PAC Checker

Mathias Fleury and Daniela Kaufmann

February 6, 2026

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 Overview **2**

2 Libraries **4**
2.1 More Polynomials . . . 4
2.2 More Ideals . . . 18

3 Finite maps and multisets **19**
3.1 Finite sets and multisets . . . 19
3.2 Finite map and multisets . . . 20
3.3 More Theorem about Loops . . . 54

4 Specification of the PAC checker **57**
4.1 Ideals . . . 57
4.2 PAC Format . . . 58

5 Hash-Map for finite mappings **67**
5.1 Operations . . . 68
5.2 Patterns . . . 70
5.3 Mapping to Normal Hashmaps . . . 70

6 Checker Algorithm **73**
6.1 Specification . . . 73
6.2 Algorithm . . . 74
6.3 Full Checker . . . 86

7 Polynomials of strings **88**
7.1 Polynomials and Variables . . . 88
7.2 Addition . . . 90
7.3 Normalisation . . . 92
7.4 Correctness . . . 92

8 Terms **97**
8.1 Ordering . . . 97
8.2 Polynomials . . . 98
8.3 Addition . . . 100
8.4 Multiplication . . . 107
8.5 Normalisation . . . 113
8.6 Multiplication and normalisation . . . 121
8.7 Correctness . . . 122

9 Executable Checker **124**
9.1 Definitions . . . 124
9.2 Correctness . . . 132

10 Various Refinement Relations **149**

11 Hash Map as association list **156**
11.1 Conversion from assoc to other map . . . 158

12 Initial Normalisation of Polynomials **159**
12.1 Sorting . . . 159
12.2 Sorting applied to monomials . . . 160
12.3 Lifting to polynomials . . . 163

13 Code Synthesis of the Complete Checker **175**

14 Correctness theorem **189**

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

1 Overview

One solution to check circuit of multipliers is to use algebraic method, like producing proofs on polynomials. We are here interested in checking PAC proofs on the Boolean ring. The idea is the following: each variable represents an input or the output of a gate and we want to prove the bitwise multiplication of the input bits yields the output, namely the bitwise representation of the multiplication of the input (modulo 2^n where n is the number of bits of the circuit).

Algebraic proof systems typically reason over polynomials in a ring $\mathbb{K}[X]$, where the variables X represent Boolean values. The aim of an algebraic proof is to derive whether a polynomial f can be derived from a given set of polynomials $G = \{g_1, \ldots, g_l\} \subseteq \mathbb{K}[X]$ together with the Boolean value constraints $B(X) = \{x_i^2 - x_i \mid x_i \in X\}$. In algebraic terms this means to show that the polynomial $f \in \langle G \cup B(X)\rangle$.

In our setting we set $\mathbb{K} = \mathbb{Z}$ and we treat the Boolean value constraints implicitly, i.e., we consider proofs in the ring $\mathbb{Z}[X]/\langle B(X)\rangle$ to admit shorter proofs

The checker takes as input 3 files:

1. an input file containing all polynomials that are initially present;
2. a target (or specification) polynomial ;

3. a "proof" file to check that contains the proof in PAC format that shows that the specification is in the ideal generated by the polynomials present initially.

Each step of the proof is either an addition of two polynomials previously derived, a multiplication from a previously derived polynomial and an arbitrary polynomial, and the deletion a derived polynomial.

One restriction on the proofs compared to generic PAC proofs is that $x^2 = x$ in the Boolean ring we are considering.

The checker can produce two outputs: valid (meaning that each derived polynomial in the proof has been correctly derived and the specification polynomial was also derived at some point [either in the proof or as input]) or invalid (without proven information what went wrong).

The development is organised as follows:

- `PAC_Specification.thy` (this file) contains the specification as described above on ideals without any peculiarities on the PAC proof format
- `PAC_Checker_Specification.thy` specialises to the PAC format and enters the nondeterminism monad to prepare the subsequent refinements.
- `PAC_Checker.thy` contains the refined version where polynomials are represented as lists.
- `PAC_Checker_Synthesis.thy` contains the efficient implementation with imperative data structure like a hash set.
- `PAC_Checker_MLton.thy` contains the code generation and the command to compile the file with the ML compiler MLton.

Here is an example of a proof and an input file (taken from the appendix of our FMCAD paper [1], available at http://fmv.jku.at/pacheck_pasteque):

```
<res.input>       <res.proof>
 1 x*y;            3  = fz, -z+1;
 2 y*z-y-z+1;      4  *  3,  y-1, -fz*y+fz-y*z+y+z-1;
                   5  +  2,    4, -fz*y+fz;
                   2  d;
                   4  d;
<res.target>       6  *  1,   fz, fz*x*y;
 -x*z+x;           1  d;
                   7  *  5,    x, -fz*x*y+fz*x;
                   8  +  6,    7, fz*x;
                   9  *  3,    x, -fz*x-x*z+x;
                  10  +  8,    9, -x*z+x;
```

Each line starts with a number that is used to index the (conclusion) polynomial. In the proof, there are four kind of steps:

1. `3 = fz, -z+1;` is an extension that introduces a new variable (in this case `fz`);

2. `4 * 3, y-1, -fz*y+fz-y*z+y+z-1;` is a multiplication of the existing polynomial with index 3 by the arbitrary polynomial `y-1` and generates the new polynomial `-fz*y+fz-y*z+y+z-1` with index 4;

3. `5 + 2, 4, -fz*y+fz;` is an addition of the existing polynomials with index 2 and 4 and generates the new polynomial `-fz*y+fz` with index 5;

4. `1 d;` deletes the polynomial with index 1 and it cannot be reused in subsequent steps.

Remark that unlike DRAT checker, we do forward checking and check every derived polynomial. The target polynomial can also be part of the input file.

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*›
by *transfer*
(*simp add*: *Const$_0$-def single-add*)

lemma *Const-mult*:
‹*Const* (*a* ∗ *b*) = *Const a* ∗ *Const b*›
by *transfer* (*simp add*: *Const$_0$-def times-monomial-monomial*)

lemma *Const$_0$-mult*:
‹*Const$_0$* (*a* ∗ *b*) = *Const$_0$ a* ∗ *Const$_0$ b*›
by *transfer* (*simp add*: *Const$_0$-def times-monomial-monomial*)

lemma *Const0*[*simp*]:
‹*Const 0* = *0*›
by *transfer* (*simp add*: *Const$_0$-def*)

lemma (**in** −) *Const-uminus*[*simp*]:
‹*Const* (−*n*) = − *Const n*›
by *transfer* (*auto simp*: *Const$_0$-def monomial-uminus*)

lemma [*simp*]: ‹*Const$_0$ 0* = *0*›
‹*MPoly 0* = *0*›
by (*auto simp*: *Const$_0$-def zero-mpoly-def*)

lemma *Const-add*:
‹*Const* (*a* + *b*) = *Const a* + *Const b*›
by *transfer* (*simp add*: *Const$_0$-def single-add*)

instance *mpoly* :: (*comm-semiring-1*) *comm-semiring-1*
by *standard*

lemma *degree-uminus*[*simp*]:
‹*degree* (−*A*) *x′* = *degree A x′*›
by (*auto simp*: *degree-def uminus-mpoly.rep-eq*)

lemma *degree-sum-notin*:
‹$x' \notin vars\ B \Longrightarrow degree\ (A + B)\ x' = degree\ A\ x'$›
apply (*auto simp*: *degree-def*)
apply (*rule arg-cong*[*of* - - *Max*])
apply *standard*+
apply (*auto simp*: *plus-mpoly.rep-eq UN-I UnE image-iff in-keys-iff subsetD vars-def lookup-add*
dest: *keys-add intro*: *in-keys-plusI1 cong*: *ball-cong-simp*)
done

lemma *degree-notin-vars*:
‹$x \notin (vars\ B) \Longrightarrow degree\ (B :: 'a :: \{monoid\text{-}add\}\ mpoly)\ x = 0$›
using *degree-sum-notin*[*of x B 0*]
by *auto*

lemma *not-in-vars-coeff0*:
‹$x \notin vars\ p \Longrightarrow MPoly\text{-}Type.coeff\ p\ (monomial\ (Suc\ 0)\ x) = 0$›
by (*subst not-not*[*symmetric*], *subst coeff-keys*[*symmetric*])
(*auto simp*: *vars-def*)

lemma *keys-add'*:
$p \in keys\ (f + g) \Longrightarrow p \in keys\ f \cup keys\ g$
by *transfer auto*

lemma *keys-mapping-sum-add*:
‹$finite\ A \Longrightarrow keys\ (mapping\text{-}of\ (\sum v \in A.\ f\ v)) \subseteq \bigcup (keys\ `\ mapping\text{-}of\ `\ f\ `\ UNIV)$›
by (*induction A rule*: *finite-induct*)
(*auto simp add*: *zero-mpoly.rep-eq plus-mpoly.rep-eq*
keys-plus-ninv-comm-monoid-add dest: *keys-add'*)

lemma *vars-sum-vars-union*:
fixes f :: ‹*int mpoly* ⇒ *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* ⊆ *?B*›)
proof
fix p
assume ‹$p \in vars\ (\sum v \mid f\ v \neq 0.\ f\ v * v)$›
then obtain x **where** ‹$x \in keys\ (mapping\text{-}of\ (\sum v \mid f\ v \neq 0.\ f\ v * v))$› **and**
p: ‹$p \in keys\ x$›
by (*auto simp*: *vars-def times-mpoly.rep-eq simp del*: *keys-mult*)
then have ‹$x \in (\bigcup x.\ keys\ (mapping\text{-}of\ (f\ x) * mapping\text{-}of\ x))$›
using *keys-mapping-sum-add*[*of* ‹$\{v.\ f\ v \neq 0\}$› ‹$\lambda x.\ f\ x * x$›] *assms*
by (*auto simp*: *vars-def times-mpoly.rep-eq*)
then have ‹$x \in (\bigcup x.\ \{a+b|\ a\ b.\ a \in keys\ (mapping\text{-}of\ (f\ x)) \wedge b \in keys\ (mapping\text{-}of\ x)\})$›
using *Union-mono*[*OF*] *keys-mult* **by** *fast*
then show ‹$p \in$ *?B*›
using p **by** (*force simp*: *vars-def zero-mpoly.rep-eq dest*!: *keys-add'*)
qed

lemma *vars-in-right-only*:
$x \in vars\ q \Longrightarrow x \notin vars\ p \Longrightarrow x \in vars\ (p+q)$
unfolding *vars-def keys-def plus-mpoly.rep-eq lookup-plus-fun*
apply *clarify*

subgoal for *xa*
by (*auto simp*: *vars-def keys-def plus-mpoly.rep-eq*
lookup-plus-fun intro!: *exI*[*of - xa*] *dest*!: *spec*[*of - xa*])
done

lemma [*simp*]:
‹*vars 0* = {}›
by (*simp add*: *vars-def zero-mpoly.rep-eq*)

lemma *vars-Un-nointer*:
‹*keys* (*mapping-of p*) ∩ *keys* (*mapping-of q*) = {} ⟹ *vars* (*p* + *q*) = *vars p* ∪ *vars q*›
by (*auto simp*: *vars-def plus-mpoly.rep-eq simp flip*: *More-MPoly-Type.keys-add dest*!: *keys-add'*)

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

lemma *polynomial-sum-monoms*:
fixes *p* :: ‹'*a* :: {*comm-monoid-add*,*cancel-comm-monoid-add*} *mpoly*›
shows
‹*p* = (∑ *x*∈*keys* (*mapping-of p*). *MPoly-Type.monom x* (*MPoly-Type.coeff p x*))›
‹*keys* (*mapping-of p*) ⊆ *I* ⟹ *finite I* ⟹ *p* = (∑ *x*∈*I*. *MPoly-Type.monom x* (*MPoly-Type.coeff p x*))›
proof −
define *J* **where** ‹*J* ≡ *keys* (*mapping-of p*)›
define *a* **where** ‹*a x* ≡ *coeff p x*› **for** *x*
have ‹*finite* (*keys* (*mapping-of p*))›
by *auto*
have ‹*p* = (∑ *x*∈*I*. *MPoly-Type.monom x* (*MPoly-Type.coeff p x*))›
if ‹*finite I*› **and** ‹*keys* (*mapping-of p*) ⊆ *I*›
for *I*
using *that*
unfolding *a-def*
proof (*induction I arbitrary*: *p rule*: *finite-induct*)
case *empty*
then have ‹*p* = *0*›
using *empty coeff-all-0 coeff-keys* **by** *blast*
then show *?case* **using** *empty* **by** (*auto simp*: *zero-mpoly.rep-eq*)
next
case (*insert x F*) **note** *fin* = *this*(*1*) **and** *xF* = *this*(*2*) **and** *IH* = *this*(*3*) **and**
incl = *this*(*4*)
let *?p* = ‹*p* − *MPoly-Type.monom x* (*MPoly-Type.coeff p x*)›
have *H*: ‹⋀*xa*. *x* ∉ *F* ⟹ *xa* ∈ *F* ⟹
MPoly-Type.monom xa (*MPoly-Type.coeff* (*p* − *MPoly-Type.monom x* (*MPoly-Type.coeff p x*)) *xa*) =
MPoly-Type.monom xa (*MPoly-Type.coeff p xa*)›
by (*metis* (*mono-tags*, *opaque-lifting*) *add-diff-cancel-right' remove-term-coeff*
remove-term-sum when-def)
have ‹*?p* = (∑ *xa*∈*F*. *MPoly-Type.monom xa* (*MPoly-Type.coeff ?p xa*))›
apply (*rule IH*)
using *incl* **apply** −
by *standard* (*smt* (*verit*) *Diff-iff Diff-insert-absorb add-diff-cancel-right'*
remove-term-keys remove-term-sum subsetD xF)
also have ‹... = (∑ *xa*∈*F*. *MPoly-Type.monom xa* (*MPoly-Type.coeff p xa*))›
by (*use xF* **in** ‹*auto intro*!: *sum.cong simp*: *H*›)
finally show *?case*

```
      apply (subst (asm) remove-term-sum[of x p, symmetric])
      apply (subst remove-term-sum[of x p, symmetric])
      using xF fin by (auto simp: ac-simps)
    qed
    from this[of I] this[of J] show
     ‹p = (∑ x∈keys (mapping-of p). MPoly-Type.monom x (MPoly-Type.coeff p x))›
     ‹keys (mapping-of p) ⊆ I ⟹ finite I ⟹ p = (∑ x∈I. MPoly-Type.monom x (MPoly-Type.coeff p
x))›
     by (auto simp: J-def)
qed


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 −

  let ?v = ‹monom (monomial (Suc 0) x') 1›
  have
    p: ‹p = (∑ x∈keys (mapping-of p). MPoly-Type.monom x (MPoly-Type.coeff p x))› (is ‹- = (∑ x ∈
?I. ?f x)›)
    using polynomial-sum-monoms(1)[of p] .
  have pv: ‹p * ?v = (∑ x ∈ ?I. ?f x * ?v)›
    by (subst p) (auto simp: field-simps sum-distrib-left)
  define I where ‹I ≡ ?I›
  have in-keysD: ‹x ∈ keys (mapping-of (∑ x∈I. MPoly-Type.monom x (h x))) ⟹ x ∈ I›
   if ‹finite I› for I and h :: ‹- ⇒ int› and x
   using that by (induction rule: finite-induct)
    (force simp: monom.rep-eq empty-iff insert-iff keys-single coeff-monom
     simp: coeff-keys simp flip: coeff-add
     simp del: coeff-add)+
  have in-keys: ‹keys (mapping-of (∑ x∈I. MPoly-Type.monom x (h x))) = (⋃ x ∈ I. (if h x = 0 then
{} else {x}))›
   if ‹finite I› for I and h :: ‹- ⇒ int› and x
   supply in-keysD[dest]
   using that by (induction rule: finite-induct)
     (auto simp: plus-mpoly.rep-eq MPoly-Type-Class.keys-plus-eqI)

  have H[simp]: ‹vars ((∑ x∈I. MPoly-Type.monom x (h x))) = (⋃ x∈I. (if h x = 0 then {} else keys
x))›
   if ‹finite I› for I and h :: ‹- ⇒ int›
   using that by (auto simp: vars-def in-keys)

  have sums: ‹(∑ x∈I.
        MPoly-Type.monom (x + a') (c x)) =
       (∑ x∈ (λx. x + a') ‘ I.
        MPoly-Type.monom x (c (x − a')))›
    if ‹finite I› for I a' c q
    using that apply (induction rule: finite-induct)
    subgoal by auto
    subgoal
      unfolding image-insert by (subst sum.insert) auto
    done
  have non-zero-keysEx: ‹p ≠ 0 ⟹ ∃ a. a ∈ keys (mapping-of p)› for p :: ‹int mpoly›
     using mapping-of-inject by (fastforce simp add: ex-in-conv)
```

```
  have ‹finite I› ‹keys (mapping-of p) ⊆ I›
    unfolding I-def by auto
  then show
    ‹vars (p * (monom (monomial (Suc 0) x') 1)) = (if p = 0 then {} else insert x' (vars p))›
    apply (subst pv, subst I-def[symmetric], subst mult-monom)
    apply (auto simp: mult-monom sums I-def)
    using Poly-Mapping.keys-add vars-def apply fastforce
     apply (auto dest!: non-zero-keysEx)
     apply (rule-tac x= ‹a + monomial (Suc 0) x'› in bexI)
      apply (auto simp: coeff-keys)
     apply (simp add: in-keys-iff lookup-add)
    apply (auto simp: vars-def)
    apply (rule-tac x= ‹xa + monomial (Suc 0) x'› in bexI)
     apply (auto simp: coeff-keys)
    apply (simp add: in-keys-iff lookup-add)
    done
qed


  term ‹(x', u, lookup u x', A)›
lemma in-mapping-mult-single:
 ‹x ∈ (λx. lookup x x') ` keys (A * (Var₀ x' :: (nat ⇒₀ nat) ⇒₀ 'b :: {monoid-mult,zero-neq-one,semiring-0}))
⟷
  x > 0 ∧ x - 1 ∈ (λx. lookup x x') ` keys (A)›
  apply (standard+; clarify)
  subgoal
    apply (auto  elim!: in-keys-timesE simp: lookup-add)
    apply (auto simp: keys-def lookup-times-monomial-right Var₀-def lookup-single image-iff)
    done
  subgoal
    apply (auto  elim!: in-keys-timesE simp: lookup-add)
    apply (auto simp: keys-def lookup-times-monomial-right Var₀-def lookup-single image-iff)
    done
  subgoal for  xa
    apply (auto  elim!: in-keys-timesE simp: lookup-add)
    apply (auto simp: keys-def lookup-times-monomial-right Var₀-def lookup-single image-iff lookup-add
      intro!: exI[of - ‹xa + Poly-Mapping.single x' 1›])
    done
  done

lemma Max-Suc-Suc-Max:
  ‹finite A ⟹ A ≠ {} ⟹ Max (insert 0 (Suc ` A)) =
    Suc (Max (insert 0 A))›
  by (induction rule: finite-induct)
   (auto simp: hom-Max-commute)

lemma [simp]:
  ‹keys (Var₀ x' :: ('a ⇒₀ nat) ⇒₀ 'b :: {zero-neq-one}) = {Poly-Mapping.single x' 1}›
  by (auto simp: Var₀-def)


lemma degree-mult-Var:
  ‹degree (A * Var x') x' = (if A = 0 then 0 else Suc (degree A x'))› for A :: ‹int mpoly›
proof −
  have [simp]: ‹A ≠ 0 ⟹
```

```
    Max (insert 0 ((λx. Suc (lookup x x')) ` keys (mapping-of A))) =
    Max (insert (Suc 0) ((λx. Suc (lookup x x')) ` keys (mapping-of A)))›
    unfolding image-image[of Suc ‹λx. lookup x x'›, symmetric] image-insert[symmetric]
   by (subst Max-Suc-Suc-Max, use mapping-of-inject in fastforce, use mapping-of-inject in fastforce)+
     (simp add: Max.hom-commute)
  have ‹A ≠ 0 ⟹
    Max (insert 0
    ((λx. lookup x x') `
    keys (mapping-of A * mapping-of (Var x')))) =
    Suc (Max (insert 0 ((λm. lookup m x') ` keys (mapping-of A))))›
    by (subst arg-cong[of - ‹insert 0
     (Suc ` ((λx. lookup x x') ` keys (mapping-of A)))› Max])
     (auto simp: image-image Var.rep-eq lookup-plus-fun in-mapping-mult-single
      hom-Max-commute Max-Suc-Suc-Max
      elim!: in-keys-timesE  split: if-splits)
  then show ?thesis
    by (auto simp: degree-def times-mpoly.rep-eq
      intro!: arg-cong[of - ‹insert 0
      (Suc ` ((λx. lookup x x') ` keys (mapping-of A)))› Max])
qed

lemma degree-mult-Var':
  ‹degree (Var x' * A) x' = (if A = 0 then 0 else Suc (degree A x'))› for A :: ‹int mpoly›
 by (simp add: degree-mult-Var semiring-normalization-rules(7))

lemma degree-times-le:
  ‹degree (A * B) x ≤ degree A x + degree B x›
  by (auto simp: degree-def times-mpoly.rep-eq
       max-def lookup-add add-mono
    dest!: set-rev-mp[OF - Poly-Mapping.keys-add]
    elim!: in-keys-timesE)

lemma monomial-inj:
  monomial c s = monomial (d::'b::zero-neq-one) t ⟷ (c = 0 ∧ d = 0) ∨ (c = d ∧ s = t)
  by (fastforce simp add: monomial-inj Poly-Mapping.single-def
    poly-mapping.Abs-poly-mapping-inject when-def fun-eq-iff
    cong: if-cong
    split: if-splits)

lemma MPoly-monomial-power':
  ‹MPoly (monomial 1 x') ^(n+1) =  MPoly (monomial (1) (((λx. x + x') ^^ n) x'))›
  by (induction n)
   (auto simp: times-mpoly.abs-eq mult-single ac-simps)

lemma MPoly-monomial-power:
  ‹n > 0 ⟹ MPoly (monomial 1 x') ^(n) =  MPoly (monomial (1) (((λx. x + x') ^^ (n - 1)) x'))›
  using MPoly-monomial-power'[of - ‹n-1›]
  by auto


lemma vars-uminus[simp]:
  ‹vars (-p) = vars p›
  by (auto simp: vars-def uminus-mpoly.rep-eq)

lemma coeff-uminus[simp]:
```

‹*MPoly-Type.coeff* (−*p*) *x* = −*MPoly-Type.coeff p x*›
by (*auto simp*: *coeff-def uminus-mpoly.rep-eq*)

definition *decrease-key*::*'a* ⇒ (*'a* ⇒$_0$ *'b*::{*monoid-add*, *minus*,*one*}) ⇒ (*'a* ⇒$_0$ *'b*) **where**
decrease-key k0 f = *Abs-poly-mapping* (λ*k*. *if k* = *k0* ∧ *lookup f k* ≠ *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*)
unfolding *decrease-key-def* **using** *finite-subset* **apply** *simp*
apply (*subst lookup-Abs-poly-mapping*)
apply (*auto intro*: *finite-subset*[*of* - ‹{*x*. *lookup f x* ≠ *0*}›])
apply (*subst lookup-Abs-poly-mapping*)
apply (*auto intro*: *finite-subset*[*of* - ‹{*x*. *lookup f x* ≠ *0*}›])
done

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 −
have [*simp*]: ‹{*x* ∈ *keys* (*mapping-of p*). *x'* ∈ *keys x*} ∪
{*x* ∈ *keys* (*mapping-of p*). *x'* ∉ *keys x*} = *keys* (*mapping-of p*)›
by *auto*
have
‹*p* = (∑ *x*∈*keys* (*mapping-of p*). *MPoly-Type.monom x* (*MPoly-Type.coeff p x*))› (**is** ‹- = (∑ *x* ∈ *?I*. *?f x*)›)
using *polynomial-sum-monoms*(*1*)[*of p*] **.**
also have ‹... = (∑ *x*∈ {*x* ∈ *?I*. *x'* ∈ *keys x*}. *?f x*) + (∑ *x*∈ {*x* ∈ *?I*. *x'* ∉ *keys x*}. *?f x*)› (**is** ‹- = *?pX* + *?qX*›)
by (*subst comm-monoid-add-class.sum.union-disjoint*[*symmetric*]) *auto*
finally have *1*: ‹*p* = *?pX* + *?qX*› **.**
have *H*: ‹*0* < *lookup x x'* ⟹ (λ*k*. (*if x'* = *k then Suc 0 else 0*) +
(*if k* = *x'* ∧ *0* < *lookup x k then lookup x k* − *1*
else lookup x k)) = *lookup x*› **for** *x x'*
by *auto*
have [*simp*]: ‹*finite* {*x*. *0* < (*Suc 0 when x'* = *x*)}› **for** *x'* :: *nat* **and** *x*
by (*smt* (*verit*) *bounded-nat-set-is-finite lessI mem-Collect-eq neq0-conv when-cong when-neq-zero*)
have *H*: ‹*x'* ∈ *keys x* ⟹ *monomial* (*Suc 0*) *x'* + *Abs-poly-mapping* (λ*k*. *if k* = *x'* ∧ *0* < *lookup x k then lookup x k* − *1 else lookup x k*) = *x*›
for *x* **and** *x'* :: *nat*
apply (*simp only*: *keys-def single.abs-eq*)
apply (*subst plus-poly-mapping.abs-eq*)
by (*auto simp*: *eq-onp-def when-def H*
intro!: *finite-subset*[*of* ‹{*xa*. (*xa* = *x'* ∧ *0* < *lookup x xa* ⟶ *Suc 0* < *lookup x x'*) ∧
(*xa* ≠ *x'* ⟶ *0* < *lookup x xa*)}› ‹{*xa*. *0* < *lookup x xa*}›])

have [*simp*]: ‹*x'* ∈ *keys x* ⟹
MPoly-Type.monom (*monomial* (*Suc 0*) *x'* + *decrease-key x' x*) *n* =
MPoly-Type.monom x n› **for** *x n* **and** *x'*
apply (*subst mpoly.mapping-of-inject*[*symmetric*], *subst poly-mapping.lookup-inject*[*symmetric*])
unfolding *mapping-of-monom lookup-single*
apply (*auto intro*!: *ext simp*: *decrease-key-def when-def H*)
done

```
  have pX: ‹?pX = monom (monomial (Suc 0) x') 1 * (∑ x∈ {x ∈ ?I. x' ∈ keys x}. MPoly-Type.monom
(decrease-key x' x) (MPoly-Type.coeff p x))›
    (is ‹- = - * ?pX'›)
    by (subst sum-distrib-left, subst mult-monom)
     (auto intro!: sum.cong)
  have ‹x' ∉ vars ?qX›
    using vars-setsum[of ‹{x. x ∈ keys (mapping-of p) ∧ x' ∉ keys x}› ‹?f›]
    by (auto dest!: vars-monom-subset[unfolded subset-eq Ball-def, rule-format])
  then show ?thesis
    using that[of ?pX' ?qX]
    unfolding pX[symmetric] 1[symmetric]
    by blast
qed


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 −
  have eq[simp]: ‹monom (monomial (Suc 0) x') 1 = Var x'›
    by (simp add: Var.abs-eq Var_0-def monom.abs-eq)
  have ‹∀ m ≤ n. ∀ P::int mpoly. degree P x' < m ⟶ (∃ A B. P = (Var x' − s) * A + B ∧ x' ∉ vars
B)› for n
  proof (induction n)
    case 0
    then show ?case by auto
  next
    case (Suc n)
    then have IH: ‹m≤n ⟹ MPoly-Type.degree P x' < m ⟹
            (∃ A B. P = (Var x' − s) * A + B ∧ x' ∉ vars B)› for m P
      by fast
    show ?case
    proof (intro allI impI)
     fix m and P :: ‹int mpoly›
     assume ‹m ≤ Suc n› and deg: ‹MPoly-Type.degree P x' < m›
     consider
       ‹m ≤ n› |
       ‹m = Suc n›
       using ‹m ≤ Suc n› by linarith
     then show ‹∃ A B. P = (Var x' − s) * A + B ∧ x' ∉ vars B›
     proof cases
       case 1
       then show ‹?thesis›
         using Suc deg by blast
     next
       case [simp]: 2
       obtain A B where
         P: ‹P = Var x' * A + B› and
         ‹x' ∉ vars B›
         using polynomial-split-on-var[of P x'] unfolding eq by blast
       have P': ‹P = (Var x' − s) * A + (s * A + B)›
         by (auto simp: field-simps P)
```

```
      have ‹A = 0 ∨ degree (s * A) x' < degree P x'›
        using deg ‹x' ∉ vars B› ‹x' ∉ vars s› degree-times-le[of s A x'] deg
        unfolding P
        by (auto simp: degree-sum-notin degree-mult-Var' degree-mult-Var degree-notin-vars
          split: if-splits)
      then obtain A' B' where
        sA: ‹s*A = (Var x' - s) * A' + B'› and
        ‹x' ∉ vars B'›
        using IH[of ‹m-1› ‹s*A›] deg ‹x' ∉ vars B› that[of 0 0]
        by (cases ‹0 < n›) (auto dest!: vars-in-right-only)
      have ‹P = (Var x' - s) * (A + A') + (B' + B)›
        unfolding P' sA by (auto simp: field-simps)
      moreover have ‹x' ∉ vars (B' + B)›
        using ‹x' ∉ vars B'› ‹x' ∉ vars B›
        by (meson UnE subset-iff vars-add)
      ultimately show ?thesis
        by fast
    qed
  qed
 qed
 then show ?thesis
   using that unfolding eq
   by blast
qed

lemma finit-whenI[intro]:
 ‹finite {x. (0 :: nat) < (y x)} ⟹ finite {x. 0 < (y x when x ≠ x')}›
 apply (rule finite-subset)
  defer apply assumption
 apply (auto simp: when-def)
 done

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 -
 let ?v = ‹monom (monomial (Suc 0) x') 1 :: int mpoly›
 have H: ‹n < m ⟹ n > 0 ⟹ ∃ q. ?v^n = ?v + q * (?v^2 - ?v)› for n m :: nat
 proof (induction m arbitrary: n)
   case 0
   then show ?case by auto
 next
   case (Suc m n) note IH = this(1-)
   consider
     ‹n < m› |
     ‹m = n› ‹n > 1› |
     ‹n = 1›
     using IH
     by (cases ‹n < m›; cases n) auto
   then show ?case
   proof cases
```

```
    case 1
    then show ?thesis using IH by auto
  next
    case 2
    have eq: ‹?v^(n) = ((?v :: int mpoly) ^ (n−2)) * (?v^2−?v) + ?v^(n−1)›
      using 2 by (auto simp: field-simps power-eq-if
        ideal.scale-right-diff-distrib)
    obtain q where
      q: ‹?v^(n−1) = ?v + q * (?v^2 − ?v)›
      using IH(1)[of ‹n−1›] 2
      by auto
    show ?thesis
      using q unfolding eq
      by (auto intro!: exI[of - ‹?v ^ (n − 2) + q›] simp: distrib-right)
  next
    case 3
    then show ‹?thesis›
      by auto
  qed
qed
have H: ‹n>0 ⟹ ∃ q. ?v^n = ?v + q * (?v^2−?v)› for n
  using H[of n ‹n+1›]
  by auto
obtain qr :: ‹nat ⇒ int mpoly› where
   qr: ‹n > 0 ⟹ ?v^n = ?v + qr n * (?v^2−?v)› for n
 using H by metis
have vn: ‹(if lookup x x' = 0 then 1 else Var x' ^ lookup x x') =
  (if lookup x x' = 0 then 1 else ?v) + (if lookup x x' = 0 then 0 else 1) * qr (lookup x x') * (?v^2−?v)›
for x
  by (simp add: qr[symmetric] Var-def Var_0-def monom.abs-eq[symmetric] cong: if-cong)

have q: ‹p = (∑ x∈keys (mapping-of p). MPoly-Type.monom x (MPoly-Type.coeff p x))›
  by (rule polynomial-sum-monoms(1)[of p])
have [simp]:
  ‹lookup x x' = 0 ⟹
  Abs-poly-mapping (λk. lookup x k when k ≠ x') = x› for x
  by (cases x, auto simp: poly-mapping.Abs-poly-mapping-inject)
    (auto intro!: ext simp: when-def)
have [simp]: ‹finite {x. 0 < (a when x' = x)}› for a :: nat
  by (metis (no-types, lifting) infinite-nat-iff-unbounded less-not-refl lookup-single lookup-single-not-eq
mem-Collect-eq)

have [simp]: ‹((λx. x + monomial (Suc 0) x') ^^ (n))
   (monomial (Suc 0) x') = Abs-poly-mapping (λk. (if k = x' then n+1 else 0))› for n
  by (induction n)
   (auto simp: single-def Abs-poly-mapping-inject plus-poly-mapping.abs-eq eq-onp-def cong:if-cong)
have [simp]: ‹0 < lookup x x' ⟹
  Abs-poly-mapping (λk. lookup x k when k ≠ x') +
  Abs-poly-mapping (λk. if k = x' then lookup x x' − Suc 0 + 1 else 0) =
  x› for x
 apply (cases x, auto simp: poly-mapping.Abs-poly-mapping-inject plus-poly-mapping.abs-eq eq-onp-def)
  apply (subst plus-poly-mapping.abs-eq)
  apply (auto simp: poly-mapping.Abs-poly-mapping-inject plus-poly-mapping.abs-eq eq-onp-def)
  apply (subst Abs-poly-mapping-inject)
  apply auto
```

done
define f **where**
$\langle f\ x = (MPoly\text{-}Type.monom\ (remove\text{-}key\ x'\ x)\ (MPoly\text{-}Type.coeff\ p\ x)) *$
$(if\ lookup\ x\ x' = 0\ then\ 1\ else\ Var\ x'\ \widehat{}\ (lookup\ x\ x'))\rangle$ **for** x
have *f-alt-def*: $\langle f\ x = MPoly\text{-}Type.monom\ x\ (MPoly\text{-}Type.coeff\ p\ x)\rangle$ **for** x
by (*auto simp*: *f-def monom-def remove-key-def Var-def MPoly-monomial-power* Var_0*-def*
mpoly.MPoly-inject monomial-inj times-mpoly.abs-eq
times-mpoly.abs-eq mult-single)
have p: $\langle p = (\sum x{\in}keys\ (mapping\text{-}of\ p).$
$MPoly\text{-}Type.monom\ (remove\text{-}key\ x'\ x)\ (MPoly\text{-}Type.coeff\ p\ x) *$
$(if\ lookup\ x\ x' = 0\ then\ 1\ else\ ?v)) +$
$(\sum x{\in}keys\ (mapping\text{-}of\ p).$
$MPoly\text{-}Type.monom\ (remove\text{-}key\ x'\ x)\ (MPoly\text{-}Type.coeff\ p\ x) *$
$(if\ lookup\ x\ x' = 0\ then\ 0$
$else\ 1) * qr\ (lookup\ x\ x')) *$
$(?v^2 - ?v)\rangle$
(**is** $\langle$- $= ?a + ?v2v\rangle$)
apply (*subst q*)
unfolding *f-alt-def*[*symmetric*, *abs-def*] *f-def vn semiring-class.distrib-left*
comm-semiring-1-class.semiring-normalization-rules(18) *semiring-0-class.sum-distrib-right*
by (*simp add*: *semiring-class.distrib-left*
sum.distrib)

have I: $\langle keys\ (mapping\text{-}of\ p) = \{x \in keys\ (mapping\text{-}of\ p).\ lookup\ x\ x' = 0\} \cup \{x \in keys\ (mapping\text{-}of\ p).\ lookup\ x\ x' \neq 0\}\rangle$
by *auto*

have $\langle p = (\sum x \mid x \in keys\ (mapping\text{-}of\ p) \wedge lookup\ x\ x' = 0.$
$MPoly\text{-}Type.monom\ x\ (MPoly\text{-}Type.coeff\ p\ x)) +$
$(\sum x \mid x \in keys\ (mapping\text{-}of\ p) \wedge lookup\ x\ x' \neq 0.$
$MPoly\text{-}Type.monom\ (remove\text{-}key\ x'\ x)\ (MPoly\text{-}Type.coeff\ p\ x)) *$
$(MPoly\text{-}Type.monom\ (monomial\ (Suc\ 0)\ x')\ 1) +$
$(\sum x \mid x \in keys\ (mapping\text{-}of\ p) \wedge lookup\ x\ x' \neq 0.$
$MPoly\text{-}Type.monom\ (remove\text{-}key\ x'\ x)\ (MPoly\text{-}Type.coeff\ p\ x) *$
$qr\ (lookup\ x\ x')) *$
$(?v^2 - ?v)\rangle$
(**is** $\langle p = ?A + ?B * - + ?C * -\rangle$)
unfolding *semiring-0-class.sum-distrib-right*[*of - -* $\langle(MPoly\text{-}Type.monom\ (monomial\ (Suc\ 0)\ x')\ 1)\rangle$]
apply (*subst p*)
apply (*subst* (2)*I*)
apply (*subst I*)
apply (*subst comm-monoid-add-class.sum.union-disjoint*)
apply *auto*[3]
apply (*subst comm-monoid-add-class.sum.union-disjoint*)
apply *auto*[3]
apply (*subst* (4) *sum.cong*[*OF refl*, *of - -* $\langle\lambda x.\ MPoly\text{-}Type.monom\ (remove\text{-}key\ x'\ x)\ (MPoly\text{-}Type.coeff\ p\ x) *$
$qr\ (lookup\ x\ x')\rangle$])
apply (*auto*; *fail*)
apply (*subst* (3) *sum.cong*[*OF refl*, *of - -* $\langle\lambda x.\ 0\rangle$])
apply (*auto*; *fail*)
apply (*subst* (2) *sum.cong*[*OF refl*, *of - -* $\langle\lambda x.\ MPoly\text{-}Type.monom\ (remove\text{-}key\ x'\ x)\ (MPoly\text{-}Type.coeff\ p\ x) *$
$(MPoly\text{-}Type.monom\ (monomial\ (Suc\ 0)\ x')\ 1)\rangle$])
apply (*auto*; *fail*)

```
    apply (subst (1) sum.cong[OF refl, of - - ‹λx. MPoly-Type.monom x (MPoly-Type.coeff p x)›])
    by (auto simp: f-def simp flip: f-alt-def)

  moreover have ‹x' ∉ vars ?A›
    using vars-setsum[of ‹{x ∈ keys (mapping-of p). lookup x x' = 0}›
      ‹λx. MPoly-Type.monom x (MPoly-Type.coeff p x)›]
    apply auto
    apply (drule set-rev-mp, assumption)
    apply (auto dest!: lookup-eq-zero-in-keys-contradict)
    by (meson lookup-eq-zero-in-keys-contradict subsetD vars-monom-subset)
  moreover have ‹x' ∉ vars ?B›
    using vars-setsum[of ‹{x ∈ keys (mapping-of p). lookup x x' ≠ 0}›
      ‹λx. MPoly-Type.monom (remove-key x' x) (MPoly-Type.coeff p x)›]
    apply auto
    apply (drule set-rev-mp, assumption)
    apply (auto dest!: lookup-eq-zero-in-keys-contradict)
    apply (drule subsetD[OF vars-monom-subset])
    apply (auto simp: remove-key-keys[symmetric])
    done
  ultimately show ?thesis apply −
    apply (rule that[of ?B ?A ?C])
    apply (auto simp: ac-simps)
    done
qed


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 −
  let ?A = ‹A * (monom (monomial (Suc 0) x') 1)›
  have ‹?A = q − b›
    using arg-cong[OF q, of ‹λa. a − b›]
    by auto
  moreover have ‹x' ∉ vars (q − b)›
    using x vars-in-right-only
    by fastforce
  ultimately have ‹x' ∉ vars (?A)›
    by simp
  then have ‹?A = 0›
    by (auto simp: vars-mult-monom split: if-splits)
  moreover have ‹?A = 0 ⟹ A = 0›
    by (metis empty-not-insert mult-zero-left vars-mult-monom)
  ultimately show ‹A = 0›
    by blast
  then show ‹q = b›
    using q by auto
qed

lemma polynomial-decomp-alien-var2:
```

