l spec* = (λ𝒱 *A*. *do{*
 dom ← *SPEC*(λ*dom*. *set-mset* (*dom-m A*) ⊆ *dom* ∧ *finite dom*);
 failed ← *SPEC*(λ-::*bool*. *True*);
 if failed
 then do {

```
     c ← remap-polys-l-dom-err;
     RETURN (error-msg (0 :: nat) c, 𝒱, fmempty)
   }
   else do {
     (b, 𝒱, A) ← FOREACH dom
       (λi (b, 𝒱,  A′).
           if i ∈# dom-m A
           then  do {
             p ← full-normalize-poly (the (fmlookup A i));
             eq ← weak-equality-l p spec;
             𝒱 ← RETURN(𝒱 ∪ vars-llist (the (fmlookup A i)));
             RETURN(b ∨ eq, 𝒱, fmupd i p A′)
           } else RETURN (b, 𝒱, A′))
       (False, 𝒱, fmempty);
     RETURN (if b then CFOUND else CSUCCESS, 𝒱, A)
 }}})›
```

definition *PAC-checker-l* **where**
```
 ‹PAC-checker-l spec A b st = do {
   (S, -) ← WHILE_T
     (λ((b, A), n). ¬is-cfailed b ∧ n ≠ [])
     (λ((bA), n). do {
        ASSERT(n ≠ []);
        S ← PAC-checker-l-step spec bA (hd n);
        RETURN (S, tl n)
     })
     ((b, A), st);
   RETURN S
 }›
```

10.2   Correctness

We now enter the locale to reason about polynomials directly.

context *poly-embed*
begin

abbreviation *pac-step-rel* **where**
 ‹*pac-step-rel* ≡ *p2rel* (⟨*Id, fully-unsorted-poly-rel O mset-poly-rel, var-rel*⟩ *pac-step-rel-raw*)›

abbreviation *fmap-polys-rel* **where**
 ‹*fmap-polys-rel* ≡ ⟨*nat-rel, sorted-poly-rel O mset-poly-rel*⟩*fmap-rel*›

lemma
 ‹*normalize-poly-p s0 s* ⟹
     (*s0, p*) ∈ *mset-poly-rel* ⟹
     (*s, p*) ∈ *mset-poly-rel*›
 ⟨*proof*⟩

lemma *vars-poly-of-vars*:
 ‹*vars (poly-of-vars a :: int mpoly*) ⊆ (φ ' *set-mset a*)›
 ⟨*proof*⟩

lemma *vars-polynomial-of-mset*:
 ‹*vars (polynomial-of-mset za*) ⊆ ⋃ (*image* φ ' (*set-mset o fst*) ' *set-mset za*)›
 ⟨*proof*⟩

lemma *fully-unsorted-poly-rel-vars-subset-vars-llist*:
⟨(A, B) ∈ *fully-unsorted-poly-rel* O *mset-poly-rel* ⟹ *vars B* ⊆ φ ' *vars-llist A*⟩
⟨*proof*⟩

lemma *fully-unsorted-poly-rel-extend-vars*:
⟨(A, B) ∈ *fully-unsorted-poly-rel* O *mset-poly-rel* ⟹
(x1c, x1a) ∈ ⟨*var-rel*⟩*set-rel* ⟹
 RETURN (x1c ∪ *vars-llist A*)
  ≤ ⇓ (⟨*var-rel*⟩*set-rel*)
    (*SPEC* ((⊆) (x1a ∪ *vars* (B))))⟩
⟨*proof*⟩

lemma *remap-polys-l-remap-polys*:
  assumes
    AB: ⟨(A, B) ∈ ⟨*nat-rel*, *fully-unsorted-poly-rel* O *mset-poly-rel*⟩*fmap-rel*⟩ **and**
    spec: ⟨(spec, spec′) ∈ *sorted-poly-rel* O *mset-poly-rel*⟩ **and**
    V: ⟨(𝒱, 𝒱′) ∈ ⟨*var-rel*⟩*set-rel*⟩
  shows ⟨*remap-polys-l spec* 𝒱 *A* ≤
    ⇓(*code-status-status-rel* ×_r ⟨*var-rel*⟩*set-rel* ×_r *fmap-polys-rel*) (*remap-polys spec*′ 𝒱′ *B*)⟩
  (**is** ⟨- ≤ ⇓ ?R -⟩)
⟨*proof*⟩

lemma *fref-to-Down-curry*:
⟨(*uncurry f*, *uncurry g*) ∈ [P]_f *A* → ⟨B⟩*nres-rel* ⟹
  (⋀x x′ y y′. *P* (x′, y′) ⟹ ((x, y), (x′, y′)) ∈ *A* ⟹ *f x y* ≤ ⇓ *B* (*g x*′ y′))⟩
⟨*proof*⟩

lemma *weak-equality-spec-weak-equality*:
⟨(p, p′) ∈ *mset-poly-rel* ⟹
  (r, r′) ∈ *mset-poly-rel* ⟹
  weak-equality-spec p r ≤ *weak-equality p*′ r′⟩
⟨*proof*⟩

lemma *weak-equality-l-weak-equality-l*′[*refine*]:
⟨*weak-equality-l p q* ≤ ⇓ *bool-rel* (*weak-equality p*′ q′)⟩
  if ⟨(p, p′) ∈ *sorted-poly-rel* O *mset-poly-rel*⟩
   ⟨(q, q′) ∈ *sorted-poly-rel* O *mset-poly-rel*⟩
  for p p′ q q′
⟨*proof*⟩

lemma *error-msg-ne-SUCCES*[*iff*]:
⟨*error-msg i m* ≠ *CSUCCESS*⟩
⟨*error-msg i m* ≠ *CFOUND*⟩
⟨*is-cfailed* (*error-msg i m*)⟩
⟨¬*is-cfound* (*error-msg i m*)⟩
⟨*proof*⟩

lemma *sorted-poly-rel-vars-llist*:
⟨(r, r′) ∈ *sorted-poly-rel* O *mset-poly-rel* ⟹
  vars r′ ⊆ φ ' *vars-llist r*⟩
⟨*proof*⟩

lemma *check-addition-l-check-add*:
  assumes ⟨(A, B) ∈ fmap-polys-rel⟩ **and** ⟨(r, r′) ∈ sorted-poly-rel O mset-poly-rel⟩
    ⟨(p, p′) ∈ Id⟩ ⟨(q, q′) ∈ Id⟩ ⟨(i, i′) ∈ nat-rel⟩
    ⟨(𝒱′, 𝒱) ∈ ⟨var-rel⟩set-rel⟩
  shows
    ⟨check-addition-l spec A 𝒱′ p q i r ≤ ⇓ {(st, b). (¬is-cfailed st ⟷ b) ∧
      (is-cfound st ⟶ spec = r)} (check-add B 𝒱 p′ q′ i′ r′)⟩
⟨*proof*⟩

lemma *check-del-l-check-del*:
  ⟨(A, B) ∈ fmap-polys-rel ⟹ (x3, x3a) ∈ Id ⟹ check-del-l spec A (pac-src1 (Del x3))
    ≤ ⇓ {(st, b). (¬is-cfailed st ⟷ b) ∧ (b ⟶ st = CSUCCESS)} (check-del B (pac-src1 (Del x3a)))⟩
  ⟨*proof*⟩

lemma *check-mult-l-check-mult*:
  assumes ⟨(A, B) ∈ fmap-polys-rel⟩ **and** ⟨(r, r′) ∈ sorted-poly-rel O mset-poly-rel⟩ **and**
    ⟨(q, q′) ∈ sorted-poly-rel O mset-poly-rel⟩
    ⟨(p, p′) ∈ Id⟩ ⟨(i, i′) ∈ nat-rel⟩ ⟨(𝒱, 𝒱′) ∈ ⟨var-rel⟩set-rel⟩
  shows
    ⟨check-mult-l spec A 𝒱 p q i r ≤ ⇓ {(st, b). (¬is-cfailed st ⟷ b) ∧
      (is-cfound st ⟶ spec = r)} (check-mult B 𝒱′ p′ q′ i′ r′)⟩
⟨*proof*⟩

lemma *normalize-poly-normalize-poly-spec*:
  assumes ⟨(r, t) ∈ unsorted-poly-rel O mset-poly-rel⟩
  shows
    ⟨normalize-poly r ≤ ⇓(sorted-poly-rel O mset-poly-rel) (normalize-poly-spec t)⟩
⟨*proof*⟩

lemma *remove1-list-rel*:
  ⟨(xs, ys) ∈ ⟨R⟩ list-rel ⟹
  (a, b) ∈ R ⟹
  IS-RIGHT-UNIQUE R ⟹
  IS-LEFT-UNIQUE R ⟹
  (remove1 a xs, remove1 b ys) ∈ ⟨R⟩list-rel⟩
  ⟨*proof*⟩

lemma *remove1-list-rel2*:
  ⟨(xs, ys) ∈ ⟨R⟩ list-rel ⟹
  (a, b) ∈ R ⟹
  (⋀c. (a, c) ∈ R ⟹ c = b) ⟹
  (⋀c. (c, b) ∈ R ⟹ c = a) ⟹
  (remove1 a xs, remove1 b ys) ∈ ⟨R⟩list-rel⟩
  ⟨*proof*⟩

lemma *remove1-sorted-poly-rel-mset-poly-rel*:
  assumes
    ⟨(r, r′) ∈ sorted-poly-rel O mset-poly-rel⟩ **and**
    ⟨([a], 1) ∈ set r⟩
  shows
    ⟨(remove1 ([a], 1) r, r′ − Var (φ a))
      ∈ sorted-poly-rel O mset-poly-rel⟩
⟨*proof*⟩

lemma *remove1-sorted-poly-rel-mset-poly-rel-minus*:
  assumes
    ⟨$(r, r') ∈$ *sorted-poly-rel O mset-poly-rel*⟩ **and**
    ⟨$([a], -1) ∈$ *set r*⟩
  shows
    ⟨$(remove1 \; ([a], -1) \; r, \; r' + Var \; (\varphi \; a))$
        $∈$ *sorted-poly-rel O mset-poly-rel*⟩
⟨*proof*⟩

lemma *insert-var-rel-set-rel*:
  ⟨$(\mathcal{V}, \mathcal{V}') ∈ ⟨var\text{-}rel⟩set\text{-}rel \implies$
  $(yb, x2) ∈ var\text{-}rel \implies$
  $(insert \; yb \; \mathcal{V}, \; insert \; x2 \; \mathcal{V}') ∈ ⟨var\text{-}rel⟩set\text{-}rel$⟩
  ⟨*proof*⟩

lemma *var-rel-set-rel-iff*:
  ⟨$(\mathcal{V}, \mathcal{V}') ∈ ⟨var\text{-}rel⟩set\text{-}rel \implies$
  $(yb, x2) ∈ var\text{-}rel \implies$
  $yb ∈ \mathcal{V} \longleftrightarrow x2 ∈ \mathcal{V}'$⟩
  ⟨*proof*⟩

lemma *check-extension-l-check-extension*:
  assumes ⟨$(A, B) ∈$ *fmap-polys-rel*⟩ **and** ⟨$(r, r') ∈$ *sorted-poly-rel O mset-poly-rel*⟩ **and**
    ⟨$(i, i') ∈$ *nat-rel*⟩ ⟨$(\mathcal{V}, \mathcal{V}') ∈ ⟨var\text{-}rel⟩set\text{-}rel$⟩ ⟨$(x, x') ∈ var\text{-}rel$⟩
  shows
    ⟨*check-extension-l spec A* $\mathcal{V}$ *i x r* $\le$
      $\Downarrow\{((st), (b)).$
        $(\neg is\text{-}cfailed \; st \longleftrightarrow b) \; \wedge$
        $(is\text{-}cfound \; st \longrightarrow spec = r)\} \; (check\text{-}extension \; B \; \mathcal{V}' \; i' \; x' \; r')$⟩
⟨*proof*⟩

lemma *full-normalize-poly-diff-ideal*:
  fixes *dom*
  assumes ⟨$(p, p') ∈$ *fully-unsorted-poly-rel O mset-poly-rel*⟩
  shows
    ⟨*full-normalize-poly p*
    $\le \Downarrow$ (*sorted-poly-rel O mset-poly-rel*)
      (*normalize-poly-spec p'*)⟩
⟨*proof*⟩

lemma *insort-key-rel-decomp*:
  ⟨$\exists ys \; zs. \; xs = ys \; @ \; zs \wedge insort\text{-}key\text{-}rel \; R \; x \; xs = ys \; @ \; x \; \# \; zs$⟩
  ⟨*proof*⟩

lemma *list-rel-append-same-length*:
  ⟨*length xs = length xs'* $\implies (xs \; @ \; ys, \; xs' \; @ \; ys') ∈ ⟨R⟩list\text{-}rel \longleftrightarrow (xs, xs') ∈ ⟨R⟩list\text{-}rel \wedge (ys, ys') ∈$
$⟨R⟩list\text{-}rel$⟩
  ⟨*proof*⟩

lemma *term-poly-list-rel-list-relD*: ⟨$(ys, cs) ∈ ⟨term\text{-}poly\text{-}list\text{-}rel \times_r int\text{-}rel⟩list\text{-}rel \implies$
    $cs = map \; (\lambda(a, y). \; (mset \; a, \; y)) \; ys$⟩