fixes $q\ A\ b :: \langle int\ mpoly \rangle$
assumes
q: $\langle q = A * (monom\ (monomial\ (Suc\ 0)\ x')\ 1 + p) + b \rangle$ **and**
x: $\langle x' \notin vars\ q \rangle\ \langle x' \notin vars\ b \rangle\ \langle x' \notin vars\ p \rangle$
shows
$\langle A = 0 \rangle$ **and**
$\langle q = b \rangle$
proof −
let $?x = \langle monom\ (monomial\ (Suc\ 0)\ x')\ 1 \rangle$
have $x'[simp]$: $\langle ?x = Var\ x' \rangle$
by (*simp add*: *Var.abs-eq Var*$_0$*-def monom.abs-eq*)
have $\langle \exists n\ Ax\ A'.\ A = ?x * Ax + A' \wedge x' \notin vars\ A' \wedge degree\ Ax\ x' = n \rangle$
using *polynomial-split-on-var*[*of A x'*] **by** *metis*
from *wellorder-class.exists-least-iff*[*THEN iffD1*, *OF this*] **obtain** $Ax\ A'\ n$ **where**
A: $\langle A = Ax * ?x + A' \rangle$ **and**
$\langle x' \notin vars\ A' \rangle$ **and**
n: $\langle MPoly\text{-}Type.degree\ Ax\ x' = n \rangle$ **and**
H: $\langle \bigwedge m\ Ax\ A'.\ m < n \longrightarrow$
$A \neq Ax * MPoly\text{-}Type.monom\ (monomial\ (Suc\ 0)\ x')\ 1 + A' \vee$
$x' \in vars\ A' \vee MPoly\text{-}Type.degree\ Ax\ x' \neq m \rangle$
unfolding *wellorder-class.exists-least-iff*[*of* $\langle \lambda n.\ \exists Ax\ A'.\ A = Ax * ?x + A' \wedge x' \notin vars\ A' \wedge$
degree Ax x' $= n \rangle$]
by (*auto simp*: *field-simps*)

have $\langle q = (A + Ax * p) * monom\ (monomial\ (Suc\ 0)\ x')\ 1 + (p * A' + b) \rangle$
unfolding *q A* **by** (*auto simp*: *field-simps*)
moreover have $\langle x' \notin vars\ q \rangle\ \langle x' \notin vars\ (p * A' + b) \rangle$
using *x* $\langle x' \notin vars\ A' \rangle$
by (*smt* (*verit*) *UnE add.assoc add.commute calculation subset-iff vars-in-right-only vars-mult*)+
ultimately have $\langle A + Ax * p = 0 \rangle\ \langle q = p * A' + b \rangle$
by (*rule polynomial-decomp-alien-var*)+

have A': $\langle A' = -Ax * ?x - Ax * p \rangle$
using $\langle A + Ax * p = 0 \rangle$ **unfolding** *A*
by (*metis* (*no-types*, *lifting*) *add-uminus-conv-diff eq-neg-iff-add-eq-0 minus-add-cancel*
mult-minus-left)

have $\langle A = -\ (Ax * p) \rangle$
using *A* **unfolding** *A'*
apply *auto*
done

obtain *Axx Ax'* **where**
Ax: $\langle Ax = ?x * Axx + Ax' \rangle$ **and**
$\langle x' \notin vars\ Ax' \rangle$
using *polynomial-split-on-var*[*of Ax x'*] **by** *metis*

have $\langle A = ?x * (-\ Axx * p) + (-\ Ax' * p) \rangle$
unfolding $\langle A = -\ (Ax * p) \rangle$ *Ax*
by (*auto simp*: *field-simps*)

moreover have $\langle x' \notin vars\ (-Ax' * p) \rangle$
using $\langle x' \notin vars\ Ax' \rangle$ **by** (*metis* (*no-types*, *opaque-lifting*) *UnE add.right-neutral*
add-minus-cancel assms(4) *subsetD vars-in-right-only vars-mult*)
moreover have $\langle Axx \neq 0 \Longrightarrow MPoly\text{-}Type.degree\ (-\ Axx * p)\ x' < degree\ Ax\ x' \rangle$

```
    using degree-times-le[of Axx p x'] x
    by (auto simp: Ax degree-sum-notin ‹x' ∉ vars Ax'› degree-mult-Var'
      degree-notin-vars)
  ultimately have [simp]: ‹Axx = 0›
    using H[of ‹MPoly-Type.degree (− Axx * p) x'› ‹− Axx * p› ‹− Ax' * p›]
    by (auto simp: n)
 then have [simp]: ‹Ax' = Ax›
  using Ax by auto

 show ‹A = 0›
  using A ‹A = − (Ax * p)› ‹x' ∉ vars (− Ax' * p)› ‹x' ∉ vars A'› polynomial-decomp-alien-var(1)
by force
 then show ‹q = b›
  using q by auto
qed

lemma vars-unE: ‹x ∈ vars (a * b) ⟹ (x ∈ vars a ⟹ thesis) ⟹ (x ∈ vars b ⟹ thesis) ⟹ thesis›
  using vars-mult[of a b] by auto


lemma in-keys-minusI1:
 assumes t ∈ keys p and t ∉ keys q
 shows t ∈ keys (p − q)
 using assms unfolding in-keys-iff lookup-minus by simp

lemma in-keys-minusI2:
 fixes t :: ‹'a› and q :: ‹'a ⇒₀ 'b :: {cancel-comm-monoid-add,group-add}›
 assumes t ∈ keys q and t ∉ keys p
 shows t ∈ keys (p − q)
 using assms unfolding in-keys-iff lookup-minus by simp


lemma in-vars-addE:
 ‹x ∈ vars (p + q) ⟹ (x ∈ vars p ⟹ thesis) ⟹ (x ∈ vars q ⟹ thesis) ⟹ thesis›
 by (meson UnE in-mono vars-add)

lemma lookup-monomial-If:
 ‹lookup (monomial v k) = (λk'. if k = k' then v else 0)›
 by (intro ext) (auto simp: lookup-single-not-eq)

lemma vars-mult-Var:
 ‹vars (Var x * p) = (if p = 0 then {} else insert x (vars p))› for p :: ‹int mpoly›
proof −
 have ‹p ≠ 0 ⟹
   ∃xa. (∃k. xa = monomial (Suc 0) x + k) ∧
        lookup (mapping-of p) (xa − monomial (Suc 0) x) ≠ 0 ∧
        0 < lookup xa x›
  by (metis (no-types, opaque-lifting) One-nat-def ab-semigroup-add-class.add.commute
    add-diff-cancel-right' aux lookup-add lookup-single-eq mapping-of-inject
    neq0-conv one-neq-zero plus-eq-zero-2 zero-mpoly.rep-eq)
 then show ?thesis
   apply (auto simp: vars-def times-mpoly.rep-eq Var.rep-eq
   elim!: in-keys-timesE dest: keys-add')
   apply (auto simp: keys-def lookup-times-monomial-left Var.rep-eq Var0-def adds-def
     lookup-add eq-diff-eq'[symmetric])
```

```
    done
qed

lemma keys-mult-monomial:
  ‹keys (monomial (n :: int) k * mapping-of a) = (if n = 0 then {} else ((+) k) ` keys (mapping-of a))›
proof −
  have [simp]: ‹(∑ aa. (if k = aa then n else 0) *
              (∑ q. lookup (mapping-of a) q when k + xa = aa + q)) =
        (∑ aa. (if k = aa then n * (∑ q. lookup (mapping-of a) q when k + xa = aa + q) else 0))›
    for xa
    by (smt (verit) Sum-any.cong mult-not-zero)
  show ?thesis
    apply (auto simp: vars-def times-mpoly.rep-eq Const.rep-eq times-poly-mapping.rep-eq
      Const0-def elim!: in-keys-timesE split: if-splits)
    apply (auto simp: lookup-monomial-If prod-fun-def
      keys-def times-poly-mapping.rep-eq)
    done
qed

lemma vars-mult-Const:
  ‹vars (Const n * a) = (if n = 0 then {} else vars a)› for a :: ‹int mpoly›
  by (auto simp: vars-def times-mpoly.rep-eq Const.rep-eq keys-mult-monomial
    Const0-def elim!: in-keys-timesE split: if-splits)

lemma coeff-minus: coeff p m − coeff q m = coeff (p−q) m
  by (simp add: coeff-def lookup-minus minus-mpoly.rep-eq)

lemma Const-1-eq-1: ‹Const (1 :: int) = (1 :: int mpoly)›
  by (simp add: Const.abs-eq Const0-one one-mpoly.abs-eq)

lemma [simp]:
  ‹vars (1 :: int mpoly) = {}›
  by (auto simp: vars-def one-mpoly.rep-eq Const-1-eq-1)
```

2.2 More Ideals

```
lemma
  fixes A :: ‹(('x ⇒0 nat) ⇒0 'a::comm-ring-1) set›
  assumes ‹p ∈ ideal A›
  shows ‹p * q ∈ ideal A›
  by (metis assms ideal.span-scale semiring-normalization-rules(7))
```

The following theorem is very close to *More-Modules.ideal* (*insert ?a ?S*) = {*x*. ∃ *k*. *x* − *k* * *?a* ∈ *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. ∃ x k. y = x + k * a ∧ x ∈ More-Modules.ideal S}›
    apply (auto simp: ideal.span-insert
      intro: exI[of - ‹- − k * a›])
   apply (rule-tac x = ‹x − k * a› in exI)
   apply auto
   apply (rule-tac x = ‹k› in exI)
   apply auto
   done
```

lemma *ideal-mult-right-in*:
‹$a \in$ *ideal* $A \Longrightarrow a * b \in$ *More-Modules.ideal* A›
by (*metis ideal.span-scale mult.commute*)

lemma *ideal-mult-right-in2*:
‹$a \in$ *ideal* $A \Longrightarrow b * a \in$ *More-Modules.ideal* A›
by (*metis ideal.span-scale*)

lemma [*simp*]: ‹*vars* (*Var* $x :: 'a :: \{$*zero-neq-one*$\}$ *mpoly*) $= \{x\}$›
by (*auto simp*: *vars-def Var.rep-eq* Var_0*-def*)

lemma *vars-minus-Var-subset*:
‹*vars* ($p' -$ *Var* $x :: 'a :: \{$*ab-group-add*,*one*,*zero-neq-one*$\}$ *mpoly*) $\subseteq \mathcal{V} \Longrightarrow$ *vars* $p' \subseteq$ *insert* $x\ \mathcal{V}$›
using *vars-add*[*of* ‹$p' -$ *Var* x› ‹*Var* x›]
by *auto*

lemma *vars-add-Var-subset*:
‹*vars* ($p' +$ *Var* $x :: 'a :: \{$*ab-group-add*,*one*,*zero-neq-one*$\}$ *mpoly*) $\subseteq \mathcal{V} \Longrightarrow$ *vars* $p' \subseteq$ *insert* $x\ \mathcal{V}$›
using *vars-add*[*of* ‹$p' +$ *Var* x› ‹$-$*Var* x›]
by *auto*

lemma *coeff-monomila-in-varsD*:
‹*coeff* p (*monomial* (*Suc 0*) x) $\neq 0 \Longrightarrow x \in$ *vars* ($p ::$ *int mpoly*)›
by (*auto simp*: *coeff-def vars-def keys-def*
intro!: *exI*[*of* - ‹*monomial* (*Suc 0*) x›])

lemma *coeff-MPoly-monomial*[*simp*]:
‹(*MPoly-Type.coeff* (*MPoly* (*monomial a m*)) m) $= a$›
by (*metis MPoly-Type.coeff-def lookup-single-eq monom.abs-eq monom.rep-eq*)

end

theory *Finite-Map-Multiset*
imports
HOL−Library.Finite-Map
Nested-Multisets-Ordinals.Duplicate-Free-Multiset
begin

notation *image-mset* (**infixr** ‹‘#› *90*)

3 Finite maps and multisets

3.1 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) $= N$›
by (*auto simp*: *fset.fset-inverse fset-mset-def*)

lemma *mset-fset-fset-mset*[*simp*]:

‹*mset-fset* (*fset-mset N*) = *remdups-mset N*›
by (*auto simp*: *fset.fset-inverse fset-mset-def Abs-fset-inverse remdups-mset-def*)

lemma *in-mset-fset-fmember*[*simp*]: ‹*x* ∈# *mset-fset N* ⟷ *x* |∈| *N*›
by *simp*

lemma *in-fset-mset-mset*[*simp*]: ‹*x* |∈| *fset-mset N* ⟷ *x* ∈# *N*›
by (*simp add*: *fset-mset-def Abs-fset-inverse*)

3.2 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*)›
unfolding *dom-m-def*
by (*cases* ‹*C* |∈| *fmdom N*›)
(*auto simp*: *mset-set.remove*)

lemma *dom-m-fmdrop-All*: ‹*dom-m* (*fmdrop C N*) = *removeAll-mset C* (*dom-m N*)›
unfolding *dom-m-def*
by (*cases* ‹*C* |∈| *fmdom N*›)
(*auto simp*: *mset-set.remove*)

lemma *dom-m-fmupd*[*simp*]: ‹*dom-m* (*fmupd k C N*) = *add-mset k* (*remove1-mset k* (*dom-m N*))›
unfolding *dom-m-def*
by (*cases* ‹*k* |∈| *fmdom N*›)
(*auto simp*: *mset-set.remove mset-set.insert-remove*)

lemma *distinct-mset-dom*: ‹*distinct-mset* (*dom-m N*)›
by (*simp add*: *distinct-mset-mset-set dom-m-def*)

lemma *in-dom-m-lookup-iff*: ‹*C* ∈# *dom-m N'* ⟷ *fmlookup N' C* ≠ *None*›
by (*auto simp*: *dom-m-def fmdom.rep-eq fmlookup-dom'-iff*)

lemma *in-dom-in-ran-m*[*simp*]: ‹*i* ∈# *dom-m N* ⟹ *the* (*fmlookup N i*) ∈# *ran-m N*›
by (*auto simp*: *ran-m-def*)

lemma *fmupd-same*[*simp*]:
‹*x1* ∈# *dom-m x1aa* ⟹ *fmupd x1* (*the* (*fmlookup x1aa x1*)) *x1aa* = *x1aa*›
by (*metis fmap-ext fmupd-lookup in-dom-m-lookup-iff option.collapse*)

lemma *ran-m-fmempty*[*simp*]: ‹*ran-m fmempty* = {#}› **and**
dom-m-fmempty[*simp*]: ‹*dom-m fmempty* = {#}›
by (*auto simp*: *ran-m-def dom-m-def*)

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

‹$a \notin xs \Longrightarrow$ *fmrestrict-set xs* (*fmupd a C N*) = *fmrestrict-set xs N*›
by (*auto simp*: *fmfilter-alt-defs*)

lemma *fset-fmdom-fmrestrict-set*:
‹*fset* (*fmdom* (*fmrestrict-set xs N*)) = *fset* (*fmdom N*) ∩ *xs*›
by (*auto simp*: *fmfilter-alt-defs*)

lemma *dom-m-fmrestrict-set*: ‹*dom-m* (*fmrestrict-set* (*set xs*) *N*) = *mset xs* ∩# *dom-m N*›
using *fset-fmdom-fmrestrict-set*[*of* ‹*set xs*› *N*] *distinct-mset-dom*[*of N*]
distinct-mset-inter-remdups-mset[*of* ‹*mset-fset* (*fmdom N*)› ‹*mset xs*›]
by (*auto simp*: *dom-m-def fset-mset-mset-fset finite-mset-set-inter multiset-inter-commute*
remdups-mset-def)

lemma *dom-m-fmrestrict-set'*: ‹*dom-m* (*fmrestrict-set xs N*) = *mset-set* (*xs* ∩ *set-mset* (*dom-m N*))›
using *fset-fmdom-fmrestrict-set*[*of* ‹*xs*› *N*] *distinct-mset-dom*[*of N*]
by (*auto simp*: *dom-m-def fset-mset-mset-fset finite-mset-set-inter multiset-inter-commute*
remdups-mset-def)

lemma *indom-mI*: ‹*fmlookup m x* = *Some y* ⟹ *x* ∈# *dom-m m*›
by (*drule fmdomI*) (*auto simp*: *dom-m-def*)

lemma *fmupd-fmdrop-id*:
assumes ‹*k* |∈| *fmdom N'*›
shows ‹*fmupd k* (*the* (*fmlookup N' k*)) (*fmdrop k N'*) = *N'*›
proof −
have [*simp*]: ‹*map-upd k* (*the* (*fmlookup N' k*))
(λ*x*. *if x* ≠ *k then fmlookup N' x else None*) =
map-upd k (*the* (*fmlookup N' k*))
(*fmlookup N'*)›
by (*auto intro*!: *ext simp*: *map-upd-def*)
have [*simp*]: ‹*map-upd k* (*the* (*fmlookup N' k*)) (*fmlookup N'*) = *fmlookup N'*›
using *assms*
by (*auto intro*!: *ext simp*: *map-upd-def*)
have [*simp*]: ‹*finite* (*dom* (λ*x*. *if x* = *k then None else fmlookup N' x*))›
by (*subst dom-if*) *auto*
show *?thesis*
apply (*auto simp*: *fmupd-def fmupd.abs-eq*[*symmetric*])
unfolding *fmlookup-drop*
apply (*simp add*: *fmlookup-inverse*)
done
qed

lemma *fm-member-split*: ‹*k* |∈| *fmdom N'* ⟹ ∃ *N''* *v*. *N'* = *fmupd k v N''* ∧ *the* (*fmlookup N' k*) = *v*
∧
k |∉| *fmdom N''*›
by (*rule exI*[*of* - ‹*fmdrop k N'*›])
(*auto simp*: *fmupd-fmdrop-id*)

lemma ‹*fmdrop k* (*fmupd k va N''*) = *fmdrop k N''*›
by (*simp add*: *fmap-ext*)

lemma *fmap-ext-fmdom*:
‹(*fmdom N* = *fmdom N'*) ⟹ (⋀ *x*. *x* |∈| *fmdom N* ⟹ *fmlookup N x* = *fmlookup N' x*) ⟹
N = *N'*›
by (*rule fmap-ext*)

(*case-tac* ‹*x* |∈| *fmdom N*›, *auto simp*: *fmdom-notD*)

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*)›
apply (*rule fmap-ext-fmdom*)
apply (*auto simp*: *fset-fmdom-fmrestrict-set*; *fail*)[]
apply (*auto simp*: *fmlookup-dom-iff*; *fail*)
done

lemma *fmrestrict-set-insert-notin*:
‹*xa* ∉ *fset* (*fmdom N*) ⟹
fmrestrict-set (*insert xa l1*) *N* = *fmrestrict-set l1 N*›
by (*rule fmap-ext-fmdom*)
(*auto simp*: *fset-fmdom-fmrestrict-set*)

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*)›
by (*simp add*: *fmrestrict-set-insert-in dom-m-def*)

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

lemma *fmlookup-restrict-set-id*: ‹*fset* (*fmdom N*) ⊆ *A* ⟹ *fmrestrict-set A N* = *N*›
by (*metis fmap-ext fmdom'-alt-def fmdom'-notD fmlookup-restrict-set subset-iff*)

lemma *fmlookup-restrict-set-id'*: ‹*set-mset* (*dom-m N*) ⊆ *A* ⟹ *fmrestrict-set A N* = *N*›
by (*rule fmlookup-restrict-set-id*)
(*auto simp*: *dom-m-def*)

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 −
define *N'* **where**
‹*N'* = *fmdrop C N*›
have *N-N'*: ‹*dom-m N* = *add-mset C* (*dom-m N'*)›
using *NC* **unfolding** *N'-def* **by** *auto*
have ‹*C* ∉# *dom-m N'*›
using *NC distinct-mset-dom*[*of N*] **unfolding** *N-N'* **by** *auto*
then show *?thesis*
by (*auto simp*: *N-N' ran-m-def mset-set.insert-remove image-mset-remove1-mset-if*
intro!: *image-mset-cong*)
qed

lemma *ran-m-mapsto-upd-notin*:
assumes *NC*: ‹*C* ∉# *dom-m N*›
shows ‹*ran-m* (*fmupd C C' N*) = *add-mset C'* (*ran-m N*)›
using *NC*
by (*auto simp*: *ran-m-def mset-set.insert-remove image-mset-remove1-mset-if*
intro!: *image-mset-cong split*: *if-splits*)

lemma *image-mset-If-eq-notin*:
‹$C \notin\# A \Longrightarrow \{\#f\ (\mathit{if}\ x = C\ \mathit{then}\ a\ x\ \mathit{else}\ b\ x).\ x \in\# A\#\} = \{\#\ f(b\ x).\ x \in\# A\ \#\}$›
by (*induction A*) *auto*

lemma *filter-mset-cong2*:
$(\bigwedge x.\ x \in\# M \Longrightarrow f\ x = g\ x) \Longrightarrow M = N \Longrightarrow \mathit{filter\text{-}mset}\ f\ M = \mathit{filter\text{-}mset}\ g\ N$
by (*hypsubst*, *rule filter-mset-cong*, *simp*)

lemma *ran-m-fmdrop*:
‹$C \in\# \mathit{dom\text{-}m}\ N \Longrightarrow \mathit{ran\text{-}m}\ (\mathit{fmdrop}\ C\ N) = \mathit{remove1\text{-}mset}\ (\mathit{the}\ (\mathit{fmlookup}\ N\ C))\ (\mathit{ran\text{-}m}\ N)$›
using *distinct-mset-dom*[*of N*]
by (*cases* ‹*fmlookup N C*›)
(*auto simp*: *ran-m-def image-mset-If-eq-notin*[*of C* - ‹$\lambda x.\ \mathit{fst}\ (\mathit{the}\ x)$›]
dest!: *multi-member-split*
intro!: *filter-mset-cong2 image-mset-cong2*)

lemma *ran-m-fmdrop-notin*:
‹$C \notin\# \mathit{dom\text{-}m}\ N \Longrightarrow \mathit{ran\text{-}m}\ (\mathit{fmdrop}\ C\ N) = \mathit{ran\text{-}m}\ N$›
using *distinct-mset-dom*[*of N*]
by (*auto simp*: *ran-m-def image-mset-If-eq-notin*[*of C* - ‹$\lambda x.\ \mathit{fst}\ (\mathit{the}\ x)$›]
dest!: *multi-member-split*
intro!: *filter-mset-cong2 image-mset-cong2*)

lemma *ran-m-fmdrop-If*:
‹$\mathit{ran\text{-}m}\ (\mathit{fmdrop}\ C\ N) = (\mathit{if}\ C \in\# \mathit{dom\text{-}m}\ N\ \mathit{then}\ \mathit{remove1\text{-}mset}\ (\mathit{the}\ (\mathit{fmlookup}\ N\ C))\ (\mathit{ran\text{-}m}\ N)\ \mathit{else}\ \mathit{ran\text{-}m}\ N)$›
using *distinct-mset-dom*[*of N*]
by (*auto simp*: *ran-m-def image-mset-If-eq-notin*[*of C* - ‹$\lambda x.\ \mathit{fst}\ (\mathit{the}\ x)$›]
dest!: *multi-member-split*
intro!: *filter-mset-cong2 image-mset-cong2*)

lemma *dom-m-empty-iff*[*iff*]:
‹$\mathit{dom\text{-}m}\ NU = \{\#\} \longleftrightarrow NU = \mathit{fmempty}$›
by (*cases NU*) (*auto simp*: *dom-m-def mset-set.insert-remove*)

end

theory *WB-Sort*
imports
Refine-Imperative-HOL.IICF
HOL−Library.Rewrite
Nested-Multisets-Ordinals.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* :: ‹$('b \Rightarrow 'b \Rightarrow \mathit{bool}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ \mathit{list} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat}\ \mathit{nres}$› **where**
‹$\mathit{choose\text{-}pivot}\ \text{-}\ \text{-}\ \text{-}\ \mathit{lo}\ \mathit{hi} = \mathit{SPEC}(\lambda k.\ k \geq \mathit{lo} \wedge k \leq \mathit{hi})$›

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

definition *isPartition-wrt* :: ‹$('b \Rightarrow 'b \Rightarrow \mathit{bool}) \Rightarrow 'b\ \mathit{list} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}$› **where**
‹$\mathit{isPartition\text{-}wrt}\ R\ \mathit{xs}\ \mathit{lo}\ \mathit{hi}\ p \equiv (\forall\ i.\ i \geq \mathit{lo} \wedge i < p \longrightarrow R\ (\mathit{xs}!i)\ (\mathit{xs}!p)) \wedge (\forall\ j.\ j > p \wedge j \leq \mathit{hi} \longrightarrow R\ (\mathit{xs}!p)\ (\mathit{xs}!j))$›

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*›
by (*simp add*: *isPartition-wrt-def*)

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* (λ*a 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*›
by (*auto simp add*: *isPartition-wrt-def conjI*)

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*›
by (*auto simp add*: *isPartition-def isPartition-wrt-def nth-Cons'*)

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

lemma *take-Suc0*:
l≠[] ⟹ *take* (*Suc 0*) *l* = [*l*!*0*]
0 < *length l* ⟹ *take* (*Suc 0*) *l* = [*l*!*0*]
Suc n ≤ *length l* ⟹ *take* (*Suc 0*) *l* = [*l*!*0*]
by (*cases l*, *auto*)+

lemma *sublist-single*: ‹*i* < *length xs* ⟹ *sublist xs i i* = [*xs*!*i*]›
by (*cases xs*) (*auto simp add*: *sublist-def take-Suc0*)

lemma *insert-eq*: ‹*insert a b* = *b* ∪ {*a*}›
by *auto*

lemma *sublist-nth*: ‹⟦*lo* ≤ *hi*; *hi* < *length xs*; *k*+*lo* ≤ *hi*⟧ ⟹ (*sublist xs lo hi*)!*k* = *xs*!(*lo*+*k*)›
by (*simp add*: *sublist-def*)

lemma *sublist-length*: ‹⟦*i* ≤ *j*; *j* < *length xs*⟧ ⟹ *length* (*sublist xs i j*) = *1* + *j* − *i*›
by (*simp add*: *sublist-def*)

lemma *sublist-not-empty*: ‹⟦*i* ≤ *j*; *j* < *length xs*; *xs* ≠ []⟧ ⟹ *sublist xs i j* ≠ []›
apply *simp*
apply (*rewrite List.length-greater-0-conv*[*symmetric*])
apply (*rewrite sublist-length*)
by *auto*

lemma *sublist-app*: ‹⟦*i1* ≤ *i2*; *i2* ≤ *i3*⟧ ⟹ *sublist xs i1 i2* @ *sublist xs* (*Suc i2*) *i3* = *sublist xs i1 i3*›

unfolding *sublist-def*
by (*smt* (*verit*) *Suc-eq-plus1-left Suc-le-mono append.assoc le-SucI le-add-diff-inverse le-trans same-append-eq take-add*)

definition *sorted-sublist-wrt* :: ‹('b ⇒ 'b ⇒ bool) ⇒ 'b list ⇒ nat ⇒ nat ⇒ bool› **where**
‹*sorted-sublist-wrt R xs lo hi* = *sorted-wrt R* (*sublist xs lo hi*)›

definition *sorted-sublist* :: ‹'a :: *linorder list* ⇒ *nat* ⇒ *nat* ⇒ *bool*› **where**
‹*sorted-sublist xs lo hi* = *sorted-sublist-wrt* (≤) *xs lo hi*›

abbreviation *sorted-sublist-map* :: ‹('b ⇒ 'b ⇒ bool) ⇒ ('a ⇒ 'b) ⇒ 'a list ⇒ nat ⇒ nat ⇒ bool› **where**
‹*sorted-sublist-map R h xs lo hi* ≡ *sorted-sublist-wrt* (λ*a b*. *R* (*h a*) (*h b*)) *xs lo hi*›

lemma *sorted-sublist-map-def'*:
‹*lo* < *length xs* ⟹ *sorted-sublist-map R h xs lo hi* ≡ *sorted-sublist-wrt R* (*map h xs*) *lo hi*›
apply (*simp add*: *sorted-sublist-wrt-def*)
by (*simp add*: *drop-map sorted-wrt-map sublist-def take-map*)

lemma *sorted-sublist-wrt-refl*: ‹*i* < *length xs* ⟹ *sorted-sublist-wrt R xs i i*›
by (*auto simp add*: *sorted-sublist-wrt-def sublist-single*)

lemma *sorted-sublist-refl*: ‹*i* < *length xs* ⟹ *sorted-sublist xs i i*›
by (*auto simp add*: *sorted-sublist-def sorted-sublist-wrt-refl*)

lemma *sublist-map*: ‹*sublist* (*map f xs*) *i j* = *map f* (*sublist xs i j*)›
apply (*auto simp add*: *sublist-def*)
by (*simp add*: *drop-map take-map*)

lemma *take-set*: ‹*j* ≤ *length xs* ⟹ *x* ∈ *set* (*take j xs*) ≡ (∃ *k*. *k* < *j* ∧ *xs*!*k* = *x*)›
by (*rule eq-reflection*) (*auto simp add*: *take-set*)

lemma *drop-set*: ‹*j* ≤ *length xs* ⟹ *x* ∈ *set* (*drop j xs*) ≡ (∃*k*. *j*≤*k*∧*k*<*length xs* ∧ *xs*!*k*=*x*)›
by (*smt* (*verit*) *Misc.in-set-drop-conv-nth*)

lemma *sublist-el*: ‹*i* ≤ *j* ⟹ *j* < *length xs* ⟹ *x* ∈ *set* (*sublist xs i j*) ≡ (∃ *k*. *k* < *Suc j*−*i* ∧ *xs*!(*i*+*k*)=*x*)›
by (*auto simp add*: *take-set sublist-def*)

lemma *sublist-el'*: ‹*i* ≤ *j* ⟹ *j* < *length xs* ⟹ *x* ∈ *set* (*sublist xs i j*) ≡ (∃ *k*. *i*≤*k*∧*k*≤*j* ∧ *xs*!*k*=*x*)›
apply (*subst sublist-el*, *assumption*, *assumption*)
by (*smt* (*verit*) *Groups.add-ac*(2) *le-add1 le-add-diff-inverse less-Suc-eq less-diff-conv nat-less-le order-refl*)

lemma *sublist-lt*: ‹*hi* < *lo* ⟹ *sublist xs lo hi* = []›
by (*auto simp add*: *sublist-def*)

lemma *nat-le-eq-or-lt*: ‹(*a* :: *nat*) ≤ *b* = (*a* = *b* ∨ *a* < *b*)›
by *linarith*

lemma *sorted-sublist-wrt-le*: ‹*hi* ≤ *lo* ⟹ *hi* < *length xs* ⟹ *sorted-sublist-wrt R xs lo hi*›
apply (*auto simp add*: *nat-le-eq-or-lt*)
unfolding *sorted-sublist-wrt-def*

subgoal apply (*rewrite sublist-single*) **by** *auto*
subgoal by (*auto simp add*: *sublist-lt*)
done

Elements in a sorted sublists are actually sorted

lemma *sorted-sublist-wrt-nth-le*:
assumes ‹*sorted-sublist-wrt R xs lo hi*› **and** ‹*lo* ≤ *hi*› **and** ‹*hi* < *length xs*› **and**
‹*lo* ≤ *i*› **and** ‹*i* < *j*› **and** ‹*j* ≤ *hi*›
shows ‹*R* (*xs*!*i*) (*xs*!*j*)›
proof −
have *A*: ‹*lo* < *length xs*› **using** *assms*(*2*) *assms*(*3*) **by** *linarith*
obtain *i'* **where** *I*: ‹*i* = *lo* + *i'*› **using** *assms*(*4*) *le-Suc-ex* **by** *auto*
obtain *j'* **where** *J*: ‹*j* = *lo* + *j'*› **by** (*meson assms*(*4*) *assms*(*5*) *dual-order.trans le-iff-add less-imp-le-nat*)
show *?thesis*
using *assms*(*1*) **apply** (*simp add*: *sorted-sublist-wrt-def I J*)
apply (*rewrite sublist-nth*[*symmetric*, **where** *k*=*i'*, **where** *lo*=*lo*, **where** *hi*=*hi*])
using *assms* **apply** *auto* **subgoal using** *I* **by** *linarith*
apply (*rewrite sublist-nth*[*symmetric*, **where** *k*=*j'*, **where** *lo*=*lo*, **where** *hi*=*hi*])
using *assms* **apply** *auto* **subgoal using** *J* **by** *linarith*
apply (*rule sorted-wrt-nth-less*)
apply *auto*
subgoal using *I J nat-add-left-cancel-less* **by** *blast*
subgoal apply (*simp add*: *sublist-length*) **using** *J* **by** *linarith*
done
qed

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

lemma *sorted-sublist-wrt-nth-le'*:
assumes *ref*: ‹⋀ *x*. *R x x*›
and ‹*sorted-sublist-wrt R xs lo hi*› **and** ‹*lo* ≤ *hi*› **and** ‹*hi* < *length xs*›
and ‹*lo* ≤ *i*› **and** ‹*i* ≤ *j*› **and** ‹*j* ≤ *hi*›
shows ‹*R* (*xs*!*i*) (*xs*!*j*)›
proof −
have ‹*i* < *j* ∨ *i* = *j*› **using** ‹*i* ≤ *j*› **by** *linarith*
then consider (*a*) ‹*i* < *j*› |
(*b*) ‹*i* = *j*› **by** *blast*
then show *?thesis*
proof *cases*
case *a*
then show *?thesis*
using *assms*(*2*−*5*,*7*) *sorted-sublist-wrt-nth-le* **by** *blast*
next
case *b*
then show *?thesis*
by (*simp add*: *ref*)
qed
qed

lemma *sorted-sublist-le*: ‹*hi* ≤ *lo* ⟹ *hi* < *length xs* ⟹ *sorted-sublist xs lo hi*›

by (*auto simp add*: *sorted-sublist-def sorted-sublist-wrt-le*)

lemma *sorted-sublist-map-le*: ‹*hi* ≤ *lo* ⟹ *hi* < *length xs* ⟹ *sorted-sublist-map R h xs lo hi*›
by (*auto simp add*: *sorted-sublist-wrt-le*)

lemma *sublist-cons*: ‹*lo* < *hi* ⟹ *hi* < *length xs* ⟹ *sublist xs lo hi* = *xs*!*lo* # *sublist xs* (*Suc lo*) *hi*›
by (*metis Cons-eq-appendI append-self-conv2 less-imp-le-nat less-or-eq-imp-le less-trans sublist-app sublist-single*)

lemma *sorted-sublist-wrt-cons'*:
‹*sorted-sublist-wrt R xs* (*lo*+*1*) *hi* ⟹ *lo* ≤ *hi* ⟹ *hi* < *length xs* ⟹ (∀ *j*. *lo*<*j*∧*j*≤*hi* ⟶ *R* (*xs*!*lo*) (*xs*!*j*)) ⟹ *sorted-sublist-wrt R xs lo hi*›
apply (*auto simp add*: *nat-le-eq-or-lt sorted-sublist-wrt-def*)
apply (*auto 5 4 simp add*: *sublist-cons sublist-el less-diff-conv add.commute*[*of* - *lo*] *dest*: *Suc-lessI sublist-single*)
done

lemma *sorted-sublist-wrt-cons*:
assumes *trans*: ‹(⋀ *x y z*. ⟦*R x y*; *R y z*⟧ ⟹ *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 −
show *?thesis*
apply (*rule sorted-sublist-wrt-cons'*) **using** *assms* **apply** *auto*
subgoal premises *assms'* **for** *j*
proof −
have *A*: ‹*j*=*lo*+*1* ∨ *j*>*lo*+*1*› **using** *assms'*(*5*) **by** *linarith*
show *?thesis*
using *A* **proof**
assume *A*: ‹*j*=*lo*+*1*› **show** *?thesis*
by (*simp add*: *A assms'*)
next
assume *A*: ‹*j*>*lo*+*1*› **show** *?thesis*
apply (*rule trans*)
apply (*rule assms*(*5*))
apply (*rule sorted-sublist-wrt-nth-le*[*OF assms*(*2*), **where** *i*=‹*lo*+*1*›, **where** *j*=*j*])
subgoal using *A assms'*(*6*) **by** *linarith*
subgoal using *assms'*(*3*) *less-imp-diff-less* **by** *blast*
subgoal using *assms'*(*5*) **by** *auto*
subgoal using *A* **by** *linarith*
subgoal by (*simp add*: *assms'*(*6*))
done
qed
qed
done
qed

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*›
by (*blast intro*: *sorted-sublist-wrt-cons*)

lemma *sublist-snoc*: ‹*lo* < *hi* ⟹ *hi* < *length xs* ⟹ *sublist xs lo hi* = *sublist xs lo* (*hi*−*1*) @ [*xs*!*hi*]›
apply (*simp add*: *sublist-def*)
proof −
assume *a1*: *lo* < *hi*
assume *hi* < *length xs*
then have *take lo xs* @ *take* (*Suc hi* − *lo*) (*drop lo xs*) = (*take lo xs* @ *take* (*hi* − *lo*) (*drop lo xs*)) @ [*xs* ! *hi*]
using *a1* **by** (*metis* (*no-types*) *Suc-diff-le add-Suc-right hd-drop-conv-nth le-add-diff-inverse less-imp-le-nat take-add take-hd-drop*)
then show *take* (*Suc hi* − *lo*) (*drop lo xs*) = *take* (*hi* − *lo*) (*drop lo xs*) @ [*xs* ! *hi*]
by *simp*
qed

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*›
apply (*simp add*: *sorted-sublist-wrt-def*)
apply (*auto simp add*: *nat-le-eq-or-lt*)
subgoal by (*simp add*: *sublist-single*)
by (*auto simp add*: *sublist-snoc sublist-el sorted-wrt-append add.commute*[*of lo*] *less-diff-conv*
simp: *leI simp flip*:*nat-le-eq-or-lt*)

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 −
show *?thesis*
apply (*rule sorted-sublist-wrt-snoc′*) **using** *assms* **apply** *auto*
subgoal premises *assms′* **for** *j*
proof −
have *A*: ‹*j*=*hi*−*1* ∨ *j*<*hi*−*1*› **using** *assms′*(*6*) **by** *linarith*
show *?thesis*
using *A* **proof**
assume *A*: ‹*j*=*hi*−*1*› **show** *?thesis*
by (*simp add*: *A assms′*)
next
assume *A*: ‹*j*<*hi*−*1*› **show** *?thesis*
apply (*rule trans*)
apply (*rule sorted-sublist-wrt-nth-le*[*OF assms*(*2*), **where** *i*=*j*, **where** *j*=‹*hi*−*1*›])
prefer *6*
apply (*rule assms*(*5*))
apply *auto*
subgoal using *A assms′*(*5*) **by** *linarith*
subgoal using *assms′*(*3*) *less-imp-diff-less* **by** *blast*
subgoal using *assms′*(*5*) **by** *auto*
subgoal using *A* **by** *linarith*
done
qed
qed
done
qed

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*›
by (*simp add*: *sublist-app*)

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*›
by (*auto simp add*: *sublist-split*[*symmetric*] *sublist-snoc*[**where** *xs*=*xs*,**where** *lo*=*lo*,**where** *hi*=*p*])

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*))›
by (*auto simp add*: *isPartition-wrt-def*)

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*)))›
by (*auto simp add*: *isPartition-wrt-def*)

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* ⟹
(∀ *i j*. *lo* ≤ *i* ∧ *i* < *p* ∧ *p* < *j* ∧ *j* ≤ *hi* ⟶ *R* (*xs*!*i*) (*xs*!*j*)) ⟹
sorted-sublist-wrt R xs lo hi›
apply (*auto simp add*: *isPartition-def isPartition-wrt-def sorted-sublist-def sorted-sublist-wrt-def sublist-map*)
apply (*simp add*: *sublist-split-part*[*symmetric*])
apply (*auto simp add*: *List.sorted-wrt-append*)
subgoal by (*auto simp add*: *sublist-el*)
subgoal by (*auto simp add*: *sublist-el*)
subgoal by (*auto simp add*: *sublist-el′*)
done

lemma *merge-sorted-wrt-partitions-between*:
‹(⋀ *x y z*. ⟦*R x y*; *R y z*⟧ ⟹ *R x z*) ⟹
isPartition-wrt R xs lo hi p ⟹
sorted-sublist-wrt R xs lo (*p*−*1*) ⟹ *sorted-sublist-wrt R xs* (*p*+*1*) *hi* ⟹
lo ≤ *hi* ⟹ *hi* < *length xs* ⟹ *lo* < *p* ⟹ *p* < *hi* ⟹ *hi* < *length xs* ⟹
sorted-sublist-wrt R xs lo hi›
by (*simp add*: *merge-sorted-wrt-partitions-between′ isPartition-wrt-trans*)

The main theorem to merge sorted lists

lemma *merge-sorted-wrt-partitions*:
‹*isPartition-wrt R xs lo hi p* ⟹
sorted-sublist-wrt R xs lo (*p* − *Suc 0*) ⟹ *sorted-sublist-wrt R xs* (*Suc p*) *hi* ⟹
lo ≤ *hi* ⟹ *lo* ≤ *p* ⟹ *p* ≤ *hi* ⟹ *hi* < *length xs* ⟹
(∀ *i j*. *lo* ≤ *i* ∧ *i* < *p* ∧ *p* < *j* ∧ *j* ≤ *hi* ⟶ *R* (*xs*!*i*) (*xs*!*j*)) ⟹
sorted-sublist-wrt R xs lo hi›
subgoal premises *assms*
proof −