$\langle proof \rangle$

lemma *term-poly-list-rel-single*: $\langle ([x32], \{\#x32\#\}) \in \text{term-poly-list-rel} \rangle$
  $\langle proof \rangle$

lemma *unsorted-poly-rel-list-rel-list-rel-uminus*:
  $\langle (map\ (\lambda(a,\ b).\ (a,\ -\ b))\ r,\ yc)$
    $\in \langle \text{unsorted-term-poly-list-rel} \times_r \text{int-rel} \rangle \text{list-rel} \Longrightarrow$
    $(r,\ map\ (\lambda(a,\ b).\ (a,\ -\ b))\ yc)$
    $\in \langle \text{unsorted-term-poly-list-rel} \times_r \text{int-rel} \rangle \text{list-rel} \rangle$
  $\langle proof \rangle$

lemma *mset-poly-rel-minus*: $\langle (\{\#(a,\ b)\#\},\ v') \in \text{mset-poly-rel} \Longrightarrow$
    $(mset\ yc,\ r') \in \text{mset-poly-rel} \Longrightarrow$
    $(r,\ yc)$
    $\in \langle \text{unsorted-term-poly-list-rel} \times_r \text{int-rel} \rangle \text{list-rel} \Longrightarrow$
    $(\text{add-mset}\ (a,\ b)\ (mset\ yc),$
     $v'\ +\ r')$
    $\in \text{mset-poly-rel} \rangle$
  $\langle proof \rangle$

lemma *fully-unsorted-poly-rel-diff*:
  $\langle ([v],\ v') \in \text{fully-unsorted-poly-rel}\ O\ \text{mset-poly-rel} \Longrightarrow$
  $(r,\ r') \in \text{fully-unsorted-poly-rel}\ O\ \text{mset-poly-rel} \Longrightarrow$
  $(v\ \#\ r,$
   $v'\ +\ r')$
  $\in \text{fully-unsorted-poly-rel}\ O\ \text{mset-poly-rel} \rangle$
  $\langle proof \rangle$

lemma *PAC-checker-l-step-PAC-checker-step*:
  assumes
    $\langle (Ast,\ Bst) \in \text{code-status-status-rel} \times_r \langle \text{var-rel} \rangle \text{set-rel} \times_r \text{fmap-polys-rel} \rangle$ **and**
    $\langle (st,\ st') \in \text{pac-step-rel} \rangle$ **and**
    $spec:\ \langle (spec,\ spec') \in \text{sorted-poly-rel}\ O\ \text{mset-poly-rel} \rangle$
  shows
    $\langle \text{PAC-checker-l-step}\ spec\ Ast\ st \leq\ \Downarrow (\text{code-status-status-rel} \times_r \langle \text{var-rel} \rangle \text{set-rel} \times_r \text{fmap-polys-rel})$
$(\text{PAC-checker-step}\ spec'\ Bst\ st') \rangle$
$\langle proof \rangle$

lemma *code-status-status-rel-discrim-iff*:
  $\langle (x1a,\ x1c) \in \text{code-status-status-rel} \Longrightarrow \text{is-cfailed}\ x1a \longleftrightarrow \text{is-failed}\ x1c \rangle$
  $\langle (x1a,\ x1c) \in \text{code-status-status-rel} \Longrightarrow \text{is-cfound}\ x1a \longleftrightarrow \text{is-found}\ x1c \rangle$
  $\langle proof \rangle$

lemma *PAC-checker-l-PAC-checker*:
  assumes
    $\langle (A,\ B) \in \langle \text{var-rel} \rangle \text{set-rel} \times_r \text{fmap-polys-rel} \rangle$ **and**
    $\langle (st,\ st') \in \langle \text{pac-step-rel} \rangle \text{list-rel} \rangle$ **and**
    $\langle (spec,\ spec') \in \text{sorted-poly-rel}\ O\ \text{mset-poly-rel} \rangle$ **and**
    $\langle (b,\ b') \in \text{code-status-status-rel} \rangle$
  shows
  $\langle \text{PAC-checker-l}\ spec\ A\ b\ st \leq\ \Downarrow (\text{code-status-status-rel} \times_r \langle \text{var-rel} \rangle \text{set-rel} \times_r \text{fmap-polys-rel})\ (\text{PAC-checker}$
$spec'\ B\ b'\ st') \rangle$
$\langle proof \rangle$

end

lemma *less-than-char-of-char*[*code-unfold*]:
 ⟨(x, y) ∈ *less-than-char* ⟷ (*of-char x* :: *nat*) < *of-char y*⟩
 ⟨*proof*⟩

lemmas [*code*] =
 add-poly-l′.simps[*unfolded var-order-rel-def*]

export-code *add-poly-l′* **in** *SML* **module-name** *test*

definition *full-checker-l*
 :: ⟨*llist-polynomial* ⇒ (*nat*, *llist-polynomial*) *fmap* ⇒ (-, *string*, *nat*) *pac-step list* ⇒
  (*string code-status* × -) *nres*⟩
where
 ⟨*full-checker-l spec A st = do* {
  spec′ ← *full-normalize-poly spec*;
  (*b*, 𝒱, *A*) ← *remap-polys-l spec′* {} *A*;
  if is-cfailed b
  then RETURN (*b*, 𝒱, *A*)
  else do {
   let 𝒱 = 𝒱 ∪ *vars-llist spec*;
   PAC-checker-l spec′ (𝒱, *A*) *b st*
  }
 }⟩

context *poly-embed*
begin

term *normalize-poly-spec*
thm *full-normalize-poly-diff-ideal*[*unfolded normalize-poly-spec-def*[*symmetric*]]
abbreviation *unsorted-fmap-polys-rel* **where**
 ⟨*unsorted-fmap-polys-rel* ≡ ⟨*nat-rel*, *fully-unsorted-poly-rel O mset-poly-rel*⟩*fmap-rel*⟩

lemma *full-checker-l-full-checker*:
 assumes
  ⟨(*A*, *B*) ∈ *unsorted-fmap-polys-rel*⟩ **and**
  ⟨(*st*, *st′*) ∈ ⟨*pac-step-rel*⟩*list-rel*⟩ **and**
  ⟨(*spec*, *spec′*) ∈ *fully-unsorted-poly-rel O mset-poly-rel*⟩
  shows
  ⟨*full-checker-l spec A st* ≤ ⇓ (*code-status-status-rel* ×ᵣ ⟨*var-rel*⟩*set-rel* ×ᵣ *fmap-polys-rel*) (*full-checker*
spec′ B st′)⟩
 ⟨*proof*⟩

lemma *full-checker-l-full-checker′*:
 ⟨(*uncurry2 full-checker-l*, *uncurry2 full-checker*) ∈
 ((*fully-unsorted-poly-rel O mset-poly-rel*) ×ᵣ *unsorted-fmap-polys-rel*) ×ᵣ ⟨*pac-step-rel*⟩*list-rel* →_f
  ⟨(*code-status-status-rel* ×ᵣ ⟨*var-rel*⟩*set-rel* ×ᵣ *fmap-polys-rel*)⟩*nres-rel*⟩
 ⟨*proof*⟩

end

definition *remap-polys-l2* :: ‹*llist-polynomial* ⇒ *string set* ⇒ (*nat*, *llist-polynomial*) *fmap* ⇒ - *nres*›
where
  ‹*remap-polys-l2 spec* = (λ𝒱 *A. do*{
  n ← *upper-bound-on-dom A*;
  b ← *RETURN* (*n* ≥ *2ˆ64*);
  if b
  then do {
    c ← *remap-polys-l-dom-err*;
    RETURN (*error-msg* (*0* ::*nat*) *c*, 𝒱, *fmempty*)
  }
  else do {
    (*b*, 𝒱, *A*) ← *nfoldli* ([*0*..<*n*]) (λ-. *True*)
    (λ*i* (*b*, 𝒱, *A′*).
      if i ∈# *dom-m A*
      then do {
        ASSERT(*fmlookup A i* ≠ *None*);
        p ← *full-normalize-poly* (*the* (*fmlookup A i*));
        eq ← *weak-equality-l p spec*;
        𝒱 ← *RETURN* (𝒱 ∪ *vars-llist* (*the* (*fmlookup A i*)));
        RETURN(*b* ∨ *eq*, 𝒱, *fmupd i p A′*)
      } *else RETURN* (*b*, 𝒱, *A′*)
)
    (*False*, 𝒱, *fmempty*);
    RETURN (*if b then CFOUND else CSUCCESS*, 𝒱, *A*)
  }
})›

lemma *remap-polys-l2-remap-polys-l*:
  ‹*remap-polys-l2 spec* 𝒱 *A* ≤ ⇓ *Id* (*remap-polys-l spec* 𝒱 *A*)›
⟨*proof*⟩

end
theory *PAC-Checker-Relation*
  imports *PAC-Checker WB-Sort Native-Word.Uint64*
begin

11 Various Refinement Relations

When writing this, it was not possible to share the definition with the IsaSAT version.

definition *uint64-nat-rel* :: (*uint64* × *nat*) *set* **where**
‹*uint64-nat-rel* = *br nat-of-uint64* (λ-. *True*)›

abbreviation *uint64-nat-assn* **where**
  ‹*uint64-nat-assn* ≡ *pure uint64-nat-rel*›

instantiation *uint32* :: *hashable*
begin
definition *hashcode-uint32* :: ‹*uint32* ⇒ *uint32*› **where**
  ‹*hashcode-uint32 n* = *n*›

definition *def-hashmap-size-uint32* :: ‹*uint32 itself* ⇒ *nat*› **where**
  ‹*def-hashmap-size-uint32* = (λ-. *16*)›
  — same as *nat*

instance
⟨*proof*⟩
end

instantiation *uint64* :: *hashable*
begin
definition *hashcode-uint64* :: ‹*uint64* ⇒ *uint32*› **where**
  ‹*hashcode-uint64 n = (uint32-of-nat (nat-of-uint64 ((n) AND ((2 :: uint64)^32 −1))))*›

definition *def-hashmap-size-uint64* :: ‹*uint64 itself* ⇒ *nat*› **where**
  ‹*def-hashmap-size-uint64 = (λ-. 16)*›
  — same as *nat*
instance
⟨*proof*⟩
end

lemma *word-nat-of-uint64-Rep-inject*[*simp*]: ‹*nat-of-uint64 ai = nat-of-uint64 bi ⟷ ai = bi*›
  ⟨*proof*⟩

instance *uint64* :: *heap*
  ⟨*proof*⟩

instance *uint64* :: *semiring-numeral*
  ⟨*proof*⟩

lemma *nat-of-uint64-012*[*simp*]: ‹*nat-of-uint64 0 = 0*› ‹*nat-of-uint64 2 = 2*› ‹*nat-of-uint64 1 = 1*›
  ⟨*proof*⟩

definition *uint64-of-nat-conv* **where**
  [*simp*]: ‹*uint64-of-nat-conv (x :: nat) = x*›
lemma *less-upper-bintrunc-id*: ‹*n < 2 ^b ⟹ n ≥ 0 ⟹ bintrunc b n = n*›
  ⟨*proof*⟩

lemma *nat-of-uint64-uint64-of-nat-id*: ‹*n < 2^64 ⟹ nat-of-uint64 (uint64-of-nat n) = n*›
  ⟨*proof*⟩

lemma [*sepref-fr-rules*]:
  ‹*(return o uint64-of-nat, RETURN o uint64-of-nat-conv) ∈ [λa. a < 2 ^64]_a nat-assn^k → uint64-nat-assn*›
  ⟨*proof*⟩

definition *string-rel* :: ‹*(String.literal × string) set*› **where**
  ‹*string-rel = {(x, y). y = String.explode x}*›

abbreviation *string-assn* :: ‹*string ⇒ String.literal ⇒ assn*› **where**
  ‹*string-assn ≡ pure string-rel*›

lemma *eq-string-eq*:
  ‹*((=), (=)) ∈ string-rel → string-rel → bool-rel*›
⟨*proof*⟩

lemmas *eq-string-eq-hnr* =
  eq-string-eq[*sepref-import-param*]

definition *string2-rel* :: ‹*(string × string) set*› **where**
  ‹*string2-rel ≡ ⟨Id⟩list-rel*›

abbreviation *string2-assn* :: ‹*string* ⇒ *string* ⇒ *assn*› **where**
  ‹*string2-assn* ≡ *pure string2-rel*›

abbreviation *monom-rel* **where**
  ‹*monom-rel* ≡ ⟨*string-rel*⟩*list-rel*›

abbreviation *monom-assn* **where**
  ‹*monom-assn* ≡ *list-assn string-assn*›

abbreviation *monomial-rel* **where**
  ‹*monomial-rel* ≡ *monom-rel* ×$_r$ *int-rel*›

abbreviation *monomial-assn* **where**
  ‹*monomial-assn* ≡ *monom-assn* ×$_a$ *int-assn*›

abbreviation *poly-rel* **where**
  ‹*poly-rel* ≡ ⟨*monomial-rel*⟩*list-rel*›

abbreviation *poly-assn* **where**
  ‹*poly-assn* ≡ *list-assn monomial-assn*›

lemma *poly-assn-alt-def*:
  ‹*poly-assn* = *pure poly-rel*›
  ⟨*proof*⟩

abbreviation *polys-assn* **where**
  ‹*polys-assn* ≡ *hm-fmap-assn uint64-nat-assn poly-assn*›

lemma *string-rel-string-assn*:
  ‹(↑ ((*c*, *a*) ∈ *string-rel*)) = *string-assn a c*›
  ⟨*proof*⟩

lemma *single-valued-string-rel*:
  ‹*single-valued string-rel*›
  ⟨*proof*⟩

lemma *IS-LEFT-UNIQUE-string-rel*:
  ‹*IS-LEFT-UNIQUE string-rel*›
  ⟨*proof*⟩

lemma *IS-RIGHT-UNIQUE-string-rel*:
  ‹*IS-RIGHT-UNIQUE string-rel*›
  ⟨*proof*⟩

lemma *single-valued-monom-rel*: ‹*single-valued monom-rel*›
  ⟨*proof*⟩

lemma *single-valued-monomial-rel*:
  ‹*single-valued monomial-rel*›
  ⟨*proof*⟩

lemma *single-valued-monom-rel′*: ‹*IS-LEFT-UNIQUE monom-rel*›
  ⟨*proof*⟩

lemma *single-valued-monomial-rel′*:
  ‹*IS-LEFT-UNIQUE monomial-rel*›
  ⟨*proof*⟩

lemma [*safe-constraint-rules*]:
  ‹*Sepref-Constraints.CONSTRAINT single-valued string-rel*›
  ‹*Sepref-Constraints.CONSTRAINT IS-LEFT-UNIQUE string-rel*›
  ⟨*proof*⟩

lemma *eq-string-monom-hnr*[*sepref-fr-rules*]:
  ‹(*uncurry* (*return oo* (=)), *uncurry* (*RETURN oo* (=))) ∈ *monom-assn*^{k} $*_a$ *monom-assn*^{k} $→_a$ *bool-assn*›
  ⟨*proof*⟩

definition *term-order-rel′* **where**
  [*simp*]: ‹*term-order-rel′ x y* = ((*x*, *y*) ∈ *term-order-rel*)›

lemma *term-order-rel*[*def-pat-rules*]:
  ‹(∈)\$(*x,y*)\$*term-order-rel* ≡ *term-order-rel′*\$*x*\$*y*›
  ⟨*proof*⟩

lemma *term-order-rel-alt-def*:
  ‹*term-order-rel* = *lexord* (*p2rel char.lexordp*)›
  ⟨*proof*⟩

instantiation *char* :: *linorder*
begin
  definition *less-char* **where** [*symmetric*, *simp*]: *less-char* = *PAC-Polynomials-Term.less-char*
  definition *less-eq-char* **where** [*symmetric*, *simp*]: *less-eq-char* = *PAC-Polynomials-Term.less-eq-char*
instance
  ⟨*proof*⟩
end

instantiation *list* :: (*linorder*) *linorder*
begin
  definition *less-list* **where** *less-list* = *lexordp* (<)
  definition *less-eq-list* **where** *less-eq-list* = *lexordp-eq*

instance
  ⟨*proof*⟩

end

lemma *term-order-rel′-alt-def-lexord*:
    ‹*term-order-rel′ x y* = *ord-class.lexordp x y*› **and**
  term-order-rel′-alt-def:
    ‹*term-order-rel′ x y* ⟷ *x* < *y*›
⟨*proof*⟩

lemma *list-rel-list-rel-order-iff*:

assumes ‹(a, b) ∈ ⟨string-rel⟩list-rel› ‹(a′, b′) ∈ ⟨string-rel⟩list-rel›
  shows ‹a < a′ ⟷ b < b′›
⟨*proof*⟩

lemma *string-rel-le*[*sepref-import-param*]:
  shows ‹((<), (<)) ∈ ⟨string-rel⟩list-rel → ⟨string-rel⟩list-rel → bool-rel›
  ⟨*proof*⟩

lemma [*sepref-import-param*]:
  assumes ‹CONSTRAINT IS-LEFT-UNIQUE R› ‹CONSTRAINT IS-RIGHT-UNIQUE R›
  shows ‹(remove1, remove1) ∈ R → ⟨R⟩list-rel → ⟨R⟩list-rel›
  ⟨*proof*⟩

instantiation *pac-step* :: (*heap, heap, heap*) *heap*
begin

instance
⟨*proof*⟩

end

end
theory *PAC-Checker-Init*
  imports *PAC-Checker WB-Sort PAC-Checker-Relation*
begin

12   Initial Normalisation of Polynomials

12.1   Sorting

Adapted from the theory *HOL−ex.MergeSort* by Tobias. We did not change much, but we
refine it to executable code and try to improve efficiency.

fun *merge* :: *- ⇒ ′a list ⇒ ′a list ⇒ ′a list*
where
  merge f (x#xs) (y#ys) =
      (if f x y then x # merge f xs (y#ys) else y # merge f (x#xs) ys)
| *merge f xs [] = xs*
| *merge f [] ys = ys*

lemma *mset-merge* [*simp*]:
  mset (merge f xs ys) = mset xs + mset ys
  ⟨*proof*⟩

lemma *set-merge* [*simp*]:
  set (merge f xs ys) = set xs ∪ set ys
  ⟨*proof*⟩

lemma *sorted-merge*:
  transp f ⟹ (⋀x y. f x y ∨ f y x) ⟹
   sorted-wrt f (merge f xs ys) ⟷ sorted-wrt f xs ∧ sorted-wrt f ys
  ⟨*proof*⟩

fun *msort* :: *- ⇒ ′a list ⇒ ′a list*
where
  msort f [] = []
| *msort f* [*x*] = [*x*]
| *msort f xs* = *merge f*
                (*msort f (take (size xs div 2) xs)*)
                (*msort f (drop (size xs div 2) xs)*)

fun *swap-ternary* :: ‹*-⇒nat⇒nat⇒ (′a × ′a × ′a) ⇒ (′a × ′a × ′a)*› **where**
  ‹*swap-ternary f m n* =
    (*if (m = 0 ∧ n = 1)*
    then (λ(a, b, c). if f a b then (a, b, c)
      else (b,a,c))
    else if (m = 0 ∧ n = 2)
    then (λ(a, b, c). if f a c then (a, b, c)
      else (c,b,a))
    else if (m = 1 ∧ n = 2)
    then (λ(a, b, c). if f b c then (a, b, c)
      else (a,c,b))
    else (λ(a, b, c). (a,b,c)))›

fun *msort2* :: *- ⇒ ′a list ⇒ ′a list*
where
  msort2 f [] = []
| *msort2 f* [*x*] = [*x*]
| *msort2 f* [*x,y*] = (*if f x y then* [*x,y*] *else* [*y,x*])
| *msort2 f xs* = *merge f*
                (*msort f (take (size xs div 2) xs)*)
                (*msort f (drop (size xs div 2) xs)*)

lemmas [*code del*] =
  msort2.simps

declare *msort2.simps*[*simp del*]
lemmas [*code*] =
  msort2.simps[*unfolded swap-ternary.simps, simplified*]

declare *msort2.simps*[*simp*]

lemma *msort-msort2*:
  fixes *xs* :: ‹*′a* :: *linorder list*›
  shows ‹*msort* (≤) *xs* = *msort2* (≤) *xs*›
  ⟨*proof*⟩

lemma *sorted-msort*:
  transp f ⟹ (⋀*x y. f x y ∨ f y x*) ⟹
  sorted-wrt f (msort f xs)
  ⟨*proof*⟩

lemma *mset-msort*[*simp*]:
  mset (msort f xs) = mset xs
  ⟨*proof*⟩

12.2 Sorting applied to monomials

lemma *merge-coeffs-alt-def*:

$\langle(RETURN\ o\ merge\text{-}coeffs)\ p =$
$REC_T(\lambda f\ p.$
  $(case\ p\ of$
    $[]\ \Rightarrow\ RETURN\ []$
  $\mid\ [\text{-}]\ =>\ RETURN\ p$
  $\mid\ ((xs,\ n)\ \#\ (ys,\ m)\ \#\ p)\ \Rightarrow$
  $(if\ xs\ =\ ys$
    $then\ if\ n\ +\ m\ \neq\ 0\ then\ f\ ((xs,\ n\ +\ m)\ \#\ p)\ else\ f\ p$
    $else\ do\ \{p\ \leftarrow\ f\ ((ys,\ m)\ \#\ p);\ RETURN\ ((xs,\ n)\ \#\ p)\})))$
  $p\rangle$
$\langle proof\rangle$

lemma *hn-invalid-recover*:
$\langle is\text{-}pure\ R\ \Longrightarrow\ hn\text{-}invalid\ R\ =\ (\lambda x\ y.\ R\ x\ y\ *\ true)\rangle$
$\langle is\text{-}pure\ R\ \Longrightarrow\ invalid\text{-}assn\ R\ =\ (\lambda x\ y.\ R\ x\ y\ *\ true)\rangle$
$\langle proof\rangle$

lemma *safe-poly-vars*:
  shows
   $[safe\text{-}constraint\text{-}rules]$:
    $is\text{-}pure\ (poly\text{-}assn)$ **and**
   $[safe\text{-}constraint\text{-}rules]$:
    $is\text{-}pure\ (monom\text{-}assn)$ **and**
   $[safe\text{-}constraint\text{-}rules]$:
    $is\text{-}pure\ (monomial\text{-}assn)$ **and**
   $[safe\text{-}constraint\text{-}rules]$:
    $is\text{-}pure\ string\text{-}assn$
  $\langle proof\rangle$

lemma *invalid-assn-distrib*:
$\langle invalid\text{-}assn\ monom\text{-}assn\ \times_a\ invalid\text{-}assn\ int\text{-}assn\ =\ invalid\text{-}assn\ (monom\text{-}assn\ \times_a\ int\text{-}assn)\rangle$
  $\langle proof\rangle$

lemma *WTF-RF-recover*:
$\langle hn\text{-}ctxt\ (invalid\text{-}assn\ monom\text{-}assn\ \times_a\ invalid\text{-}assn\ int\text{-}assn)\ xb$
    $x'a\ \vee_A$
   $hn\text{-}ctxt\ monomial\text{-}assn\ xb\ x'a\ \Longrightarrow_t$
   $hn\text{-}ctxt\ (monomial\text{-}assn)\ xb\ x'a\rangle$
$\langle proof\rangle$

lemma *WTF-RF*:
$\langle hn\text{-}ctxt\ (invalid\text{-}assn\ monom\text{-}assn\ \times_a\ invalid\text{-}assn\ int\text{-}assn)\ xb\ x'a\ *$
   $(hn\text{-}invalid\ poly\text{-}assn\ la\ l'a\ *\ hn\text{-}invalid\ int\text{-}assn\ a2'\ a2\ *$
    $hn\text{-}invalid\ monom\text{-}assn\ a1'\ a1\ *$
    $hn\text{-}invalid\ poly\text{-}assn\ l\ l'\ *$
    $hn\text{-}invalid\ monomial\text{-}assn\ xa\ x'\ *$
    $hn\text{-}invalid\ poly\text{-}assn\ ax\ px)\ \Longrightarrow_t$
   $hn\text{-}ctxt\ (monomial\text{-}assn)\ xb\ x'a\ *$
   $hn\text{-}ctxt\ poly\text{-}assn$
    $la\ l'a\ *$
   $hn\text{-}ctxt\ poly\text{-}assn\ l\ l'\ *$
   $(hn\text{-}invalid\ int\text{-}assn\ a2'\ a2\ *$
    $hn\text{-}invalid\ monom\text{-}assn\ a1'\ a1\ *$
    $hn\text{-}invalid\ monomial\text{-}assn\ xa\ x'\ *$
    $hn\text{-}invalid\ poly\text{-}assn\ ax\ px)\rangle$