```
    have C: ‹lo=p∧p=hi ∨ lo=p∧p<hi ∨ lo<p∧p=hi ∨ lo<p∧p<hi›
      using assms by linarith
    show ?thesis
      using C apply auto
      subgoal — lo=p=hi
        apply (rule sorted-sublist-wrt-refl)
        using assms by auto
      subgoal — lo=p<hi
        using assms by (simp add: isPartition-def isPartition-wrt-def sorted-sublist-wrt-cons')
      subgoal — lo<p=hi
        using assms by (simp add: isPartition-def isPartition-wrt-def sorted-sublist-wrt-snoc')
      subgoal — lo<p<hi
        using assms
        apply (rewrite merge-sorted-wrt-partitions-between'[where p=p])
        by auto
      done
  qed
  done

theorem merge-sorted-map-partitions:
  ‹(⋀ x y z. ⟦R (h x) (h y); R (h y) (h z)⟧ ⟹ R (h x) (h z)) ⟹
    isPartition-map R h xs lo hi p ⟹
    sorted-sublist-map R h xs lo (p − Suc 0) ⟹ sorted-sublist-map R h xs (Suc p) hi ⟹
    lo ≤ hi ⟹ lo ≤ p ⟹ p ≤ hi ⟹ hi < length xs ⟹
    sorted-sublist-map R h xs lo hi›
  apply (rule merge-sorted-wrt-partitions) apply auto
  by (simp add: merge-sorted-wrt-partitions isPartition-map-trans)


lemma partition-wrt-extend:
  ‹isPartition-wrt R xs lo' hi' p ⟹
  hi < length xs ⟹
  lo ≤ lo' ⟹ lo' ≤ hi ⟹ hi' ≤ hi ⟹
  lo' ≤ p ⟹ p ≤ hi' ⟹
  (⋀ i. lo≤i ⟹ i <lo' ⟹ R (xs!i) (xs!p)) ⟹
  (⋀ j. hi'<j ⟹ j≤hi ⟹ R (xs!p) (xs!j)) ⟹
  isPartition-wrt R xs lo hi p›
  unfolding isPartition-wrt-def
  apply (intro conjI)
  subgoal
    by (force simp: not-le)
  subgoal
    using leI by blast
  done

lemma partition-map-extend:
  ‹isPartition-map R h xs lo' hi' p ⟹
  hi < length xs ⟹
  lo ≤ lo' ⟹ lo' ≤ hi ⟹ hi' ≤ hi ⟹
  lo' ≤ p ⟹ p ≤ hi' ⟹
  (⋀ i. lo≤i ⟹ i <lo' ⟹ R (h (xs!i)) (h (xs!p))) ⟹
  (⋀ j. hi'<j ⟹ j≤hi ⟹ R (h (xs!p)) (h (xs!j))) ⟹
  isPartition-map R h xs lo hi p›
  by (auto simp add: partition-wrt-extend)
```

lemma *isPartition-empty*:
‹(⋀ *j*. ⟦*lo* < *j*; *j* ≤ *hi*⟧ ⟹ *R* (*xs* ! *lo*) (*xs* ! *j*)) ⟹
isPartition-wrt R xs lo hi lo›
by (*auto simp add*: *isPartition-wrt-def*)

lemma *take-ext*:
‹(∀ *i*<*k*. *xs'*!*i*=*xs*!*i*) ⟹
k < *length xs* ⟹ *k* < *length xs'* ⟹
take k xs' = *take k xs*›
by (*simp add*: *nth-take-lemma*)

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*›
apply (*rewrite* **in** ‹*drop* - ⌑ = -› *List.rev-rev-ident*[*symmetric*])
apply (*rewrite* **in** ‹- = *drop* - ⌑› *List.rev-rev-ident*[*symmetric*])
apply (*rewrite* **in** ‹⌑ = -› *List.drop-rev*)
apply (*rewrite* **in** ‹- = ⌑› *List.drop-rev*)
apply *simp*
apply (*rule take-ext*)
by (*auto simp add*: *rev-nth*)

lemma *drop-ext*:
‹(∀ *i*. *i*≥*k* ∧ *i*<*length xs* ⟶ *xs'*!*i*=*xs*!*i*) ⟹
length xs'=*length xs* ⟹
drop k xs' = *drop k xs*›
apply (*cases xs*)
apply *auto*
apply (*cases k*)
subgoal by (*simp add*: *nth-equalityI*)
subgoal apply (*rule drop-ext'*) **by** *auto*
done

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*›
apply (*simp add*: *sublist-def*)
apply (*rule take-ext*)
by *auto*

lemma *lt-Suc*: ‹(*a* < *b*) = (*Suc a* = *b* ∨ *Suc a* < *b*)›
by *auto*

lemma *sublist-until-end-eq-drop*: ‹*Suc hi* = *length xs* ⟹ *sublist xs lo hi* = *drop lo xs*›
by (*simp add*: *sublist-def*)

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*›
  apply (*auto simp add*: *lt-Suc*[**where** *a*=*hi*])
  subgoal by (*auto simp add*: *sublist-until-end-eq-drop drop-ext*)
  subgoal by (*auto simp add*: *sublist-ext'*)
  done

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 −
  have *l*: ‹*lo* < *lo'* − *1* ∨ *lo* ≥ *lo'*−*1*›
    by *linarith*
  show *?thesis*
    using *l* **apply** *auto*
    subgoal — lo < lo' - 1
      apply (*auto simp add*: *sorted-sublist-wrt-def*)
      apply (*rewrite sublist-ext*[**where** *xs*=*xs*])
      using *assms* **by** (*auto simp add*: *sorted-sublist-wrt-def*)
    subgoal — lo >= lo' - 1
      using *assms* **by** (*auto simp add*: *sorted-sublist-wrt-le*)
    done
qed

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*)›
  using *assms* **by** (*rule sorted-wrt-lower-sublist-still-sorted*)

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 −
  have *l*: ‹*hi'* + *1* < *hi* ∨ *hi'* + *1* ≥ *hi*›
    by *linarith*
  show *?thesis*
    using *l* **apply** *auto*
    subgoal — hi' + 1 < h
      apply (*auto simp add*: *sorted-sublist-wrt-def*)
      apply (*rewrite sublist-ext*[**where** *xs*=*xs*])
      using *assms* **by** (*auto simp add*: *sorted-sublist-wrt-def*)
    subgoal — *hi* ≤ *hi'* + *1*
      using *assms* **by** (*auto simp add*: *sorted-sublist-wrt-le*)
    done
qed

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*›
using *assms* **by** (*rule sorted-wrt-upper-sublist-still-sorted*)

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*)›
by (*metis in-set-conv-nth length-take min.commute min.strict-boundedE nth-take*)

lemma *mset-drop-upto*: ‹*mset* (*drop a N*) = {#*N*!*i*. *i* ∈# *mset-set* {*a*..<*length N*}#}›
proof (*induction N arbitrary*: *a*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons c N*)
have *upt*: ‹{*0*..<*Suc* (*length N*)} = *insert 0* {*1*..<*Suc* (*length N*)}›
by *auto*
then have *H*: ‹*mset-set* {*0*..<*Suc* (*length N*)} = *add-mset 0* (*mset-set* {*1*..<*Suc* (*length N*)})›
unfolding *upt* **by** *auto*
have *mset-case-Suc*: ‹{#*case x of 0* ⇒ *c* | *Suc x* ⇒ *N* ! *x* . *x* ∈# *mset-set* {*Suc a*..<*Suc b*}#} =
{#*N* ! (*x*−*1*) . *x* ∈# *mset-set* {*Suc a*..<*Suc b*}#}› **for** *a b*
by (*rule image-mset-cong*) (*auto split*: *nat.splits*)
have *Suc-Suc*: ‹{*Suc a*..<*Suc b*} = *Suc* ‘ {*a*..<*b*}› **for** *a b*
by *auto*
then have *mset-set-Suc-Suc*: ‹*mset-set* {*Suc a*..<*Suc b*} = {#*Suc n*. *n* ∈# *mset-set* {*a*..<*b*}#}› **for** *a b*
unfolding *Suc-Suc* **by** (*subst image-mset-mset-set*[*symmetric*]) *auto*
have ∗: ‹{#*N* ! (*x*−*Suc 0*) . *x* ∈# *mset-set* {*Suc a*..<*Suc b*}#} = {#*N* ! *x* . *x* ∈# *mset-set* {*a*..<*b*}#}›
for *a b*
by (*auto simp add*: *mset-set-Suc-Suc multiset.map-comp comp-def*)
show *?case*
apply (*cases a*)
using *Cons*[*of 0*] *Cons* **by** (*auto simp*: *nth-Cons drop-Cons H mset-case-Suc* ∗)
qed

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 −

```
  define xs1 xs2 xs3 xs1' xs2' xs3' where
    ‹xs1 = take lo xs› and
    ‹xs2 = take (Suc hi − lo) (drop lo xs)› and
    ‹xs3 = drop (Suc hi) xs› and
    ‹xs1' = take lo xs'› and
    ‹xs2' = take (Suc hi − lo) (drop lo xs')› and
    ‹xs3' = drop (Suc hi) xs'›
  have [simp]: ‹length xs' = length xs›
    using Perm by (auto dest: mset-eq-length)
  have [simp]: ‹mset xs1 = mset xs1'›
    using Fix(1) unfolding xs1-def xs1'-def
    by (metis Perm le-cases mset-eq-length nth-take-lemma take-all)
  have [simp]: ‹mset xs3 = mset xs3'›
    using Fix(2) unfolding xs3-def xs3'-def mset-drop-upto
    by (auto intro: image-mset-cong)
  have ‹xs = xs1 @ xs2 @ xs3› ‹xs' = xs1' @ xs2' @ xs3'›
    using I unfolding xs1-def xs2-def xs3-def xs1'-def xs2'-def xs3'-def
    by (metis append.assoc append-take-drop-id le-SucI le-add-diff-inverse order-trans take-add)+
  moreover have ‹xs ! i = xs2 ! (i − lo)› ‹i ≥ length xs1›
    using I Bounds unfolding xs2-def xs1-def by (auto simp: nth-take min-def)
  moreover have  ‹x ∈ set xs2'›
    using I Bounds unfolding xs2'-def
    by (auto simp: in-set-take-conv-nth
      intro!: exI[of - ‹i − lo›])
  ultimately have ‹x ∈ set xs2›
    using Perm I by (auto dest: mset-eq-setD)
  then obtain j where ‹xs ! (lo + j) = x› ‹j ≤ hi − lo›
    unfolding in-set-conv-nth xs2-def
    by auto
  then show ?thesis
    using Bounds I
    by (auto intro: exI[of - ‹lo+j›])
qed
```

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
  fix x
  assume ‹x ∈ set (sublist xs' lo hi)›
  then have ‹∃ i. lo≤i∧i≤hi ∧ xs'!i=x›
    by (metis assms(1) bounds(1) bounds(2) size-mset sublist-el')
  then obtain i where I: ‹lo≤i› ‹i≤hi› ‹xs'!i=x› by blast
  have ‹∃ j. lo≤j∧j≤hi ∧ xs!j=x›
    using Perm I bounds(2) Fix by (rule mathias, auto)
  then show ‹x ∈ set (sublist xs lo hi)›
    by (simp add: bounds(1) bounds(2) sublist-el')
qed


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
show ‹*set* (*sublist xs′ lo hi*) ⊆ *set* (*sublist xs lo hi*)›
apply (*rule mset-sublist-incl*)
using *assms* **by** *auto*
show ‹*set* (*sublist xs lo hi*) ⊆ *set* (*sublist xs′ lo hi*)›
by (*rule mset-sublist-incl*) (*metis assms size-mset*)+
qed

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 (λ*f* (*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′*›
by (*auto simp add*: *quicksort-post-def*)

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

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 −

First boilerplate code step: 'unfold' the HOL definitions in the assumptions and convert them to Pure

```
have pre: ‹lo ≤ hi› ‹hi < length xs›
  using pre by (auto simp add: quicksort-pre-def)

have part: ‹mset xs' = mset xs› True
  ‹isPartition-map R h xs' lo hi p› ‹lo ≤ p› ‹p ≤ hi›
  ‹⋀ i. i<lo ⟹ xs'!i=xs!i› ‹⋀ i. ⟦hi<i; i<length xs'⟧ ⟹ xs'!i=xs!i›
  using part by (auto simp add: partition-spec-def)


have sorted-lower: ‹sorted-sublist-map R h xs' lo (p − Suc 0)›
proof −
  show ?thesis
    apply (rule sorted-sublist-wrt-le)
    subgoal using ifs(1) by auto
    subgoal using ifs(1) mset-eq-length part(1) pre(1) pre(2) by fastforce
    done
qed

have sorted-upper: ‹sorted-sublist-map R h xs' (Suc p) hi›
proof −
  show ?thesis
    apply (rule sorted-sublist-wrt-le)
    subgoal using ifs(2) by auto
    subgoal using ifs(1) mset-eq-length part(1) pre(1) pre(2) by fastforce
    done
qed

have sorted-middle: ‹sorted-sublist-map R h xs' lo hi›
proof −
  show ?thesis
    apply (rule merge-sorted-map-partitions[where p=p])
    subgoal by (rule trans)
    subgoal by (rule part)
    subgoal by (rule sorted-lower)
    subgoal by (rule sorted-upper)
    subgoal using pre(1) by auto
    subgoal by (simp add: part(4))
    subgoal by (simp add: part(5))
    subgoal by (metis part(1) pre(2) size-mset)
    done
qed

show ?thesis
proof (intro quicksort-postI)
  show ‹mset xs' = mset xs›
    by (simp add: part(1))
next
  show ‹sorted-sublist-map R h xs' lo hi›
    by (rule sorted-middle)
next
    show ‹⋀i. i < lo ⟹ xs' ! i = xs ! i›
    using part(6) by blast
next
```

```
    show ‹⋀j. ⟦hi < j; j < length xs⟧ ⟹ xs' ! j = xs ! j›
      by (metis part(1) part(7) size-mset)
  qed
qed
```

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 −
```

First boilerplate code step: 'unfold' the HOL definitions in the assumptions and convert them to Pure

```
  have pre: ‹lo ≤ hi› ‹hi < length xs› ‹mset xs = mset xs0›
    using pre by (auto simp add: quicksort-pre-def)
  have part: ‹mset xs' = mset xs› True
    ‹isPartition-map R h xs' lo hi p› ‹lo ≤ p› ‹p ≤ hi›
    ‹⋀ i. i<lo ⟹ xs'!i=xs!i› ‹⋀ i. ⟦hi<i; i<length xs'⟧ ⟹ xs'!i=xs!i›
    using part by (auto simp add: partition-spec-def)
  show ?thesis
    unfolding quicksort-pre-def
  proof (intro conjI)
    show ‹Suc p ≤ hi›
      using ifs by linarith
    show ‹hi < length xs'›
      by (metis part(1) pre(2) size-mset)
    show ‹mset xs' = mset xs0›
      using pre(3) part(1) by (auto dest: mset-eq-setD)
  qed
qed
```

```
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 −
  have ‹mset xs' = mset xs› ‹⋀ i. i<lo ⟹ xs'!i = xs!i› ‹⋀ j. ⟦hi<j; j<length xs⟧ ⟹ xs'!j = xs!j›
    using assms by (auto simp add: quicksort-post-def)
  then show ?thesis
    using bounds by (rule mset-sublist-eq, auto)
qed
```

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 −
```

First boilerplate code step: 'unfold' the HOL definitions in the assumptions and convert them to Pure

```
 have pre: ‹lo ≤ hi› ‹hi < length xs› ‹mset xs = mset xs0›
  using pre by (auto simp add: quicksort-pre-def)
 have part: ‹mset xs' = mset xs› True
  ‹isPartition-map R h xs' lo hi p› ‹lo ≤ p› ‹p ≤ hi›
  ‹⋀ i. i<lo ⟹ xs'!i=xs!i› ‹⋀ i. ⟦hi<i; i<length xs'⟧ ⟹ xs'!i=xs!i›
  using part by (auto simp add: partition-spec-def)
 have IH1: ‹mset xs'' = mset xs'› ‹sorted-sublist-map R h xs'' (Suc p) hi›
    ‹⋀ i. i<Suc p ⟹ xs'' ! i = xs' ! i› ‹⋀ j. ⟦hi < j; j < length xs'⟧ ⟹ xs'' ! j = xs' ! j›
  using IH1' by (auto simp add: quicksort-post-def)
 note IH1-perm = quicksort-post-set[OF IH1']

 have still-partition: ‹isPartition-map R h xs'' lo hi p›
 proof(intro isPartition-wrtI)
  fix i assume ‹lo ≤ i› ‹i < p›
  show ‹R (h (xs'' ! i)) (h (xs'' ! p))›
```

This holds because this part hasn't changed

```
    using IH1(3) ‹i < p› ‹lo ≤ i› isPartition-wrt-def part(3) by fastforce
  next
    fix j assume ‹p < j› ‹j ≤ hi›
```

Obtain the position *posJ* where $xs'' ! j$ was stored in xs'.

```
    have ‹xs''!j ∈ set (sublist xs'' (Suc p) hi)›
     by (metis IH1(1) Suc-leI ‹j ≤ hi› ‹p < j› less-le-trans mset-eq-length part(1) pre(2) sublist-el')
    then have ‹xs''!j ∈ set (sublist xs' (Suc p) hi)›
     by (metis IH1-perm ifs(2) nat-le-linear part(1) pre(2) size-mset)
    then have ‹∃ posJ. Suc p≤posJ∧posJ≤hi ∧ xs''!j = xs'!posJ›
     by (metis Suc-leI ‹j ≤ hi› ‹p < j› less-le-trans part(1) pre(2) size-mset sublist-el')
    then obtain posJ :: nat where PosJ: ‹Suc p≤posJ› ‹posJ≤hi› ‹xs''!j = xs'!posJ› by blast

    then show ‹R (h (xs'' ! p)) (h (xs'' ! j))›
     by (metis IH1(3) Suc-le-lessD isPartition-wrt-def lessI part(3))
 qed

 have sorted-lower: ‹sorted-sublist-map R h xs'' lo (p − Suc 0)›
 proof −
  show ?thesis
   apply (rule sorted-sublist-wrt-le)
   subgoal by (simp add: ifs(1))
   subgoal using IH1(1) mset-eq-length part(1) part(5) pre(2) by fastforce
   done
 qed

 note sorted-upper = IH1(2)

 have sorted-middle: ‹sorted-sublist-map R h xs'' lo hi›
 proof −
  show ?thesis
```

```
    apply (rule merge-sorted-map-partitions[where p=p])
    subgoal by (rule trans)
    subgoal by (rule still-partition)
    subgoal by (rule sorted-lower)
    subgoal by (rule sorted-upper)
    subgoal using pre(1) by auto
    subgoal by (simp add: part(4))
    subgoal by (simp add: part(5))
    subgoal by (metis IH1(1) part(1) pre(2) size-mset)
    done
  qed


  show ?thesis
  proof (intro quicksort-postI)
    show ‹mset xs'' = mset xs›
      using part(1) IH1(1) by auto — I was faster than sledgehammer :-)
  next
    show ‹sorted-sublist-map R h xs'' lo hi›
      by (rule sorted-middle)
  next
    show ‹⋀i. i < lo ⟹ xs'' ! i = xs ! i›
      using IH1(3) le-SucI part(4) part(6) by auto
  next show ‹⋀j. hi < j ⟹ j < length xs ⟹ xs'' ! j = xs ! j›
      by (metis IH1(4) part(1) part(7) size-mset)
  qed
qed
```

In the 4th case, we have to show that the premise holds for $(lo,\ p - 1,\ xs')$, in case $\neg\ p - 1 \leq lo$

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 −
```

First boilerplate code step: 'unfold' the HOL definitions in the assumptions and convert them to Pure

```
  have pre: ‹lo ≤ hi› ‹hi < length xs› ‹mset xs0 = mset xs›
    using pre by (auto simp add: quicksort-pre-def)
  have part: ‹mset xs' = mset xs› True
    ‹isPartition-map R h xs' lo hi p› ‹lo ≤ p› ‹p ≤ hi›
    ‹⋀ i. i<lo ⟹ xs'!i=xs!i› ‹⋀ i. ⟦hi<i; i<length xs'⟧ ⟹ xs'!i=xs!i›
    using part by (auto simp add: partition-spec-def)

  show ?thesis
    unfolding quicksort-pre-def
  proof (intro conjI)
    show ‹lo ≤ p − Suc 0›
      using ifs by linarith
    show ‹p − Suc 0 < length xs'›
```

```
    using mset-eq-length part(1) part(5) pre(2) by fastforce
  show ‹mset xs' = mset xs0›
    using pre(3) part(1) by (auto dest: mset-eq-setD)
 qed
qed
```

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 −
```

First boilerplate code step: 'unfold' the HOL definitions in the assumptions and convert them to Pure

```
 have pre: ‹lo ≤ hi› ‹hi < length xs›
   using pre by (auto simp add: quicksort-pre-def)
 have part: ‹mset xs' = mset xs› True
   ‹isPartition-map R h xs' lo hi p› ‹lo ≤ p› ‹p ≤ hi›
   ‹⋀ i. i<lo ⟹ xs'!i=xs!i› ‹⋀ i. ⟦hi<i; i<length xs'⟧ ⟹ xs'!i=xs!i›
   using part by (auto simp add: partition-spec-def)
 have IH1: ‹mset xs'' = mset xs'› ‹sorted-sublist-map R h xs'' lo (p − Suc 0)›
   ‹⋀ i. i<lo ⟹ xs''!i = xs'!i› ‹⋀ j. ⟦p−Suc 0<j; j<length xs'⟧ ⟹ xs''!j = xs'!j›
   using IH1' by (auto simp add: quicksort-post-def)
 note IH1-perm = quicksort-post-set[OF IH1']


 have still-partition: ‹isPartition-map R h xs'' lo hi p›
 proof(intro isPartition-wrtI)
   fix i assume ‹lo ≤ i› ‹i < p›
```

Obtain the position *posI* where $xs'' ! i$ was stored in xs'.

```
     have ‹xs''!i ∈ set (sublist xs'' lo (p−Suc 0))›
      by (metis (no-types, lifting) IH1(1) Suc-leI Suc-pred ‹i < p› ‹lo ≤ i› le-less-trans less-imp-diff-less
mset-eq-length not-le not-less-zero part(1) part(5) pre(2) sublist-el')
     then have ‹xs''!i ∈ set (sublist xs' lo (p−Suc 0))›
        by (metis IH1-perm ifs(1) le-less-trans less-imp-diff-less mset-eq-length nat-le-linear part(1)
part(5) pre(2))
     then have ‹∃ posI. lo≤posI∧posI≤p−Suc 0 ∧ xs''!i = xs'!posI›
     proof − — sledgehammer
       have p − Suc 0 < length xs
         by (meson diff-le-self le-less-trans part(5) pre(2))
       then show ?thesis
        by (metis (no-types) ‹xs'' ! i ∈ set (sublist xs' lo (p − Suc 0))› ifs(1) mset-eq-length nat-le-linear
part(1) sublist-el')
     qed
     then obtain posI :: nat where PosI: ‹lo≤posI› ‹posI≤p−Suc 0› ‹xs''!i = xs'!posI› by blast
     then show ‹R (h (xs'' ! i)) (h (xs'' ! p))›
        by (metis (no-types, lifting) IH1(4) ‹i < p› diff-Suc-less isPartition-wrt-def le-less-trans
mset-eq-length not-le not-less-eq part(1) part(3) part(5) pre(2) zero-less-Suc)
   next
```

```
    fix j assume ‹p < j› ‹j ≤ hi›
    then show ‹R (h (xs'' ! p)) (h (xs'' ! j))›
```

This holds because this part hasn't changed

```
      by (smt (verit) IH1(4) add-diff-cancel-left' add-diff-inverse-nat diff-Suc-eq-diff-pred diff-le-self
ifs(1) isPartition-wrt-def le-less-Suc-eq less-le-trans mset-eq-length nat-less-le part(1) part(3) part(4)
plus-1-eq-Suc pre(2))
  qed


  note sorted-lower = IH1(2)

  have sorted-upper: ‹sorted-sublist-map R h xs'' (Suc p) hi›
  proof −
    show ?thesis
      apply (rule sorted-sublist-wrt-le)
      subgoal by (simp add: ifs(2))
      subgoal using IH1(1) mset-eq-length part(1) part(5) pre(2) by fastforce
      done
  qed


  have sorted-middle: ‹sorted-sublist-map R h xs'' lo hi›
  proof −
    show ?thesis
      apply (rule merge-sorted-map-partitions[where p=p])
      subgoal by (rule trans)
      subgoal by (rule still-partition)
      subgoal by (rule sorted-lower)
      subgoal by (rule sorted-upper)
      subgoal using pre(1) by auto
      subgoal by (simp add: part(4))
      subgoal by (simp add: part(5))
      subgoal by (metis IH1(1) part(1) pre(2) size-mset)
      done
  qed


  show ?thesis
  proof (intro quicksort-postI)
    show ‹mset xs'' = mset xs›
      by (simp add: IH1(1) part(1))
  next
    show ‹sorted-sublist-map R h xs'' lo hi›
      by (rule sorted-middle)
  next
    show ‹⋀i. i < lo ⟹ xs'' ! i = xs ! i›
      by (simp add: IH1(3) part(6))
  next
    show ‹⋀j. hi < j ⟹ j < length xs ⟹ xs'' ! j = xs ! j›
      by (metis IH1(4) diff-le-self dual-order.strict-trans2 mset-eq-length part(1) part(5) part(7))
  qed
qed
```

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 −

First boilerplate code step: 'unfold' the HOL definitions in the assumptions and convert them to Pure

have *pre*: ‹*lo ≤ hi*› ‹*hi < length xs*› ‹*mset xs0 = mset xs*›
using *pre* **by** (*auto simp add*: *quicksort-pre-def*)
have *part*: ‹*mset xs' = mset xs*› *True*
‹*isPartition-map R h xs' lo hi p*› ‹*lo ≤ p*› ‹*p ≤ hi*›
‹⋀ *i*. *i*<*lo* ⟹ *xs'*!*i*=*xs*!*i*› ‹⋀ *i*. ⟦*hi*<*i*; *i*<*length xs'*⟧ ⟹ *xs'*!*i*=*xs*!*i*›
using *part* **by** (*auto simp add*: *partition-spec-def*)
have *IH1*: ‹*mset xs'' = mset xs'*› ‹*sorted-sublist-map R h xs'' lo (p − Suc 0)*›
‹⋀ *i*. *i*<*lo* ⟹ *xs''*!*i* = *xs'*!*i*› ‹⋀ *j*. ⟦*p*−*Suc 0*<*j*; *j*<*length xs'*⟧ ⟹ *xs''*!*j* = *xs'*!*j*›
using *IH1* **by** (*auto simp add*: *quicksort-post-def*)

show *?thesis*
unfolding *quicksort-pre-def*
proof (*intro conjI*)
show ‹*Suc p ≤ hi*›
using *ifs*(*2*) **by** *linarith*
show ‹*hi < length xs''*›
using *IH1*(*1*) *mset-eq-length part*(*1*) *pre*(*2*) **by** *fastforce*
show ‹*mset xs'' = mset xs0*›
using *pre*(*3*) *part*(*1*) *IH1*(*1*) **by** (*auto dest*: *mset-eq-setD*)
qed
qed

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 −

First boilerplate code step: 'unfold' the HOL definitions in the assumptions and convert them to Pure

have *pre*: ‹*lo ≤ hi*› ‹*hi < length xs*›
using *pre* **by** (*auto simp add*: *quicksort-pre-def*)
have *part*: ‹*mset xs' = mset xs*› *True*
‹*isPartition-map R h xs' lo hi p*› ‹*lo ≤ p*› ‹*p ≤ hi*›
‹⋀ *i*. *i*<*lo* ⟹ *xs'*!*i*=*xs*!*i*› ‹⋀ *i*. ⟦*hi*<*i*; *i*<*length xs'*⟧ ⟹ *xs'*!*i*=*xs*!*i*›
using *part* **by** (*auto simp add*: *partition-spec-def*)
have *IH1*: ‹*mset xs'' = mset xs'*› ‹*sorted-sublist-map R h xs'' lo (p − Suc 0)*›

‹⋀ i. i<lo ⟹ xs''!i = xs'!i› ‹⋀ j. ⟦p−Suc 0<j; j<length xs'⟧ ⟹ xs''!j = xs'!j›
using *IH1'* **by** (*auto simp add*: *quicksort-post-def*)
note *IH1-perm* = *quicksort-post-set*[*OF IH1'*]
have *IH2*: ‹*mset xs''' = mset xs''*› ‹*sorted-sublist-map R h xs''' (Suc p) hi*›
‹⋀ i. i<Suc p ⟹ xs'''!i = xs''!i› ‹⋀ j. ⟦hi<j; j<length xs''⟧ ⟹ xs'''!j = xs''!j›
using *IH2'* **by** (*auto simp add*: *quicksort-post-def*)
note *IH2-perm* = *quicksort-post-set*[*OF IH2'*]

We still have a partition after the first call (same as in case 5)

have *still-partition1*: ‹*isPartition-map R h xs'' lo hi p*›
proof(*intro isPartition-wrtI*)
fix *i* **assume** ‹*lo* ≤ *i*› ‹*i* < *p*›

Obtain the position *posI* where *xs''* ! *i* was stored in *xs'*.

have ‹*xs''!i* ∈ *set* (*sublist xs'' lo* (*p*−*Suc 0*))›
by (*metis* (*no-types*, *lifting*) *IH1*(*1*) *Suc-leI Suc-pred* ‹*i* < *p*› ‹*lo* ≤ *i*› *le-less-trans less-imp-diff-less mset-eq-length not-le not-less-zero part*(*1*) *part*(*5*) *pre*(*2*) *sublist-el'*)
then have ‹*xs''!i* ∈ *set* (*sublist xs' lo* (*p*−*Suc 0*))›
by (*metis IH1-perm ifs*(*1*) *le-less-trans less-imp-diff-less mset-eq-length nat-le-linear part*(*1*) *part*(*5*) *pre*(*2*))
then have ‹∃ *posI*. *lo*≤*posI*∧*posI*≤*p*−*Suc 0* ∧ *xs''!i* = *xs'!posI*›
proof − — sledgehammer
have *p* − *Suc 0* < *length xs*
by (*meson diff-le-self le-less-trans part*(*5*) *pre*(*2*))
then show *?thesis*
by (*metis* (*no-types*) ‹*xs''* ! *i* ∈ *set* (*sublist xs' lo* (*p* − *Suc 0*))› *ifs*(*1*) *mset-eq-length nat-le-linear part*(*1*) *sublist-el'*)
qed
then obtain *posI* :: *nat* **where** *PosI*: ‹*lo*≤*posI*› ‹*posI*≤*p*−*Suc 0*› ‹*xs''!i* = *xs'!posI*› **by** *blast*
then show ‹*R* (*h* (*xs''* ! *i*)) (*h* (*xs''* ! *p*))›
by (*metis* (*no-types*, *lifting*) *IH1*(*4*) ‹*i* < *p*› *diff-Suc-less isPartition-wrt-def le-less-trans mset-eq-length not-le not-less-eq part*(*1*) *part*(*3*) *part*(*5*) *pre*(*2*) *zero-less-Suc*)
next
fix *j* **assume** ‹*p* < *j*› ‹*j* ≤ *hi*›
then show ‹*R* (*h* (*xs''* ! *p*)) (*h* (*xs''* ! *j*))›

This holds because this part hasn't changed

by (*smt* (*verit*) *IH1*(*4*) *add-diff-cancel-left' add-diff-inverse-nat diff-Suc-eq-diff-pred diff-le-self ifs*(*1*) *isPartition-wrt-def le-less-Suc-eq less-le-trans mset-eq-length nat-less-le part*(*1*) *part*(*3*) *part*(*4*) *plus-1-eq-Suc pre*(*2*))
qed

We still have a partition after the second call (similar as in case 3)

have *still-partition2*: ‹*isPartition-map R h xs''' lo hi p*›
proof(*intro isPartition-wrtI*)
fix *i* **assume** ‹*lo* ≤ *i*› ‹*i* < *p*›
show ‹*R* (*h* (*xs'''* ! *i*)) (*h* (*xs'''* ! *p*))›

This holds because this part hasn't changed

using *IH2*(*3*) ‹*i* < *p*› ‹*lo* ≤ *i*› *isPartition-wrt-def still-partition1* **by** *fastforce*
next
fix *j* **assume** ‹*p* < *j*› ‹*j* ≤ *hi*›

Obtain the position *posJ* where *xs'''* ! *j* was stored in *xs'''*.

```
    have ‹xs'''!j ∈ set (sublist xs''' (Suc p) hi)›
      by (metis IH1(1) IH2(1) Suc-leI ‹j ≤ hi› ‹p < j› ifs(2) nat-le-linear part(1) pre(2) size-mset
sublist-el')
    then have ‹xs'''!j ∈ set (sublist xs'' (Suc p) hi)›
      by (metis IH1(1) IH2-perm ifs(2) mset-eq-length nat-le-linear part(1) pre(2))
    then have ‹∃ posJ. Suc p≤posJ∧posJ≤hi ∧ xs'''!j = xs''!posJ›
      by (metis IH1(1) ifs(2) mset-eq-length nat-le-linear part(1) pre(2) sublist-el')
    then obtain posJ :: nat where PosJ: ‹Suc p≤posJ› ‹posJ≤hi› ‹xs'''!j = xs''!posJ› by blast

    then show ‹R (h (xs''' ! p)) (h (xs''' ! j))›
    proof − — sledgehammer
      have ∀ n na as p. (p (as ! na::'a) (as ! posJ) ∨ posJ ≤ na) ∨ ¬ isPartition-wrt p as n hi na
        by (metis (no-types) PosJ(2) isPartition-wrt-def not-less)
      then show ?thesis
        by (metis IH2(3) PosJ(1) PosJ(3) lessI not-less-eq-eq still-partition1)
    qed
  qed
```

We have that the lower part is sorted after the first recursive call

```
  note sorted-lower1 = IH1(2)
```

We show that it is still sorted after the second call.