⟨*hn-ctxt (invalid-assn monom-assn ×ₐ invalid-assn int-assn) xa x′ ∗*
    (hn-ctxt poly-assn l l′ ∗ hn-invalid poly-assn ax px) ⟹ₜ
    hn-ctxt (monomial-assn) xa x′ ∗
    hn-ctxt poly-assn l l′ ∗
    hn-ctxt poly-assn ax px ∗
    emp⟩
⟨*proof*⟩

The refinement frameword is completely lost here when synthesizing the constants – it does not understant what is pure (actually everything) and what must be destroyed.

sepref-definition *merge-coeffs-impl*
  is ⟨*RETURN o merge-coeffs*⟩
  :: ⟨*poly-assn^d →ₐ poly-assn*⟩
  ⟨*proof*⟩

definition *full-quicksort-poly* **where**
  ⟨*full-quicksort-poly = full-quicksort-ref (λx y. x = y ∨ (x, y) ∈ term-order-rel) fst*⟩

lemma *down-eq-id-list-rel*: ⟨⇓(⟨Id⟩list-rel) x = x⟩
  ⟨*proof*⟩

definition *quicksort-poly*:: ⟨*nat ⇒ nat ⇒ llist-polynomial ⇒ (llist-polynomial) nres*⟩ **where**
  ⟨*quicksort-poly x y z = quicksort-ref (≤) fst (x, y, z)*⟩

term *partition-between-ref*

definition *partition-between-poly* :: ⟨*nat ⇒ nat ⇒ llist-polynomial ⇒ (llist-polynomial × nat) nres*⟩ **where**
  ⟨*partition-between-poly = partition-between-ref (≤) fst*⟩

definition *partition-main-poly* :: ⟨*nat ⇒ nat ⇒ llist-polynomial ⇒ (llist-polynomial × nat) nres*⟩ **where**
  ⟨*partition-main-poly = partition-main (≤) fst*⟩

lemma *string-list-trans*:
  ⟨*(xa ::char list list, ya) ∈ lexord (lexord {(x, y). x < y}) ⟹*
  (ya, z) ∈ lexord (lexord {(x, y). x < y}) ⟹
    (xa, z) ∈ lexord (lexord {(x, y). x < y})⟩
  ⟨*proof*⟩

lemma *full-quicksort-sort-poly-spec*:
  ⟨*(full-quicksort-poly, sort-poly-spec) ∈ ⟨Id⟩list-rel →f ⟨⟨Id⟩list-rel⟩nres-rel*⟩
⟨*proof*⟩

12.3   Lifting to polynomials

definition *merge-sort-poly* :: ⟨*-*⟩ **where**
⟨*merge-sort-poly = msort (λa b. fst a ≤ fst b)*⟩

definition *merge-monoms-poly* :: ⟨*-*⟩ **where**
⟨*merge-monoms-poly = msort (≤)*⟩

definition *merge-poly* :: ⟨*-*⟩ **where**
⟨*merge-poly = merge (λa b. fst a ≤ fst b)*⟩

definition *merge-monoms* :: ⟨*-*⟩ **where**

‹*merge-monoms* = *merge* (≤)›

definition *msort-poly-impl* :: ‹(*String.literal list* × *int*) *list* ⇒ -› **where**
‹*msort-poly-impl* = *msort* (λ*a b. fst a* ≤ *fst b*)›

definition *msort-monoms-impl* :: ‹(*String.literal list*) ⇒ -› **where**
‹*msort-monoms-impl* = *msort* (≤)›

lemma *msort-poly-impl-alt-def*:
  ‹*msort-poly-impl xs* =
    (*case xs of*
      [] ⇒ []
    | [*a*] ⇒ [*a*]
    | [*a,b*] ⇒ *if fst a* ≤ *fst b then* [*a,b*]*else* [*b,a*]
    | *xs* ⇒ *merge-poly*
                    (*msort-poly-impl* (*take* ((*length xs*) *div 2*) *xs*))
                    (*msort-poly-impl* (*drop* ((*length xs*) *div 2*) *xs*)))›
  ⟨*proof*⟩

lemma *le-term-order-rel′*:
  ‹(≤) = (λ*x y. x* = *y* ∨ *term-order-rel′ x y*)›
  ⟨*proof*⟩

fun *lexord-eq* **where**
  ‹*lexord-eq* [] - = *True*› |
  ‹*lexord-eq* (*x* # *xs*) (*y* # *ys*) = (*x* < *y* ∨ (*x* = *y* ∧ *lexord-eq xs ys*))› |
  ‹*lexord-eq* - - = *False*›

lemma [*simp*]:
  ‹*lexord-eq* [] [] = *True*›
  ‹*lexord-eq* (*a* # *b*)[] = *False*›
  ‹*lexord-eq* [] (*a* # *b*) = *True*›
  ⟨*proof*⟩

lemma *var-order-rel′*:
  ‹(≤) = (λ*x y. x* = *y* ∨ (*x,y*) ∈ *var-order-rel*)›
  ⟨*proof*⟩

lemma *var-order-rel″*:
  ‹(*x,y*) ∈ *var-order-rel* ⟷ *x* < *y*›
  ⟨*proof*⟩

lemma *lexord-eq-alt-def1*:
  ‹*a* ≤ *b* = *lexord-eq a b*› **for** *a b* :: ‹*String.literal list*›
  ⟨*proof*⟩

lemma *lexord-eq-alt-def2*:
  ‹(*RETURN oo lexord-eq*) *xs ys* =
    REC$_T$ (λ*f* (*xs, ys*).
      case (*xs, ys*) *of*
        ([], -) ⇒ *RETURN True*
      | (*x* # *xs, y* # *ys*) ⇒
        if x < *y then RETURN True*
        else if x = *y then f* (*xs, ys*) *else RETURN False*

```
        | - ⇒ RETURN False)
       (xs, ys)›
⟨proof⟩
```

definition *var-order′* **where**
  [*simp*]: ‹*var-order′* = *var-order*›

lemma *var-order-rel*[*def-pat-rules*]:
  ‹(∈)\$(x,y)\$*var-order-rel* ≡ *var-order′*\$x\$y›
  ⟨*proof*⟩

lemma *var-order-rel-alt-def*:
  ‹*var-order-rel* = *p2rel char.lexordp*›
  ⟨*proof*⟩

lemma *var-order-rel-var-order*:
  ‹(x, y) ∈ *var-order-rel* ⟷ *var-order x y*›
  ⟨*proof*⟩

lemma *var-order-string-le*[*sepref-import-param*]:
  ‹((<), *var-order′*) ∈ *string-rel* → *string-rel* → *bool-rel*›
  ⟨*proof*⟩

lemma [*sepref-import-param*]:
  ‹((≤), (≤)) ∈ *monom-rel* → *monom-rel* →*bool-rel*›
  ⟨*proof*⟩

lemma [*sepref-import-param*]:
  ‹((<), (<)) ∈ *string-rel* → *string-rel* →*bool-rel*›
  ⟨*proof*⟩

lemma [*sepref-import-param*]:
  ‹((≤), (≤)) ∈ *string-rel* → *string-rel* →*bool-rel*›
  ⟨*proof*⟩

sepref-register *lexord-eq*
sepref-definition *lexord-eq-term*
  is ‹*uncurry* (*RETURN oo lexord-eq*)›
  :: ‹*monom-assn^k* *$_a$ *monom-assn^k* →$_a$ *bool-assn*›
  ⟨*proof*⟩

declare *lexord-eq-term.refine*[*sepref-fr-rules*]

lemmas [*code del*] = *msort-poly-impl-def msort-monoms-impl-def*
lemmas [*code*] =
  msort-poly-impl-def[*unfolded lexord-eq-alt-def1*[*abs-def*]]
  msort-monoms-impl-def[*unfolded msort-msort2*]

lemma *term-order-rel-trans*:
  ‹   (a, aa) ∈ *term-order-rel* ⟹
      (aa, ab) ∈ *term-order-rel* ⟹ (a, ab) ∈ *term-order-rel*›
  ⟨*proof*⟩

lemma *merge-sort-poly-sort-poly-spec*:
 ‹(*RETURN o merge-sort-poly*, *sort-poly-spec*) ∈ ⟨*Id*⟩*list-rel* →$_f$ ⟨⟨*Id*⟩*list-rel*⟩*nres-rel*›
 ⟨*proof*⟩

lemma *msort-alt-def*:
 ‹*RETURN o* (*msort f*) =
    REC$_T$ (λ*g xs*.
      case xs of
        [] ⇒ *RETURN* []
      | [*x*] ⇒ *RETURN* [*x*]
      | - ⇒ *do* {
          a ← *g* (*take* (*size xs div 2*) *xs*);
          b ← *g* (*drop* (*size xs div 2*) *xs*);
          RETURN (*merge f a b*)})›
 ⟨*proof*⟩

lemma *monomial-rel-order-map*:
 ‹(*x*, *a*, *b*) ∈ *monomial-rel* ⟹
    (*y*, *aa*, *bb*) ∈ *monomial-rel* ⟹
    fst x ≤ *fst y* ⟷ *a* ≤ *aa*›
 ⟨*proof*⟩

lemma *step-rewrite-pure*:
 fixes *K* :: ‹(′*olbl* × ′*lbl*) *set*›
 shows
  ‹*pure* (*p2rel* (⟨*K*, *V*, *R*⟩*pac-step-rel-raw*)) = *pac-step-rel-assn* (*pure K*) (*pure V*) (*pure R*)›
  ‹*monomial-assn* = *pure* (*monom-rel* ×$_r$ *int-rel*)› **and**
 poly-assn-list:
  ‹*poly-assn* = *pure* (⟨*monom-rel* ×$_r$ *int-rel*⟩*list-rel*)›
 ⟨*proof*⟩

lemma *safe-pac-step-rel-assn*[*safe-constraint-rules*]:
 is-pure K ⟹ *is-pure V* ⟹ *is-pure R* ⟹ *is-pure* (*pac-step-rel-assn K V R*)
 ⟨*proof*⟩

lemma *merge-poly-merge-poly*:
 ‹(*merge-poly*, *merge-poly*)
  ∈ *poly-rel* → *poly-rel* → *poly-rel*›
  ⟨*proof*⟩

lemmas [*fcomp-norm-unfold*] =
 poly-assn-list[*symmetric*]
 step-rewrite-pure(*1*)

lemma *merge-poly-merge-poly2*:
 ‹(*a*, *b*) ∈ *poly-rel* ⟹ (*a′*, *b′*) ∈ *poly-rel* ⟹
  (*merge-poly a a′*, *merge-poly b b′*) ∈ *poly-rel*›
 ⟨*proof*⟩

lemma *list-rel-takeD*:
 ‹(*a*, *b*) ∈ ⟨*R*⟩*list-rel* ⟹ (*n*, *n′*)∈ *Id* ⟹ (*take n a*, *take n′ b*) ∈ ⟨*R*⟩*list-rel*›
 ⟨*proof*⟩

lemma *list-rel-dropD*:
‹(*a*, *b*) ∈ ⟨*R*⟩*list-rel* ⟹ (*n*, *n′*)∈ *Id* ⟹ (*drop n a*, *drop n′ b*) ∈ ⟨*R*⟩*list-rel*›
⟨*proof*⟩

lemma *merge-sort-poly*[*sepref-import-param*]:
‹(*msort-poly-impl*, *merge-sort-poly*)
∈ *poly-rel* → *poly-rel*›
⟨*proof*⟩

lemmas [*sepref-fr-rules*] = *merge-sort-poly*[*FCOMP merge-sort-poly-sort-poly-spec*]

sepref-definition *partition-main-poly-impl*
is ‹*uncurry2 partition-main-poly*›
:: ‹*nat-assn^k *$_a$ nat-assn^k *$_a$ poly-assn^k →$_a$ prod-assn poly-assn nat-assn* ›
⟨*proof*⟩

declare *partition-main-poly-impl.refine*[*sepref-fr-rules*]

sepref-definition *partition-between-poly-impl*
is ‹*uncurry2 partition-between-poly*›
:: ‹*nat-assn^k *$_a$ nat-assn^k *$_a$ poly-assn^k →$_a$ prod-assn poly-assn nat-assn* ›
⟨*proof*⟩

declare *partition-between-poly-impl.refine*[*sepref-fr-rules*]

sepref-definition *quicksort-poly-impl*
is ‹*uncurry2 quicksort-poly*›
:: ‹*nat-assn^k *$_a$ nat-assn^k *$_a$ poly-assn^k →$_a$ poly-assn*›
⟨*proof*⟩

lemmas [*sepref-fr-rules*] = *quicksort-poly-impl.refine*

sepref-register *quicksort-poly*
sepref-definition *full-quicksort-poly-impl*
is ‹*full-quicksort-poly*›
:: ‹*poly-assn^k →$_a$ poly-assn*›
⟨*proof*⟩

lemmas *sort-poly-spec-hnr* =
full-quicksort-poly-impl.refine[*FCOMP full-quicksort-sort-poly-spec*]

declare *merge-coeffs-impl.refine*[*sepref-fr-rules*]

sepref-definition *normalize-poly-impl*
is ‹*normalize-poly*›
:: ‹*poly-assn^k →$_a$ poly-assn*›
⟨*proof*⟩

declare *normalize-poly-impl.refine*[*sepref-fr-rules*]

definition *full-quicksort-vars* **where**

‹*full-quicksort-vars = full-quicksort-ref* ($\lambda x\ y.\ x = y \lor (x, y) \in$ *var-order-rel*) *id*›

definition *quicksort-vars*:: ‹*nat* $\Rightarrow$ *nat* $\Rightarrow$ *string list* $\Rightarrow$ (*string list*) *nres*› **where**
  ‹*quicksort-vars x y z = quicksort-ref* ($\leq$) *id* (*x, y, z*)›

definition *partition-between-vars* :: ‹*nat* $\Rightarrow$ *nat* $\Rightarrow$ *string list* $\Rightarrow$ (*string list* $\times$ *nat*) *nres*› **where**
  ‹*partition-between-vars = partition-between-ref* ($\leq$) *id*›

definition *partition-main-vars* :: ‹*nat* $\Rightarrow$ *nat* $\Rightarrow$ *string list* $\Rightarrow$ (*string list* $\times$ *nat*) *nres*› **where**
  ‹*partition-main-vars = partition-main* ($\leq$) *id*›

lemma *total-on-lexord-less-than-char-linear2*:
  ‹*xs* $\neq$ *ys* $\Longrightarrow$ (*xs, ys*) $\notin$ *lexord* (*less-than-char*) $\longleftrightarrow$
    (*ys, xs*) $\in$ *lexord less-than-char*›
  $\langle proof \rangle$

lemma *string-trans*:
  ‹(*xa, ya*) $\in$ *lexord* {(*x::char, y::char*). $x < y$} $\Longrightarrow$
  (*ya, z*) $\in$ *lexord* {(*x::char, y::char*). $x < y$} $\Longrightarrow$
  (*xa, z*) $\in$ *lexord* {(*x::char, y::char*). $x < y$}›
  $\langle proof \rangle$

lemma *full-quicksort-sort-vars-spec*:
  ‹(*full-quicksort-vars, sort-coeff*) $\in \langle Id \rangle$ *list-rel* $\rightarrow_f \langle\langle Id \rangle$ *list-rel* $\rangle$ *nres-rel*›
$\langle proof \rangle$

sepref-definition *partition-main-vars-impl*
  is ‹*uncurry2 partition-main-vars*›
  :: ‹*nat-assn*^{k} $*_a$ *nat-assn*^{k} $*_a$ (*monom-assn*)^k $\rightarrow_a$ *prod-assn* (*monom-assn*) *nat-assn*›
  $\langle proof \rangle$

declare *partition-main-vars-impl.refine*[*sepref-fr-rules*]

sepref-definition *partition-between-vars-impl*
  is ‹*uncurry2 partition-between-vars*›
  :: ‹*nat-assn*^{k} $*_a$ *nat-assn*^{k} $*_a$ *monom-assn*^{k} $\rightarrow_a$ *prod-assn monom-assn nat-assn* ›
  $\langle proof \rangle$

declare *partition-between-vars-impl.refine*[*sepref-fr-rules*]

sepref-definition *quicksort-vars-impl*
  is ‹*uncurry2 quicksort-vars*›
  :: ‹*nat-assn*^{k} $*_a$ *nat-assn*^{k} $*_a$ *monom-assn*^{k} $\rightarrow_a$ *monom-assn*›
  $\langle proof \rangle$

lemmas [*sepref-fr-rules*] = *quicksort-vars-impl.refine*

sepref-register *quicksort-vars*

lemma *le-var-order-rel*:
  ‹($\leq$) = ($\lambda x\ y.\ x = y \lor (x, y) \in$ *var-order-rel*)›

⟨*proof*⟩

sepref-definition *full-quicksort-vars-impl*
  is ⟨*full-quicksort-vars*⟩
  :: ⟨*monom-assn*^{k} →$_a$ *monom-assn*⟩
  ⟨*proof*⟩

lemmas *sort-vars-spec-hnr* =
  full-quicksort-vars-impl.refine[*FCOMP full-quicksort-sort-vars-spec*]

lemma *string-rel-order-map*:
  ⟨(*x*, *a*) ∈ *string-rel* ⟹
      (*y*, *aa*) ∈ *string-rel* ⟹
      x ≤ *y* ⟷ *a* ≤ *aa*⟩
  ⟨*proof*⟩

lemma *merge-monoms-merge-monoms*:
  ⟨(*merge-monoms*, *merge-monoms*) ∈ *monom-rel* → *monom-rel* → *monom-rel*⟩
  ⟨*proof*⟩

lemma *merge-monoms-merge-monoms2*:
  ⟨(*a*, *b*) ∈ *monom-rel* ⟹ (*a′*, *b′*) ∈ *monom-rel* ⟹
  (*merge-monoms a a′*, *merge-monoms b b′*) ∈ *monom-rel*⟩
  ⟨*proof*⟩

lemma *msort-monoms-impl*:
  ⟨(*msort-monoms-impl*, *merge-monoms-poly*)
  ∈ *monom-rel* → *monom-rel*⟩
  ⟨*proof*⟩

lemma *merge-sort-monoms-sort-monoms-spec*:
  ⟨(*RETURN o merge-monoms-poly*, *sort-coeff*) ∈ ⟨*Id*⟩*list-rel* →$_f$ ⟨⟨*Id*⟩*list-rel*⟩*nres-rel*⟩
  ⟨*proof*⟩

sepref-register *sort-coeff*
lemma [*sepref-fr-rules*]:
  ⟨(*return o msort-monoms-impl*, *sort-coeff*) ∈ *monom-assn*^{k} →$_a$ *monom-assn*⟩
  ⟨*proof*⟩

sepref-definition *sort-all-coeffs-impl*
  is ⟨*sort-all-coeffs*⟩
  :: ⟨*poly-assn*^{k} →$_a$ *poly-assn*⟩
  ⟨*proof*⟩

declare *sort-all-coeffs-impl.refine*[*sepref-fr-rules*]

lemma *merge-coeffs0-alt-def*:
  ⟨(*RETURN o merge-coeffs0*) *p* =
  REC_T(λ*f p*.
    (*case p of*
       [] ⟹ *RETURN* []
    | [*p*] => *if snd p = 0 then RETURN* [] *else RETURN* [*p*]
    | ((*xs*, *n*) # (*ys*, *m*) # *p*) ⟹

```
   (if xs = ys
     then if n + m ≠ 0 then f ((xs, n + m) # p) else f p
     else if n = 0 then
         do {p ← f ((ys, m) # p);
             RETURN p}
       else do {p ← f ((ys, m) # p);
             RETURN ((xs, n) # p)}})))
   p⟩
 ⟨proof⟩
```

Again, Sepref does not understand what is going here.

sepref-definition *merge-coeffs0-impl*
 is ⟨*RETURN o merge-coeffs0*⟩
 :: ⟨*poly-assn^k →$_a$ poly-assn*⟩
 ⟨*proof*⟩

declare *merge-coeffs0-impl.refine*[*sepref-fr-rules*]

sepref-definition *fully-normalize-poly-impl*
 is ⟨*full-normalize-poly*⟩
 :: ⟨*poly-assn^k →$_a$ poly-assn*⟩
 ⟨*proof*⟩

declare *fully-normalize-poly-impl.refine*[*sepref-fr-rules*]

end
theory *PAC-Version*
 imports *Main*
begin

This code was taken from IsaFoR and adapted to git.

local-setup ⟨
 let
   val version = 2020−AFP
(∗   *trim-line (#1 (Isabelle-System.bash-output (cd $ISAFOL/ && git rev−parse −−short HEAD ||*
echo unknown))) ∗)
 in
   Local-Theory.define
     ((**binding** ⟨*version*⟩, *NoSyn*),
       ((**binding** ⟨*version-def*⟩, []), *HOLogic.mk-literal version*)) #> #2*
 end
⟩

declare *version-def* [*code*]

end
theory *PAC-Checker-Synthesis*
 imports *PAC-Checker WB-Sort PAC-Checker-Relation*
   PAC-Checker-Init More-Loops PAC-Version
begin

13 Code Synthesis of the Complete Checker

We here combine refine the full checker, using the initialisation provided in another file.

abbreviation *vars-assn* **where**
‹*vars-assn ≡ hs.assn string-assn*›

fun *vars-of-monom-in* **where**
‹*vars-of-monom-in [] - = True*› |
‹*vars-of-monom-in (x # xs) V ⟷ x ∈ V ∧ vars-of-monom-in xs V*›

fun *vars-of-poly-in* **where**
‹*vars-of-poly-in [] - = True*› |
‹*vars-of-poly-in ((x, -) # xs) V ⟷ vars-of-monom-in x V ∧ vars-of-poly-in xs V*›

lemma *vars-of-monom-in-alt-def*:
‹*vars-of-monom-in xs V ⟷ set xs ⊆ V*›
⟨*proof*⟩

lemma *vars-llist-alt-def*:
‹*vars-llist xs ⊆ V ⟷ vars-of-poly-in xs V*›
⟨*proof*⟩

lemma *vars-of-monom-in-alt-def2*:
‹*vars-of-monom-in xs V ⟷ fold (λx b. b ∧ x ∈ V) xs True*›
⟨*proof*⟩

sepref-definition *vars-of-monom-in-impl*
  is ‹*uncurry (RETURN oo vars-of-monom-in)*›
  :: ‹*(list-assn string-assn)^k *$_a$ vars-assn^k →$_a$ bool-assn*›
⟨*proof*⟩

declare *vars-of-monom-in-impl.refine*[*sepref-fr-rules*]

lemma *vars-of-poly-in-alt-def2*:
‹*vars-of-poly-in xs V ⟷ fold (λ(x, -) b. b ∧ vars-of-monom-in x V) xs True*›
⟨*proof*⟩

sepref-definition *vars-of-poly-in-impl*
  is ‹*uncurry (RETURN oo vars-of-poly-in)*›
  :: ‹*(poly-assn)^k *$_a$ vars-assn^k →$_a$ bool-assn*›
⟨*proof*⟩

declare *vars-of-poly-in-impl.refine*[*sepref-fr-rules*]

definition *union-vars-monom* :: ‹*string list ⇒ string set ⇒ string set*› **where**
‹*union-vars-monom xs V = fold insert xs V*›

definition *union-vars-poly* :: ‹*llist-polynomial ⇒ string set ⇒ string set*› **where**
‹*union-vars-poly xs V = fold (λ(xs, -) V. union-vars-monom xs V) xs V*›