```
  have sorted-lower2: ‹sorted-sublist-map R h xs''' lo (p−Suc 0)›
  proof −
    show ?thesis
      using sorted-lower1 apply (rule sorted-wrt-lower-sublist-still-sorted)
      subgoal by (rule part)
      subgoal
        using IH1(1) mset-eq-length part(1) part(5) pre(2) by fastforce
      subgoal
        by (simp add: IH2(3))
      subgoal
        by (metis IH2(1) size-mset)
      done
  qed
```

The second IH gives us the the upper list is sorted after the second recursive call

```
  note sorted-upper2 = IH2(2)
```

Finally, we have to show that the entire list is sorted after the second recursive call.

```
  have sorted-middle: ‹sorted-sublist-map R h xs''' lo hi›
  proof −
    show ?thesis
      apply (rule merge-sorted-map-partitions[where p=p])
      subgoal by (rule trans)
      subgoal by (rule still-partition2)
      subgoal by (rule sorted-lower2)
      subgoal by (rule sorted-upper2)
      subgoal using pre(1) by auto
      subgoal by (simp add: part(4))
      subgoal by (simp add: part(5))
      subgoal by (metis IH1(1) IH2(1) part(1) pre(2) size-mset)
      done
  qed
```

show *?thesis*
proof (*intro quicksort-postI*)
show ‹*mset xs''' = mset xs*›
by (*simp add*: *IH1*(*1*) *IH2*(*1*) *part*(*1*))
next
show ‹*sorted-sublist-map R h xs''' lo hi*›
by (*rule sorted-middle*)
next
show ‹⋀*i*. *i* < *lo* ⟹ *xs'''* ! *i* = *xs* ! *i*›
using *IH1*(*3*) *IH2*(*3*) *part*(*4*) *part*(*6*) **by** *auto*
next
show ‹⋀*j*. *hi* < *j* ⟹ *j* < *length xs* ⟹ *xs'''* ! *j* = *xs* ! *j*›
by (*metis IH1*(*1*) *IH1*(*4*) *IH2*(*4*) *diff-le-self ifs*(*2*) *le-SucI less-le-trans nat-le-eq-or-lt not-less part*(*1*) *part*(*7*) *size-mset*)
qed

qed

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*(λ*xs*. *quicksort-post R h lo0 hi0 xs0 xs*))›
proof −
have *wf*: ‹*wf* (*measure* (λ(*lo*, *hi*, *xs*). *Suc hi* − *lo*))›
by *auto*
define *pre* **where** ‹*pre* = (λ(*lo*,*hi*,*xs*). *quicksort-pre R h xs0 lo hi xs*)›
define *post* **where** ‹*post* = (λ(*lo*,*hi*,*xs*). *quicksort-post R h lo hi xs*)›
have *pre*: ‹*pre* (*lo0*,*hi0*,*xs0*)›
unfolding *quicksort-pre-def pre-def* **by** (*simp add*: *Pre*)

We first generalize the goal a over all states.

have ‹*WB-Sort.quicksort R h* (*lo0*,*hi0*,*xs0*) ≤ ⇓ *Id* (*SPEC* (*post* (*lo0*,*hi0*,*xs0*)))›
unfolding *quicksort-def prod.case*
apply (*rule RECT-rule*)
apply (*refine-mono*)
apply (*rule wf*)
apply (*rule pre*)
subgoal premises *IH* **for** *f x*
apply (*refine-vcg ASSERT-leI*)
unfolding *pre-def post-def*

subgoal — First premise (assertion) for partition
using *IH*(*2*) **by** (*simp add*: *quicksort-pre-def pre-def*)
subgoal — Second premise (assertion) for partition
using *IH*(*2*) **by** (*simp add*: *quicksort-pre-def pre-def*)
subgoal
using *IH*(*2*) **by** (*auto simp add*: *quicksort-pre-def pre-def dest*: *mset-eq-setD*)

Termination case: *p* − *1* ≤ *lo'* and *hi'* ≤ *p* + *1*; directly show postcondition

subgoal unfolding *partition-spec-def* **by** (*auto dest*: *mset-eq-setD*)
subgoal — Postcondition (after partition)
apply −

using *IH*(*2*) **unfolding** *pre-def* **apply** (*simp*, *elim conjE*, *split prod.splits*)
using *trans lin* **apply** (*rule quicksort-correct-case1*) **by** *auto*

Case $p - 1 \leq lo'$ and $hi' < p + 1$ (Only second recursive call)

subgoal
apply (*rule IH*(*1*)[*THEN order-trans*])

Show that the invariant holds for the second recursive call

subgoal
using *IH*(*2*) **unfolding** *pre-def* **apply** (*simp*, *elim conjE*, *split prod.splits*)
apply (*rule quicksort-correct-case2*) **by** *auto*

Wellfoundness (easy)

subgoal by (*auto simp add*: *quicksort-pre-def partition-spec-def*)

Show that the postcondition holds

subgoal
apply (*simp add*: *Misc.subset-Collect-conv post-def*, *intro allI impI*, *elim conjE*)
using *trans lin* **apply** (*rule quicksort-correct-case3*)
using *IH*(*2*) **unfolding** *pre-def* **by** *auto*
done

Case: At least the first recursive call

subgoal
apply (*rule IH*(*1*)[*THEN order-trans*])

Show that the precondition holds for the first recursive call

subgoal
using *IH*(*2*) **unfolding** *pre-def post-def* **apply** (*simp*, *elim conjE*, *split prod.splits*) **apply** *auto*
apply (*rule quicksort-correct-case4*) **by** *auto*

Wellfoundness for first recursive call (easy)

subgoal by (*auto simp add*: *quicksort-pre-def partition-spec-def*)

Simplify some refinement suff...

apply (*simp add*: *Misc.subset-Collect-conv ASSERT-leI*, *intro allI impI conjI*, *elim conjE*)
apply (*rule ASSERT-leI*)
apply (*simp-all add*: *Misc.subset-Collect-conv ASSERT-leI*)
subgoal unfolding *quicksort-post-def pre-def post-def* **by** (*auto dest*: *mset-eq-setD*)

Only the first recursive call: show postcondition

subgoal
using *trans lin* **apply** (*rule quicksort-correct-case5*)
using *IH*(*2*) **unfolding** *pre-def post-def* **by** *auto*

apply (*rule ASSERT-leI*)
subgoal unfolding *quicksort-post-def pre-def post-def* **by** (*auto dest*: *mset-eq-setD*)

Both recursive calls.

subgoal
apply (*rule IH*(*1*)[*THEN order-trans*])

Show precondition for second recursive call (after the first call)

subgoal
unfolding *pre-def post-def*
apply *auto*
apply (*rule quicksort-correct-case6*)
using *IH*(*2*) **unfolding** *pre-def post-def* **by** *auto*

Wellfoundedness for second recursive call (easy)

subgoal by (*auto simp add*: *quicksort-pre-def partition-spec-def*)

Show that the postcondition holds (after both recursive calls)

subgoal
apply (*simp add*: *Misc.subset-Collect-conv*, *intro allI impI*, *elim conjE*)
using *trans lin* **apply** (*rule quicksort-correct-case7*)
using *IH*(*2*) **unfolding** *pre-def post-def* **by** *auto*
done
done
done
done

Finally, apply the generalized lemma to show the thesis.

then show *?thesis* **unfolding** *post-def* **by** *auto*
qed

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* are smaller than the pivot
(∀ *k*. *k* ≥ *i* ∧ *k* < *j* ⟶ *R* (*h* (*xs*!*hi*)) (*h* (*xs*!*k*))) ∧ — All elements from *i* to *j* − *1* 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*.
(λ(*i*,*j*,*xs*). *j* < *hi*)
(λ(*i*,*j*,*xs*). *do* {
ASSERT(*i* < *length xs* ∧ *j* < *length xs*);
if R (*h* (*xs*!*j*)) *pivot*

```
      then RETURN (i+1, j+1, swap xs i j)
      else RETURN (i,   j+1, xs)
     })
     (lo, lo, xs0); — i and j are both initialized to lo
   ASSERT(i < length xs ∧ j = hi ∧ lo ≤ i ∧ hi < length xs ∧ mset xs = mset xs0);
   RETURN (swap xs i hi, i)
 }›

lemma partition-main-correct:
 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 ‹partition-main R h lo hi xs ≤ SPEC(λ(xs', p). mset xs = mset xs' ∧
   lo ≤ p ∧ p ≤ hi ∧ isPartition-map R h xs' lo hi p ∧ (∀ i. i<lo ⟶ xs'!i=xs!i) ∧ (∀ i. hi<i∧i<length
xs' ⟶ xs'!i=xs!i))›
proof −
 have K: ‹b ≤ hi − Suc n ⟹ n > 0 ⟹ Suc n ≤ hi ⟹ Suc b ≤ hi − n› for b hi n
   by auto
 have L: ‹~ R (h x) (h y) ⟹ R (h y) (h x)› for x y — Corollary of linearity
   using assms by blast
 have M: ‹a < Suc b ≡ a = b ∨ a < b› for a b
   by linarith
 have N: ‹(a::nat) ≤ b ≡ a = b ∨ a < b› for a b
   by arith

 show ?thesis
   unfolding partition-main-def choose-pivot-def
   apply (refine-vcg WHILEIT-rule[where R = ‹measure(λ(i,j,xs). hi−j)›])
   subgoal using assms by blast — We feed our assumption to the assertion
   subgoal by auto — WF
   subgoal — Invariant holds before the first iteration
     unfolding partition-main-inv-def
     using assms apply simp by linarith
   subgoal unfolding partition-main-inv-def by simp
   subgoal unfolding partition-main-inv-def by simp
   subgoal
     unfolding partition-main-inv-def
     apply (auto dest: mset-eq-length)
     done
   subgoal unfolding partition-main-inv-def by (auto dest: mset-eq-length)
   subgoal
     unfolding partition-main-inv-def apply (auto dest: mset-eq-length)
     by (metis L M mset-eq-length nat-le-eq-or-lt)

   subgoal unfolding partition-main-inv-def by simp — assertions, etc
   subgoal unfolding partition-main-inv-def by simp
   subgoal unfolding partition-main-inv-def by (auto dest: mset-eq-length)
   subgoal unfolding partition-main-inv-def by simp
   subgoal unfolding partition-main-inv-def by (auto dest: mset-eq-length)
   subgoal unfolding partition-main-inv-def by (auto dest: mset-eq-length)
   subgoal unfolding partition-main-inv-def by (auto dest: mset-eq-length)
   subgoal unfolding partition-main-inv-def by simp
   subgoal unfolding partition-main-inv-def by simp
```

subgoal — After the last iteration, we have a partitioning! :-)
unfolding *partition-main-inv-def* **by** (*auto simp add*: *isPartition-wrt-def*)
subgoal — And the lower out-of-bounds parts of the list haven't been changed
unfolding *partition-main-inv-def* **by** *auto*
subgoal — And the upper out-of-bounds parts of the list haven't been changed
unfolding *partition-main-inv-def* **by** *auto*
done
qed

definition *partition-between* :: ‹('*b* ⇒ '*b* ⇒ *bool*) ⇒ ('*a* ⇒ '*b*) ⇒ *nat* ⇒ *nat* ⇒ '*a list* ⇒ ('*a list* × *nat*) *nres*› **where**
‹*partition-between R h lo hi xs0* = *do* {
ASSERT(*hi* < *length xs0* ∧ *lo* ≤ *hi*);
k ← *choose-pivot R h xs0 lo hi*; — choice of pivot
ASSERT(*k* < *length xs0*);
xs ← *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-between-correct*:
assumes ‹*hi* < *length xs*› **and** ‹*lo* ≤ *hi*› **and**
‹⋀ *x y z*. ⟦*R* (*h x*) (*h y*); *R* (*h y*) (*h z*)⟧ ⟹ *R* (*h x*) (*h z*)› **and** ‹⋀*x y*. *R* (*h x*) (*h y*) ∨ *R* (*h y*) (*h x*)›
shows ‹*partition-between R h lo hi xs* ≤ *SPEC*(*uncurry* (*partition-spec R h xs lo hi*))›
proof −
have *K*: ‹*b* ≤ *hi* − *Suc n* ⟹ *n* > *0* ⟹ *Suc n* ≤ *hi* ⟹ *Suc b* ≤ *hi* − *n*› **for** *b hi n*
by *auto*
show *?thesis*
unfolding *partition-between-def choose-pivot-def*
apply (*refine-vcg partition-main-correct*)
using *assms* **apply** (*auto dest*: *mset-eq-length simp add*: *partition-spec-def*)
by (*metis dual-order.strict-trans2 less-imp-not-eq2 mset-eq-length swap-nth*)
qed

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* ∧ *R mid end*) ∨ (*R end mid* ∧ *R mid start*) *then RETURN k*
else if (*R start end* ∧ *R end mid*) ∨ (*R mid end* ∧ *R end start*) *then RETURN hi*
else RETURN lo
}›

— 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* ≤ ⇓ *Id* (*choose-pivot R h xs lo hi*)›
unfolding *choose-pivot3-def choose-pivot-def*
using *assms* **by** (*auto intro*!: *ASSERT-leI simp*: *Let-def*)

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

definition *partition-between-ref*
:: ‹('*b* ⇒ '*b* ⇒ *bool*) ⇒ ('*a* ⇒ '*b*) ⇒ *nat* ⇒ *nat* ⇒ '*a list* ⇒ ('*a list* × *nat*) *nres*›
where
‹*partition-between-ref R h lo hi xs0* = *do* {
ASSERT(*hi* < *length xs0* ∧ *hi* < *length xs0* ∧ *lo* ≤ *hi*);
k ← *choose-pivot3 R h xs0 lo hi*; — choice of pivot
ASSERT(*k* < *length xs0*);
xs ← *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*
≤ ⇓ ((λ *a b c d*. *Id*) *a b c d*) (*partition-main R h lo hi xs*)›
by *auto*

lemma *Down-id-eq*:
‹⇓*Id x* = *x*›
by *auto*

lemma *partition-between-ref-partition-between*:
‹*partition-between-ref R h lo hi xs* ≤ (*partition-between R h lo hi xs*)›
proof −
have *swap*: ‹(*swap xs k hi*, *swap xs ka hi*) ∈ *Id*› **if** ‹*k* = *ka*›
for *k ka*
using *that* **by** *auto*
have [*refine0*]: ‹(*h* (*xsa* ! *hi*), *h* (*xsaa* ! *hi*)) ∈ *Id*›
if ‹(*xsa*, *xsaa*) ∈ *Id*›
for *xsa xsaa*
using *that* **by** *auto*

show *?thesis*
apply (*subst* (*2*) *Down-id-eq*[*symmetric*])
unfolding *partition-between-ref-def*
partition-between-def
OP-def
apply (*refine-vcg choose-pivot3-choose-pivot swap partition-main-correct*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*

subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
by (*auto intro*: *Refine-Basic.Id-refine dest*: *mset-eq-length*)
qed

Technical lemma for sepref

lemma *partition-between-ref-partition-between'*:
‹(*uncurry2* (*partition-between-ref R h*), *uncurry2* (*partition-between R h*)) ∈
(*nat-rel* ×$_r$ *nat-rel*) ×$_r$ ⟨*Id*⟩*list-rel* →$_f$ ⟨⟨*Id*⟩*list-rel* ×$_r$ *nat-rel*⟩*nres-rel*›
by (*intro frefI nres-relI*)
(*auto intro*: *partition-between-ref-partition-between*)

Example instantiation for pivot

definition *choose-pivot3-impl* **where**
‹*choose-pivot3-impl* = *choose-pivot3* (≤) *id*›

lemma *partition-between-ref-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*)›
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 −
show *?thesis*
apply (*rule partition-between-ref-partition-between*[*THEN order-trans*])
using *bounds* **apply** (*rule partition-between-correct*[**where** *h*=*h*])
subgoal by (*rule trans*)
subgoal by (*rule lin*)
done
qed

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'*))›
unfolding *fref-def uncurry-def nres-rel-def*

by *auto*

lemma *fref-to-Down-curry*:
‹$(f,\ g) \in [P]_f\ A \to \langle B\rangle nres\text{-}rel \Longrightarrow$
$(\bigwedge x\ x'\ .\ P\ x' \Longrightarrow (x,\ x') \in A \Longrightarrow$
$f\ x\ \le \Downarrow B\ (g\ x'))$›
unfolding *fref-def uncurry-def nres-rel-def*
by *auto*

lemma *quicksort-ref-quicksort*:
assumes *bounds*: ‹*hi* < *length xs*› ‹*lo* ≤ *hi*› **and**
trans: ‹$\bigwedge$ *x y z*. ⟦*R* (*h x*) (*h y*); *R* (*h y*) (*h z*)⟧ ⟹ *R* (*h x*) (*h z*)› **and** *lin*: ‹$\bigwedge$*x y*. *R* (*h x*) (*h y*) ∨ *R* (*h y*) (*h x*)›
shows ‹*quicksort-ref R h x0* ≤ ⇓ *Id* (*quicksort R h x0*)›
proof −
have *wf*: ‹*wf* (*measure* (λ(*lo*, *hi*, *xs*). *Suc hi* − *lo*))›
by *auto*
have *pre*: ‹*x0* = *x0'* ⟹ (*x0*, *x0'*) ∈ *Id* ×$_r$ *Id* ×$_r$ ⟨*Id*⟩*list-rel*› **for** *x0 x0'* :: ‹*nat* × *nat* × *'b list*›
by *auto*
have [*refine0*]: ‹(*x1e* = *x1d*) ⟹ (*x1e*,*x1d*) ∈ *Id*› **for** *x1e x1d* :: ‹*'b list*›
by *auto*

show *?thesis*
unfolding *quicksort-def quicksort-ref-def*
apply (*refine-vcg pre partition-between-ref-partition-between'*[*THEN fref-to-Down-curry2*])

First assertion (premise for partition)

subgoal
by *auto*

First assertion (premise for partition)

subgoal
by *auto*
subgoal
by (*auto dest*: *mset-eq-length*)
subgoal
by (*auto dest*: *mset-eq-length mset-eq-setD*)

Correctness of the concrete partition function

subgoal
apply (*simp*, *rule partition-between-ref-correct*)
subgoal by (*rule trans*)
subgoal by (*rule lin*)
subgoal by *auto* — first premise
subgoal by *auto* — second premise
done
subgoal
by (*auto dest*: *mset-eq-length mset-eq-setD*)
subgoal by (*auto simp*: *partition-spec-def isPartition-wrt-def*)
subgoal by (*auto simp*: *partition-spec-def isPartition-wrt-def dest*: *mset-eq-length*)
subgoal
by (*auto dest*: *mset-eq-length mset-eq-setD*)
subgoal

```
    by (auto dest: mset-eq-length mset-eq-setD)
  subgoal
    by (auto dest: mset-eq-length mset-eq-setD)
  subgoal
    by (auto dest: mset-eq-length mset-eq-setD)

  by simp+
qed

— Sort the entire list
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 −
  show ?thesis
    unfolding full-quicksort-ref-def full-quicksort-def
    using lin
    apply (intro frefI nres-relI)
    apply (auto intro!: quicksort-ref-quicksort[unfolded Down-id-eq] intro: trans)
    done
qed


lemma sublist-entire:
  ‹sublist xs 0 (length xs − 1) = xs›
  by (simp add: sublist-def)


lemma sorted-sublist-wrt-entire:
  assumes ‹sorted-sublist-wrt R xs 0 (length xs − 1)›
  shows ‹sorted-wrt R xs›
proof −
  have ‹sorted-wrt R (sublist xs 0 (length xs − 1))›
    using assms by (simp add: sorted-sublist-wrt-def )
  then show ?thesis
    by (metis sublist-entire)
qed

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 −
```

show *?thesis*
using *assms* **by** (*rule sorted-sublist-wrt-entire*)
qed

Final correctness lemma

theorem *full-quicksort-correct-sorted*:
assumes
trans: ‹$\bigwedge x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)$› **and** *lin*: ‹$\bigwedge x\ y.\ x \neq y \Longrightarrow R\ (h\ x)\ (h\ y) \vee R\ (h\ y)\ (h\ x)$›
shows ‹*full-quicksort R h xs* $\leq \Downarrow$ *Id* (*SPEC*($\lambda xs'.$ *mset* $xs' =$ *mset xs* $\wedge$ *sorted-wrt* ($\lambda\ x\ y.\ R\ (h\ x)\ (h\ y)$) xs'))›
proof −
show *?thesis*
unfolding *full-quicksort-def*
apply (*refine-vcg*)
subgoal by *simp* — case xs=[]
subgoal by *simp* — case xs=[]

apply (*rule quicksort-correct*[*THEN order-trans*])
subgoal by (*rule trans*)
subgoal by (*rule lin*)
subgoal by *linarith*
subgoal by *simp*

apply (*simp add*: *Misc.subset-Collect-conv*, *intro allI impI conjI*)
subgoal
by (*auto simp add*: *quicksort-post-def*)
subgoal
apply (*rule sorted-sublist-map-entire*)
by (*auto simp add*: *quicksort-post-def dest*: *mset-eq-length*)
done
qed

lemma *full-quicksort-correct*:
assumes
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 ‹*full-quicksort R h xs* $\leq \Downarrow$ *Id* (*SPEC*($\lambda xs'.$ *mset* $xs' =$ *mset xs*))›
by (*rule order-trans*[*OF full-quicksort-correct-sorted*])
(*use assms* **in** *auto*)

end

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

3.3 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*›
by *auto*

lemma *while-upt-while-direct1*:
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
}
apply (*rewrite at* ‹- ≤ ⌑› *Down-id-eq*[*symmetric*])
apply (*refine-vcg WHILET-refine*[**where** *R* = ‹{((*l*, *x'*), (*i*::*nat*, *x*::'*a*)). *x*= *x'* ∧ *i* ≤ *b* ∧ *i* ≥ *a* ∧ *l* = *drop* (*i*−*a*) [*a..<b*]}›])
subgoal by *auto*
subgoal by (*auto simp*: *FOREACH-cond-def*)
subgoal by (*auto simp*: *FOREACH-body-def intro*!: *bind-refine*[*OF Id-refine*])
subgoal by *auto*
done

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
}
apply (*rewrite at* ‹- ≤ ⌑› *Down-id-eq*[*symmetric*])
apply (*refine-vcg WHILET-refine*[**where** *R* = ‹{((*i*::*nat*, *x*::'*a*), (*l*, *x'*)). *x*= *x'* ∧ *i* ≤ *b* ∧ *i* ≥ *a* ∧ *l* = *drop* (*i*−*a*) [*a..<b*]}›])
subgoal by *auto*
subgoal by (*auto simp*: *FOREACH-cond-def*)
subgoal by (*auto simp*: *FOREACH-body-def intro*!: *bind-refine*[*OF Id-refine*])
subgoal by (*auto simp*: *FOREACH-body-def intro*!: *bind-refine*[*OF Id-refine*])
subgoal by *auto*
done

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
}
apply (*rewrite at* ‹- ≤ ⌑› *Down-id-eq*[*symmetric*])
apply (*refine-vcg WHILET-refine*[**where** *R* = ‹{((*i*::*nat*, *x*::'*a*), (*l*, *x'*)). *x*= *x'* ∧ *i* ≤ *b* ∧ *i* ≥ *a* ∧ *l* = *drop* (*i*−*a*) [*a..<b*]}›])
subgoal by *auto*
subgoal by (*auto simp*: *FOREACH-cond-def*)
subgoal by (*auto simp*: *FOREACH-body-def intro*!: *bind-refine*[*OF Id-refine*])
subgoal by (*auto simp*: *FOREACH-body-def intro*!: *bind-refine*[*OF Id-refine*])
subgoal by *auto*
done

```
}) (a,σ);
    RETURN σ
  }
  using while-upt-while-direct1[of a b] while-upt-while-direct2[of a b]
  unfolding order-eq-iff by fast

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
  apply (induct l arbitrary: σ)
  apply (subst WHILET-unfold)
  apply (simp add: FOREACH-cond-def)

  apply (subst WHILET-unfold)
  apply (auto
    simp: FOREACH-cond-def FOREACH-body-def
    intro: bind-mono Refine-Basic.bind-mono(1))
 done
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 (induct l arbitrary: σ)
  case Nil thus ?case by (subst WHILEIT-unfold) (auto simp: FOREACH-cond-def)
next
  case (Cons x ls)
  show ?case
  proof (cases c σ)
    case False thus ?thesis
      apply (subst WHILEIT-unfold)
      unfolding FOREACH-cond-def
      by simp
  next
    case [simp]: True
    from Cons show ?thesis
      apply (subst WHILEIT-unfold)
      unfolding FOREACH-cond-def FOREACH-body-def
      apply clarsimp
      apply (rule Refine-Basic.bind-mono)
      apply simp-all
      done
  qed
qed

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
  apply (rule antisym)
```

apply (*rule while-nfoldli*)
apply (*rule order-trans*[*OF nfoldli-while*[**where** *I*=λ-. *True*]])
apply (*simp add*: *WHILET-def*)
done

end

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

4 Specification of the PAC checker

4.1 Ideals

type-synonym *int-poly* = ‹*int mpoly*›
definition *polynomial-bool* :: ‹*int-poly set*› **where**
‹*polynomial-bool* = (λ*c*. *Var c* ^ 2 − *Var c*) ‘ *UNIV*›

definition *pac-ideal* **where**
‹*pac-ideal A* ≡ *ideal* (*A* ∪ *polynomial-bool*)›

lemma *X2-X-in-pac-ideal*:
‹*Var c* ^ 2 − *Var c* ∈ *pac-ideal A*›
unfolding *polynomial-bool-def pac-ideal-def*
by (*auto intro*: *ideal.span-base*)

lemma *pac-idealI1*[*intro*]:
‹*p* ∈ *A* ⟹ *p* ∈ *pac-ideal A*›
unfolding *pac-ideal-def*
by (*auto intro*: *ideal.span-base*)

lemma *pac-idealI2*[*intro*]:
‹*p* ∈ *ideal A* ⟹ *p* ∈ *pac-ideal A*›
using *ideal.span-subspace-induct pac-ideal-def* **by** *blast*

lemma *pac-idealI3*[*intro*]:
‹*p* ∈ *ideal A* ⟹ *p*∗*q* ∈ *pac-ideal A*›
by (*metis ideal.span-scale mult.commute pac-idealI2*)

lemma *pac-ideal-Xsq2-iff*:
‹*Var c* ^ 2 ∈ *pac-ideal A* ⟷ *Var c* ∈ *pac-ideal A*›
unfolding *pac-ideal-def*
apply (*subst* (2) *ideal.span-add-eq*[*symmetric*, *OF X2-X-in-pac-ideal*[*of c*, *unfolded pac-ideal-def*]])
apply *auto*
done

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 −
have *insert p polynomial-bool* ⊆ *pac-ideal A*
using *a1* **unfolding** *pac-ideal-def* **by** (*meson ideal.span-superset insert-subset le-sup-iff*)
then show *?thesis*

using *a2* **unfolding** *pac-ideal-def* **by** (*metis* (*no-types*) *ideal.eq-span-insert-eq ideal.span-subset-spanI ideal.span-superset insert-subset subsetD*)
qed

lemma *diff-in-polynomial-bool-pac-idealI2*:
assumes *a1*: $p \in A$
assumes *a2*: $p - p' \in$ *More-Modules.ideal polynomial-bool*
shows ‹$p' \in$ *pac-ideal A*›
using *diff-in-polynomial-bool-pac-idealI*[*OF - assms*(*2*), *of A*] *assms*(*1*)
by (*auto simp*: *ideal.span-base*)

lemma *pac-ideal-alt-def*:
‹*pac-ideal A* = *ideal* (*A* ∪ *ideal polynomial-bool*)›
unfolding *pac-ideal-def*
by (*meson ideal.span-eq ideal.span-mono ideal.span-superset le-sup-iff subset-trans sup-ge2*)

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*$_{VI}$ **where**
‹*restricted-ideal-to*$_{VI}$ *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*›
unfolding *restricted-ideal-to-def*
by *auto*

lemma *pac-ideal-insert-already-in*:
‹*pq* ∈ *pac-ideal* (*set-mset A*) ⟹ *pac-ideal* (*insert pq* (*set-mset A*)) = *pac-ideal* (*set-mset A*)›
by (*auto simp*: *pac-ideal-alt-def ideal.span-insert-idI*)

lemma *pac-ideal-add*:
‹*p* ∈# *A* ⟹ *q* ∈# *A* ⟹ *p* + *q* ∈ *pac-ideal* (*set-mset A*)›
by (*simp add*: *ideal.span-add ideal.span-base pac-ideal-def*)
lemma *pac-ideal-mult*:
‹*p* ∈# *A* ⟹ *p* ∗ *q* ∈ *pac-ideal* (*set-mset A*)›
by (*simp add*: *ideal.span-base pac-idealI3*)

lemma *pac-ideal-mono*:
‹*A* ⊆ *B* ⟹ *pac-ideal A* ⊆ *pac-ideal B*›
using *ideal.span-mono*[*of* ‹*A* ∪ -› ‹*B* ∪ -›]
by (*auto simp*: *pac-ideal-def intro*: *ideal.span-mono*)

4.2 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* ($\mathcal{V}$, *A*) ($\mathcal{V}$, *add-mset p′ A*)›
if
‹*p* ∈# *A*› ‹*q* ∈# *A*›
‹*p*+*q* − *p′* ∈ *ideal polynomial-bool*›
‹*vars p′* ⊆ $\mathcal{V}$› |
mult:
‹*PAC-Format* ($\mathcal{V}$, *A*) ($\mathcal{V}$, *add-mset p′ A*)›
if
‹*p* ∈# *A*›
‹*p*∗*q* − *p′* ∈ *ideal polynomial-bool*›
‹*vars p′* ⊆ $\mathcal{V}$›
‹*vars q* ⊆ $\mathcal{V}$› |
del:
‹*p* ∈# *A* ⟹ *PAC-Format* ($\mathcal{V}$, *A*) ($\mathcal{V}$, *A* − {#*p*#})› |
extend-pos:
‹*PAC-Format* ($\mathcal{V}$, *A*) ($\mathcal{V}$ ∪ {*x′* ∈ *vars* (−*Var x* + *p′*). *x′* ∉ $\mathcal{V}$}, *add-mset* (−*Var x* + *p′*) *A*)›
if
‹(*p′*)² − *p′* ∈ *ideal polynomial-bool*›
‹*vars p′* ⊆ $\mathcal{V}$›
‹*x* ∉ $\mathcal{V}$›

In the PAC format above, we have a technical condition on the normalisation: *vars* $p' \subseteq$ *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 process 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* ($\mathcal{V}$, *A*) ($\mathcal{V}'$, *A′*)› **and**
cases:
‹⋀*p q p′ A* $\mathcal{V}$. *p* ∈# *A* ⟹ *q* ∈# *A* ⟹ *p*+*q* − *p′* ∈ *ideal polynomial-bool* ⟹ *vars p′* ⊆ $\mathcal{V}$ ⟹ *P* $\mathcal{V}$ *A* $\mathcal{V}$ (*add-mset p′ A*)›
‹⋀*p q p′ A* $\mathcal{V}$. *p* ∈# *A* ⟹ *p*∗*q* − *p′* ∈ *ideal polynomial-bool* ⟹ *vars p′* ⊆ $\mathcal{V}$ ⟹ *vars q* ⊆ $\mathcal{V}$ ⟹ *P* $\mathcal{V}$ *A* $\mathcal{V}$ (*add-mset p′ A*)›
‹⋀*p A* $\mathcal{V}$. *p* ∈# *A* ⟹ *P* $\mathcal{V}$ *A* $\mathcal{V}$ (*A* − {#*p*#})›
‹⋀*p′ x r*.
(*p′*)^2 − (*p′*) ∈ *ideal polynomial-bool* ⟹ *vars p′* ⊆ $\mathcal{V}$ ⟹
x ∉ $\mathcal{V}$ ⟹ *P* $\mathcal{V}$ *A* ($\mathcal{V}$ ∪ {*x′* ∈ *vars* (*p′* − *Var x*). *x′* ∉ $\mathcal{V}$}) (*add-mset* (*p′* − *Var x*) *A*)›
shows
‹*P* $\mathcal{V}$ *A* $\mathcal{V}'$ *A′*›

```
  using assms(1) apply −
  by (induct V≡𝒱 A≡A 𝒱' A' rule: PAC-Format-induct-split)
   (auto intro: assms(1) cases)
```

The theorem below (based on the proof ideal by Manuel Kauers) is the correctness theorem of extensions. Remark that the assumption $vars\ q \subseteq \mathcal{V}$ 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 −
  define p' where ‹p' ≡ p − Var x'›
  let ?v = ‹Var x' :: int mpoly›
  have p-p': ‹p = ?v + p'›
    by (auto simp: p'-def)
  define q' where ‹q' ≡ Var x' − p›
  have q-q': ‹p = ?v − q'›
    by (auto simp: q'-def)
  have diff: ‹q'^2 − q' ∈ More-Modules.ideal polynomial-bool›
    using diff unfolding q-q' by auto

  have [simp]: ‹vars ((Var c)^2 − Var c :: int mpoly) = {c}› for c
    apply (auto simp: vars-def Var-def Var0-def mpoly.MPoly-inverse keys-def lookup-minus-fun
      lookup-times-monomial-right single.rep-eq split: if-splits)
    apply (auto simp: vars-def Var-def Var0-def mpoly.MPoly-inverse keys-def lookup-minus-fun
      lookup-times-monomial-right single.rep-eq when-def ac-simps adds-def lookup-plus-fun
      power2-eq-square times-mpoly.rep-eq minus-mpoly.rep-eq split: if-splits)
    apply (rule-tac x = ‹(2 :: nat ⇒₀ nat) + monomial (Suc 0) c› in exI)
    apply (auto dest: monomial-0D simp: plus-eq-zero-2 lookup-plus-fun mult-2)
    by (meson Suc-neq-Zero monomial-0D plus-eq-zero-2)


  have eq: ‹More-Modules.ideal (insert p (set-mset A ∪ polynomial-bool)) =
      More-Modules.ideal (insert p (set-mset A ∪ (λc. Var c ^ 2 − Var c) ` {c. c ≠ x'}))›
      (is ‹?A = ?B› is ‹- = More-Modules.ideal ?trimmed›)
  proof −
    let ?C = ‹insert p (set-mset A ∪ (λc. Var c ^ 2 − Var c) ` {c. c ≠ x'})›
    let ?D = ‹(λc. Var c ^ 2 − Var c) ` {c. c ≠ x'}›
    have diff: ‹q'^2 − q' ∈ More-Modules.ideal ?D› (is ‹?q ∈ -›)
    proof −
      obtain r t where
        q: ‹?q = (∑a∈t. r a * a)› and
        fin-t: ‹finite t› and
        t: ‹t ⊆ polynomial-bool›
        using diff unfolding ideal.span-explicit
        by auto
      show ?thesis
      proof (cases ‹?v^2−?v ∉ t›)
```

```
    case True
    then show ‹?thesis›
      using q fin-t t unfolding ideal.span-explicit
      by (auto intro!: exI[of - ‹t − {?v^2 −?v}›] exI[of - r]
        simp: polynomial-bool-def sum-diff1)
  next
    case False
    define t' where ‹t' = t − {?v^2 − ?v}›
    have t-t': ‹t = insert (?v^2 − ?v) t'› and
      notin: ‹?v^2 − ?v ∉ t'› and
      ‹t' ⊆ (λc. Var c ^ 2 − Var c) ` {c. c ≠ x'}›
      using False t unfolding t'-def polynomial-bool-def by auto
    have mon: ‹monom (monomial (Suc 0) x') 1 = Var x'›
      by (auto simp: coeff-def minus-mpoly.rep-eq Var-def Var_0-def monom-def
        times-mpoly.rep-eq lookup-minus lookup-times-monomial-right mpoly.MPoly-inverse)
    then have ‹∀ a. ∃ g h. r a = ?v * g + h ∧ x' ∉ vars h›
      using polynomial-split-on-var[of ‹r -› x']
      by metis
    then obtain g h where
      r: ‹r a = ?v * g a + h a› and
      x'-h: ‹x' ∉ vars (h a)› for a
      using polynomial-split-on-var[of ‹r a› x']
      by metis
    have ‹?q = ((∑ a∈t'. g a * a) + r (?v^2−?v) * (?v − 1)) * ?v + (∑ a∈t'. h a * a)›
      using fin-t notin unfolding t-t' q r
      by (auto simp: field-simps comm-monoid-add-class.sum.distrib
        power2-eq-square ideal.scale-left-commute sum-distrib-left)
    moreover have ‹x' ∉ vars ?q›
      by (metis (no-types, opaque-lifting) Groups.add-ac(2) Un-iff add-diff-cancel-left'
        diff-minus-eq-add in-mono leading q'-def semiring-normalization-rules(29)
        vars-in-right-only vars-mult)
    moreover {
      have ‹x' ∉ (⋃ m∈t' − {?v^2−?v}. vars (h m * m))›
        using fin-t x'-h vars-mult[of ‹h -›] ‹t ⊆ polynomial-bool›
        by (auto simp: polynomial-bool-def t-t' elim!: vars-unE)
      then have ‹x' ∉ vars (∑ a∈t'. h a * a)›
        using vars-setsum[of ‹t'› ‹λa. h a * a›] fin-t x'-h t notin
        by (auto simp: t-t')
    }
    ultimately have ‹?q = (∑ a∈t'. h a * a)›
      unfolding mon[symmetric]
      by (rule polynomial-decomp-alien-var(2)[unfolded])
    then show ?thesis
      using t fin-t ‹t' ⊆ (λc. Var c ^ 2 − Var c) ` {c. c ≠ x'}›
      unfolding ideal.span-explicit t-t'
      by auto
  qed
qed
have eq1: ‹More-Modules.ideal (insert p (set-mset A ∪ polynomial-bool)) =
  More-Modules.ideal (insert (?v^2 − ?v) ?C)›
  (is ‹More-Modules.ideal - = More-Modules.ideal (insert - ?C)›)
  by (rule arg-cong[of - - More-Modules.ideal])
   (auto simp: polynomial-bool-def)
moreover have ‹?v^2 − ?v ∈ More-Modules.ideal ?C›
proof −
```

```
    have ‹?v − q′ ∈ More-Modules.ideal ?C›
      by (auto simp: q-q′ ideal.span-base)
   from ideal.span-scale[OF this, of ‹?v + q′ − 1›] have ‹(?v − q′) ∗ (?v + q′ − 1) ∈ More-Modules.ideal
?C›
      by (auto simp: field-simps)
    moreover have ‹q′^2 − q′ ∈ More-Modules.ideal ?C›
      using diff by (smt (verit) Un-insert-right ideal.span-mono insert-subset subsetD sup-ge2)
    ultimately have ‹(?v − q′) ∗ (?v + q′ − 1) + (q′^2 − q′) ∈ More-Modules.ideal ?C›
      by (rule ideal.span-add)
    moreover have ‹?v^2 − ?v = (?v − q′) ∗ (?v + q′ − 1) + (q′^2 − q′)›
      by (auto simp: p′-def q-q′ field-simps power2-eq-square)
    ultimately show ?thesis by simp
  qed
  ultimately show ?thesis
    using ideal.span-insert-idI by blast
 qed

 have ‹n < m ⟹ n > 0 ⟹ ∃ q. ?v^n = ?v + q ∗ (?v^2 − ?v)› for n m :: nat
 proof (induction m arbitrary: n)
  case 0
  then show ?case by auto
 next
  case (Suc m n) note IH = this(1−)
  consider
    ‹n < m› |
    ‹m = n› ‹n > 1› |
    ‹n = 1›
    using IH
    by (cases ‹n < m›; cases n) auto
  then show ?case
  proof cases
    case 1
    then show ?thesis using IH by auto
  next
    case 2
    have eq: ‹?v^(n) = ((?v :: int mpoly) ^ (n−2)) ∗ (?v^2−?v) + ?v^(n−1)›
      using 2 by (auto simp: field-simps power-eq-if
        ideal.scale-right-diff-distrib)
    obtain q where
      q: ‹?v^(n−1) = ?v + q ∗ (?v^2 − ?v)›
      using IH(1)[of ‹n−1›] 2
      by auto
    show ?thesis
      using q unfolding eq
      by (auto intro!: exI[of - ‹Var x' ^ (n − 2) + q›] simp: distrib-right)
  next
    case 3
    then show ‹?thesis›
      by auto
  qed
 qed

 obtain r t where
   q: ‹q = (∑ a∈t. r a ∗ a)› and
   fin-t: ‹finite t› and
```

```
  t: ‹t ⊆ ?trimmed›
  using q unfolding eq unfolding ideal.span-explicit
  by auto


define t' where ‹t' ≡ t − {p}›
have t': ‹t = (if p ∈ t then insert p t' else t')› and
  t''[simp]: ‹p ∉ t'›
  unfolding t'-def by auto
show ?thesis
proof (cases ‹r p = 0 ∨ p ∉ t›)
  case True
  have
    q: ‹q = (∑ a∈t'. r a * a)› and
   fin-t: ‹finite t'› and
    t: ‹t' ⊆ set-mset A ∪ polynomial-bool›
    using q fin-t t True t''
    apply (subst (asm) t')
    apply (auto intro: sum.cong simp: sum.insert-remove t'-def)
    using q fin-t t True t''
    apply (auto intro: sum.cong simp: sum.insert-remove t'-def polynomial-bool-def)
    done
  then show ?thesis
    by (auto simp: ideal.span-explicit)
next
  case False
  then have ‹r p ≠ 0› and ‹p ∈ t›
    by auto
  then have t: ‹t = insert p t'›
    by (auto simp: t'-def)

 have ‹x' ∉ vars (− p')›
   using leading p'-def vars-in-right-only by fastforce
 have mon: ‹monom (monomial (Suc 0) x') 1 = Var x'›
   by (auto simp:coeff-def minus-mpoly.rep-eq Var-def Var0-def monom-def
     times-mpoly.rep-eq lookup-minus lookup-times-monomial-right mpoly.MPoly-inverse)
 then have ‹∀ a. ∃ g h. r a = (?v + p') * g + h ∧ x' ∉ vars h›
   using polynomial-split-on-var2[of x' ‹−p'› ‹r ->]  ‹x' ∉ vars (− p')›
   by (metis diff-minus-eq-add)
 then obtain g h where
   r: ‹r a = p * g a + h a› and
   x'-h: ‹x' ∉ vars (h a)› for a
   using polynomial-split-on-var2[of x' p' ‹r a›] unfolding p-p'[symmetric]
   by metis


have ISABLLE-come-on: ‹a * (p * g a) = p * (a * g a)› for a
  by auto
have q1: ‹q = p * (∑ a∈t'. g a * a) + (∑ a∈t'. h a * a) + p * r p›
  (is ‹- = - + ?NOx' + -›)
  using fin-t t'' unfolding q t ISABLLE-come-on r
  apply (subst semiring-class.distrib-right)+
  apply (auto simp: comm-monoid-add-class.sum.distrib semigroup-mult-class.mult.assoc
    ISABLLE-come-on simp flip: semiring-0-class.sum-distrib-right
      semiring-0-class.sum-distrib-left)
```

by (*auto simp*: *field-simps*)
also have ‹... = ((∑ a∈t'. g a * a) + r p) * p + (∑ a∈t'. h a * a)›
by (*auto simp*: *field-simps*)
finally have *q-decomp*: ‹q = ((∑ a∈t'. g a * a) + r p) * p + (∑ a∈t'. h a * a)›
(**is** ‹q = ?X * p + ?NOx'›).