lemma *union-vars-monom-alt-def*:
‹*union-vars-monom xs V = V ∪ set xs*›
⟨*proof*⟩

lemma *union-vars-poly-alt-def*:
  ‹*union-vars-poly xs* $\mathcal{V}$ = $\mathcal{V}$ ∪ *vars-llist xs*›
  ⟨*proof*⟩

sepref-definition *union-vars-monom-impl*
  is ‹*uncurry* (*RETURN oo union-vars-monom*)›
  :: ‹*monom-assn^k* $*_a$ *vars-assn^d* $\rightarrow_a$ *vars-assn*›
  ⟨*proof*⟩

declare *union-vars-monom-impl.refine*[*sepref-fr-rules*]

sepref-definition *union-vars-poly-impl*
  is ‹*uncurry* (*RETURN oo union-vars-poly*)›
  :: ‹*poly-assn^k* $*_a$ *vars-assn^d* $\rightarrow_a$ *vars-assn*›
  ⟨*proof*⟩

declare *union-vars-poly-impl.refine*[*sepref-fr-rules*]

hide-const (**open**) *Autoref-Fix-Rel.CONSTRAINT*

fun *status-assn* **where**
  ‹*status-assn - CSUCCESS CSUCCESS = emp*› |
  ‹*status-assn - CFOUND CFOUND = emp*› |
  ‹*status-assn R* (*CFAILED a*) (*CFAILED b*) = *R a b*› |
  ‹*status-assn - - - = false*›

lemma *SUCCESS-hnr*[*sepref-fr-rules*]:
  ‹(*uncurry0* (*return CSUCCESS*), *uncurry0* (*RETURN CSUCCESS*)) ∈ *unit-assn^k* $\rightarrow_a$ *status-assn R*›
  ⟨*proof*⟩

lemma *FOUND-hnr*[*sepref-fr-rules*]:
  ‹(*uncurry0* (*return CFOUND*), *uncurry0* (*RETURN CFOUND*)) ∈ *unit-assn^k* $\rightarrow_a$ *status-assn R*›
  ⟨*proof*⟩

lemma *is-success-hnr*[*sepref-fr-rules*]:
  ‹*CONSTRAINT is-pure R* $\Longrightarrow$
  ((*return o is-cfound*), (*RETURN o is-cfound*)) ∈ (*status-assn R*)^k $\rightarrow_a$ *bool-assn*›
  ⟨*proof*⟩

lemma *is-cfailed-hnr*[*sepref-fr-rules*]:
  ‹*CONSTRAINT is-pure R* $\Longrightarrow$
  ((*return o is-cfailed*), (*RETURN o is-cfailed*)) ∈ (*status-assn R*)^k $\rightarrow_a$ *bool-assn*›
  ⟨*proof*⟩

lemma *merge-cstatus-hnr*[*sepref-fr-rules*]:
  ‹*CONSTRAINT is-pure R* $\Longrightarrow$
  (*uncurry* (*return oo merge-cstatus*), *uncurry* (*RETURN oo merge-cstatus*)) ∈
    (*status-assn R*)^k $*_a$ (*status-assn R*)^k $\rightarrow_a$ *status-assn R*›
  ⟨*proof*⟩

sepref-definition *add-poly-impl*
  is ‹*add-poly-l*›
  :: ‹(*poly-assn* $\times_a$ *poly-assn*)^k $\rightarrow_a$ *poly-assn*›

91

$\langle proof \rangle$

declare *add-poly-impl.refine*[*sepref-fr-rules*]

sepref-register *mult-monomials*
lemma *mult-monoms-alt-def*:
  $\langle(RETURN\ oo\ mult\text{-}monoms)\ x\ y = REC_T$
    $(\lambda f\ (p,\ q).$
      $case\ (p,\ q)\ of$
        $([],\ \text{-}) \Rightarrow RETURN\ q$
        $|\ (\text{-},\ []) \Rightarrow RETURN\ p$
        $|\ (x\ \#\ p,\ y\ \#\ q) \Rightarrow$
        $(if\ x = y\ then\ do\ \{$
          $pq \leftarrow f\ (p,\ q);$
            $RETURN\ (x\ \#\ pq)\}$
          $else\ if\ (x,\ y) \in var\text{-}order\text{-}rel$
          $then\ do\ \{$
            $pq \leftarrow f\ (p,\ y\ \#\ q);$
            $RETURN\ (x\ \#\ pq)\}$
          $else\ do\ \{$
            $pq \leftarrow\ f\ (x\ \#\ p,\ q);$
            $RETURN\ (y\ \#\ pq)\}))$
        $(x,\ y)\rangle$
  $\langle proof \rangle$

sepref-definition *mult-monoms-impl*
  is $\langle uncurry\ (RETURN\ oo\ mult\text{-}monoms)\rangle$
  $:: \langle(monom\text{-}assn)^k *_a (monom\text{-}assn)^k \rightarrow_a (monom\text{-}assn)\rangle$
  $\langle proof \rangle$

declare *mult-monoms-impl.refine*[*sepref-fr-rules*]

sepref-definition *mult-monomials-impl*
  is $\langle uncurry\ (RETURN\ oo\ mult\text{-}monomials)\rangle$
  $:: \langle(monomial\text{-}assn)^k *_a (monomial\text{-}assn)^k \rightarrow_a (monomial\text{-}assn)\rangle$
  $\langle proof \rangle$

lemma *map-append-alt-def2*:
  $\langle(RETURN\ o\ (map\text{-}append\ f\ b))\ xs = REC_T$
    $(\lambda g\ xs.\ case\ xs\ of\ [] \Rightarrow RETURN\ b$
      $|\ x\ \#\ xs \Rightarrow do\ \{$
          $y \leftarrow g\ xs;$
          $RETURN\ (f\ x\ \#\ y)$
      $\})\ xs\rangle$
  $\langle proof \rangle$

definition *map-append-poly-mult* **where**
  $\langle map\text{-}append\text{-}poly\text{-}mult\ x = map\text{-}append\ (mult\text{-}monomials\ x)\rangle$

declare *mult-monomials-impl.refine*[*sepref-fr-rules*]

sepref-definition *map-append-poly-mult-impl*
  is ⟨*uncurry2* (*RETURN ooo map-append-poly-mult*)⟩
  :: ⟨*monomial-assn^k $*_a$ poly-assn^k $*_a$ poly-assn^k $\rightarrow_a$ poly-assn*⟩
  ⟨*proof*⟩

declare *map-append-poly-mult-impl.refine*[*sepref-fr-rules*]

TODO *foldl* (λ*l x. l* @ [*?f x*]) [] *?l* = *map ?f ?l* is the worst possible implementation of map!

sepref-definition *mult-poly-raw-impl*
  is ⟨*uncurry* (*RETURN oo mult-poly-raw*)⟩
  :: ⟨*poly-assn^k $*_a$ poly-assn^k $\rightarrow_a$ poly-assn*⟩
  ⟨*proof*⟩

declare *mult-poly-raw-impl.refine*[*sepref-fr-rules*]

sepref-definition *mult-poly-impl*
  is ⟨*uncurry mult-poly-full*⟩
  :: ⟨*poly-assn^k $*_a$ poly-assn^k $\rightarrow_a$ poly-assn*⟩
  ⟨*proof*⟩

declare *mult-poly-impl.refine*[*sepref-fr-rules*]

lemma *inverse-monomial*:
  ⟨*monom-rel^{-1} $\times_r$ int-rel* = (*monom-rel $\times_r$ int-rel*)$^{-1}$⟩
  ⟨*proof*⟩

lemma *eq-poly-rel-eq*[*sepref-import-param*]:
  ⟨((=), (=)) ∈ *poly-rel* → *poly-rel* → *bool-rel*⟩
  ⟨*proof*⟩

sepref-definition *weak-equality-l-impl*
  is ⟨*uncurry weak-equality-l*⟩
  :: ⟨*poly-assn^k $*_a$ poly-assn^k $\rightarrow_a$ bool-assn*⟩
  ⟨*proof*⟩

declare *weak-equality-l-impl.refine*[*sepref-fr-rules*]
sepref-register *add-poly-l mult-poly-full*

abbreviation *raw-string-assn* :: ⟨*string* ⇒ *string* ⇒ *assn*⟩ **where**
  ⟨*raw-string-assn* ≡ *list-assn id-assn*⟩

definition *show-nat* :: ⟨*nat* ⇒ *string*⟩ **where**
  ⟨*show-nat i* = *show i*⟩

lemma [*sepref-import-param*]:
  ⟨(*show-nat, show-nat*) ∈ *nat-rel* → ⟨*Id*⟩*list-rel*⟩
  ⟨*proof*⟩

lemma *status-assn-pure-conv*:
  ⟨*status-assn* (*id-assn*) *a b* = *id-assn a b*⟩
  ⟨*proof*⟩

lemma [*sepref-fr-rules*]:
⟨(*uncurry3* (λx y. *return oo* (*error-msg-not-equal-dom x y*)), *uncurry3 check-not-equal-dom-err*) ∈
poly-assn^k *$_a$ *poly-assn^k* *$_a$ *poly-assn^k* *$_a$ *poly-assn^k* →$_a$ *raw-string-assn*⟩
⟨*proof*⟩

lemma [*sepref-fr-rules*]:
⟨(*return o* (*error-msg-notin-dom o nat-of-uint64*), *RETURN o error-msg-notin-dom*)
  ∈ *uint64-nat-assn^k* →$_a$ *raw-string-assn*⟩
⟨(*return o* (*error-msg-reused-dom o nat-of-uint64*), *RETURN o error-msg-reused-dom*)
  ∈ *uint64-nat-assn^k* →$_a$ *raw-string-assn*⟩
⟨(*uncurry* (*return oo* (λi. *error-msg* (*nat-of-uint64 i*))), *uncurry* (*RETURN oo error-msg*))
  ∈ *uint64-nat-assn^k* *$_a$ *raw-string-assn^k* →$_a$ *status-assn raw-string-assn*⟩
⟨(*uncurry* (*return oo error-msg*), *uncurry* (*RETURN oo error-msg*))
  ∈ *nat-assn^k* *$_a$ *raw-string-assn^k* →$_a$ *status-assn raw-string-assn*⟩
⟨*proof*⟩

sepref-definition *check-addition-l-impl*
  is ⟨*uncurry6 check-addition-l*⟩
  :: ⟨*poly-assn^k* *$_a$ *polys-assn^k* *$_a$ *vars-assn^k* *$_a$ *uint64-nat-assn^k* *$_a$ *uint64-nat-assn^k* *$_a$
      uint64-nat-assn^k *$_a$ *poly-assn^k* →$_a$ *status-assn raw-string-assn*⟩
  ⟨*proof*⟩

declare *check-addition-l-impl.refine*[*sepref-fr-rules*]

sepref-register *check-mult-l-dom-err*

definition *check-mult-l-dom-err-impl* **where**
  ⟨*check-mult-l-dom-err-impl pd p ia i* =
    (*if pd then* ″*The polynomial with id* ″ @ *show* (*nat-of-uint64 p*) @ ″ *was not found*″ *else* ″″) @
    (*if ia then* ″*The id of the resulting id* ″ @ *show* (*nat-of-uint64 i*) @ ″ *was already given*″ *else* ″″)⟩

definition *check-mult-l-mult-err-impl* **where**
  ⟨*check-mult-l-mult-err-impl p q pq r* =
    ″*Multiplying* ″ @ *show p* @ ″ *by* ″ @ *show q* @ ″ *gives* ″ @ *show pq* @ ″ *and not* ″ @ *show r*⟩

lemma [*sepref-fr-rules*]:
⟨(*uncurry3* ((λx y. *return oo* (*check-mult-l-dom-err-impl x y*))),
  uncurry3 (*check-mult-l-dom-err*)) ∈ *bool-assn^k* *$_a$ *uint64-nat-assn^k* *$_a$ *bool-assn^k* *$_a$ *uint64-nat-assn^k*
→$_a$ *raw-string-assn*⟩
  ⟨*proof*⟩

lemma [*sepref-fr-rules*]:
⟨(*uncurry3* ((λx y. *return oo* (*check-mult-l-mult-err-impl x y*))),
  uncurry3 (*check-mult-l-mult-err*)) ∈ *poly-assn^k* *$_a$ *poly-assn^k* *$_a$ *poly-assn^k* *$_a$ *poly-assn^k* →$_a$ *raw-string-assn*⟩
  ⟨*proof*⟩

sepref-definition *check-mult-l-impl*
  is ⟨*uncurry6 check-mult-l*⟩
  :: ⟨*poly-assn^k* *$_a$ *polys-assn^k* *$_a$ *vars-assn^k* *$_a$ *uint64-nat-assn^k* *$_a$ *poly-assn^k* *$_a$ *uint64-nat-assn^k* *$_a$
poly-assn^k →$_a$ *status-assn raw-string-assn*⟩
  ⟨*proof*⟩

declare *check-mult-l-impl.refine*[*sepref-fr-rules*]

definition *check-ext-l-dom-err-impl* :: ‹*uint64* $\Rightarrow$ *-*› **where**
‹*check-ext-l-dom-err-impl p* =
  ''*There is already a polynomial with index* '' @ *show* (*nat-of-uint64 p*)›

lemma [*sepref-fr-rules*]:
‹(((*return o* (*check-ext-l-dom-err-impl*))),
  (*check-extension-l-dom-err*)) $\in$ *uint64-nat-assn*^{k} $\rightarrow_a$ *raw-string-assn*›
  ⟨*proof*⟩

definition *check-extension-l-no-new-var-err-impl* :: ‹*-* $\Rightarrow$ *-*› **where**
‹*check-extension-l-no-new-var-err-impl p* =
  ''*No new variable could be found in polynomial* '' @ *show p*›

lemma [*sepref-fr-rules*]:
‹(((*return o* (*check-extension-l-no-new-var-err-impl*))),
  (*check-extension-l-no-new-var-err*)) $\in$ *poly-assn*^{k} $\rightarrow_a$ *raw-string-assn*›
  ⟨*proof*⟩

definition *check-extension-l-side-cond-err-impl* :: ‹*-* $\Rightarrow$ *-*› **where**
‹*check-extension-l-side-cond-err-impl v p r s* =
  ''*Error while checking side conditions of extensions polynow, var is* '' @ *show v* @
  '' *polynomial is* '' @ *show p* @ ''*side condition* $p*p - p =$ '' @ *show s* @ '' *and should be 0*''›

lemma [*sepref-fr-rules*]:
‹(((*uncurry3* ($\lambda x\ y.\ return\ oo$ (*check-extension-l-side-cond-err-impl x y*)))),
  uncurry3 (*check-extension-l-side-cond-err*)) $\in$ *string-assn*^{k} $*_a$ *poly-assn*^{k} $*_a$ *poly-assn*^{k} $*_a$ *poly-assn*^{k}
$\rightarrow_a$ *raw-string-assn*›
  ⟨*proof*⟩

definition *check-extension-l-new-var-multiple-err-impl* :: ‹*-* $\Rightarrow$ *-*› **where**
‹*check-extension-l-new-var-multiple-err-impl v p* =
  ''*Error while checking side conditions of extensions polynow, var is* '' @ *show v* @
  '' *but it either appears at least once in the polynomial or another new variable is created* '' @
  show p @ '' *but should not.*''›

lemma [*sepref-fr-rules*]:
‹(((*uncurry* (*return oo* (*check-extension-l-new-var-multiple-err-impl*)))),
  uncurry (*check-extension-l-new-var-multiple-err*)) $\in$ *string-assn*^{k} $*_a$ *poly-assn*^{k} $\rightarrow_a$ *raw-string-assn*›
  ⟨*proof*⟩

sepref-register *check-extension-l-dom-err fmlookup$'$*
  check-extension-l-side-cond-err check-extension-l-no-new-var-err
  check-extension-l-new-var-multiple-err

definition *uminus-poly* :: ‹*llist-polynomial* $\Rightarrow$ *llist-polynomial*› **where**
‹*uminus-poly p$'$* = *map* ($\lambda(a, b).\ (a, - b)$) *p$'$*›

sepref-register *uminus-poly*
lemma [*sepref-import-param*]:
‹(*map* ($\lambda(a, b).\ (a, - b)$), *uminus-poly*) $\in$ *poly-rel* $\rightarrow$ *poly-rel*›
  ⟨*proof*⟩

sepref-register *vars-of-poly-in*
  weak-equality-l

lemma [*safe-constraint-rules*]:
  ‹*Sepref-Constraints.CONSTRAINT single-valued (the-pure monomial-assn)*› **and**
  single-valued-the-monomial-assn:
    ‹*single-valued (the-pure monomial-assn)*›
    ‹*single-valued ((the-pure monomial-assn)$^{-1}$)*›
  ⟨*proof*⟩

sepref-definition *check-extension-l-impl*
  is ‹*uncurry5 check-extension-l*›
  :: ‹*poly-assn^k $*_a$ polys-assn^k $*_a$ vars-assn^k $*_a$ uint64-nat-assn^k $*_a$ string-assn^k $*_a$ poly-assn^k $\to_a$
    status-assn raw-string-assn*›
  ⟨*proof*⟩

declare *check-extension-l-impl.refine*[*sepref-fr-rules*]

sepref-definition *check-del-l-impl*
  is ‹*uncurry2 check-del-l*›
  :: ‹*poly-assn^k $*_a$ polys-assn^k $*_a$ uint64-nat-assn^k $\to_a$ status-assn raw-string-assn*›
  ⟨*proof*⟩

lemmas [*sepref-fr-rules*] = *check-del-l-impl.refine*

abbreviation *pac-step-rel* **where**
  ‹*pac-step-rel ≡ p2rel (⟨Id, ⟨monomial-rel⟩list-rel, Id⟩ pac-step-rel-raw)*›

sepref-register *PAC-Polynomials-Operations.normalize-poly*
  pac-src1 pac-src2 new-id pac-mult case-pac-step check-mult-l
  check-addition-l check-del-l check-extension-l

lemma *pac-step-rel-assn-alt-def2*:
  ‹*hn-ctxt (pac-step-rel-assn nat-assn poly-assn id-assn) b bi =*
      hn-val
       (p2rel
         (⟨nat-rel, poly-rel, Id :: (string × -) set⟩pac-step-rel-raw)) b bi›
  ⟨*proof*⟩

lemma *is-AddD-import*[*sepref-fr-rules*]:
  assumes ‹*CONSTRAINT is-pure K*› ‹*CONSTRAINT is-pure V*›
  shows
    ‹*(return o pac-res, RETURN o pac-res) ∈ [λx. is-Add x ∨ is-Mult x ∨ is-Extension x]$_a$*
      (pac-step-rel-assn K V R)^k → V›
    ‹*(return o pac-src1, RETURN o pac-src1) ∈ [λx. is-Add x ∨ is-Mult x ∨ is-Del x]$_a$ (pac-step-rel-assn*
  K V R)^k → K›
    ‹*(return o new-id, RETURN o new-id) ∈ [λx. is-Add x ∨ is-Mult x ∨ is-Extension x]$_a$ (pac-step-rel-assn*
  K V R)^k → K›
    ‹*(return o is-Add, RETURN o is-Add) ∈ (pac-step-rel-assn K V R)^k $\to_a$ bool-assn*›
    ‹*(return o is-Mult, RETURN o is-Mult) ∈ (pac-step-rel-assn K V R)^k $\to_a$ bool-assn*›
    ‹*(return o is-Del, RETURN o is-Del) ∈ (pac-step-rel-assn K V R)^k $\to_a$ bool-assn*›
    ‹*(return o is-Extension, RETURN o is-Extension) ∈ (pac-step-rel-assn K V R)^k $\to_a$ bool-assn*›
  ⟨*proof*⟩

lemma [*sepref-fr-rules*]:
  ‹*CONSTRAINT is-pure K* $\implies$
  (*return o pac-src2*, *RETURN o pac-src2*) $\in$ [$\lambda x.$ *is-Add x*]$_a$ (*pac-step-rel-assn K V R*)^k $\to$ *K*›
  ‹*CONSTRAINT is-pure V* $\implies$
  (*return o pac-mult*, *RETURN o pac-mult*) $\in$ [$\lambda x.$ *is-Mult x*]$_a$ (*pac-step-rel-assn K V R*)^k $\to$ *V*›
  ‹*CONSTRAINT is-pure R* $\implies$
  (*return o new-var*, *RETURN o new-var*) $\in$ [$\lambda x.$ *is-Extension x*]$_a$ (*pac-step-rel-assn K V R*)^k $\to$ *R*›
  ⟨*proof*⟩

lemma *is-Mult-lastI*:
  ‹¬ *is-Add b* $\implies$ ¬*is-Mult b* $\implies$ ¬*is-Extension b* $\implies$ *is-Del b*›
  ⟨*proof*⟩

sepref-register *is-cfailed is-Del*

definition *PAC-checker-l-step′* :: - **where**
  ‹*PAC-checker-l-step′ a b c d = PAC-checker-l-step a (b, c, d)*›

lemma *PAC-checker-l-step-alt-def*:
  ‹*PAC-checker-l-step a bcd e = (let (b,c,d) = bcd in PAC-checker-l-step′ a b c d e)*›
  ⟨*proof*⟩

sepref-decl-intf ($'k$) *acode-status* **is** ($'k$) *code-status*
sepref-decl-intf ($'k$, $'b$, $'lbl$) *apac-step* **is** ($'k$, $'b$, $'lbl$) *pac-step*

sepref-register *merge-cstatus full-normalize-poly new-var is-Add*

lemma *poly-rel-the-pure*:
  ‹*poly-rel = the-pure poly-assn*› **and**
  nat-rel-the-pure:
  ‹*nat-rel = the-pure nat-assn*› **and**
  WTF-RF: ‹*pure (the-pure nat-assn) = nat-assn*›
  ⟨*proof*⟩

lemma [*safe-constraint-rules*]:
    ‹*CONSTRAINT IS-LEFT-UNIQUE uint64-nat-rel*› **and**
  single-valued-uint64-nat-rel[*safe-constraint-rules*]:
    ‹*CONSTRAINT single-valued uint64-nat-rel*›
  ⟨*proof*⟩

sepref-definition *check-step-impl*
  is ‹*uncurry4 PAC-checker-l-step′*›
  :: ‹*poly-assn*^{k} $*_a$ (*status-assn raw-string-assn*)^d $*_a$ *vars-assn*^{d} $*_a$ *polys-assn*^{d} $*_a$ (*pac-step-rel-assn*
(*uint64-nat-assn*) *poly-assn* (*string-assn* :: *string* $\Rightarrow$ -))^d $\to_a$
    status-assn raw-string-assn $\times_a$ *vars-assn* $\times_a$ *polys-assn*›
  ⟨*proof*⟩

declare *check-step-impl.refine*[*sepref-fr-rules*]

sepref-register *PAC-checker-l-step PAC-checker-l-step′ fully-normalize-poly-impl*

definition *PAC-checker-l′* **where**
  ‹*PAC-checker-l′ p $\mathcal{V}$ A status steps = PAC-checker-l p ($\mathcal{V}$, A) status steps*›

lemma *PAC-checker-l-alt-def*:
⟨*PAC-checker-l p $\mathcal{V}A$ status steps =*
   (let $(\mathcal{V}, A) = \mathcal{V}A$ in PAC-checker-l' p $\mathcal{V}$ A status steps)⟩
⟨*proof*⟩

sepref-definition *PAC-checker-l-impl*
  is ⟨*uncurry4 PAC-checker-l'*⟩
  :: ⟨*poly-assn^k $*_a$ vars-assn^d $*_a$ polys-assn^d $*_a$ (status-assn raw-string-assn)^d $*_a$*
       (list-assn (pac-step-rel-assn (uint64-nat-assn) poly-assn string-assn))^k $\to_a$
     status-assn raw-string-assn $\times_a$ vars-assn $\times_a$ polys-assn⟩
  ⟨*proof*⟩

declare *PAC-checker-l-impl.refine*[*sepref-fr-rules*]

abbreviation *polys-assn-input* **where**
  ⟨*polys-assn-input $\equiv$ iam-fmap-assn nat-assn poly-assn*⟩

definition *remap-polys-l-dom-err-impl* :: ⟨*-*⟩ **where**
  ⟨*remap-polys-l-dom-err-impl =*
     "Error during initialisation. Too many polynomials where provided. If this happens," @
     "please report the example to the authors, because something went wrong during " @
     "code generation (code generation to arrays is likely to be broken)."⟩

lemma [*sepref-fr-rules*]:
  ⟨*((uncurry0 (return (remap-polys-l-dom-err-impl))),*
    uncurry0 (remap-polys-l-dom-err)) $\in$ unit-assn^k $\to_a$ raw-string-assn⟩
  ⟨*proof*⟩