have [*iff*]: ‹*monomial* (*Suc* 0) c = 0 − *monomial* (*Suc* 0) c = *False*› **for** c
by (*metis One-nat-def diff-is-0-eq' le-eq-less-or-eq less-Suc-eq-le monomial-0-iff single-diff zero-neq-one*)
have ‹x ∈ t' ⟹ x' ∈ *vars* x ⟹ *False*› **for** x
using ‹t ⊆ ?trimmed› t *assms*(2,3)
apply (*auto simp*: *polynomial-bool-def dest*!: *multi-member-split*)
apply (*frule set-rev-mp*)
apply *assumption*
apply (*auto dest*!: *multi-member-split*)
done
then have ‹x' ∉ (⋃ m∈t'. *vars* (h m * m))›
using *fin-t x'-h vars-mult*[*of* ‹h -›]
by (*auto simp*: t *elim*!: *vars-unE*)
then have ‹x' ∉ *vars* ?NOx'›
using *vars-setsum*[*of* ‹t'› ‹λa. h a * a›] *fin-t x'-h*
by (*auto simp*: t)

moreover {
have ‹x' ∉ *vars* p'›
using *assms*(7)
unfolding *p'-def*
by *auto*
then have ‹x' ∉ *vars* (h p * p')›
using *vars-mult*[*of* ‹h p› p'] *x'-h*
by *auto*
}
ultimately have
‹x' ∉ *vars* q›
‹x' ∉ *vars* ?NOx'›
‹x' ∉ *vars* p'›
using x' *vars-q vars-add*[*of* ‹h p * p'› ‹∑ a∈t'. h a * a›] *x'-h*
leading p'-def
by *auto*
then have ‹?X = 0› **and** *q-decomp*: ‹q = ?NOx'›
unfolding *mon*[*symmetric*] *p-p'*
using *polynomial-decomp-alien-var2*[*OF q-decomp*[*unfolded p-p' mon*[*symmetric*]]]
by *auto*

then have ‹r p = (∑ a∈t'. (− g a) * a)›
(**is** ‹- = ?CL›)
unfolding *add.assoc add-eq-0-iff equation-minus-iff*
by (*auto simp*: *sum-negf ac-simps*)

then have *q2*: ‹q = (∑ a∈t'. a * (r a − p * g a))›
using *fin-t* **unfolding** q
apply (*auto simp*: t r q
comm-monoid-add-class.sum.distrib[*symmetric*]
sum-distrib-left

```
        sum-distrib-right
        left-diff-distrib
       intro!: sum.cong)
   apply (auto simp: field-simps)
   done
 then show ‹?thesis›
   using t fin-t ‹t ⊆ ?trimmed› unfolding ideal.span-explicit
   by (auto intro!: exI[of - t'] exI[of - ‹λa. r a − p ∗ g a›]
     simp: field-simps polynomial-bool-def)
 qed
qed

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 −
 have ‹q ∈ More-Modules.ideal (insert (− p) (set-mset A ∪ polynomial-bool))›
   by (metis ideal.span-breakdown-eq minus-mult-minus q)

 then show ?thesis
   using extensions-are-safe[of x' ‹−p› 𝒱 A q] assms
   using vars-in-right-only by force
qed
```

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›
 by (metis One-nat-def Var.abs-eq Var0-def group-eq-aux monom.abs-eq mult-numeral-1 polynomial-decomp-alien-var(1)
zero-neq-numeral)

lemma vars-subst-in-left-only-diff-iff:
 fixes p :: ‹int mpoly›
 assumes ‹x ∉ vars p›
 shows ‹vars (p − Var x) = insert x (vars p)›
proof −
 have ‹⋀xa. x ∉ vars p ⟹ xa ∈ vars (p − Var x) ⟹ xa ∉ vars p ⟹ xa = x›
   by (metis (no-types, opaque-lifting) diff-0-right diff-minus-eq-add empty-iff in-vars-addE insert-iff
     keys-single minus-diff-eq monom-one mult.right-neutral one-neq-zero single-zero
     vars-monom-keys vars-mult-Var vars-uminus)
 moreover have ‹⋀xa. x ∉ vars p ⟹ xa ∈ vars p ⟹ xa ∈ vars (p − Var x)›
   by (metis add.inverse-inverse diff-minus-eq-add empty-iff insert-iff keys-single minus-diff-eq
     monom-one mult.right-neutral one-neq-zero single-zero vars-in-right-only vars-monom-keys
     vars-mult-Var vars-uminus)
 ultimately show ?thesis
   using assms
   by (auto simp: vars-subst-in-left-only)
qed
```

lemma *vars-subst-in-left-only-iff*:
‹$x \notin$ *vars* $p \Longrightarrow$ *vars* $(p + Var\ x) =$ *insert* x (*vars* p)› **for** $p :: ‹$*int mpoly*›
using *vars-subst-in-left-only-diff-iff*[*of* x ‹$-p$›]
by (*metis diff-0 diff-diff-add vars-uminus*)

lemma *coeff-add-right-notin*:
‹$x \notin$ *vars* $p \Longrightarrow$ *MPoly-Type.coeff* $(Var\ x - p)$ (*monomial* $(Suc\ 0)\ x) = 1$›
apply (*auto simp flip*: *coeff-minus simp*: *not-in-vars-coeff0*)
by (*simp add*: *MPoly-Type.coeff-def Var.rep-eq* Var_0*-def*)

lemma *coeff-add-left-notin*:
‹$x \notin$ *vars* $p \Longrightarrow$ *MPoly-Type.coeff* $(p - Var\ x)$ (*monomial* $(Suc\ 0)\ x) = -1$› **for** $p :: ‹$*int mpoly*›
apply (*auto simp flip*: *coeff-minus simp*: *not-in-vars-coeff0*)
by (*simp add*: *MPoly-Type.coeff-def Var.rep-eq* Var_0*-def*)

lemma *ideal-insert-polynomial-bool-swap*: ‹$r - s \in$ *ideal polynomial-bool* $\Longrightarrow$
More-Modules.ideal (*insert* r $(A \cup$ *polynomial-bool*)) = *More-Modules.ideal* (*insert* s $(A \cup$ *polynomial-bool*))›
apply *auto*
using *ideal.eq-span-insert-eq ideal.span-mono sup-ge2* **apply** *blast+*
done

lemma *PAC-Format-subset-ideal*:
‹*PAC-Format* $(\mathcal{V}, A)$ $(\mathcal{V}', B) \Longrightarrow \bigcup$(*vars* ‘ *set-mset* $A) \subseteq \mathcal{V} \Longrightarrow$
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* $B) \subseteq \mathcal{V}'$›
unfolding *restricted-ideal-to-def*
apply (*induction rule*:*PAC-Format-induct*)
subgoal for p q *pq* A $\mathcal{V}$
using *vars-add*
by (*force simp*: *ideal.span-add-eq ideal.span-base pac-ideal-insert-already-in*[*OF diff-in-polynomial-bool-pac-idealI*[*of* ‹$p + q$› ‹-› *pq*]]
pac-ideal-add
intro!: *diff-in-polynomial-bool-pac-idealI*[*of* ‹$p + q$› ‹-› *pq*])
subgoal for p q *pq*
using *vars-mult*[*of* p q]
by (*force simp*: *ideal.span-add-eq ideal.span-base pac-ideal-mult*
pac-ideal-insert-already-in[*OF diff-in-polynomial-bool-pac-idealI*[*of* ‹$p*q$› ‹-› *pq*]])
subgoal for p A
using *pac-ideal-mono*[*of* ‹*set-mset* $(A - \{\#p\#\})$› ‹*set-mset* A›]
by (*auto dest*: *in-diffD*)
subgoal for p x' r'
apply (*subgoal-tac* ‹$x' \notin$ *vars* p›)
using *extensions-are-safe-uminus*[*of* x' ‹$-Var\ x' + p$› $\mathcal{V}$ A] **unfolding** *pac-ideal-def*
apply (*auto simp*: *vars-subst-in-left-only coeff-add-left-notin*)
done
done

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›
by (*auto simp add*: *Collect-disj-eq pac-idealI1 restricted-ideal-to-def*)

lemma *rtranclp-PAC-Format-subset-ideal*:

‹*rtranclp PAC-Format* ($\mathcal{V}$, *A*) ($\mathcal{V}'$, *B*) $\Longrightarrow$ $\bigcup$(*vars* ‘ *set-mset A*) $\subseteq$ $\mathcal{V}$ $\Longrightarrow$
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*) $\subseteq$ $\mathcal{V}'$›
apply (*induction rule*:*rtranclp-induct*[*of PAC-Format* ‹(-, -)› ‹(-, -)›, *split-format*(*complete*)])
subgoal
by (*simp add*: *restricted-ideal-to-restricted-ideal-to*$_I$*D*)
subgoal
by (*drule PAC-Format-subset-ideal*)
(*auto simp*: *restricted-ideal-to-def Collect-mono-iff*)
done

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* ∧
(∀ *i j*. (*i*, *j*) ∈ *K* ⟶ (*j* ∈# *dom-m m2*) = (*i* ∈# *dom-m m1*))}
›
unfolding *fmap-rel-def dom-m-def*
by *auto*

lemma *fmdom-empty-fmempty-iff*[*simp*]: ‹*fmdom m* = {||} ⟷ *m* = *fmempty*›
by (*metis fmdom-empty fmdrop-fset-fmdom fmdrop-fset-null*)

lemma *fmap-rel-empty1-simp*[*simp*]:
(*fmempty*,*m*)∈⟨*K*,*V*⟩*fmap-rel* ⟷ *m*=*fmempty*
apply (*cases* ‹*fmdom m* = {||}›)
apply (*auto simp*: *fmap-rel-def*)[]
by (*auto simp add*: *fmap-rel-def simp del*: *fmdom-empty-fmempty-iff*)

lemma *fmap-rel-empty2-simp*[*simp*]:
(*m*,*fmempty*)∈⟨*K*,*V*⟩*fmap-rel* ⟷ *m*=*fmempty*
apply (*cases* ‹*fmdom m* = {||}›)

apply (*auto simp*: *fmap-rel-def*)[]
by (*fastforce simp add*: *fmap-rel-def simp del*: *fmdom-empty-fmempty-iff*)

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

lemma [*synth-rules*]: ⟦*INTF-OF-REL K TYPE*('*k*); *INTF-OF-REL V TYPE*('*v*)⟧
⟹ *INTF-OF-REL* (⟨*K*,*V*⟩*fmap-rel*) *TYPE*(('*k*,'*v*) *f-map*) **by** *simp*

5.1 Operations

sepref-decl-op *fmap-empty*: *fmempty* :: ⟨*K*,*V*⟩*fmap-rel* .

sepref-decl-op *fmap-is-empty*: (=) *fmempty* :: ⟨*K*,*V*⟩*fmap-rel* → *bool-rel*
apply (*rule fref-ncI*)
apply *parametricity*
apply (*rule fun-relI*; *auto*)
done

lemma *fmap-rel-fmupd-fmap-rel*:
‹(*A*, *B*) ∈ ⟨*K*, *R*⟩*fmap-rel* ⟹ (*p*, *p'*) ∈ *K* ⟹ (*q*, *q'*) ∈ *R* ⟹
(*fmupd p q A*, *fmupd p' q' B*) ∈ ⟨*K*, *R*⟩*fmap-rel*›
if *single-valued K single-valued* (*K*$^{-1}$)
using *that*
unfolding *fmap-rel-alt-def*
apply (*case-tac* ‹*p'* ∈# *dom-m B*›)
apply (*auto simp add*: *all-conj-distrib IS-RIGHT-UNIQUED dest*!: *multi-member-split*)
done

sepref-decl-op *fmap-update*: *fmupd* :: *K* → *V* → ⟨*K*,*V*⟩*fmap-rel* → ⟨*K*,*V*⟩*fmap-rel*
where *single-valued K single-valued* (*K*$^{-1}$)
apply (*rule fref-ncI*)
apply *parametricity*
apply (*intro fun-relI*)
by (*rule fmap-rel-fmupd-fmap-rel*)

lemma *remove1-mset-eq-add-mset-iff*:
‹*remove1-mset a A* = *add-mset a A'* ⟷ *A* = *add-mset a* (*add-mset a A'*)›
by (*metis add-mset-add-single add-mset-diff-bothsides diff-zero remove1-mset-eqE*)

lemma *fmap-rel-fmdrop-fmap-rel*:
‹(*fmdrop p A*, *fmdrop p' B*) ∈ ⟨*K*, *R*⟩*fmap-rel*›
if *single*: *single-valued K single-valued* (*K*$^{-1}$) **and**
H0: ‹(*A*, *B*) ∈ ⟨*K*, *R*⟩*fmap-rel*› ‹(*p*, *p'*) ∈ *K*›
proof −
have *H*: ‹⋀*Aa j*.
∀ *i*. *i* ∈# *dom-m B* ⟶ (∀ *j*. (*j*, *i*) ∈ *K* ⟶ (*the* (*fmlookup A j*), *the* (*fmlookup B i*)) ∈ *R*) ⟹
remove1-mset p' (*dom-m B*) = *add-mset p' Aa* ⟹ (*j*, *p'*) ∈ *K* ⟹ *False*›
by (*metis dom-m-fmdrop fmlookup-drop in-dom-m-lookup-iff union-single-eq-member*)
have *H2*: ‹⋀*i Aa j*.
(*p*, *p'*) ∈ *K* ⟹
∀ *i*. *i* ∈# *dom-m B* ⟶ (∀ *j*. (*j*, *i*) ∈ *K* ⟶ (*the* (*fmlookup A j*), *the* (*fmlookup B i*)) ∈ *R*) ⟹
∀ *i j*. (*i*, *j*) ∈ *K* ⟶ (*j* ∈# *dom-m B*) = (*i* ∈# *dom-m A*) ⟹
remove1-mset p' (*dom-m B*) = *add-mset i Aa* ⟹
(*j*, *i*) ∈ *K* ⟹

```
        (the (fmlookup A j), the (fmlookup B i)) ∈ R ∧ j ∈# remove1-mset p (dom-m A) ∧
      i ∈# remove1-mset p' (dom-m B)›
    ‹⋀i j Aa.
      (p, p') ∈ K ⟹
      single-valued K ⟹
      single-valued (K⁻¹) ⟹
      ∀ i. i ∈# dom-m B ⟶ (∀ j. (j, i) ∈ K ⟶ (the (fmlookup A j), the (fmlookup B i)) ∈ R) ⟹
      fset (fmdom A) ⊆ Domain K ⟹
      fset (fmdom B) ⊆ Range K ⟹
      ∀ i j. (i, j) ∈ K ⟶ (j ∈# dom-m B) = (i ∈# dom-m A) ⟹
      (i, j) ∈ K ⟹ remove1-mset p (dom-m A) = add-mset i Aa ⟹ j ∈# remove1-mset p' (dom-m
B)›
    using single
    by (metis IS-RIGHT-UNIQUED converse.intros dom-m-fmdrop fmlookup-drop in-dom-m-lookup-iff
        union-single-eq-member)+
  show ‹(fmdrop p A, fmdrop p' B) ∈ ⟨K, R⟩fmap-rel›
    using that
    unfolding fmap-rel-alt-def
    by (auto simp add: all-conj-distrib IS-RIGHT-UNIQUED
        dest!: multi-member-split dest: H H2)
qed

sepref-decl-op fmap-delete: fmdrop :: K → ⟨K,V⟩fmap-rel → ⟨K,V⟩fmap-rel
  where single-valued K single-valued (K⁻¹)
  apply (rule fref-ncI)
  apply parametricity
  by (auto simp add: fmap-rel-fmdrop-fmap-rel)

lemma fmap-rel-nat-the-fmlookup[intro]:
  ‹(A, B) ∈ ⟨S, R⟩fmap-rel ⟹ (p, p') ∈ S ⟹ p' ∈# dom-m B ⟹
     (the (fmlookup A p), the (fmlookup B p')) ∈ R›
  by (auto simp: fmap-rel-alt-def distinct-mset-dom)

lemma fmap-rel-in-dom-iff:
  ‹(aa, a'a) ∈ ⟨K, V⟩fmap-rel ⟹
    (a, a') ∈ K ⟹
    a' ∈# dom-m a'a ⟷
    a ∈# dom-m aa›
  unfolding fmap-rel-alt-def
  by auto

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›
  using fmap-rel-nat-the-fmlookup[of aa a'a K V a a']
    fmap-rel-in-dom-iff[of aa a'a K V a a']
    in-dom-m-lookup-iff[of a' a'a]
    in-dom-m-lookup-iff[of a aa]
  by (cases ‹a' ∈# dom-m a'a›)
    (auto simp del: fmap-rel-nat-the-fmlookup)


sepref-decl-op fmap-lookup: fmlookup :: ⟨K,V⟩fmap-rel → K → ⟨V⟩option-rel
  apply (rule fref-ncI)
  apply parametricity
```

apply (*intro fun-relI*)
apply (*rule fmap-rel-fmlookup-rel*; *assumption*)
done

lemma *in-fdom-alt*: $k \in \#\, dom\text{-}m\ m \longleftrightarrow \neg is\text{-}None\ (fmlookup\ m\ k)$
by (*auto split*: *option.split intro*: *fmdom-notI fmdomI simp*: *dom-m-def*)

sepref-decl-op *fmap-contains-key*: $\lambda k\ m.\ k \in \#\, dom\text{-}m\ m :: K \to \langle K, V\rangle fmap\text{-}rel \to bool\text{-}rel$
unfolding *in-fdom-alt*
apply (*rule fref-ncI*)
apply *parametricity*
apply (*rule fmap-rel-fmlookup-rel*; *assumption*)
done

5.2 Patterns

lemma *pat-fmap-empty*[*pat-rules*]: $fmempty \equiv op\text{-}fmap\text{-}empty$ **by** *simp*

lemma *pat-map-is-empty*[*pat-rules*]:
$(=)\ \$m\$fmempty \equiv op\text{-}fmap\text{-}is\text{-}empty\$m$
$(=)\ \$fmempty\$m \equiv op\text{-}fmap\text{-}is\text{-}empty\$m$
$(=)\ \$(dom\text{-}m\$m)\$\{\#\} \equiv op\text{-}fmap\text{-}is\text{-}empty\$m$
$(=)\ \$\{\#\}\$(dom\text{-}m\$m) \equiv op\text{-}fmap\text{-}is\text{-}empty\$m$
unfolding *atomize-eq*
by (*auto dest*: *sym*)

lemma *op-map-contains-key*[*pat-rules*]:
$(\in\#)\ \$\ k\ \$\ (dom\text{-}m\$m) \equiv op\text{-}fmap\text{-}contains\text{-}key\$'k\$'m$
by (*auto intro*!: *eq-reflection*)

5.3 Mapping to Normal Hashmaps

abbreviation *map-of-fmap* :: $\langle('k \Rightarrow 'v\ option) \Rightarrow ('k, 'v)\ fmap\rangle$ **where**
$\langle map\text{-}of\text{-}fmap\ h \equiv Abs\text{-}fmap\ h\rangle$

definition *map-fmap-rel* **where**
$\langle map\text{-}fmap\text{-}rel = br\ map\text{-}of\text{-}fmap\ (\lambda a.\ finite\ (dom\ a))\rangle$

lemma *fmdrop-set-None*:
$\langle(op\text{-}map\text{-}delete, fmdrop) \in Id \to map\text{-}fmap\text{-}rel \to map\text{-}fmap\text{-}rel\rangle$
apply (*auto simp*: *map-fmap-rel-def br-def*)
apply (*subst fmdrop.abs-eq*)
apply (*auto simp*: *eq-onp-def fmap.Abs-fmap-inject*
map-drop-def map-filter-finite
intro!: *ext*)
apply (*auto simp*: *map-filter-def*)
done

lemma *map-upd-fmupd*:
$\langle(op\text{-}map\text{-}update, fmupd) \in Id \to Id \to map\text{-}fmap\text{-}rel \to map\text{-}fmap\text{-}rel\rangle$
apply (*auto simp*: *map-fmap-rel-def br-def*)
apply (*subst fmupd.abs-eq*)
apply (*auto simp*: *eq-onp-def fmap.Abs-fmap-inject*
map-drop-def map-filter-finite map-upd-def
intro!: *ext*)
done

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))›
by (*simp add*: *fold-is-None in-fdom-alt*)

lemma *op-map-lookup-fmlookup*:
‹(*op-map-lookup*, *fmlookup'*) ∈ *Id* → *map-fmap-rel* → ⟨*Id*⟩*option-rel*›
by (*auto simp*: *map-fmap-rel-def br-def fmap.Abs-fmap-inverse*)

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*›
by (*auto simp*: *map-fmap-rel-def br-def fmempty-def frefI nres-relI*)

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*›
by (*auto simp*: *map-fmap-rel-def br-def fmempty-def frefI nres-relI op-iam-fmap-empty-def*)

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 −
have [*simp*]: ‹*finite* (*dom b*) ⟹ *i* ∈ *fset* (*fmdom* (*map-of-fmap b*)) ⟷ *i* ∈ *dom b*› **for** *i b*
by (*subst fmdom.abs-eq*)
(*auto simp*: *eq-onp-def fset.Abs-fset-inverse*)
have *2*: ‹*nat-rel* = *the-pure* (*nat-assn*)› **and**
3: ‹*nat-assn* = *pure nat-rel*›
by *auto*
have [*simp*]: ‹*the-pure* (λ*a c* :: *nat*. ↑ (*c* = *a*)) = *nat-rel*›
apply (*subst 2*)
apply (*subst 3*)
apply (*subst pure-def*)
apply *auto*
done

have [*simp*]: ‹(*iam-of-list l*, *b*) ∈ *the-pure* (λ*a c* :: *nat*. ↑ (*c* = *a*)) → ⟨*the-pure V*⟩*option-rel* ⟹
b i = *Some y* ⟹ *i* < *length l*› **for** *i b l y*
by (*auto dest*!: *fun-relD*[*of - - - - i i*] *simp*: *option-rel-def*
iam-of-list-def split: *if-splits*)
show *?thesis*
by *sepref-to-hoare*
(*sep-auto simp*: *upper-bound-on-dom-def hr-comp-def iam.assn-def map-rel-def*
map-fmap-rel-def is-iam-def br-def dom-m-def)
qed

lemma *fmap-rel-nat-rel-dom-m*[*simp*]:
‹(*A*, *B*) ∈ ⟨*nat-rel*, *R*⟩*fmap-rel* ⟹ *dom-m A* = *dom-m B*›
by (*subst distinct-set-mset-eq-iff*[*symmetric*])
(*auto simp*: *fmap-rel-alt-def distinct-mset-dom*
simp del: *fmap-rel-nat-the-fmlookup*)

lemma *ref-two-step′*:
‹*A* ≤ *B* ⟹ ⇓ *R A* ≤ ⇓ *R B*›
using *ref-two-step* **by** *auto*

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*›
  by (*cases a*) *auto*

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(*V*$_0$, *A*$_0$) *spec* (*V*, *A*) (*b*, *B*).
      (*is-success b* ⟶
        ⋃(*vars* ‘ *set-mset A*$_0$) ⊆ *V*$_0$ ∧
        ⋃(*vars* ‘ *set-mset A*) ⊆ *V* ∧ *PAC-Format*** (*V*$_0$, *A*$_0$) (*V*, *A*) ∧ *PAC-Format*** (*V*, *A*) *B*) ∧
      (*is-found b* ⟶
        ⋃(*vars* ‘ *set-mset A*$_0$) ⊆ *V*$_0$ ∧
        ⋃(*vars* ‘ *set-mset A*) ⊆ *V* ∧ *PAC-Format*** (*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* (λr. $p - r$ ∈ *ideal polynomial-bool* ∧ *vars* r ⊆ *vars* p)›

lemma *normalize-poly-spec-alt-def*:
‹*normalize-poly-spec* p = *SPEC* (λr. $r - p$ ∈ *ideal polynomial-bool* ∧ *vars* r ⊆ *vars* p)›
unfolding *normalize-poly-spec-def*
by (*auto dest*: *ideal.span-neg*)

definition *mult-poly-spec* :: ‹*int mpoly* ⇒ *int mpoly* ⇒ *int mpoly nres*› **where**
‹*mult-poly-spec* p q = *SPEC* (λr. $p * q - r$ ∈ *ideal polynomial-bool*)›

definition *check-add* :: ‹(*nat*, *int mpoly*) *fmap* ⇒ *nat set* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *int mpoly* ⇒ *bool nres*› **where**
‹*check-add* A $\mathcal{V}$ p q i r =
SPEC(λb. b ⟶ p ∈# *dom-m* A ∧ q ∈# *dom-m* A ∧ i ∉# *dom-m* A ∧ *vars* r ⊆ $\mathcal{V}$ ∧
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 $\mathcal{V}$ p q i r =
SPEC(λb. b ⟶ p ∈# *dom-m* A ∧i ∉# *dom-m* A ∧ *vars* q ⊆ $\mathcal{V}$ ∧ *vars* r ⊆ $\mathcal{V}$ ∧
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 $\mathcal{V}$ i v p =
SPEC(λb. b ⟶ (i ∉# *dom-m* A ∧
(v ∉ $\mathcal{V}$ ∧
(p+*Var* v)^2 − (p+*Var* v) ∈ *ideal polynomial-bool* ∧
vars (p+*Var* v) ⊆ $\mathcal{V}$)))›

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(λb. b ⟶ *True*)›

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 = (λspec (stat, (𝒱, A)) st. case st of
   Add - - - - ⇒
     do {
       r ← normalize-poly-spec (pac-res st);
      eq ← check-add A 𝒱 (pac-src1 st) (pac-src2 st) (new-id st) r;
      st' ← SPEC(λst'. (¬is-failed st' ∧ is-found st' ⟶ r − spec ∈ ideal polynomial-bool));
      if eq
      then RETURN (merge-status stat st',
        𝒱, fmupd (new-id st) r A)
      else RETURN (FAILED, (𝒱, A))
  }
  | Del - ⇒
     do {
      eq ← check-del A (pac-src1 st);
      if eq
      then RETURN (stat, (𝒱, fmdrop (pac-src1 st) A))
      else RETURN (FAILED, (𝒱, A))
  }
  | Mult - - - - ⇒
     do {
       r ← normalize-poly-spec (pac-res st);
       q ← normalize-poly-spec (pac-mult st);
      eq ← check-mult A 𝒱 (pac-src1 st) q (new-id st) r;
      st' ← SPEC(λst'. (¬is-failed st' ∧ is-found st' ⟶ r − spec ∈ ideal polynomial-bool));
      if eq
      then RETURN (merge-status stat st',
        𝒱, fmupd (new-id st) r A)
      else RETURN (FAILED, (𝒱, A))
  }
  | Extension - - - ⇒
     do {
       r ← normalize-poly-spec (pac-res st − Var (new-var st));
      (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*›
using *distinct-mset-dom*[*of A*]
apply (*auto simp*: *polys-rel-def ran-m-mapsto-upd ran-m-mapsto-upd-notin*
ran-m-fmdrop)
apply (*subst ran-m-mapsto-upd-notin*)
apply (*auto dest*: *in-diffD dest*!: *multi-member-split simp*: *ran-m-fmdrop ran-m-fmdrop-If distinct-mset-remove1-All ran-m-def*
add-mset-eq-add-mset removeAll-notin
split: *if-splits intro*!: *image-mset-cong*)
done

lemma *polys-rel-in-dom-inD*:
‹(*A*, *B*) ∈ *polys-rel* ⟹
x12 ∈# *dom-m A* ⟹
the (*fmlookup A x12*) ∈# *B*›
by (*auto simp*: *polys-rel-def*)

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* ⟹
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)›
subgoal
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule PAC-Format.add*[*of ‹the (fmlookup A x12)› B ‹the (fmlookup A x12)›*])
apply (*auto dest*: *polys-rel-in-dom-inD*)
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule PAC-Format.del*[*of ‹the (fmlookup A x12)›*])
apply (*auto dest*: *polys-rel-in-dom-inD*)
done
subgoal *H2*
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule PAC-Format.add*[*of ‹the (fmlookup A x11)› B ‹the (fmlookup A x12)›*])
apply (*auto dest*: *polys-rel-in-dom-inD*)
done
subgoal
apply (*rule rtranclp-trans*)
apply (*rule H2*; *assumption*)
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule PAC-Format.del*[*of ‹the (fmlookup A x12)›*])
apply (*auto dest*: *polys-rel-in-dom-inD*)
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule PAC-Format.del*[*of ‹the (fmlookup A x11)›*])
apply (*auto dest*: *polys-rel-in-dom-inD*)
apply (*auto simp*: *polys-rel-def ran-m-def add-mset-eq-add-mset dest*!: *multi-member-split*)
done
subgoal *H2*
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule PAC-Format.mult*[*of ‹the (fmlookup A x11)› B ‹x32› r*])
apply (*auto dest*: *polys-rel-in-dom-inD*)
done
subgoal
apply (*rule rtranclp-trans*)
apply (*rule H2*; *assumption*)
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule PAC-Format.del*[*of ‹the (fmlookup A x11)›*])
apply (*auto dest*: *polys-rel-in-dom-inD*)
done
subgoal
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule PAC-Format.del*[*of ‹the (fmlookup A x12)› B*])

apply (*auto dest*: *polys-rel-in-dom-inD*)
done
subgoal
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule PAC-Format.extend-pos*[*of* ‹$p' + Var\ x$› - x])
using *coeff-monomila-in-varsD*[*of* ‹$p' - Var\ x$› x]
apply (*auto dest*: *polys-rel-in-dom-inD simp*: *vars-in-right-only vars-subst-in-left-only*)
apply (*subgoal-tac* ‹$\mathcal{V} \cup \{x' \in vars\ (p').\ x' \notin \mathcal{V}\} = insert\ x\ \mathcal{V}$›)
apply *simp*
using *coeff-monomila-in-varsD*[*of* p' x]
apply (*auto dest*: *vars-add-Var-subset vars-minus-Var-subset polys-rel-in-dom-inD simp*: *vars-subst-in-left-only-iff*)
using *vars-in-right-only vars-subst-in-left-only* **by** *force*
done

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'*)›
by (*cases a*; *cases st'*; *auto*; *fail*)+

lemma *status-rel-merge-status*:
‹(*merge-status a b*, *SUCCESS*) ∉ *status-rel* ⟷
(*a* = *FAILED*) ∨ (*b* = *FAILED*) ∨
a = *FOUND* ∨ (*b* = *FOUND*)›
by (*cases a*; *cases b*; *auto*)

lemma *Ex-status-iff*:
‹(∃ *a*. *P a*) ⟷ *P SUCCESS* ∨ *P FOUND* ∨ (*P* (*FAILED*))›
apply *auto*
apply (*case-tac a*; *auto*)
done

lemma *is-failed-alt-def*:
‹*is-failed st'* ⟷ ¬*is-success st'* ∧ ¬*is-found st'*›
by (*cases st'*) *auto*

lemma *merge-status-eq-iff*[*simp*]:
‹*merge-status a SUCCESS* = *SUCCESS* ⟷ *a* = *SUCCESS*›
‹*merge-status a SUCCESS* = *FOUND* ⟷ *a* = *FOUND*›
‹*merge-status SUCCESS a* = *SUCCESS* ⟷ *a* = *SUCCESS*›
‹*merge-status SUCCESS a* = *FOUND* ⟷ *a* = *FOUND*›
‹*merge-status SUCCESS a* = *FAILED* ⟷ *a* = *FAILED*›
‹*merge-status a SUCCESS* = *FAILED* ⟷ *a* = *FAILED*›
‹*merge-status FOUND a* = *FAILED* ⟷ *a* = *FAILED*›
‹*merge-status a FOUND* = *FAILED* ⟷ *a* = *FAILED*›
‹*merge-status a FOUND* = *SUCCESS* ⟷ *False*›
‹*merge-status a b* = *FOUND* ⟷ (*a* = *FOUND* ∨ *b* = *FOUND*) ∧ (*a* ≠ *FAILED* ∧ *b* ≠ *FAILED*)›
apply (*cases a*; *auto*; *fail*)+
apply (*cases a*; *cases b*; *auto*; *fail*)+
done

lemma *fmdrop-irrelevant*: ‹*x11* ∉# *dom-m A* ⟹ *fmdrop x11 A* = *A*›
by (*simp add*: *fmap-ext in-dom-m-lookup-iff*)

lemma *PAC-checker-step-PAC-checker-specification2*:
fixes *a* :: ‹*status*›
assumes *AB*: ‹((*V*, *A*),(*V*$_B$, *B*)) ∈ *polys-rel-full*› **and**
‹¬*is-failed a*› **and**
[*simp*,*intro*]: ‹*a* = *FOUND* ⟹ *spec* ∈ *pac-ideal* (*set-mset A*$_0$)› **and**
A$_0$*B*: ‹*PAC-Format*** (*V*$_0$, *A*$_0$) (*V*, *B*)› **and**
spec$_0$: ‹*vars spec* ⊆ *V*$_0$› **and**
vars-A$_0$: ‹⋃ (*vars* ‘ *set-mset A*$_0$) ⊆ *V*$_0$›
shows ‹*PAC-checker-step spec* (*a*, (*V*, *A*)) *st* ≤ ⇓ (*status-rel* ×$_r$ *polys-rel-full*) (*PAC-checker-specification-step2* (*V*$_0$, *A*$_0$) *spec* (*V*, *B*))›
proof −
have
‹*V*$_B$ = *V*›**and**
[*simp*, *intro*]:‹(*A*, *B*) ∈ *polys-rel*›
using *AB*
by (*auto simp*: *polys-rel-full-def*)
have *H0*: ‹*2* ∗ *the* (*fmlookup A x12*) − *r* ∈ *More-Modules.ideal polynomial-bool* ⟹
r ∈ *pac-ideal*
(*insert* (*the* (*fmlookup A x12*))
((λ*x*. *the* (*fmlookup A x*)) ‘ *set-mset Aa*))› **for** *x12 r Aa*
by (*metis* (*no-types*, *lifting*) *ab-semigroup-mult-class.mult.commute*
diff-in-polynomial-bool-pac-idealI
ideal.span-base pac-idealI3 set-image-mset set-mset-add-mset-insert union-single-eq-member)+
then have *H0′*: ‹⋀*Aa*. *2* ∗ *the* (*fmlookup A x12*) − *r* ∈ *More-Modules.ideal polynomial-bool* ⟹
r − *spec* ∈ *More-Modules.ideal polynomial-bool* ⟹
spec ∈ *pac-ideal* (*insert* (*the* (*fmlookup A x12*)) ((λ*x*. *the* (*fmlookup A x*)) ‘ *set-mset Aa*))›
for *r x12*
by (*metis* (*no-types*, *lifting*) *diff-in-polynomial-bool-pac-idealI*)

have *H1*: ‹ *x12* ∈# *dom-m A* ⟹
2 ∗ *the* (*fmlookup A x12*) − *r* ∈ *More-Modules.ideal polynomial-bool* ⟹
r − *spec* ∈ *More-Modules.ideal polynomial-bool* ⟹
vars spec ⊆ *vars r* ⟹
spec ∈ *pac-ideal* (*set-mset B*)› **for** *x12 r*
using ‹(*A*,*B*) ∈ *polys-rel*›
ideal.span-add[*OF ideal.span-add*[*OF ideal.span-neg ideal.span-neg*,
of ‹*the* (*fmlookup A x12*)› - ‹*the* (*fmlookup A x12*)›],
of ‹*set-mset B* ∪ *polynomial-bool*› ‹*2* ∗ *the* (*fmlookup A x12*) − *r*›]
unfolding *polys-rel-def*
by (*auto dest*!: *multi-member-split simp*: *ran-m-def*
intro: *H0′*)
have *H2′*: ‹*the* (*fmlookup A x11*) + *the* (*fmlookup A x12*) − *r* ∈ *More-Modules.ideal polynomial-bool* ⟹
B = *add-mset* (*the* (*fmlookup A x11*)) {#*the* (*fmlookup A x*). *x* ∈# *Aa*#} ⟹
(*the* (*fmlookup A x11*) + *the* (*fmlookup A x12*) − *r*
∈ *More-Modules.ideal*
(*insert* (*the* (*fmlookup A x11*))
((λ*x*. *the* (*fmlookup A x*)) ‘ *set-mset Aa* ∪ *polynomial-bool*)) ⟹
− *r*
∈ *More-Modules.ideal*
(*insert* (*the* (*fmlookup A x11*))
((λ*x*. *the* (*fmlookup A x*)) ‘ *set-mset Aa* ∪ *polynomial-bool*))) ⟹

```
    r ∈ pac-ideal (insert (the (fmlookup A x11)) ((λx. the (fmlookup A x)) ‘ set-mset Aa))›
   for r x12 x11 A Aa
   by (metis (mono-tags, lifting) Un-insert-left diff-diff-eq2 diff-in-polynomial-bool-pac-idealI diff-zero
     ideal.span-diff ideal.span-neg minus-diff-eq pac-idealI1 pac-ideal-def set-image-mset
     set-mset-add-mset-insert union-single-eq-member)
  have H2: ‹x11 ∈# dom-m A ⟹
     x12 ∈# dom-m A ⟹
     the (fmlookup A x11) + the (fmlookup A x12) − r
     ∈ More-Modules.ideal polynomial-bool ⟹
     r − spec ∈ More-Modules.ideal polynomial-bool ⟹
     spec ∈ pac-ideal (set-mset B)› for x12 r x11
   using ‹(A,B) ∈ polys-rel›
    ideal.span-add[OF ideal.span-add[OF ideal.span-neg ideal.span-neg,
      of ‹the (fmlookup A x11)› - ‹the (fmlookup A x12)›],
    of ‹set-mset B ∪ polynomial-bool› ‹the (fmlookup A x11) + the (fmlookup A x12) − r›]
   unfolding polys-rel-def
   by (subgoal-tac ‹r ∈ pac-ideal (set-mset B)›)
    (auto dest!: multi-member-split simp: ran-m-def ideal.span-base
      intro: diff-in-polynomial-bool-pac-idealI simp: H2')

  have H3': ‹the (fmlookup A x12) ∗ q − r ∈ More-Modules.ideal polynomial-bool ⟹
       r − spec ∈ More-Modules.ideal polynomial-bool ⟹
       r ∈ pac-ideal (insert (the (fmlookup A x12)) ((λx. the (fmlookup A x)) ‘ set-mset Aa))›
   for Aa x12 r q
   by (metis (no-types, lifting) ab-semigroup-mult-class.mult.commute diff-in-polynomial-bool-pac-idealI
     ideal.span-base pac-idealI3 set-image-mset set-mset-add-mset-insert union-single-eq-member)

 have H3: ‹x12 ∈# dom-m A ⟹
     the (fmlookup A x12) ∗ q − r ∈ More-Modules.ideal polynomial-bool ⟹
     r − spec ∈ More-Modules.ideal polynomial-bool ⟹
     spec ∈ pac-ideal (set-mset B)› for x12 r q
   using ‹(A,B) ∈ polys-rel›
    ideal.span-add[OF ideal.span-add[OF ideal.span-neg ideal.span-neg,
      of ‹the (fmlookup A x12)› - ‹the (fmlookup A x12)›],
    of ‹set-mset B ∪ polynomial-bool› ‹2 ∗ the (fmlookup A x12) − r›]
   unfolding polys-rel-def
   by (subgoal-tac ‹r ∈ pac-ideal (set-mset B)›)
    (auto dest!: multi-member-split simp: ran-m-def H3'
      intro: diff-in-polynomial-bool-pac-idealI)

 have [intro]: ‹spec ∈ pac-ideal (set-mset B) ⟹ spec ∈ pac-ideal (set-mset A0)› and
   vars-B: ‹⋃ (vars ‘ set-mset B) ⊆ 𝒱›and
   vars-B: ‹⋃ (vars ‘ set-mset (ran-m A)) ⊆ 𝒱›
    using rtranclp-PAC-Format-subset-ideal[OF A0B vars-A0] spec0 ‹(A, B) ∈ polys-rel›[unfolded
polys-rel-def, simplified]
   by (smt (verit) in-mono mem-Collect-eq restricted-ideal-to-def)+

 have eq-successI: ‹st' ≠ FAILED ⟹
     st' ≠ FOUND ⟹ st' = SUCCESS› for st'
   by (cases st') auto
 have vars-diff-inv: ‹vars (Var x2 − r) = vars (r − Var x2 :: int mpoly)› for x2 r
   using vars-uminus[of ‹Var x2 − r›]
   by (auto simp del: vars-uminus)
 have vars-add-inv: ‹vars (Var x2 + r) = vars (r + Var x2 :: int mpoly)› for x2 r
   unfolding add.commute[of ‹Var x2› r] ..
```

have [*iff*]: ‹*a* ≠ *FAILED*› **and**
[*intro*]: ‹*a* ≠ *SUCCESS* ⟹ *a* = *FOUND*› **and**
[*simp*]: ‹*merge-status a FOUND* = *FOUND*›
using *assms*(*2*) **by** (*cases a*; *auto*)+
note [[*goals-limit=1*]]
show *?thesis*
unfolding *PAC-checker-step-def PAC-checker-specification-step-spec-def*
normalize-poly-spec-alt-def check-mult-def check-add-def
check-extension-def polys-rel-full-def
apply (*cases st*)
apply *clarsimp-all*
subgoal for *x11 x12 x13 x14*
apply (*refine-vcg lhs-step-If*)
subgoal for *r eqa st'*
using *assms vars-B* **apply** −
apply (*rule RETURN-SPEC-refine*)
apply (*rule-tac x* = ‹(*merge-status a st'*,*V*,*add-mset r B*)› **in** *exI*)
by (*auto simp*: *polys-rel-update-remove ran-m-mapsto-upd-notin*
intro: *PAC-Format-add-and-remove H2 dest*: *rtranclp-PAC-Format-subset-ideal*)
subgoal
by (*rule RETURN-SPEC-refine*)
(*auto simp*: *Ex-status-iff dest*: *rtranclp-PAC-Format-subset-ideal*)
done
subgoal for *x11 x12 x13 x14*
apply (*refine-vcg lhs-step-If*)
subgoal for *r q eqa st'*
using *assms vars-B* **apply** −
apply (*rule RETURN-SPEC-refine*)
apply (*rule-tac x* = ‹(*merge-status a st'*,*V*,*add-mset r B*)› **in** *exI*)
by (*auto intro*: *polys-rel-update-remove intro*: *PAC-Format-add-and-remove*(*3*−) *H3*
dest: *rtranclp-PAC-Format-subset-ideal*)
subgoal
by (*rule RETURN-SPEC-refine*)
(*auto simp*: *Ex-status-iff*)
done
subgoal for *x31 x32 x34*
apply (*refine-vcg lhs-step-If*)
subgoal for *r x*
using *assms vars-B* **apply** −
apply (*rule RETURN-SPEC-refine*)
apply (*rule-tac x* = ‹(*a*,*insert x32 V*, *add-mset r B*)› **in** *exI*)
apply (*auto simp*: *intro*!: *polys-rel-update-remove PAC-Format-add-and-remove*(*5*−)
dest: *rtranclp-PAC-Format-subset-ideal*)
done
subgoal
by (*rule RETURN-SPEC-refine*)
(*auto simp*: *Ex-status-iff*)
done
subgoal for *x11*
unfolding *check-del-def*
apply (*refine-vcg lhs-step-If*)
subgoal for *eq*
using *assms vars-B* **apply** −
apply (*rule RETURN-SPEC-refine*)

```
      apply (cases ‹x11 ∈# dom-m A›)
      subgoal
        apply (rule-tac x = ‹(a,𝒱, remove1-mset (the (fmlookup A x11)) B)› in exI)
        apply (auto simp: polys-rel-update-remove PAC-Format-add-and-remove
            is-failed-def is-success-def is-found-def
          dest!: eq-successI
          split: if-splits
          dest: rtranclp-PAC-Format-subset-ideal
          intro: PAC-Format-add-and-remove H3)
        done
      subgoal
        apply (rule-tac x = ‹(a,𝒱, B)› in exI)
        apply (auto simp: fmdrop-irrelevant
            is-failed-def is-success-def is-found-def
          dest!: eq-successI
          split: if-splits
          dest: rtranclp-PAC-Format-subset-ideal
          intro: PAC-Format-add-and-remove)
        done
      done
    subgoal
      by (rule RETURN-SPEC-refine)
        (auto simp: Ex-status-iff)
    done
  done
qed


definition PAC-checker
  :: ‹int-poly ⇒ fpac-step ⇒ status ⇒ (int-poly, nat, nat) pac-step list ⇒
    (status × fpac-step) nres›
where
  ‹PAC-checker spec A b st = do {
    (S, -) ← WHILE_T
      (λ((b :: status, A :: fpac-step), st). ¬is-failed b ∧ st ≠ [])
      (λ((bA), st). do {
        ASSERT(st ≠ []);
        S ← 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) ⟹
    PAC-checker-specification-step-spec A spec x2 (st', x1a) ⟹
    PAC-checker-specification-spec spec A (st', x1a)›
 unfolding PAC-checker-specification-spec-def
   PAC-checker-specification-step-spec-def
 apply auto
using is-failed-alt-def apply blast+
done
```

lemma *RES-SPEC-eq*:
‹*RES* Φ = *SPEC*(λ*P*. *P* ∈ Φ)›
by *auto*

lemma *is-failed-is-success-completeD*:
‹¬ *is-failed x* ⟹ ¬*is-success x* ⟹ *is-found x*›
by (*cases x*) *auto*

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*)›
unfolding *PAC-checker-def conc-fun-RES*
apply (*subst RES-SPEC-eq*)
apply (*refine-vcg WHILET-rule*[**where**
I = ‹λ((*bB*), *st*). *bB* ∈ (*status-rel* ×$_r$ *polys-rel-full*)$^{-1}$ ‘‘
Collect (*PAC-checker-specification-spec spec B*)›
and *R* = ‹*measure* (λ(-, *st*). *Suc* (*length st*))›])
subgoal by *auto*
subgoal apply (*auto simp*: *PAC-checker-specification-spec-def*)
apply (*cases B*; *cases A*)
apply (*auto simp*:*polys-rel-def polys-rel-full-def Image-iff*)
done
subgoal by *auto*
subgoal
apply *auto*
apply (*rule*
PAC-checker-step-PAC-checker-specification2[*of* - - - - - - - ‹*fst B*›, *THEN order-trans*])
apply *assumption*
apply *assumption*
apply (*auto intro*: *PAC-checker-specification-spec-trans simp*: *conc-fun-RES*)
apply (*auto simp*: *PAC-checker-specification-spec-def polys-rel-full-def polys-rel-def*
dest: *PAC-Format-subset-ideal*
dest: *is-failed-is-success-completeD*; *fail*)+
by (*auto simp*: *Image-iff intro*: *PAC-checker-specification-spec-trans*
simp: *polys-rel-def polys-rel-full-def*)
subgoal
by *auto*
done

definition *remap-polys-polynomial-bool* :: ‹*int mpoly* ⇒ *nat set* ⇒ (*nat*, *int-poly*) *fmap* ⇒ (*status* × *fpac-step*) *nres*› **where**
‹*remap-polys-polynomial-bool spec* = (λ$\mathcal{V}$ *A*.
SPEC(λ(*st*, $\mathcal{V}'$, *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*)) ⊆ $\mathcal{V}'$ ∧
⋃(*vars* ‘ *set-mset* (*ran-m A'*)) ⊆ $\mathcal{V}'$) ∧
(*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* = (λ$\mathcal{V}$ *A*. *SPEC* (λ(*st*, $\mathcal{V}'$, *A'*).
(¬*is-failed st* ⟶
pac-ideal (*set-mset* (*ran-m A*)) = *pac-ideal* (*set-mset* (*ran-m A'*)) ∧
⋃(*vars* ‘ *set-mset* (*ran-m A*)) ⊆ $\mathcal{V}'$ ∧
⋃(*vars* ‘ *set-mset* (*ran-m A'*)) ⊆ $\mathcal{V}'$) ∧
(*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*))›
by (*metis fmap-ext in-dom-m-lookup-iff option.expand*)

lemma *ideal-remap-incl*:
‹*finite A'* ⟹ (∀ *a'*∈*A'*. ∃ *a*∈*A*. *a*−*a'* ∈ *B*) ⟹ *ideal* (*A'* ∪ *B*) ⊆ *ideal* (*A* ∪ *B*)›
apply (*induction A' rule*: *finite-induct*)
apply (*auto intro*: *ideal.span-mono*)
using *ideal.span-mono sup-ge2* **apply** *blast*
proof −
fix *x* :: *'a* **and** *F* :: *'a set* **and** *xa* :: *'a* **and** *a* :: *'a*
assume *a1*: *a* ∈ *A*
assume *a2*: *a* − *x* ∈ *B*
assume *a3*: *xa* ∈ *More-Modules.ideal* (*insert x* (*F* ∪ *B*))
assume *a4*: *More-Modules.ideal* (*F* ∪ *B*) ⊆ *More-Modules.ideal* (*A* ∪ *B*)
have *x* ∈ *More-Modules.ideal* (*A* ∪ *B*)
using *a2 a1* **by** (*metis* (*no-types*, *lifting*) *Un-upper1 Un-upper2 add-diff-cancel-left' diff-add-cancel ideal.module-axioms ideal.span-diff in-mono module.span-superset*)
then show *xa* ∈ *More-Modules.ideal* (*A* ∪ *B*)
using *a4 a3 ideal.span-insert-subset* **by** *blast*
qed

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 ((λ*x*. *the* (*fmlookup b x*)) ‘ *set-mset* (*dom-m ba*)) = *pac-ideal* ((λ*x*. *the* (*fmlookup ba x*)) ‘ *set-mset* (*dom-m ba*))›
unfolding *pac-ideal-alt-def*
apply *standard*
subgoal
apply (*rule ideal-remap-incl*)
apply (*auto dest*!: *multi-member-split*
dest: *ideal.span-neg*)
apply (*drule ideal.span-neg*)
apply *auto*
done
subgoal
by (*rule ideal-remap-incl*)
(*auto dest*!: *multi-member-split*)
done

lemma *remap-polys-polynomial-bool-remap-polys-change-all*:
‹*remap-polys-polynomial-bool spec* $\mathcal{V}$ *A* ≤ *remap-polys-change-all spec* $\mathcal{V}$ *A*›
unfolding *remap-polys-polynomial-bool-def remap-polys-change-all-def*
apply (*simp add*: *ideal.span-zero fmap-eq-dom-iff ideal.span-eq*)
apply (*auto dest*: *multi-member-split simp*: *ran-m-def ideal.span-base pac-ideal-remap-eq*

```
    add-mset-eq-add-mset
    eq-commute[of ‹add-mset - -› ‹dom-m (A :: (nat, int mpoly)fmap)› for A])
  done


definition remap-polys :: ‹int mpoly ⇒ nat set ⇒ (nat, int-poly) fmap ⇒ (status × fpac-step) nres›
where
  ‹remap-polys spec = (λ𝒱 A. do{
   dom ← SPEC(λdom. 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
       (λi (b, 𝒱, A').
          if i ∈# dom-m A
          then do {
           p ← SPEC(λp. the (fmlookup A i) − p ∈ ideal polynomial-bool ∧ vars p ⊆ vars (the (fmlookup
A i)));
            eq ← SPEC(λeq. 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›
  unfolding remap-polys-def remap-polys-polynomial-bool-def
  apply (refine-vcg FOREACH-rule[where
    I = ‹λdom (b, 𝒱, A').
      set-mset (dom-m A') =  set-mset (dom-m A) − dom ∧
     (∀ i ∈ set-mset (dom-m A) − dom. the (fmlookup A i) − the (fmlookup A' i) ∈ ideal polynomial-bool)
∧
     ⋃(vars ‘ set-mset (ran-m (fmrestrict-set (set-mset (dom-m A')) A))) ⊆ 𝒱 ∧
     ⋃(vars ‘ set-mset (ran-m A')) ⊆ 𝒱 ∧
      (b ⟶ spec ∈# ran-m A')›])
  subgoal by auto
  subgoal by auto
  subgoal by auto
  subgoal by auto
  subgoal by auto
  subgoal by auto
  subgoal by auto
  subgoal
    by auto
  subgoal by auto
  subgoal using ideal.span-add by auto
  subgoal by auto
  subgoal by auto
```

subgoal by *clarsimp auto*
subgoal
supply[[*goals-limit=1*]]
by (*auto simp add*: *ran-m-mapsto-upd-notin dom-m-fmrestrict-set′ subset-eq*)
subgoal
supply[[*goals-limit=1*]]
by (*auto simp add*: *ran-m-mapsto-upd-notin dom-m-fmrestrict-set′ subset-eq*)
subgoal
by (*auto simp*: *ran-m-mapsto-upd-notin*)
subgoal
by *auto*
subgoal
by *auto*
subgoal
by (*auto simp add*: *ran-m-mapsto-upd-notin dom-m-fmrestrict-set′ subset-eq*)
subgoal
by (*auto simp add*: *ran-m-mapsto-upd-notin dom-m-fmrestrict-set′ subset-eq*)
subgoal
by *auto*
subgoal
by (*auto simp*: *distinct-set-mset-eq-iff*[*symmetric*] *distinct-mset-dom*)
subgoal
by (*auto simp*: *distinct-set-mset-eq-iff*[*symmetric*] *distinct-mset-dom*)
subgoal
by (*auto simp add*: *ran-m-mapsto-upd-notin dom-m-fmrestrict-set′ subset-eq*
fmlookup-restrict-set-id′)
subgoal
by (*auto simp add*: *ran-m-mapsto-upd-notin dom-m-fmrestrict-set′ subset-eq*)
subgoal
by (*auto simp add*: *ran-m-mapsto-upd-notin dom-m-fmrestrict-set′ subset-eq*
fmlookup-restrict-set-id′)
done

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*, $\mathcal{V}$, *A*) ← *remap-polys-change-all spec* {} *A*;
if is-failed st then
RETURN (*st*, $\mathcal{V}$, *A*)
else do {
$\mathcal{V}$ ← *SPEC*(λ$\mathcal{V}'$. $\mathcal{V}$ ∪ *vars spec0* ⊆ $\mathcal{V}'$);
PAC-checker spec ($\mathcal{V}$, *A*) *st pac*
}
}›

lemma *restricted-ideal-to-mono*:
‹*restricted-ideal-to*$_I$ $\mathcal{V}$ *I* ⊆ *restricted-ideal-to*$_I$ $\mathcal{V}'$ *J* ⟹
$\mathcal{U}$ ⊆ $\mathcal{V}$ ⟹
restricted-ideal-to$_I$ $\mathcal{U}$ *I* ⊆ *restricted-ideal-to*$_I$ $\mathcal{U}$ *J*›
by (*auto simp*: *restricted-ideal-to-def*)

lemma *pac-ideal-idemp*: ‹*pac-ideal* (*pac-ideal A*) = *pac-ideal A*›

by (*metis dual-order.antisym ideal.span-subset-spanI ideal.span-superset le-sup-iff pac-ideal-def*)

lemma *full-checker-spec*:
assumes ‹(*A*, *A'*) ∈ *polys-rel*›
shows
‹*full-checker spec A pac* ≤ ⇓{((*st*, *G*), (*st'*, *G'*)). (*st*, *st'*) ∈ *status-rel* ∧
(*st* ≠ *FAILED* ⟶ (*G*, *G'*) ∈ *polys-rel-full*)}
(*PAC-checker-specification spec* (*A'*))›
proof −
have *H*: ‹*set-mset b* ⊆ *pac-ideal* (*set-mset* (*ran-m A*)) ⟹
x ∈ *pac-ideal* (*set-mset b*) ⟹ *x* ∈ *pac-ideal* (*set-mset A'*)› **for** *b x*
using *assms* **apply** −
by (*drule pac-ideal-mono*) (*auto simp*: *polys-rel-def pac-ideal-idemp*)
have *1*: ‹*x* ∈ {(*st*, *V'*, *A'*).
(¬ *is-failed st* ⟶ *pac-ideal* (*set-mset* (*ran-m x2*)) =
pac-ideal (*set-mset* (*ran-m A'*)) ∧
⋃ (*vars* ‘ *set-mset* (*ran-m ABC*)) ⊆ *V'* ∧
⋃ (*vars* ‘ *set-mset* (*ran-m A'*)) ⊆ *V'*) ∧
(*st* = *FOUND* ⟶ *speca* ∈# *ran-m A'*)} ⟹
x = (*st*, *x'*) ⟹ *x'* = (*V*, *Aa*) ⟹((*V'*, *Aa*), *V'*, *ran-m Aa*) ∈ *polys-rel-full*› **for** *Aa speca x2 st x*
V' V x' ABC
by (*auto simp*: *polys-rel-def polys-rel-full-def*)
have *H1*: ‹⋀*a aa b xa x x1a x1 x2 speca*.
vars spec ⊆ *x1b* ⟹
⋃ (*vars* ‘ *set-mset* (*ran-m A*)) ⊆ *x1b* ⟹
⋃ (*vars* ‘ *set-mset* (*ran-m x2a*)) ⊆ *x1b* ⟹
restricted-ideal-to$_I$ *x1b b* ⊆ *restricted-ideal-to*$_I$ *x1b* (*ran-m x2a*) ⟹
xa ∈ *restricted-ideal-to*$_I$ (⋃ (*vars* ‘ *set-mset* (*ran-m A*)) ∪ *vars spec*) *b* ⟹
xa ∈ *restricted-ideal-to*$_I$ (⋃ (*vars* ‘ *set-mset* (*ran-m A*)) ∪ *vars spec*) (*ran-m x2a*)›
for *x1b b xa x2a*
by (*drule restricted-ideal-to-mono*[*of* - - - - ‹⋃ (*vars* ‘ *set-mset* (*ran-m A*)) ∪ *vars spec*›])
auto
have *H2*: ‹⋀*a aa b speca x2 x1a x1b x2a*.
spec − *speca* ∈ *More-Modules.ideal polynomial-bool* ⟹
vars spec ⊆ *x1b* ⟹
⋃ (*vars* ‘ *set-mset* (*ran-m A*)) ⊆ *x1b* ⟹
⋃ (*vars* ‘ *set-mset* (*ran-m x2a*)) ⊆ *x1b* ⟹
speca ∈ *pac-ideal* (*set-mset* (*ran-m x2a*)) ⟹
restricted-ideal-to$_I$ *x1b b* ⊆ *restricted-ideal-to*$_I$ *x1b* (*ran-m x2a*) ⟹
spec ∈ *pac-ideal* (*set-mset* (*ran-m x2a*))›
by (*metis* (*no-types*, *lifting*) *group-eq-aux ideal.span-add ideal.span-base in-mono*
pac-ideal-alt-def sup.cobounded2)

show *?thesis*
supply[[*goals-limit=1*]]
unfolding *full-checker-def normalize-poly-spec-def*
PAC-checker-specification-def remap-polys-change-all-def
apply (*refine-vcg PAC-checker-PAC-checker-specification2*[*THEN order-trans*, *of* - -]
lhs-step-If)
subgoal by (*auto simp*: *is-failed-def RETURN-RES-refine-iff*)
apply (*rule 1*; *assumption*)
subgoal
using *fmap-ext assms* **by** (*auto simp*: *polys-rel-def ran-m-def*)
subgoal
by *auto*

subgoal
by *auto*
subgoal for *speca x1 x2 x x1a x2a x1b*
apply (*rule ref-two-step*[*OF conc-fun-R-mono*])
apply *auto*[]
using *assms*
by (*auto simp add*: *PAC-checker-specification-spec-def conc-fun-RES polys-rel-def H1 H2*
polys-rel-full-def
dest!: *rtranclp-PAC-Format-subset-ideal dest*: *is-failed-is-success-completeD*)
done
qed

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*›
using *full-checker-spec*
by (*auto intro*!: *frefI nres-relI*)

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 by using an injective function that converts a string to a natural number. It exists because strings are countable. Remark that the whole development is independent of the function.

7.1 Polynomials and Variables

lemma *poly-embed-EX*:
‹∃φ. *bij* (φ :: *string* ⇒ *nat*)›
by (*rule countableE-infinite*[*of* ‹*UNIV* :: *string set*›])
(*auto intro*!: *infinite-UNIV-listI*)

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*›
by (*auto simp*: *normalized-poly-def*)

lemma *normalized-poly-mono*:
‹*normalized-poly B* ⟹ *A* ⊆# *B* ⟹ *normalized-poly A*›
unfolding *normalized-poly-def*
by (*auto intro*: *distinct-mset-mono image-mset-subseteq-mono*)

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* ⇒ *mset-polynomial* ⇒ *mset-polynomial*› **where**
‹*mult-poly-raw p q* =
(*sum-mset* ((λ*y*. *mult-poly-by-monom y q*) ‘# *p*))›

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

definition *all-vars-mset* :: ‹*mset-polynomial* ⇒ *string multiset*› **where**
‹*all-vars-mset p* = ∑# (*fst* ‘# *p*)›

abbreviation *all-vars* :: ‹*mset-polynomial* ⇒ *string set*› **where**
‹*all-vars p* ≡ *set-mset* (*all-vars-mset p*)›

definition *add-to-coefficient* :: ‹- ⇒ *mset-polynomial* ⇒ *mset-polynomial*› **where**
‹*add-to-coefficient* = (λ(*a*, *n*) *b*. {#(*a*′, -) ∈# *b*. *a*′ ≠ *a*#} +
(*if n* + *sum-mset* (*snd* ‘# {#(*a*′, -) ∈# *b*. *a*′ = *a*#}) = *0 then* {#}
else {#(*a*, *n* + *sum-mset* (*snd* ‘# {#(*a*′, -) ∈# *b*. *a*′ = *a*#}))#}))›

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

lemma *add-to-coefficient-simps*:
‹*n* + *sum-mset* (*snd* ‘# {#(*a*′, -) ∈# *b*. *a*′ = *a*#}) ≠ *0* ⟹
add-to-coefficient (*a*, *n*) *b* = {#(*a*′, -) ∈# *b*. *a*′ ≠ *a*#} +
{#(*a*, *n* + *sum-mset* (*snd* ‘# {#(*a*′, -) ∈# *b*. *a*′ = *a*#}))#}›
‹*n* + *sum-mset* (*snd* ‘# {#(*a*′, -) ∈# *b*. *a*′ = *a*#}) = *0* ⟹
add-to-coefficient (*a*, *n*) *b* = {#(*a*′, -) ∈# *b*. *a*′ ≠ *a*#}› **and**
add-to-coefficient-simps-If:
‹*add-to-coefficient* (*a*, *n*) *b* = {#(*a*′, -) ∈# *b*. *a*′ ≠ *a*#} +
(*if n* + *sum-mset* (*snd* ‘# {#(*a*′, -) ∈# *b*. *a*′ = *a*#}) = *0 then* {#}
else {#(*a*, *n* + *sum-mset* (*snd* ‘# {#(*a*′, -) ∈# *b*. *a*′ = *a*#}))#})›
by (*auto simp*: *add-to-coefficient-def*)

interpretation *comp-fun-commute* ‹*add-to-coefficient*›
proof −
have [*iff*]:
‹*a* ≠ *aa* ⟹
((*case x of* (*a*′, -) ⇒ *a*′ = *a*) ∧ (*case x of* (*a*′, -) ⇒ *a*′ ≠ *aa*)) ⟷

```
    (case x of (a', -) ⇒ a' = a)› for a' aa a x
    by auto
  show ‹comp-fun-commute add-to-coefficient›
    unfolding add-to-coefficient-def
    by standard
     (auto intro!: ext simp: filter-filter-mset ac-simps add-eq-0-iff)
qed

lemma normalized-poly-normalize-poly[simp]:
  ‹normalized-poly (normalize-poly p)›
  unfolding normalize-poly-def
  apply (induction p)
  subgoal by auto
  subgoal for x p
    by (cases x)
      (auto simp: add-to-coefficient-simps-If
      intro: normalized-poly-mono)
  done
```

7.2 Addition

```
inductive add-poly-p :: ‹mset-polynomial × mset-polynomial × mset-polynomial ⇒ mset-polynomial ×
mset-polynomial × mset-polynomial ⇒ 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)›
  apply (induction p arbitrary: r)
  subgoal by auto
  subgoal
    by (metis (no-types, lifting) add-new-coeff-l r-into-rtranclp
      rtranclp-trans union-mset-add-mset-left union-mset-add-mset-right)
  done

lemma add-poly-p-empty-r:
  ‹add-poly-p** (p, q, r) (p, {#}, q + r)›
  apply (induction q arbitrary: r)
  subgoal by auto
  subgoal
    by (metis (no-types, lifting) add-new-coeff-r r-into-rtranclp
      rtranclp-trans union-mset-add-mset-left union-mset-add-mset-right)
  done
```

lemma *add-poly-p-sym*:
‹*add-poly-p* (*p*, *q*, *r*) (*p'*, *q'*, *r'*) ⟷ *add-poly-p* (*q*, *p*, *r*) (*q'*, *p'*, *r'*)›
apply (*rule iffI*)
subgoal
by (*cases rule*: *add-poly-p.cases*, *assumption*)
(*auto intro*: *add-poly-p.intros*)
subgoal
by (*cases rule*: *add-poly-p.cases*, *assumption*)
(*auto intro*: *add-poly-p.intros*)
done

lemma *wf-if-measure-in-wf*:
‹*wf R* ⟹ (⋀*a b*. (*a*, *b*) ∈ *S* ⟹ (*ν a*, *ν b*)∈*R*) ⟹ *wf S*›
by (*metis in-inv-image wfE-min wfI-min wf-inv-image*)

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*)))›
apply (*cases n*)
apply *simp*
by (*auto simp*: *map-prod-def image-iff lex-prod-def*)

lemma *wf-add-poly-p*:
‹*wf* {(*x*, *y*). *add-poly-p y x*}›
by (*rule wf-if-measure-in-wf*[**where** *R* = ‹*lexn less-than 3*› **and**
ν = ‹λ(*a*,*b*,*c*). [*size a* , *size b*, *size c*]›])
(*auto simp*: *add-poly-p.simps wf-lexn*
simp: *lexn-n simp del*: *lexn.simps*(*2*))

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 −
interpret *comp-fun-commute* ‹(λ*xs*. *add-mset* (*xs* + *t*))› **for** *t*
by *standard auto*
show
‹*mult-poly-by-monom t* (*ps* + *qs*) = *mult-poly-by-monom t ps* + *mult-poly-by-monom t qs*› **for** *t*
by (*induction ps*)
(*auto simp*: *mult-poly-by-monom-def*)
show
‹*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*)›
‹*mult-poly-by-monom t* {#} = {#}›**for** *t*
by (*auto simp*: *mult-poly-by-monom-def*)
qed

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 *φ* :: ‹*string* ⇒ *nat*›
assumes *φ-inj*: ‹*inj φ*›
begin

definition *poly-of-vars* :: *term-poly* ⇒ ('*a* :: {*comm-semiring-1*}) *mpoly* **where**
‹*poly-of-vars xs* = *fold-mset* (λ*a b*. *Var* (*φ a*) ∗ *b*) (*1* :: '*a mpoly*) *xs*›

lemma *poly-of-vars-simps*[*simp*]:
shows
‹*poly-of-vars* (*add-mset x xs*) = *Var* (*φ 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 −
interpret *comp-fun-commute* ‹(λ*a b*. (*b* :: '*a* :: {*comm-semiring-1*} *mpoly*) ∗ *Var* (*φ a*))›
by *standard*
(*auto simp*: *algebra-simps ac-simps*
Var-def times-monomial-monomial intro!: *ext*)

show *?A*
by (*auto simp*: *poly-of-vars-def comp-fun-commute-axioms fold-mset-fusion*
ac-simps)
show *?B*
apply (*auto simp*: *poly-of-vars-def ac-simps*)
by (*simp add*: *local.comp-fun-commute-axioms local.fold-mset-fusion*
semiring-normalization-rules(*18*))
qed

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

interpretation *comp-fun-commute* ‹*mononom-of-vars*›
by *standard*
(*auto simp*: *algebra-simps ac-simps mononom-of-vars-def*

Var-def times-monomial-monomial intro!: *ext*)

lemma [*simp*]:
‹*poly-of-vars* {#} = *1*›
by (*auto simp*: *poly-of-vars-def*)

lemma *mononom-of-vars-add*[*simp*]:
‹*NO-MATCH 0 b* ⟹ *mononom-of-vars xs b* = *Const* (*snd xs*) ∗ *poly-of-vars* (*fst xs*) + *b*›
by (*cases xs*)
(*auto simp*: *ac-simps mononom-of-vars-def*)

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*›
by (*auto simp*: *ac-simps Const-def polynomial-of-mset-def*)

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*›
by (*cases x*)
(*auto simp*: *ac-simps polynomial-of-mset-def mononom-of-vars-def*)

lemma *polynomial-of-mset-empty*[*simp*]:
‹*polynomial-of-mset* {#} = *0*›
by (*auto simp*: *polynomial-of-mset-def*)

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*)›
by (*induction ys*)
(*auto simp*: *Const-mult algebra-simps*)

lemma *polynomial-of-mset-mult-poly-raw*[*simp*]:
‹*polynomial-of-mset* (*mult-poly-raw xs ys*) = *polynomial-of-mset xs* ∗ *polynomial-of-mset ys*›
unfolding *mult-poly-raw-def*
by (*induction xs arbitrary*: *ys*)
(*auto simp*: *Const-mult algebra-simps*)

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

lemma *X2-X-polynomial-bool-mult-in*:
‹*Var* (*x1*) ∗ (*Var* (*x1*) ∗ *p*) − *Var* (*x1*) ∗ *p* ∈ *More-Modules.ideal polynomial-bool*›
using *ideal-mult-right-in*[*OF X2-X-in-pac-ideal*[*of x1* ‹{}›, *unfolded pac-ideal-def*], *of p*]
by (*auto simp*: *right-diff-distrib ac-simps power2-eq-square*)

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

```
  case empty
  then show ‹?case› by (auto simp: remove-powers-def ideal.span-zero)
next
  case (add x xs)
  have H1: ‹x1 ∈# x2 ⟹
       Var (φ x1) * poly-of-vars x2 − p ∈ More-Modules.ideal polynomial-bool ⟷
       poly-of-vars x2 − p ∈ More-Modules.ideal polynomial-bool
       › for x1 x2 p
    apply (subst (2) ideal.span-add-eq[symmetric,
      of ‹Var (φ x1) * poly-of-vars x2 − poly-of-vars x2›])
    apply (drule multi-member-split)
    apply (auto simp: X2-X-polynomial-bool-mult-in)
    done

  have diff: ‹poly-of-vars (x) − poly-of-vars (remdups-mset (x)) ∈ ideal polynomial-bool› for x
    by (induction x)
     (auto simp: remove-powers-def ideal.span-zero H1
      simp flip: right-diff-distrib intro!: ideal.span-scale)
  have [simp]: ‹polynomial-of-mset xs −
    polynomial-of-mset (apfst remdups-mset ‘# xs)
    ∈ More-Modules.ideal polynomial-bool ⟹
    poly-of-vars ys * poly-of-vars ys −
    poly-of-vars ys * poly-of-vars (remdups-mset ys)
    ∈ More-Modules.ideal polynomial-bool ⟹
    polynomial-of-mset xs + Const y * poly-of-vars ys −
    (polynomial-of-mset (apfst remdups-mset ‘# xs) +
    Const y * poly-of-vars (remdups-mset ys))
    ∈ More-Modules.ideal polynomial-bool› for y ys
    by (metis add-diff-add diff ideal.scale-right-diff-distrib ideal.span-add ideal.span-scale)
  show ?case
    using add
    apply (cases x)
    subgoal for ys y
      using ideal-mult-right-in2[OF diff, of ‹poly-of-vars ys› ys]
      by (auto simp: remove-powers-def right-diff-distrib
        ideal.span-diff ideal.span-add field-simps)
    done
qed