MLton is not able to optimise the calls to pow.

lemma *pow-2-64*: ⟨*(2::nat) $\hat{\ }$ 64 = 18446744073709551616*⟩
  ⟨*proof*⟩

sepref-register *upper-bound-on-dom op-fmap-empty*

sepref-definition *remap-polys-l-impl*
  is ⟨*uncurry2 remap-polys-l2*⟩
  :: ⟨*poly-assn^k $*_a$ vars-assn^d $*_a$ polys-assn-input^d $\to_a$*
    status-assn raw-string-assn $\times_a$ vars-assn $\times_a$ polys-assn⟩
  ⟨*proof*⟩

lemma *remap-polys-l2-remap-polys-l*:
  ⟨*(uncurry2 remap-polys-l2, uncurry2 remap-polys-l) $\in$ (Id $\times_r$ ⟨Id⟩set-rel) $\times_r$ Id $\to_f$ ⟨Id⟩nres-rel*⟩
  ⟨*proof*⟩

lemma [*sepref-fr-rules*]:
  ⟨*(uncurry2 remap-polys-l-impl,*
    uncurry2 remap-polys-l) $\in$ poly-assn^k $*_a$ vars-assn^d $*_a$ polys-assn-input^d $\to_a$
     status-assn raw-string-assn $\times_a$ vars-assn $\times_a$ polys-assn⟩
  ⟨*proof*⟩

sepref-register *remap-polys-l*

sepref-definition *full-checker-l-impl*
  is ⟨*uncurry2 full-checker-l*⟩

$$:: \langle poly\text{-}assn^k *_a polys\text{-}assn\text{-}input^d *_a (list\text{-}assn (pac\text{-}step\text{-}rel\text{-}assn (uint64\text{-}nat\text{-}assn) poly\text{-}assn string\text{-}assn))^k$$
$$\rightarrow_a$$
$$status\text{-}assn raw\text{-}string\text{-}assn \times_a vars\text{-}assn \times_a polys\text{-}assn\rangle$$
$\langle proof \rangle$

sepref-definition *PAC-update-impl*
  is $\langle uncurry2 (RETURN \; ooo \; fmupd) \rangle$
  $:: \langle nat\text{-}assn^k *_a poly\text{-}assn^k *_a (polys\text{-}assn\text{-}input)^d \rightarrow_a polys\text{-}assn\text{-}input \rangle$
  $\langle proof \rangle$

sepref-definition *PAC-empty-impl*
  is $\langle uncurry0 (RETURN \; fmempty) \rangle$
  $:: \langle unit\text{-}assn^k \rightarrow_a polys\text{-}assn\text{-}input \rangle$
  $\langle proof \rangle$

sepref-definition *empty-vars-impl*
  is $\langle uncurry0 (RETURN \; \{\}) \rangle$
  $:: \langle unit\text{-}assn^k \rightarrow_a vars\text{-}assn \rangle$
  $\langle proof \rangle$

This is a hack for performance. There is no need to recheck that that a char is valid when working on chars coming from strings... It is not that important in most cases, but in our case the preformance difference is really large.

definition *unsafe-asciis-of-literal* $:: \langle \text{-} \rangle$ **where**
  $\langle unsafe\text{-}asciis\text{-}of\text{-}literal \; xs = String.asciis\text{-}of\text{-}literal \; xs \rangle$

definition *unsafe-asciis-of-literal′* $:: \langle \text{-} \rangle$ **where**
  [*simp, symmetric, code*]: $\langle unsafe\text{-}asciis\text{-}of\text{-}literal' = unsafe\text{-}asciis\text{-}of\text{-}literal \rangle$

code-printing
  constant *unsafe-asciis-of-literal′* $\rightharpoonup$
    (*SML*) !(*List.map* (*fn c => let val k = Char.ord c in IntInf.fromInt k end*) /o *String.explode*)

Now comes the big and ugly and unsafe hack.

Basically, we try to avoid the conversion to IntInf when calculating the hash. The performance gain is roughly 40%, which is a LOT and definitively something we need to do. We are aware that the SML semantic encourages compilers to optimise conversions, but this does not happen here, corroborating our early observation on the verified SAT solver IsaSAT.x

definition *raw-explode* **where**
  [*simp*]: $\langle raw\text{-}explode = String.explode \rangle$
code-printing
  constant *raw-explode* $\rightharpoonup$
    (*SML*) *String.explode*

definition $\langle hashcode\text{-}literal' \; s \equiv$
    $foldl \; (\lambda h \; x. \; h * 33 + uint32\text{-}of\text{-}int \; (of\text{-}char \; x)) \; 5381$
      $(raw\text{-}explode \; s) \rangle$

lemmas [*code*] =
  hashcode-literal-def[*unfolded String.explode-code*
    unsafe-asciis-of-literal-def[*symmetric*]]

definition *uint32-of-char* **where**
  [*symmetric, code-unfold*]: $\langle uint32\text{-}of\text{-}char \; x = uint32\text{-}of\text{-}int \; (int\text{-}of\text{-}char \; x) \rangle$

code-printing
  constant *uint32-of-char* ⇀
    (*SML*) !(*Word32.fromInt* /*o* (*Char.ord*))

lemma [*code*]: ‹*hashcode s = hashcode-literal′ s*›
  ⟨*proof*⟩

We do not include

export-code *PAC-checker-l-impl PAC-update-impl PAC-empty-impl the-error is-cfailed is-cfound*
  int-of-integer Del Add Mult nat-of-integer String.implode remap-polys-l-impl
  fully-normalize-poly-impl union-vars-poly-impl empty-vars-impl
  full-checker-l-impl check-step-impl CSUCCESS
  Extension hashcode-literal′ version
  in *SML-imp* **module-name** *PAC-Checker*
  file-prefix *checker*

compile-generated-files -
  external-files
    ‹*code/parser.sml*›
    ‹*code/pasteque.sml*›
    ‹*code/pasteque.mlb*›
  where ‹*fn dir =>*
    let
      val exec = Generated-Files.execute (Path.append dir (Path.basic code));
      val - = exec ‹*rename file*› *mv checker.ML checker.sml*
      val - =
        exec ‹*Compilation*›
          (*File.bash-path* **path** ‹*$ISABELLE-MLTON*› ̂ ̂
            −*const ′MLton.safe false′* −*verbose 1* −*default*−*type int64* −*output pasteque* ̂
            −*codegen native* −*inline 700* −*cc*−*opt* −*O3 pasteque.mlb*);
    in () *end*›

14   Correctness theorem

context *poly-embed*
begin

definition *full-poly-assn* **where**
  ‹*full-poly-assn = hr-comp poly-assn (fully-unsorted-poly-rel O mset-poly-rel)*›

definition *full-poly-input-assn* **where**
  ‹*full-poly-input-assn = hr-comp*
      (*hr-comp polys-assn-input*
        (‹*nat-rel, fully-unsorted-poly-rel O mset-poly-rel*›*fmap-rel*))
      polys-rel›

definition *fully-pac-assn* **where**
  ‹*fully-pac-assn = (list-assn*
      (*hr-comp (pac-step-rel-assn uint64-nat-assn poly-assn string-assn)*
        (*p2rel*
          (‹*nat-rel,*
           fully-unsorted-poly-rel O

$$mset\text{-}poly\text{-}rel, \ var\text{-}rel\rangle pac\text{-}step\text{-}rel\text{-}raw)))))\rangle$$

definition *code-status-assn* **where**
 ‹*code-status-assn = hr-comp* (*status-assn raw-string-assn*)
$$code\text{-}status\text{-}status\text{-}rel\rangle$$

definition *full-vars-assn* **where**
 ‹*full-vars-assn = hr-comp* (*hs.assn string-assn*)
$$(\langle var\text{-}rel\rangle set\text{-}rel)\rangle$$

lemma *polys-rel-full-polys-rel*:
 ‹*polys-rel-full = Id* $\times_r$ *polys-rel*›
 ‹*proof*›

definition *full-polys-assn* :: ‹-› **where**
‹*full-polys-assn = hr-comp* (*hr-comp polys-assn*
$$(\langle nat\text{-}rel,$$
$$sorted\text{-}poly\text{-}rel \ O \ mset\text{-}poly\text{-}rel\rangle fmap\text{-}rel))$$
$$polys\text{-}rel\rangle$$

Below is the full correctness theorems. It basically states that:

1. assuming that the input polynomials have no duplicate variables

Then:

1. if the checker returns *CFOUND*, the spec is in the ideal and the PAC file is correct

2. if the checker returns *CSUCCESS*, the PAC file is correct (but there is no information on the spec, aka checking failed)

3. if the checker return *CFAILED err*, then checking failed (and *err might* give you an indication of the error, but the correctness theorem does not say anything about that).

    The input parameters are:

4. the specification polynomial represented as a list

5. the input polynomials as hash map (as an array of option polynomial)

6. a represention of the PAC proofs.

lemma *PAC-full-correctness*:
 ‹(*uncurry2 full-checker-l-impl*,
   uncurry2 (λ*spec A -. PAC-checker-specification spec A*))
  $\in$ (*full-poly-assn*)^k $*_a$ (*full-poly-input-assn*)^d $*_a$ (*fully-pac-assn*)^k $\rightarrow_a$ *hr-comp*
   (*code-status-assn* $\times_a$ *full-vars-assn* $\times_a$ *hr-comp polys-assn*
$$(\langle nat\text{-}rel, \ sorted\text{-}poly\text{-}rel \ O \ mset\text{-}poly\text{-}rel\rangle fmap\text{-}rel))$$
$$\{((st, \ G), \ st', \ G').$$
$$st = st' \land (st \neq FAILED \longrightarrow (G, \ G') \in Id \ \times_r \ polys\text{-}rel)\}\rangle$$
 ‹*proof*›

It would be more efficient to move the parsing to Isabelle, as this would be more memory efficient (and also reduce the TCB). But now comes the fun part: It cannot work. A stream (of a file) is consumed by side effects. Assume that this would work. The code could look like:

Let (*read-file file*) *f*

This code is equal to (in the HOL sense of equality): *let - = read-file file in Let* (*read-file file*) *f*
However, as an hypothetical *read-file* changes the underlying stream, we would get the next token. Remark that this is already a weird point of ML compilers. Anyway, I see currently two solutions to this problem:

1. The meta-argument: use it only in the Refinement Framework in a setup where copies are disallowed. Basically, this works because we can express the non-duplication constraints on the type level. However, we cannot forbid people from expressing things directly at the HOL level.

2. On the target language side, model the stream as the stream and the position. Reading takes two arguments. First, the position to read. Second, the stream (and the current position) to read. If the position to read does not match the current position, return an error. This would fit the correctness theorem of the code generation (roughly "if it terminates without exception, the answer is the same"), but it is still unsatisfactory.

end

definition φ :: ⟨*string* $\Rightarrow$ *nat*⟩ **where**
  ⟨$\varphi = (SOME\ \varphi.\ bij\ \varphi)$⟩

lemma *bij-φ*: ⟨*bij* φ⟩
  ⟨*proof*⟩

global-interpretation *PAC*: *poly-embed* **where**
  $\varphi = \varphi$
  ⟨*proof*⟩

The full correctness theorem is (*uncurry2 full-checker-l-impl*, *uncurry2* (λ*spec A -. PAC-checker-specification spec A*)) $\in$ *PAC.full-poly-assn^k* $*_a$ *PAC.full-poly-input-assn^d* $*_a$ *PAC.fully-pac-assn^k* $\rightarrow_a$ *hr-comp* (*PAC.code-status-assn* $\times_a$ *PAC.full-vars-assn* $\times_a$ *hr-comp polys-assn* (⟨*nat-rel, sorted-poly-rel O PAC.mset-poly-rel*⟩*fmap-rel*)) {((st, G), st', G'). $st = st' \wedge (st \neq FAILED \longrightarrow (G, G') \in Id \times_r polys\text{-}rel$)}.

end

References

[1] D. Kaufmann, M. Fleury, and A. Biere. The proof checkers pacheck and pasteque for the practical algebraic calculus. In O. Strichman and A. Ivrii, editors, *Formal Methods in Computer-Aided Design, FMCAD 2020, September 21-24, 2020.* IEEE, 2020.