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')›
  apply (induction rule: add-poly-p-induct)
  subgoal
    by auto
  subgoal
    by auto
  subgoal
    by (auto simp: algebra-simps Const-add)
  subgoal
    by (auto simp: algebra-simps Const-add)
  subgoal
    by (auto simp: algebra-simps Const-add)
  done
```

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'*)›
  by (*induction rule*: *rtranclp-induct*[*of add-poly-p* ‹(-, -, -)› ‹(-, -, -)›, *split-format*(*complete*), *of* **for** *r*])
    (*auto dest*: *add-poly-p-polynomial-of-mset*)

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*)›
  by (*drule rtranclp-add-poly-p-polynomial-of-mset*)
    (*auto simp*: *ac-simps add-eq-0-iff*)

lemma *poly-of-vars-remdups-mset*:
  ‹*poly-of-vars* (*remdups-mset* (*xs*)) − (*poly-of-vars xs*)
    ∈ *More-Modules.ideal polynomial-bool*›
  apply (*induction xs*)
  subgoal by (*auto simp*: *ideal.span-zero*)
  subgoal for *x xs*
    apply (*cases* ‹*x* ∈# *xs*›)
     apply (*metis* (*no-types*, *lifting*) *X2-X-polynomial-bool-mult-in diff-add-cancel diff-diff-eq2*
        ideal.span-diff insert-DiffM poly-of-vars-simps(*1*) *remdups-mset-singleton-sum*)
    by (*metis* (*no-types*, *lifting*) *ideal.span-scale poly-of-vars-simps*(*1*) *remdups-mset-singleton-sum*
        right-diff-distrib)
  done

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 (*induction q*)
  case *empty*
  then show *?case* **by** (*auto simp*: *algebra-simps ideal.span-zero*)
next
  case (*add x q*)
  then have *uP*: ‹− *?P q* ∈ *More-Modules.ideal polynomial-bool*›
    using *ideal.span-neg* **by** *blast*
  have ‹ *Const b* ∗ (*Const m* ∗ *poly-of-vars* (*remdups-mset* (*a* + *xs*))) −
          Const b ∗ (*Const m* ∗ (*poly-of-vars a* ∗ *poly-of-vars xs*))
          ∈ *More-Modules.ideal polynomial-bool*› **for** *a b*
    by (*auto simp*: *Const-mult simp flip*: *right-diff-distrib'* *poly-of-vars-simps*
        intro!: *ideal.span-scale poly-of-vars-remdups-mset*)
  then show *?case*
    apply (*subst ideal.span-add-eq2*[*symmetric*, *OF uP*])
    apply (*cases x*)
    apply (*auto simp*: *field-simps Const-mult simp flip*:
        intro!: *ideal.span-scale poly-of-vars-remdups-mset*)
    done
qed

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 (induction rule: mult-poly-p-induct)
  case (mult-step xs n p r)
  show ?case
    by (auto simp: algebra-simps polynomial-of-mset-mult-map)
qed

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›
 apply (induction p' r' rule: rtranclp-induct[of ‹mult-poly-p q› ‹(p, r)› ‹(p', q')› for p' q', split-format(complete)])
  subgoal
    by (auto simp: ideal.span-zero)
  subgoal for a b aa ba
    apply (drule mult-poly-p-mult-ideal)
    apply (drule ideal.span-add)
    apply assumption
    by (auto simp: group-add-class.diff-add-eq-diff-diff-swap
        add.inverse-distrib-swap ac-simps add-diff-eq
      simp flip:  diff-add-eq-diff-diff-swap)
  done

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›
  by (drule rtranclp-mult-poly-p-mult-ideal) auto

lemma normalize-poly-p-poly-of-mset:
  ‹normalize-poly-p p q ⟹ polynomial-of-mset p = polynomial-of-mset q›
  apply (induction rule: normalize-poly-p.induct)
  apply (auto simp: Const-add algebra-simps)
  done


lemma rtranclp-normalize-poly-p-poly-of-mset:
  ‹normalize-poly-p** p q ⟹ polynomial-of-mset p = polynomial-of-mset q›
  by (induction rule: rtranclp-induct)
    (auto simp: normalize-poly-p-poly-of-mset)

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
```

‹$\varphi' = \textit{the-inv-into}\ V\ \varphi$›

lemma φ'-φ[*simp*]:
‹$x \in V \Longrightarrow \varphi'\ (\varphi\ x) = x$›
using φ-*bij* **unfolding** φ'-*def*
by (*meson bij-betw-imp-inj-on the-inv-into-f-f*)

lemma φ-φ'[*simp*]:
‹$x \in N \Longrightarrow \varphi\ (\varphi'\ x) = x$›
using φ-*bij* **unfolding** φ'-*def*
by (*meson f-the-inv-into-f-bij-betw*)

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) $\in [P]_f\ A \to \langle B\rangle$*nres-rel* $\Longrightarrow$
($\bigwedge a\ b\ x'.\ P\ x' \Longrightarrow ((a,\ b),\ x') \in A \Longrightarrow f\ a\ b \leq \Downarrow B\ (g\ x')$)›
unfolding *fref-def uncurry-def nres-rel-def*
by *auto*

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) $\in [P]_f\ A \to \langle B\rangle$*nres-rel* $\Longrightarrow$
($\bigwedge a\ b\ x'.\ P\ (a,b) \Longrightarrow (x',\ (a,\ b)) \in A \Longrightarrow f\ x' \leq \Downarrow B\ (g\ a\ b)$)›
unfolding *fref-def uncurry-def nres-rel-def*
by *auto*

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* ⇒ *char* ⇒ *bool*› **where**
‹*less-eq-char c d* = (((*of-char c*) :: *nat*) ≤ *of-char d*)›

definition *less-char* :: ‹*char* ⇒ *char* ⇒ *bool*› **where**

‹*less-char c d* = (((*of-char c*) :: *nat*) < *of-char d*)›

global-interpretation *char*: *linorder less-eq-char less-char*
using *linorder-char*
unfolding *linorder-class-def class.linorder-def*
less-eq-char-def[*symmetric*] *less-char-def*[*symmetric*]
class.order-def order-class-def
class.preorder-def preorder-class-def
ord-class-def
apply *auto*
done

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

lemma *less-than-char-def*:
‹(*x*,*y*) ∈ *less-than-char* ⟷ *less-char x y*›
by (*auto simp*: *p2rel-def*)

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*›
by (*auto simp*: *less-than-char-def trans-def irrefl-def antisym-def*)

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'* = {#}›
by (*cases s'*)
(*auto simp*: *sorted-poly-list-rel-wrt-def list-mset-rel-def br-def*)

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'* = {#}›
by (*auto simp*: *fully-unsorted-poly-list-rel-def list-mset-rel-def br-def*)

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'* = {#}›
by (*auto simp*: *poly-list-rel-with0-def list-mset-rel-def br-def*)

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*›
apply (*auto simp*: *term-poly-list-rel-def*; *fail*)+
by (*metis* (*mono-tags*, *lifting*) *CollectD UnI2 rel2p-def sorted-wrt-mono-rel split-conv*
term-poly-list-rel-def)

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

8.3 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
     (λadd-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›
  by (auto simp: nonzero-coeffs-def)

lemma nonzero-coeffsD:
  ‹nonzero-coeffs a ⟹ (x, n) ∈# a ⟹ n ≠ 0›
  by (auto simp: nonzero-coeffs-def)

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›
  unfolding sorted-poly-list-rel-wrt-def prod.case mem-Collect-eq
    list-rel-def
  apply (clarsimp)
  apply (subst (asm) list.rel-sel)
  apply (intro conjI)
  apply (rename-tac y, rule-tac b = ‹tl y› in relcompI)
  apply (auto simp: sorted-poly-list-rel-wrt-def list-mset-rel-def br-def
    list.tl-def term-poly-list-rel-def nonzero-coeffs-def split: list.splits)
  done

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*›
apply (*rule iffI*)
subgoal
by (*auto dest*!: *sorted-poly-list-rel-ConsD*)
subgoal
unfolding *sorted-poly-list-rel-wrt-def prod.case mem-Collect-eq*
list-rel-def
apply (*clarsimp*)
apply (*intro conjI*)
apply (*rename-tac y*; *rule-tac b* = ‹(*mset ys*, *n*) # *y*› **in** *relcompI*)
by (*auto simp*: *sorted-poly-list-rel-wrt-def list-mset-rel-def br-def*
term-poly-list-rel-def add-mset-eq-add-mset eq-commute[*of* - ‹*mset* -›]
nonzero-coeffs-def
dest!: *multi-member-split*)
done

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*›
unfolding *sorted-repeat-poly-list-rel-wrt-def prod.case mem-Collect-eq*
list-rel-def
apply (*clarsimp*)
apply (*subst* (*asm*) *list.rel-sel*)
apply (*intro conjI*)
apply (*rename-tac y*, *rule-tac b* = ‹*tl y*› **in** *relcompI*)
apply (*auto simp*: *sorted-poly-list-rel-wrt-def list-mset-rel-def br-def*
list.tl-def term-poly-list-rel-def nonzero-coeffs-def split: *list.splits*)
done

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*›
apply (*rule iffI*)
subgoal
by (*auto dest*!: *sorted-repeat-poly-list-rel-ConsD*)
subgoal
unfolding *sorted-repeat-poly-list-rel-wrt-def prod.case mem-Collect-eq*
list-rel-def
apply (*clarsimp*)
apply (*intro conjI*)
apply (*rename-tac y*, *rule-tac b* = ‹(*mset ys*, *n*) # *y*› **in** *relcompI*)
by (*auto simp*: *sorted-repeat-poly-list-rel-wrt-def list-mset-rel-def br-def*
term-poly-list-rel-def add-mset-eq-add-mset eq-commute[*of* - ‹*mset* -›]
nonzero-coeffs-def
dest!: *multi-member-split*)
done

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*)›
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule add-poly-p.add-new-coeff-r*)
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule add-poly-p.add-same-coeff-l*)
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*auto intro*: *add-poly-p.rem-0-coeff*)
done

lemma *monoms-add-poly-l'D*:
‹(*aa*, *ba*) ∈ *set* (*add-poly-l'* *x*) ⟹ *aa* ∈ *fst* ‘ *set* (*fst x*) ∨ *aa* ∈ *fst* ‘ *set* (*snd x*)›
by (*induction x rule*: *add-poly-l'.induct*)
(*auto split*: *if-splits*)

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'*)›
apply (*induction rule*: *rtranclp-induct*[*of add-poly-p* ‹(-, -, -)› ‹(-, -, -)›, *split-format*(*complete*), *of* **for** *r*])
subgoal by *auto*
by (*elim add-poly-pE*)
(*metis* (*no-types*, *lifting*) *Pair-inject add-poly-p.intros rtranclp.simps union-mset-add-mset-right*)+

lemma *add-poly-p-add-mset-comb*:
‹*add-poly-p*** (*A*, *Aa*, {#}) ({#}, {#}, *r*) ⟹
*add-poly-p***
(*add-mset* (*xs*, *n*) *A*, *Aa*, {#})
({#}, {#}, *add-mset* (*xs*, *n*) *r*)›
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule add-poly-p.add-new-coeff-l*)
using *add-poly-p-add-to-result*[*of A Aa* ‹{#}› ‹{#}› ‹{#}› *r* ‹{#(*xs*, *n*)#}›]
by *auto*

lemma *add-poly-p-add-mset-comb2*:
‹*add-poly-p*** (*A*, *Aa*, {#}) ({#}, {#}, *r*) ⟹
*add-poly-p***
(*add-mset* (*ys*, *n*) *A*, *add-mset* (*ys*, *m*) *Aa*, {#})
({#}, {#}, *add-mset* (*ys*, *n* + *m*) *r*)›
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule add-poly-p.add-new-coeff-r*)
apply (*rule converse-rtranclp-into-rtranclp*)
apply (*rule add-poly-p.add-same-coeff-l*)
using *add-poly-p-add-to-result*[*of A Aa* ‹{#}› ‹{#}› ‹{#}› *r* ‹{#(*ys*, *n*+*m*)#}›]
by *auto*

lemma *add-poly-p-add-mset-comb3*:
‹*add-poly-p*** (*A*, *Aa*, {#}) ({#}, {#}, *r*) ⟹
*add-poly-p***

```
      (A, add-mset (ys, m) Aa, {#})
      ({#}, {#}, add-mset (ys, m) r)›
  apply (rule converse-rtranclp-into-rtranclp)
  apply (rule add-poly-p.add-new-coeff-r)
  using add-poly-p-add-to-result[of A Aa ‹{#}› ‹{#}› ‹{#}› r ‹{#(ys, m)#}›]
  by auto

lemma total-on-lexord:
  ‹Relation.total-on UNIV R ⟹ Relation.total-on UNIV (lexord R)›
  apply (auto simp: Relation.total-on-def)
  by (meson lexord-linear)

lemma antisym-lexord:
  ‹antisym R ⟹ irrefl R ⟹ antisym (lexord R)›
  by (auto simp: antisym-def lexord-def irrefl-def
    elim!: list-match-lel-lel)

lemma less-than-char-linear:
  ‹(a, b) ∈ less-than-char ∨
        a = b ∨ (b, a) ∈ less-than-char›
  by (auto simp: less-than-char-def)

lemma total-on-lexord-less-than-char-linear:
  ‹xs ≠ ys ⟹ (xs, ys) ∉ lexord (lexord less-than-char) ⟷
      (ys, xs) ∈ lexord (lexord less-than-char)›
   using lexord-linear[of ‹lexord less-than-char› xs ys]
   using lexord-linear[of ‹less-than-char›] less-than-char-linear
   using lexord-irrefl[OF irrefl-less-than-char]
    antisym-lexord[OF antisym-lexord[OF antisym-less-than-char irrefl-less-than-char]]
   apply (auto simp: antisym-def Relation.total-on-def)
   done

lemma sorted-poly-list-rel-nonzeroD:
  ‹(p, r) ∈ sorted-poly-list-rel term-order ⟹
      nonzero-coeffs (r)›
  ‹(p, r) ∈ sorted-poly-list-rel (rel2p (lexord (lexord less-than-char))) ⟹
      nonzero-coeffs (r)›
  by (auto simp: sorted-poly-list-rel-wrt-def nonzero-coeffs-def)


lemma add-poly-l'-add-poly-p:
  assumes ‹(pq, pq') ∈ sorted-poly-rel ×r sorted-poly-rel›
  shows ‹∃ r. (add-poly-l' pq, r) ∈ sorted-poly-rel ∧
                    add-poly-p** (fst pq', snd pq', {#}) ({#}, {#}, r)›
  supply [[goals-limit=1]]
  using assms
  apply (induction ‹pq› arbitrary: pq' rule: add-poly-l'.induct)
  subgoal for p pq'
    using add-poly-p-empty-l[of ‹fst pq'› ‹{#}› ‹{#}›]
    by (cases pq') (auto intro!: exI[of - ‹fst pq'›])
  subgoal for x p pq'
    using add-poly-p-empty-r[of  ‹{#}› ‹snd pq'› ‹{#}›]
    by (cases pq')  (auto intro!: exI[of - ‹snd pq'›])
  subgoal premises p for xs n p ys m q pq'
    apply (cases pq') — Isabelle does a completely stupid case distinction here
```

```
    apply (cases ‹xs = ys›)
    subgoal
      apply (cases ‹n + m = 0›)
      subgoal
        using p(1)[of ‹(remove1-mset (mset xs, n) (fst pq'), remove1-mset (mset ys, m)  (snd pq'))›]
p(5−)
        apply (auto dest!: sorted-poly-list-rel-ConsD multi-member-split
          )
      using add-poly-p-add-mset-sum-0 by blast
    subgoal
        using p(2)[of ‹(remove1-mset (mset xs, n) (fst pq'), remove1-mset (mset ys, m)  (snd pq'))›]
p(5−)
      apply (auto dest!: sorted-poly-list-rel-ConsD multi-member-split)
      apply (rule-tac x = ‹add-mset (mset ys, n + m) r› in exI)
      apply (fastforce dest!: monoms-add-poly-l'D simp: sorted-poly-list-rel-Cons-iff rel2p-def
          sorted-poly-list-rel-nonzeroD var-order-rel-def
        intro: add-poly-p-add-mset-comb2)
      done
   done
    subgoal
      apply (cases ‹(xs, ys) ∈ term-order-rel›)
      subgoal
        using p(3)[of ‹(remove1-mset (mset xs, n) (fst pq'), (snd pq'))›] p(5−)
        apply (auto dest!: multi-member-split simp: sorted-poly-list-rel-Cons-iff rel2p-def)
        apply (rule-tac x = ‹add-mset (mset xs, n) r› in exI)
        apply (auto dest!: monoms-add-poly-l'D)
        apply (auto intro: lexord-trans add-poly-p-add-mset-comb simp: lexord-transI var-order-rel-def)
        apply (rule lexord-trans)
        apply assumption
        apply (auto intro: lexord-trans add-poly-p-add-mset-comb simp: lexord-transI
          sorted-poly-list-rel-nonzeroD var-order-rel-def)
        using total-on-lexord-less-than-char-linear by fastforce

      subgoal
        using p(4)[of ‹(fst pq', remove1-mset (mset ys, m) (snd pq'))›] p(5−)
        apply (auto dest!: multi-member-split simp: sorted-poly-list-rel-Cons-iff rel2p-def
          var-order-rel-def)
        apply (rule-tac x = ‹add-mset (mset ys, m) r› in exI)
        apply (auto dest!: monoms-add-poly-l'D
          simp: total-on-lexord-less-than-char-linear)
        apply (auto intro: lexord-trans add-poly-p-add-mset-comb  simp: lexord-transI
          total-on-lexord-less-than-char-linear var-order-rel-def)
        apply (rule lexord-trans)
        apply assumption
        apply (auto intro: lexord-trans add-poly-p-add-mset-comb3 simp: lexord-transI
          sorted-poly-list-rel-nonzeroD var-order-rel-def)
        using total-on-lexord-less-than-char-linear by fastforce
      done
  done
 done


lemma add-poly-l-add-poly:
  ‹add-poly-l x = RETURN (add-poly-l' x)›
  unfolding add-poly-l-def
```

by (*induction x rule*: *add-poly-l'.induct*)
(*solves* ‹*subst RECT-unfold*, *refine-mono*, *simp split*: *list.split*›)+

lemma *add-poly-l-spec*:
‹(*add-poly-l*, *uncurry* (λ*p q*. *SPEC*(λ*r*. *add-poly-p*** (*p*, *q*, {#}) ({#}, {#}, *r*)))) ∈
sorted-poly-rel ×$_r$ *sorted-poly-rel* →$_f$ ⟨*sorted-poly-rel*⟩*nres-rel*›
unfolding *add-poly-l-add-poly*
apply (*intro nres-relI frefI*)
apply (*drule add-poly-l'-add-poly-p*)
apply (*auto simp*: *conc-fun-RES*)
done

definition *sort-poly-spec* :: ‹*llist-polynomial* ⇒ *llist-polynomial nres*› **where**
‹*sort-poly-spec p* =
SPEC(λ*p'*. *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 −
obtain *y* **where**
py: ‹(*p*, *y*) ∈ ⟨*term-poly-list-rel* ×$_r$ *int-rel*⟩*list-rel*› **and**
p'-y: ‹*p'* = *mset y*› **and**
zero: ‹*0* ∉# *snd* `# *p'*›
using *assms*
unfolding *sort-poly-spec-def poly-list-rel-def sorted-poly-list-rel-wrt-def*
by (*auto simp*: *list-mset-rel-def br-def*)
then have [*simp*]: ‹*length y* = *length p*›
by (*auto simp*: *list-rel-def list-all2-conv-all-nth*)
have *H*: ‹(*x*, *p'*)
∈ ⟨*term-poly-list-rel* ×$_r$ *int-rel*⟩*list-rel O list-mset-rel*›
if *px*: ‹*mset p* = *mset x*› **and** ‹*sorted-wrt* (*rel2p* (*Id* ∪ *lexord var-order-rel*)) (*map fst x*)›
for *x* :: ‹*llist-polynomial*›
proof −
from *px* **have** ‹*length x* = *length p*›
by (*metis size-mset*)
from *px* **have** ‹*mset x* = *mset p*›
by *simp*
then obtain *f* **where** ‹*f permutes* {..<*length p*}› ‹*permute-list f p* = *x*›
by (*rule mset-eq-permutation*)
with ‹*length x* = *length p*› **have** *f*: ‹*bij-betw f* {..<*length x*} {..<*length p*}›
by (*simp add*: *permutes-imp-bij*)
from ‹*f permutes* {..<*length p*}› ‹*permute-list f p* = *x*› [*symmetric*]
have [*simp*]: ‹⋀*i*. *i* < *length x* ⟹ *x* ! *i* = *p* ! (*f i*)›
by (*simp add*: *permute-list-nth*)
let *?y* = ‹*map* (λ*i*. *y* ! *f i*) [*0* ..< *length x*]›
have ‹*i* < *length y* ⟹ (*p* ! *f i*, *y* ! *f i*) ∈ *term-poly-list-rel* ×$_r$ *int-rel*› **for** *i*
using *list-all2-nthD*[*of* - *p y*
‹*f i*›, *OF py*[*unfolded list-rel-def mem-Collect-eq prod.case*]]
mset-eq-length[*OF px*] *f*
by (*auto simp*: *list-rel-def list-all2-conv-all-nth bij-betw-def*)
then have ‹(*x*, *?y*) ∈ ⟨*term-poly-list-rel* ×$_r$ *int-rel*⟩*list-rel*› **and**
xy: ‹*length x* = *length y*›
using *py list-all2-nthD*[*of* ‹*rel2p* (*term-poly-list-rel* ×$_r$ *int-rel*)› *p y*
‹*f i*› **for** *i*, *simplified*] *mset-eq-length*[*OF px*]

```
    by (auto simp: list-rel-def list-all2-conv-all-nth)
  moreover {
    have f: ‹mset-set {0..<length x} = f `# mset-set {0..<length x}›
      using f mset-eq-length[OF px]
      by (auto simp: bij-betw-def lessThan-atLeast0 image-mset-mset-set)
    have ‹mset y = {#y ! f x. x ∈# mset-set {0..<length x}#}›
      by (subst drop-0[symmetric], subst mset-drop-upto, subst xy[symmetric], subst f)
        auto
    then have ‹(?y, p') ∈ list-mset-rel›
      by (auto simp: list-mset-rel-def br-def p'-y)
  }
  ultimately show ?thesis
    by (auto intro!: relcompI[of - ?y])
qed
show ?thesis
  using zero
  unfolding sort-poly-spec-def poly-list-rel-def sorted-repeat-poly-list-rel-wrt-def
  by refine-rcg (auto intro: H)
qed
```

8.4 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' = {#}›
  by (auto simp: term-poly-list-rel-def)

lemma mset-eqD-set-mset: ‹mset xs = A ⟹ set xs = set-mset A›
  by auto

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)›
  by (auto simp: term-poly-list-rel-def rel2p-def dest!: multi-member-split mset-eqD-set-mset)

lemma var-order-rel-antisym[simp]:
  ‹(y, y) ∉ var-order-rel›
  by (simp add: less-than-char-def lexord-irreflexive var-order-rel-def)

lemma term-poly-list-rel-remdups-mset:
  ‹(p, p') ∈ term-poly-list-rel ⟹
    (p, remdups-mset p') ∈ term-poly-list-rel›
 by (auto simp: term-poly-list-rel-def distinct-mset-remdups-mset-id simp flip: distinct-mset-mset-distinct)

lemma var-notin-notin-mult-monomsD:
  ‹y ∈ set (mult-monoms p q) ⟹ y ∈ set p ∨ y ∈ set q›
  by (induction p q arbitrary: p' q' rule: mult-monoms.induct) (auto split: if-splits)
```

lemma *term-poly-list-rel-set-mset*:
‹(*p*, *q*) ∈ *term-poly-list-rel* ⟹ *set p* = *set-mset q*›
by (*auto simp*: *term-poly-list-rel-def*)

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 −
have [*dest*]: ‹*remdups-mset* (*A* + *Aa*) = *add-mset y Ab* ⟹ *y* ∉# *A* ⟹
y ∉# *Aa* ⟹
False› **for** *Aa Ab y A*
by (*metis set-mset-remdups-mset union-iff union-single-eq-member*)
show *?thesis*
apply (*intro fun-relI*)
apply (*rename-tac p p′ q q′*)
subgoal for *p p′ q q′*
apply (*induction p q arbitrary*: *p′ q′ rule*: *mult-monoms.induct*)
subgoal by (*auto simp*: *term-poly-list-rel-Cons-iff rel2p-def term-poly-list-rel-remdups-mset*)
subgoal for *x p p′ q′*
by (*auto simp*: *term-poly-list-rel-Cons-iff rel2p-def term-poly-list-rel-remdups-mset*
dest!: *multi-member-split*[*of* - *q′*])
subgoal premises *p* **for** *x p y q p′ q′*
apply (*cases* ‹*x* = *y*›)
subgoal
using *p*(*1*)[*of* ‹*remove1-mset y p′*› ‹*remove1-mset y q′*›] *p*(*4*−)
by (*auto simp*: *term-poly-list-rel-Cons-iff rel2p-def*
dest!: *var-notin-notin-mult-monomsD*
dest!: *multi-member-split*)
apply (*cases* ‹(*x*, *y*) ∈ *var-order-rel*›)
subgoal
using *p*(*2*)[*of* ‹*remove1-mset x p′*› ‹*q′*›] *p*(*4*−)
apply (*auto simp*: *term-poly-list-rel-Cons-iff*
term-poly-list-rel-set-mset rel2p-def var-order-rel-def
dest!: *multi-member-split*[*of* - *p′*] *multi-member-split*[*of* - *q′*]
var-notin-notin-mult-monomsD
split: *if-splits*)
apply (*meson lexord-cons-cons list.inject total-on-lexord-less-than-char-linear*)
apply (*meson lexord-cons-cons list.inject total-on-lexord-less-than-char-linear*)
apply (*meson lexord-cons-cons list.inject total-on-lexord-less-than-char-linear*)
using *lexord-trans trans-less-than-char var-order-rel-antisym*
unfolding *var-order-rel-def* **apply** *blast*+
done
subgoal
using *p*(*3*)[*of* ‹*p′*› ‹*remove1-mset y q′*›] *p*(*4*−)
apply (*auto simp*: *term-poly-list-rel-Cons-iff rel2p-def*
term-poly-list-rel-set-mset rel2p-def var-order-rel-antisym
dest!: *multi-member-split*[*of* - *p′*] *multi-member-split*[*of* - *q′*]
var-notin-notin-mult-monomsD
split: *if-splits*)
using *lexord-trans trans-less-than-char var-order-rel-antisym*
unfolding *var-order-rel-def* **apply** *blast*
apply (*meson lexord-cons-cons list.inject total-on-lexord-less-than-char-linear*)
by (*meson less-than-char-linear lexord-linear lexord-trans trans-less-than-char*)
done

```
    done
  done
qed

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 (λb 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›
  by (induction f b xs rule: map-append.induct)
   auto

lemma foldl-append-empty:
  ‹NO-MATCH [] xs ⟹ foldl (λb x. f x @ b) xs p = foldl (λb x. f x @ b) [] p @ xs›
  apply (induction p arbitrary: xs)
   apply simp
  by (metis (mono-tags, lifting) NO-MATCH-def append.assoc append-self-conv foldl-Cons)


lemma poly-list-rel-empty-iff[simp]:
  ‹([], r) ∈ poly-list-rel R ⟷ r = {#}›
  by (auto simp: poly-list-rel-def list-mset-rel-def br-def)

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›
  subgoal by (auto simp: mult-poly-raw-def)
  subgoal by (induction p) (auto simp: mult-poly-raw-def foldl-append-empty)
  done

lemma sorted-poly-list-relD:
  ‹(q, q') ∈ sorted-poly-list-rel R ⟹ q' = (λ(a, b). (mset a, b)) `# mset q›
  apply (induction q arbitrary: q')
  apply (auto simp: sorted-poly-list-rel-wrt-def list-mset-rel-def br-def
    list-rel-split-right-iff)
  apply (subst (asm)(2) term-poly-list-rel-def)
  apply (simp add: relcomp.relcompI)
  done

lemma list-all2-in-set-ExD:
  ‹list-all2 R p q ⟹ x ∈ set p ⟹ ∃y ∈ set q. R x y›
  by (induction p q rule: list-all2-induct)
    auto

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′*)›
 apply (*induction rule*: *rtranclp-induct*[*of* ‹*mult-poly-p q′*› ‹(-, *r*)› ‹(*p′*, *q′′*)› **for** *p′ q′′*, *split-format*(*complete*)])
 subgoal by *simp*
 apply (*rule rtranclp.rtrancl-into-rtrancl*)
 apply *assumption*
 by (*auto elim*!: *mult-poly-p-elim intro*: *mult-poly-p.intros*
 intro: *rtranclp.rtrancl-into-rtrancl simp*: *add-mset-commute*[*of x*])

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 −
 have *H*: ‹(*q*, *q′*) ∈ *sorted-poly-list-rel term-order* ⟹ *n* < *length q* ⟹
 distinct aa ⟹ *sorted-wrt var-order aa* ⟹
 (*mult-monoms aa* (*fst* (*q* ! *n*)),
 mset (*mult-monoms aa* (*fst* (*q* ! *n*))))
 ∈ *term-poly-list-rel*› **for** *aa n*
 using *mult-monoms-spec*[*unfolded fun-rel-def*, *simplified*] **apply** −
 apply (*drule bspec*[*of* - - ‹(*aa*, (*mset aa*))›])
 apply (*auto simp*: *term-poly-list-rel-def*)[]
 unfolding *prod.case sorted-poly-list-rel-wrt-def*
 apply *clarsimp*
 subgoal for *y*
 apply (*drule bspec*[*of* - - ‹(*fst* (*q* ! *n*), *mset* (*fst* (*q* ! *n*)))›])
 apply (*cases* ‹*q* ! *n*›; *cases* ‹*y* ! *n*›)
 using *param-nth*[*of n y n q* ‹*term-poly-list-rel* ×$_r$ *int-rel*›]
 by (*auto simp*: *list-rel-imp-same-length term-poly-list-rel-def*)
 done

 have *H′*: ‹(*q*, *q′*) ∈ *sorted-poly-list-rel term-order* ⟹
 distinct aa ⟹ *sorted-wrt var-order aa* ⟹
 (*ab*, *ba*) ∈ *set q* ⟹
 remdups-mset (*mset aa* + *mset ab*) = *mset* (*mult-monoms aa ab*)› **for** *aa n ab ba*
 using *mult-monoms-spec*[*unfolded fun-rel-def*, *simplified*] **apply** −
 apply (*drule bspec*[*of* - - ‹(*aa*, (*mset aa*))›])
 apply (*auto simp*: *term-poly-list-rel-def*)[]
 unfolding *prod.case sorted-poly-list-rel-wrt-def*
 apply *clarsimp*
 subgoal for *y*
 apply (*drule bspec*[*of* - - ‹(*ab*, *mset ab*)›])
 apply (*auto simp*: *list-rel-imp-same-length term-poly-list-rel-def list-rel-def*
 dest: *list-all2-in-set-ExD*)
 done
 done

 have *H*: ‹(*q*, *q′*) ∈ *sorted-poly-list-rel term-order* ⟹
 a = (*aa*, *b*) ⟹
 (*pq*, *r*) ∈ *unsorted-poly-rel* ⟹
 p′ = *add-mset* (*mset aa*, *b*) *A* ⟹
 ∀ *x*∈*set p*. *term-order aa* (*fst x*) ⟹
 sorted-wrt var-order aa ⟹
 distinct aa ⟹ *b*≠ *0* ⟹
 (⋀*aaa*. (*aaa*, *0*) ∉# *q′*) ⟹
 (*pq* @

```
      map (mult-monomials (aa, b)) q,
      {#case x of (ys, n) ⇒ (remdups-mset (mset aa + ys), n * b)
      . x ∈# q'#} +
      r)
     ∈ unsorted-poly-rel› for a p p' pq aa b r
  apply (auto simp: poly-list-rel-def)
  apply (rule-tac b = ‹y @ map (λ(a,b). (mset a, b)) (map (mult-monomials (aa, b)) q)› in relcompI)
  apply (auto simp: list-rel-def list-all2-append list-all2-lengthD H
    list-mset-rel-def br-def mult-monomials-def case-prod-beta intro!: list-all2-all-nthI
    simp: sorted-poly-list-relD)
    apply (subst sorted-poly-list-relD[of q q' term-order])
    apply (auto simp: case-prod-beta H' intro!: image-mset-cong)
  done

 show ?thesis
  using assms
  apply (induction p arbitrary: p')
  subgoal
    by auto
  subgoal premises p for a p p'
    using p(1)[of ‹remove1-mset (mset (fst a), snd a) p'›] p(2−)
    apply (cases a)
    apply (auto simp: sorted-poly-list-rel-Cons-iff
      dest!: multi-member-split)
    apply (rule-tac x = ‹(λ(ys, n). (remdups-mset (mset (fst a) + ys), n * snd a)) '# q' + r› in exI)
    apply (auto 5 3 intro: mult-poly-p.intros simp: intro!: H
      dest: sorted-poly-list-rel-nonzeroD nonzero-coeffsD)
    apply (rule rtranclp-trans)
    apply (rule mult-poly-p-add-mset-same)
    apply assumption
    apply (rule converse-rtranclp-into-rtranclp)
    apply (auto intro!: mult-poly-p.intros simp: ac-simps)
    done
  done
qed

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›
 by (induction p rule: merge-coeffs.induct)  auto


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

apply (*induction p rule*: *merge-coeffs.induct*)
subgoal
by *auto*
subgoal
by *auto*
subgoal
by (*auto split*: *if-splits*)
done

lemma *distinct-merge-coeffs*:
assumes ‹*sorted-wrt R* (*map fst xs*)› **and** ‹*transp R*› ‹*antisymp R*›
shows ‹*distinct* (*map fst* (*merge-coeffs xs*))›
using *assms*
by (*induction xs rule*: *merge-coeffs.induct*)
(*auto 5 4 dest*: *antisympD dest*!: *fst-normalize-polynomial-subsetD*)

lemma *in-set-merge-coeffsD*:
‹(*a*, *b*) ∈ *set* (*merge-coeffs p*) ⟹∃ *b*. (*a*, *b*) ∈ *set p*›
by (*auto dest*!: *fst-normalize-polynomial-subsetD*)

lemma *rtranclp-normalize-poly-add-mset*:
‹*normalize-poly-p*** *A r* ⟹ *normalize-poly-p*** (*add-mset x A*) (*add-mset x r*)›
by (*induction rule*: *rtranclp-induct*)
(*auto dest*: *normalize-poly-p.keep-coeff*[*of* - - *x*])

lemma *nonzero-coeffs-diff*:
‹*nonzero-coeffs A* ⟹ *nonzero-coeffs* (*A* − *B*)›
by (*auto simp*: *nonzero-coeffs-def dest*: *in-diffD*)

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*›
apply (*induction xs arbitrary*: *ys rule*: *merge-coeffs.induct*)
subgoal by (*auto simp*: *sorted-repeat-poly-list-rel-wrt-def sorted-poly-list-rel-wrt-def*)
subgoal
by (*auto simp*: *sorted-repeat-poly-list-rel-wrt-def sorted-poly-list-rel-wrt-def*)
subgoal premises *p* **for** *xs n ys m p ysa*
apply (*cases* ‹*xs* = *ys*›, *cases* ‹*m*+*n* ≠ *0*›)
subgoal
using *p*(*1*)[*of* ‹*add-mset* (*mset ys*, *m*+*n*) *ysa* − {#(*mset ys*, *m*), (*mset ys*, *n*)#}›] *p*(*4*−)
apply (*auto simp*: *sorted-poly-list-rel-Cons-iff ac-simps add-mset-commute*
remove1-mset-add-mset-If nonzero-coeffs-diff sorted-repeat-poly-list-rel-Cons-iff)
apply (*rule-tac x* = ‹*r*› **in** *exI*)
using *normalize-poly-p.merge-dup-coeff*[*of* ‹*ysa* − {#(*mset ys*, *m*), (*mset ys*, *n*)#}› ‹*ysa* − {#(*mset ys*, *m*), (*mset ys*, *n*)#}› ‹*mset ys*› *m n*]
by (*auto dest*!: *multi-member-split simp del*: *normalize-poly-p.merge-dup-coeff*
simp: *add-mset-commute*
intro: *converse-rtranclp-into-rtranclp*)
subgoal
using *p*(*2*)[*of* ‹*ysa* − {#(*mset ys*, *m*), (*mset ys*, *n*)#}›] *p*(*4*−)
apply (*auto simp*: *sorted-poly-list-rel-Cons-iff ac-simps add-mset-commute*
remove1-mset-add-mset-If nonzero-coeffs-diff sorted-repeat-poly-list-rel-Cons-iff)
apply (*rule-tac x* = ‹*r*› **in** *exI*)
using *normalize-poly-p.rem-0-coeff*[*of* ‹*add-mset* (*mset ys*, *m* +*n*) *ysa* − {#(*mset ys*, *m*), (*mset*

ys, *n*)#}› ‹*add-mset* (*mset ys*, *m* +*n*) *ysa* − {#(*mset ys*, *m*), (*mset ys*, *n*)#}› ‹*mset ys*›]
using *normalize-poly-p.merge-dup-coeff*[*of* ‹*ysa* − {#(*mset ys*, *m*), (*mset ys*, *n*)#}› ‹*ysa* − {#(*mset ys*, *m*), (*mset ys*, *n*)#}› ‹*mset ys*› *m n*]
by (*force intro*: *add-mset-commute*[*of* ‹(*mset ys*, *n*)› ‹(*mset ys*, −*n*)›]
converse-rtranclp-into-rtranclp
dest!: *multi-member-split*
simp del: *normalize-poly-p.rem-0-coeff*
simp: *add-eq-0-iff2*
intro: *normalize-poly-p.rem-0-coeff*)
subgoal
using *p*(*3*)[*of* ‹*add-mset* (*mset ys*, *m*) *ysa* − {#(*mset xs*, *n*), (*mset ys*, *m*)#}›] *p*(*4*−)
apply (*auto simp*: *sorted-poly-list-rel-Cons-iff ac-simps add-mset-commute*
remove1-mset-add-mset-If sorted-repeat-poly-list-rel-Cons-iff)
apply (*rule-tac x* = ‹*add-mset* (*mset xs*, *n*) *r*› **in** *exI*)
apply (*auto dest*!: *in-set-merge-coeffsD*)
apply (*auto intro*: *normalize-poly-p.intros rtranclp-normalize-poly-add-mset*
simp: *rel2p-def var-order-rel-def*
dest!: *multi-member-split*
dest: *sorted-poly-list-rel-nonzeroD*)
using *total-on-lexord-less-than-char-linear* **apply** *fastforce*
using *total-on-lexord-less-than-char-linear* **apply** *fastforce*
done
done
done

8.5 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*(λ*xs*. *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›
by (*induction a*) (*auto simp*: *rel2p-def*)

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 −
have *H*: ‹
∀ *x*∈*set xs*′. *sorted-wrt var-order* (*fst x*) ⟹
sorted-wrt (*rel2p* (*Id* ∪ *var-order-rel*)) *x* ⟹
(*aaa*, *ba*) ∈ *set xs*′ ⟹

```
      sorted-wrt (rel2p (Id ∪ var-order-rel)) aaa› for xs  xs' ba aa b x aaa
    by (metis UnCI fst-eqD rel2p-def sorted-wrt-mono-rel)
  show ?thesis
    using assms
    unfolding sort-all-coeffs-def sort-coeff-def
    apply (induction xs arbitrary: xs')
    subgoal
      using assms
      by auto
    subgoal premises p for a xs
      using p(2−)
    apply (cases a, simp only: monadic-nfoldli-simp bind-to-let-conv Let-def if-True Refine-Basic.nres-monad3
        intro-spec-refine-iff prod.case)
      by (auto 5 3 simp: intro-spec-refine-iff image-Un
        dest: same-mset-distinct-iff
        intro!: p(1)[THEN order-trans] distinct-var-order-Id-var-order
        simp: H)
    done
qed

definition shuffle-coefficients where
  ‹shuffle-coefficients xs = (SPEC(λys. 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)›
  unfolding sort-all-coeffs-def shuffle-coefficients-def
  by (rule sort-all-coeffs-gen[THEN order-trans])
   auto

lemma unsorted-term-poly-list-rel-mset:
  ‹(ys, aa) ∈ unsorted-term-poly-list-rel ⟹ mset ys = aa›
  by (auto simp: unsorted-term-poly-list-rel-def)

lemma RETURN-map-alt-def:
  ‹RETURN o (map f) =
    REC_T (λg xs.
      case xs of
        [] ⇒ RETURN []
      | x # xs ⇒ do {xs ← g xs; RETURN (f x # xs)})›
  unfolding comp-def
  apply (subst eq-commute)
  apply (intro ext)
  apply (induct-tac x)
  subgoal
    apply (subst RECT-unfold)
    apply refine-mono
    apply auto
    done
  subgoal
    apply (subst RECT-unfold)
    apply refine-mono
    apply auto
```

done
done

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*›
apply (*auto simp*: *poly-list-rel-def list-rel-split-right-iff list-mset-rel-def br-def*
unsorted-term-poly-list-rel-def
nonzero-coeffs-def fully-unsorted-poly-list-rel-def dest!: *multi-member-split*)
apply *blast*
apply (*rule-tac b* = ‹(*mset ys*, *n*) # *y*› **in** *relcompI*)
apply *auto*
done

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*›
by (*induction s*) (*auto simp*: *term-poly-list-rel-def*
distinct-var-order-Id-var-order)

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*)›
by (*induction p arbitrary*: *y*)
(*auto simp*: *list-rel-split-right-iff*
unsorted-term-poly-list-rel-def)

lemma *shuffle-terms-distinct-iff*:
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 −
have ‹∀ *x*∈*set s*. *distinct* (*fst x*)›
if *m*: ‹*map* (λ(*a*, *y*). (*mset a*, *y*)) *p* = *map* (λ(*a*, *y*). (*mset a*, *y*)) *s*› **and**
dist: ‹∀ *x*∈*set p*. *distinct* (*fst x*)›
for *s p*
proof *standard*+
fix *x*
assume *x*: ‹*x* ∈ *set s*›
obtain *v n* **where** [*simp*]: ‹*x* = (*v*, *n*)› **by** (*cases x*)
then have ‹(*mset v*, *n*) ∈ *set* (*map* (λ(*a*, *y*). (*mset a*, *y*)) *p*)›
using *x* **unfolding** *m* **by** *auto*
then obtain *v*′ **where**
‹(*v*′, *n*) ∈ *set p*› **and**
‹*mset v*′ = *mset v*›
by (*auto simp*: *image-iff*)
then show ‹*distinct* (*fst x*)›
using *dist* **by** (*metis* ‹*x* = (*v*, *n*)› *distinct-mset-mset-distinct fst-conv*)
qed
from *this*[*of p s*] *this*[*of s p*]
show ‹*?thesis*›
unfolding *assms*

by *blast*
qed

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

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 −
have *H*: ‹(*map* (λ(*a*, *y*). (*mset a*, *y*)) (*rev p*)) =
map (λ(*a*, *y*). (*mset a*, *y*)) *s* ⟷
(*map* (λ(*a*, *y*). (*mset a*, *y*)) *p*) =
map (λ(*a*, *y*). (*mset a*, *y*)) (*rev s*)› **for** *s*
by (*auto simp flip*: *rev-map simp*: *eq-commute*[*of* ‹*rev* (*map* - -)› ‹*map* - -›])
have *1*: ‹⋀*s y*. (*p*, *y*) ∈ ⟨*unsorted-term-poly-list-rel* ×$_r$ *int-rel*⟩*list-rel* ⟹
p' = *mset y* ⟹
map (λ(*a*, *y*). (*mset a*, *y*)) (*rev p*) = *map* (λ(*a*, *y*). (*mset a*, *y*)) *s* ⟹
∀ *x*∈*set s*. *sorted-wrt var-order* (*fst x*) ⟹
(*s*, *map* (λ(*a*, *y*). (*mset a*, *y*)) *s*)
∈ ⟨*term-poly-list-rel* ×$_r$ *int-rel*⟩*list-rel*›
by (*auto 4 4 simp*: *rel2p-def*
dest!: *list-rel-unsorted-term-poly-list-relD*
dest: *shuffle-terms-distinct-iff*[*THEN iffD1*]
intro!: *map-mset-unsorted-term-poly-list-rel*
sorted-wrt-mono-rel[*of* - ‹*rel2p* (*var-order-rel*)› ‹*rel2p* (*Id* ∪ *var-order-rel*)›])
have *2*: ‹⋀*s y*. (*p*, *y*) ∈ ⟨*unsorted-term-poly-list-rel* ×$_r$ *int-rel*⟩*list-rel* ⟹
p' = *mset y* ⟹
map (λ(*a*, *y*). (*mset a*, *y*)) (*rev p*) = *map* (λ(*a*, *y*). (*mset a*, *y*)) *s* ⟹
∀ *x*∈*set s*. *sorted-wrt var-order* (*fst x*) ⟹
mset y = {#*case x of* (*a*, *x*) ⇒ (*mset a*, *x*). *x* ∈# *mset s*#}›
by (*metis* (*no-types*, *lifting*) *list-rel-unsorted-term-poly-list-relD mset-map mset-rev*)
show *?thesis*
apply (*rule sort-all-coeffs*[*THEN order-trans*])
using *assms*
by (*auto simp*: *shuffle-coefficients-def poly-list-rel-def*
RETURN-def fully-unsorted-poly-list-rel-def list-mset-rel-def
br-def dest: *list-rel-unsorted-term-poly-list-relD*
intro!: *RES-refine relcompI*[*of* - ‹*map* (λ(*a*, *y*). (*mset a*, *y*)) (*rev p*)›]
1 2)
qed

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 −
obtain *y* **where**
py: ‹(*p*, *y*) ∈ ⟨*term-poly-list-rel* ×$_r$ *int-rel*⟩*list-rel*› **and**
p'-y: ‹*p'* = *mset y*›
using *assms*
unfolding *fully-unsorted-poly-list-rel-def poly-list-rel-def sorted-poly-list-rel-wrt-def*
by (*auto simp*: *list-mset-rel-def br-def*)
then have [*simp*]: ‹*length y* = *length p*›

```
    by (auto simp: list-rel-def list-all2-conv-all-nth)
  have H: ‹(x, p')
      ∈ ⟨term-poly-list-rel ×r int-rel⟩list-rel O list-mset-rel›
    if px: ‹mset p = mset x› and ‹sorted-wrt (rel2p (Id ∪ lexord var-order-rel)) (map fst x)›
    for x :: ‹llist-polynomial›
  proof −
    from px have ‹length x = length p›
      by (metis size-mset)
    from px have ‹mset x = mset p›
      by simp
    then obtain f where ‹f permutes {..<length p}› ‹permute-list f p = x›
      by (rule mset-eq-permutation)
    with ‹length x = length p› have f: ‹bij-betw f {..<length x} {..<length p}›
      by (simp add: permutes-imp-bij)
    from ‹f permutes {..<length p}› ‹permute-list f p = x› [symmetric]
    have [simp]: ‹⋀i. i < length x ⟹ x ! i = p ! (f i)›
      by (simp add: permute-list-nth)
    let ?y = ‹map (λi. y ! f i) [0 ..< length x]›
    have ‹i < length y ⟹ (p ! f i, y ! f i) ∈ term-poly-list-rel ×r int-rel› for i
      using list-all2-nthD[of - p y
        ‹f i›, OF py[unfolded list-rel-def mem-Collect-eq prod.case]]
        mset-eq-length[OF px] f
      by (auto simp: list-rel-def list-all2-conv-all-nth bij-betw-def)
    then have ‹(x, ?y) ∈ ⟨term-poly-list-rel ×r int-rel⟩list-rel› and
      xy: ‹length x = length y›
      using py list-all2-nthD[of ‹rel2p (term-poly-list-rel ×r int-rel)› p y
        ‹f i› for i, simplified] mset-eq-length[OF px]
      by (auto simp: list-rel-def list-all2-conv-all-nth)
    moreover {
      have f: ‹mset-set {0..<length x} = f `# mset-set {0..<length x}›
        using f mset-eq-length[OF px]
        by (auto simp: bij-betw-def lessThan-atLeast0 image-mset-mset-set)
      have ‹mset y = {#y ! f x. x ∈# mset-set {0..<length x}#}›
        by (subst drop-0[symmetric], subst mset-drop-upto, subst xy[symmetric], subst f)
          auto
      then have ‹(?y, p') ∈ list-mset-rel›
        by (auto simp: list-mset-rel-def br-def p'-y)
    }
    ultimately show ?thesis
      by (auto intro!: relcompI[of - ?y])
  qed
  show ?thesis
    unfolding sort-poly-spec-def poly-list-rel-def sorted-repeat-poly-list-rel-with0-wrt-def
    by refine-rcg (auto intro: H)
qed


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›
unfolding *sorted-repeat-poly-list-rel-with0-wrt-def prod.case mem-Collect-eq*
list-rel-def
apply (*clarsimp*)
apply (*subst* (*asm*) *list.rel-sel*)
apply (*intro conjI*)
apply (*rule-tac b* = ‹*tl y*› **in** *relcompI*)
apply (*auto simp*: *sorted-poly-list-rel-wrt-def list-mset-rel-def br-def*)
apply (*case-tac* ‹*lead-coeff y*›; *case-tac y*)
apply (*auto simp*: *term-poly-list-rel-def*)
apply (*case-tac* ‹*lead-coeff y*›; *case-tac y*)
apply (*auto simp*: *term-poly-list-rel-def*)
apply (*case-tac* ‹*lead-coeff y*›; *case-tac y*)
apply (*auto simp*: *term-poly-list-rel-def*)
apply (*case-tac* ‹*lead-coeff y*›; *case-tac y*)
apply (*auto simp*: *term-poly-list-rel-def*)
done

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›
apply (*rule iffI*)
subgoal
by (*auto dest*!: *sorted-repeat-poly-list-rel-with0-wrt-ConsD*)
subgoal
unfolding *sorted-poly-list-rel-wrt-def prod.case mem-Collect-eq*
list-rel-def sorted-repeat-poly-list-rel-with0-wrt-def
apply (*clarsimp*)
apply (*rule-tac b* = ‹(*mset ys*, *n*) # *y*› **in** *relcompI*)
by (*auto simp*: *sorted-poly-list-rel-wrt-def list-mset-rel-def br-def*
term-poly-list-rel-def add-mset-eq-add-mset eq-commute[*of* - ‹*mset* -›]
nonzero-coeffs-def
dest!: *multi-member-split*)
done

lemma *fst-normalize0-polynomial-subsetD*:
‹(*a*, *b*) ∈ *set* (*merge-coeffs0 p*) ⟹ *a* ∈ *mononoms p*›
apply (*induction p rule*: *merge-coeffs0.induct*)
subgoal
by *auto*
subgoal
by (*auto split*: *if-splits*)
subgoal
by (*auto split*: *if-splits*)
done

lemma *in-set-merge-coeffs0D*:

‹(a, b) ∈ set (merge-coeffs0 p) ⟹∃ b. (a, b) ∈ set p›
by (*auto dest*!: *fst-normalize0-polynomial-subsetD*)

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*›
apply (*induction xs arbitrary*: *ys rule*: *merge-coeffs0.induct*)
subgoal by (*auto simp*: *sorted-repeat-poly-list-rel-wrt-def sorted-poly-list-rel-wrt-def*
sorted-repeat-poly-list-rel-with0-wrt-def list-mset-rel-def br-def)
subgoal for *xs n ys*
by (*force simp*: *sorted-repeat-poly-list-rel-wrt-def sorted-poly-list-rel-wrt-def*
sorted-repeat-poly-list-rel-with0-wrt-def list-mset-rel-def br-def
list-rel-split-right-iff)
subgoal premises *p* **for** *xs n ys m p ysa*
apply (*cases* ‹*xs* = *ys*›, *cases* ‹*m*+*n* ≠ *0*›)
subgoal
using *p*(*1*)[*of* ‹*add-mset* (*mset ys*, *m*+*n*) *ysa* − {#(*mset ys*, *m*), (*mset ys*, *n*)#}›] *p*(*5*−)
apply (*auto simp*: *sorted-repeat-poly-list-rel-with0-wrtl-Cons-iff ac-simps add-mset-commute*
remove1-mset-add-mset-If nonzero-coeffs-diff sorted-repeat-poly-list-rel-Cons-iff)
apply (*auto intro*: *normalize-poly-p.intros add-mset-commute add-mset-commute*
converse-rtranclp-into-rtranclp dest!: *multi-member-split*
simp del: *normalize-poly-p.merge-dup-coeff*)
apply (*rule-tac x* = ‹*r*› **in** *exI*)
using *normalize-poly-p.merge-dup-coeff*[*of* ‹*ysa* − {#(*mset ys*, *m*), (*mset ys*, *n*)#}› ‹*ysa* − {#(*mset ys*, *m*), (*mset ys*, *n*)#}› ‹*mset ys*› *m n*]
by (*auto intro*: *normalize-poly-p.intros*
converse-rtranclp-into-rtranclp dest!: *multi-member-split*
simp: *add-mset-commute*[*of* ‹(*mset ys*, *n*)› ‹(*mset ys*, *m*)›]
simp del: *normalize-poly-p.merge-dup-coeff*)
subgoal
using *p*(*2*)[*of* ‹*ysa* − {#(*mset ys*, *m*), (*mset ys*, *n*)#}›] *p*(*5*−)
apply (*auto simp*: *sorted-repeat-poly-list-rel-with0-wrtl-Cons-iff ac-simps add-mset-commute*
remove1-mset-add-mset-If nonzero-coeffs-diff sorted-repeat-poly-list-rel-Cons-iff)
apply (*rule-tac x* = ‹*r*› **in** *exI*)
using *normalize-poly-p.rem-0-coeff*[*of* ‹*add-mset* (*mset ys*, *m* +*n*) *ysa* − {#(*mset ys*, *m*), (*mset ys*, *n*)#}› ‹*add-mset* (*mset ys*, *m* +*n*) *ysa* − {#(*mset ys*, *m*), (*mset ys*, *n*)#}› ‹*mset ys*›]
using *normalize-poly-p.merge-dup-coeff*[*of* ‹*ysa* − {#(*mset ys*, *m*), (*mset ys*, *n*)#}› ‹*ysa* − {#(*mset ys*, *m*), (*mset ys*, *n*)#}› ‹*mset ys*› *m n*]
by (*force intro*: *normalize-poly-p.intros converse-rtranclp-into-rtranclp*
dest!: *multi-member-split*
simp del: *normalize-poly-p.rem-0-coeff*
simp: *add-mset-commute*[*of* ‹(*mset ys*, *n*)› ‹(*mset ys*, *m*)›])
apply (*cases* ‹*n* = *0*›)
subgoal
using *p*(*3*)[*of* ‹*add-mset* (*mset ys*, *m*) *ysa* − {#(*mset xs*, *n*), (*mset ys*, *m*)#}›] *p*(*4*−)
apply (*auto simp*: *sorted-repeat-poly-list-rel-with0-wrtl-Cons-iff ac-simps add-mset-commute*
remove1-mset-add-mset-If sorted-repeat-poly-list-rel-Cons-iff)
apply (*rule-tac x* = ‹*r*› **in** *exI*)
apply (*auto dest*!: *in-set-merge-coeffsD*)
by (*force intro*: *rtranclp-normalize-poly-add-mset converse-rtranclp-into-rtranclp*
simp: *rel2p-def var-order-rel-def sorted-poly-list-rel-Cons-iff*
dest!: *multi-member-split*
dest: *sorted-poly-list-rel-nonzeroD*)
subgoal

```
    using p(4)[of ‹add-mset (mset ys, m) ysa − {#(mset xs, n), (mset ys, m)#}›] p(5−)
   apply (auto simp: sorted-repeat-poly-list-rel-with0-wrtl-Cons-iff ac-simps add-mset-commute
    remove1-mset-add-mset-If sorted-repeat-poly-list-rel-Cons-iff)
   apply (rule-tac x = ‹add-mset (mset xs, n) r› in exI)
   apply (auto dest!: in-set-merge-coeffs0D)
   apply (auto intro: normalize-poly-p.intros rtranclp-normalize-poly-add-mset
    simp: rel2p-def var-order-rel-def sorted-poly-list-rel-Cons-iff
    dest!: multi-member-split
    dest: sorted-poly-list-rel-nonzeroD)
    using in-set-merge-coeffs0D total-on-lexord-less-than-char-linear apply fastforce
    using in-set-merge-coeffs0D total-on-lexord-less-than-char-linear apply fastforce
    done
   done
  done

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›
  by (induction xs rule: sorted-remdups.induct)
   auto

lemma distinct-sorted-remdups:
  ‹sorted-wrt R xs ⟹ antisymp R ⟹ distinct (sorted-remdups xs)›
  by (induction xs rule: sorted-remdups.induct)
    (auto dest: antisympD)

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 −
  have 1: ‹?B = do {
    p' ← RETURN p';
    p' ← RETURN p';
    SPEC (λr. normalize-poly-p** p' r)
   }›
    by auto
  have [refine0]: ‹sort-all-coeffs p ≤ SPEC(λp. (p, p') ∈ unsorted-poly-rel-with0)›
    by (rule sort-all-coeffs-unsorted-poly-rel-with0[OF assms, THEN order-trans])
     (auto simp: conc-fun-RES RETURN-def)
  have [refine0]: ‹sort-poly-spec p ≤ SPEC (λc. (c, p') ∈ sorted-repeat-poly-rel-with0)›
    if ‹(p, p') ∈ unsorted-poly-rel-with0›
    for p p'
    by (rule sort-poly-spec-id'[THEN order-trans, OF that])
```

```
    (auto simp: conc-fun-RES RETURN-def)
  show ?thesis
    apply (subst 1)
    unfolding full-normalize-poly-def
    by (refine-rcg)
     (auto intro!: RES-refine
       dest!: merge-coeffs0-is-normalize-poly-p
       simp: RETURN-def)
qed

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 −
  have 1: ‹SPEC (λr. normalize-poly-p** p' r) = do {
      p' ← RETURN p';
      SPEC (λr. normalize-poly-p** p' r)
   }›
   by auto
  show ?thesis
    unfolding normalize-poly-def
    apply (subst 1)
    apply (refine-rcg sort-poly-spec-id[OF assms]
      merge-coeffs-is-normalize-poly-p)
    subgoal
      by (drule merge-coeffs-is-normalize-poly-p)
        (auto intro!: RES-refine simp: RETURN-def)
    done
qed
```

8.6 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 −
  have ‹term-poly-list-rel  ×r int-rel ⊆ unsorted-term-poly-list-rel ×r int-rel›
    by (auto simp: unsorted-term-poly-list-rel-def term-poly-list-rel-def)
  from list-rel-mono[OF this]
  show ?thesis
    unfolding poly-list-rel-def fully-unsorted-poly-list-rel-def
    by (auto simp:)
qed

lemma mult-poly-full-mult-poly-p':
```

assumes ‹$(p,\ p') \in$ *sorted-poly-rel*› ‹$(q,\ q') \in$ *sorted-poly-rel*›
shows ‹*mult-poly-full p q* $\leq \Downarrow$ (*sorted-poly-rel*) (*mult-poly-p' p' q'*)›
unfolding *mult-poly-full-def mult-poly-p'-def*
apply (*refine-rcg full-normalize-poly-normalize-poly-p*
normalize-poly-normalize-poly-p)
apply (*subst RETURN-RES-refine-iff*)
apply (*subst Bex-def*)
apply (*subst mem-Collect-eq*)
apply (*subst conj-commute*)
apply (*rule mult-poly-raw-mult-poly-p*[*OF assms*(*1*,*2*)])
subgoal
by *blast*
done

definition *add-poly-spec* :: ‹-› **where**
‹*add-poly-spec p q* = *SPEC* ($\lambda r.\ p + q - r \in$ *ideal polynomial-bool*)›

definition *add-poly-p'* :: ‹-› **where**
‹*add-poly-p' p q* = *SPEC*($\lambda r.$ *add-poly-p*** (p, q, {#}) ({#}, {#}, r))›

lemma *add-poly-l-add-poly-p'*:
assumes ‹$(p,\ p') \in$ *sorted-poly-rel*› ‹$(q,\ q') \in$ *sorted-poly-rel*›
shows ‹*add-poly-l* (p, q) $\leq \Downarrow$ (*sorted-poly-rel*) (*add-poly-p' p' q'*)›
unfolding *add-poly-p'-def*
apply (*refine-rcg add-poly-l-spec*[*THEN fref-to-Down-curry-right*, *THEN order-trans*, *of* - *p' q'*])
subgoal by *auto*
subgoal using *assms* **by** *auto*
subgoal
by *auto*
done

8.7 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*:
‹$(p,\ p') \in$ *mset-poly-rel* $\Longrightarrow$
SPEC ($\lambda r.$ *normalize-poly-p*** *p r*) $\leq \Downarrow$*mset-poly-rel* (*normalize-poly-spec p'*)›
by (*auto simp*: *mset-poly-rel-def rtranclp-normalize-poly-p-poly-of-mset ideal.span-zero*
normalize-poly-spec-def intro!: *RES-refine*)

lemma *mult-poly-p'-mult-poly-spec*:
‹$(p,\ p') \in$ *mset-poly-rel* $\Longrightarrow$ $(q,\ q') \in$ *mset-poly-rel* $\Longrightarrow$
mult-poly-p' p q $\leq \Downarrow$*mset-poly-rel* (*mult-poly-spec p' q'*)›
unfolding *mult-poly-p'-def mult-poly-spec-def*
apply *refine-rcg*
apply (*auto simp*: *mset-poly-rel-def dest*!: *rtranclp-mult-poly-p-mult-ideal-final*)
apply (*intro RES-refine*)

using *ideal.span-add-eq2 ideal.span-zero*
by (*fastforce dest*!: *rtranclp-normalize-poly-p-poly-of-mset intro*: *ideal.span-add-eq2*)

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'*)›
unfolding *add-poly-p'-def add-poly-spec-def*
apply (*auto simp*: *mset-poly-rel-def dest*!: *rtranclp-add-poly-p-polynomial-of-mset-full*)
apply (*intro RES-refine*)
apply (*auto simp*: *rtranclp-add-poly-p-polynomial-of-mset-full ideal.span-zero*)
done

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* (λ*r*. *r* ⟷ *p* = *q*)›

definition *weak-equality-spec* :: ‹*mset-polynomial* ⇒ *mset-polynomial* ⇒ *bool nres*› **where**
‹*weak-equality-spec p q* = *SPEC* (λ*r*. *r* ⟷ *p* = *q*)›

lemma *term-poly-list-rel-same-rightD*:
‹(*a*, *aa*) ∈ *term-poly-list-rel* ⟹ (*a*, *ab*) ∈ *term-poly-list-rel* ⟹ *aa* = *ab*›
by (*auto simp*: *term-poly-list-rel-def*)

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*›
by (*induction xa arbitrary*: *y ya*)
(*auto simp*: *list-rel-split-right-iff*
dest: *term-poly-list-rel-same-rightD*)

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*›
by (*intro frefI nres-relI*)
(*auto simp*: *weak-equality-l-def weak-equality-spec-def*
sorted-poly-list-rel-wrt-def list-mset-rel-def br-def
dest: *list-rel-term-poly-list-rel-same-rightD*)

end
theory *PAC-Misc*
imports *Main*
begin

I believe this should be added to the simplifier by default...

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

lemma *in-set-conv-iff*:
  $x \in set\ (take\ n\ xs) \longleftrightarrow (\exists i{<}n.\ i < length\ xs \wedge xs\ !\ i = x)$
  by (*metis in-set-conv-nth length-take min-less-iff-conj nth-take*)

lemma *in-set-take-conv-nth*:
  $x \in set\ (take\ n\ xs) \longleftrightarrow (\exists i{<}min\ n\ (length\ xs).\ xs\ !\ i = x)$
  by (*simp add*: *in-set-conv-iff*)

lemma *in-set-remove1D*:
  $a \in set\ (remove1\ x\ xs) \Longrightarrow a \in set\ xs$
  by (*meson notin-set-remove1*)

end

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

9 Executable Checker

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

9.1 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*›
by (*cases a*; *cases b*; *auto simp*: *code-status-status-rel-def*; *fail*)+

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*)›
 by (*auto split*: *pac-step.splits*)

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 −
  have [intro]: ‹ab ∉ V ⟹
      vars ba ⊆ V ⟹
      MPoly-Type.coeff (ba + Var ab) (monomial (Suc 0) ab) = 1› for ab ba
    by (subst coeff-add[symmetric], subst not-in-vars-coeff0)
      (auto simp flip: coeff-add monom.abs-eq
        simp: not-in-vars-coeff0 MPoly-Type.coeff-def
          Var.abs-eq Var0-def lookup-single-eq monom.rep-eq)
  have [simp]: ‹MPoly-Type.coeff p (monomial (Suc 0) ab) = −1›
    if ‹vars (p + Var ab) ⊆ V›
      ‹ab ∉ V›
    for ab
   proof −
    define q where ‹q ≡ p + Var ab›
    then have p: ‹p = q − Var ab›
      by auto
    show ?thesis
      unfolding p
      apply (subst coeff-minus[symmetric], subst not-in-vars-coeff0)
      using that unfolding q-def[symmetric]
```

```
      by (auto simp flip: coeff-minus simp: not-in-vars-coeff0
          Var.abs-eq Var_0-def simp flip: monom.abs-eq
          simp: not-in-vars-coeff0 MPoly-Type.coeff-def
          Var.abs-eq Var_0-def lookup-single-eq monom.rep-eq)
  qed
  have [simp]: ‹vars (p − Var ab) = vars (Var ab − p)› for ab
   using vars-uminus[of ‹p − Var ab›]
   by simp
  show ?thesis
   unfolding check-extension-def
   apply (auto 5 5 simp: check-extension-def weak-equality-def
     mult-poly-spec-def field-simps
     add-poly-spec-def power2-eq-square cong: if-cong
     intro!: intro-spec-refine[where R=Id, simplified]
     split: option.splits dest: ideal.span-add)
   done
qed


lemma RES-RES-RETURN-RES: ‹RES A ⤜ (λT. RES (f T)) = RES (⋃(f ‘ A))›
  by (auto simp: pw-eq-iff refine-pw-simps)


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 −
  have 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 SPEC(λb. 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)
     else
       SPEC(λb. 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)}›
   (is ‹- = ?B›)
    by (auto simp: check-add-def RES-RES-RETURN-RES)
   have ‹?A ≤ ⇓ Id (check-add A 𝒱 p q i r)›
    apply refine-vcg
    apply ((auto simp: check-add-alt-def weak-equality-def
       add-poly-spec-def RES-RES-RETURN-RES summarize-ASSERT-conv
     cong: if-cong
     intro!: ideal.span-zero;fail)+)
```

```
    done
  then show ?thesis
    unfolding check-add-alt-def[symmetric]
    by simp
qed

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
     }
  }›
  unfolding check-mult-def
  apply (rule refine-IdD)
  by refine-vcg
   (auto simp: check-mult-def weak-equality-def
      mult-poly-spec-def RES-RES-RETURN-RES
    intro!:  ideal.span-zero
      exI[of - ‹the (fmlookup A p) * q›])

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)›
  by (induction xs)
   auto

lemma sorted-wrt-insort-key-rel:
   ‹totalp-on (insert x (set xs)) R ⟹ transp R ⟹ reflp R ⟹
    sorted-wrt R xs ⟹ sorted-wrt R (insort-key-rel R x xs)›
  by (induction xs)
   (auto dest: transpD reflpD simp: totalp-on-def)

lemma sorted-wrt-insort-key-rel2:
   ‹totalp-on (insert x (set xs)) R ⟹ transp R ⟹ x ∉ set xs ⟹
    sorted-wrt R xs ⟹ sorted-wrt R (insort-key-rel R x xs)›
  by (induction xs)
   (auto dest: transpD simp: totalp-on-def in-mono)

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›
  by (auto intro!: ext simp: relAPP-def)

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 (λp'.
    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* = (λ$\mathcal{V}$ *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*, $\mathcal{V}$, *fmempty*)
}
else do {
(*b*, $\mathcal{V}$, *A*) ← *FOREACH dom*
(λ*i* (*b*, $\mathcal{V}$, *A*′).
if i ∈# *dom-m A*
then do {
p ← *full-normalize-poly* (*the* (*fmlookup A i*));
eq ← *weak-equality-l p spec*;
$\mathcal{V}$ ← *RETURN*($\mathcal{V}$ ∪ *vars-llist* (*the* (*fmlookup A i*)));
RETURN(*b* ∨ *eq*, $\mathcal{V}$, *fmupd i p A*′)
} *else RETURN* (*b*, $\mathcal{V}$, *A*′))
(*False*, $\mathcal{V}$, *fmempty*);
RETURN (*if b then CFOUND else CSUCCESS*, $\mathcal{V}$, *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
}›

9.2 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›
  by (auto simp: mset-poly-rel-def normalize-poly-p-poly-of-mset)

lemma vars-poly-of-vars:
  ‹vars (poly-of-vars a :: int mpoly) ⊆ (φ ‘ set-mset a)›
  by (induction a)
   (auto simp: vars-mult-Var)

lemma vars-polynomial-of-mset:
  ‹vars (polynomial-of-mset za) ⊆ ⋃(image φ ‘ (set-mset o fst) ‘ set-mset za)›
  apply (induction za)
  using vars-poly-of-vars
  by (fastforce elim!: in-vars-addE simp: vars-mult-Const split: if-splits)+

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›
  by (auto simp: fully-unsorted-poly-list-rel-def mset-poly-rel-def
      set-rel-def var-rel-def br-def vars-llist-def list-rel-append2 list-rel-append1
      list-rel-split-right-iff list-mset-rel-def image-iff
      unsorted-term-poly-list-rel-def list-rel-split-left-iff
    dest!: set-rev-mp[OF - vars-polynomial-of-mset] split-list
    dest: multi-member-split
    dest: arg-cong[of ‹mset -› ‹add-mset - -› set-mset])

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))))›
  using fully-unsorted-poly-rel-vars-subset-vars-llist[of A B]
  apply (subst RETURN-RES-refine-iff)
  apply clarsimp
  apply (rule exI[of - ‹x1a ∪ φ ‘ vars-llist A›])
  apply (auto simp: set-rel-def var-rel-def br-def
    dest: fully-unsorted-poly-rel-vars-subset-vars-llist)
  done

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 −
  have 1: ‹inj-on id (dom :: nat set)› for dom
    by auto
  have H: ‹x ∈# dom-m A ⟹
     (⋀p. (the (fmlookup A x), p) ∈ fully-unsorted-poly-rel ⟹
       (p, the (fmlookup B x)) ∈ mset-poly-rel ⟹ thesis) ⟹
     thesis› for x thesis
     using fmap-rel-nat-the-fmlookup[OF AB, of x x] fmap-rel-nat-rel-dom-m[OF AB] by auto
```

```
  have full-normalize-poly: ‹full-normalize-poly (the (fmlookup A x))
      ≤ ⇓ (sorted-poly-rel O mset-poly-rel)
        (SPEC
          (λp. the (fmlookup B x') − p ∈ More-Modules.ideal polynomial-bool ∧
              vars p ⊆ vars (the (fmlookup B x'))))›
      if x-dom: ‹x ∈# dom-m A› and ‹(x, x') ∈ Id› for x x'
      apply (rule H[OF x-dom])
      subgoal for p
      apply (rule full-normalize-poly-normalize-poly-p[THEN order-trans])
      apply assumption
      subgoal
        using that(2) apply −
        unfolding conc-fun-chain[symmetric]
        by (rule ref-two-step', rule RES-refine)
         (auto simp: rtranclp-normalize-poly-p-poly-of-mset
          mset-poly-rel-def ideal.span-zero)
      done
      done

  have H': ‹(p, pa) ∈ sorted-poly-rel O mset-poly-rel ⟹
     weak-equality-l p spec ≤ SPEC (λeqa. eqa ⟶ pa = spec')› for p pa
    using spec by (auto simp: weak-equality-l-def weak-equality-spec-def
        list-mset-rel-def br-def mset-poly-rel-def
      dest: list-rel-term-poly-list-rel-same-rightD sorted-poly-list-relD)

  have emp: ‹(V, V') ∈ ⟨var-rel⟩set-rel ⟹
    ((False, V, fmempty), False, V', fmempty) ∈ bool-rel ×r ⟨var-rel⟩set-rel ×r fmap-polys-rel› for V V'
    by auto
  show ?thesis
    using assms
    unfolding remap-polys-l-def remap-polys-l-dom-err-def
      remap-polys-def prod.case
    apply (refine-rcg full-normalize-poly fmap-rel-fmupd-fmap-rel)
    subgoal
      by auto
    subgoal
      by auto
    subgoal
      by (auto simp: error-msg-def)
    apply (rule 1)
    subgoal by auto
    apply (rule emp)
    subgoal
      using V by auto
    subgoal by auto
    subgoal by auto
    subgoal by (rule H')
    apply (rule fully-unsorted-poly-rel-extend-vars)
    subgoal by (auto intro!: fmap-rel-nat-the-fmlookup)
    subgoal by (auto intro!: fmap-rel-fmupd-fmap-rel)
    subgoal by (auto intro!: fmap-rel-fmupd-fmap-rel)
    subgoal by auto
    subgoal by auto
    done
qed
```

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′*))›
unfolding *fref-def uncurry-def nres-rel-def*
by *auto*

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′*›
unfolding *weak-equality-spec-def weak-equality-def*
by (*auto simp*: *mset-poly-rel-def*)

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′*
using *that*
by (*auto intro*!: *weak-equality-l-weak-equality-spec*[*THEN fref-to-Down-curry*, *THEN order-trans*]
ref-two-step′
weak-equality-spec-weak-equality
simp flip: *conc-fun-chain*)

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*)›
by (*auto simp*: *error-msg-def*)

lemma *sorted-poly-rel-vars-llist*:
‹(*r*, *r′*) ∈ *sorted-poly-rel O mset-poly-rel* ⟹
vars r′ ⊆ *φ* ‘ *vars-llist r*›
apply (*auto simp*: *mset-poly-rel-def*
set-rel-def var-rel-def br-def vars-llist-def list-rel-append2 list-rel-append1
list-rel-split-right-iff list-mset-rel-def image-iff sorted-poly-list-rel-wrt-def
dest!: *set-rev-mp*[*OF - vars-polynomial-of-mset*]
dest!: *split-list*)
apply (*auto dest*!: *multi-member-split simp*: *list-rel-append1*
term-poly-list-rel-def eq-commute[*of - ‹mset -›*]
list-rel-split-right-iff list-rel-append2 list-rel-split-left-iff
dest: *arg-cong*[*of ‹mset -› ‹add-mset - -› set-mset*])
done

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 −
  have [refine]:
    ‹add-poly-l (p, q) ≤ ⇓ (sorted-poly-rel O mset-poly-rel) (add-poly-spec 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'
    using that
    by (auto intro!: add-poly-l-add-poly-p'[THEN order-trans] ref-two-step'
        add-poly-p'-add-poly-spec
      simp flip: conc-fun-chain)

  show ?thesis
    using assms
    unfolding check-addition-l-def
      check-not-equal-dom-err-def apply −
    apply (rule order-trans)
    defer
    apply (rule ref-two-step')
    apply (rule check-add-alt-def)
    apply refine-rcg
    subgoal
      by (drule sorted-poly-rel-vars-llist)
       (auto simp: set-rel-def var-rel-def br-def)
    subgoal
      by auto
    subgoal
      by auto
    subgoal
      by auto
    subgoal
      by auto
    subgoal
      by auto
    subgoal
      by auto
    subgoal
      by (auto simp: weak-equality-l-def bind-RES-RETURN-eq)
    done
qed

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)))›
  unfolding check-del-l-def check-del-def
  by (refine-vcg lhs-step-If RETURN-SPEC-refine)
    (auto simp: fmap-rel-nat-rel-dom-m bind-RES-RETURN-eq)

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 −
have [*refine*]:
‹*mult-poly-full p q* ≤ ⇓ (*sorted-poly-rel O mset-poly-rel*) (*mult-poly-spec 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′*
using *that*
by (*auto intro*!: *mult-poly-full-mult-poly-p′*[*THEN order-trans*] *ref-two-step′*
mult-poly-p′-mult-poly-spec
simp flip: *conc-fun-chain*)

show *?thesis*
using *assms*
unfolding *check-mult-l-def*
check-mult-l-mult-err-def check-mult-l-dom-err-def **apply** −
apply (*rule order-trans*)
defer
apply (*rule ref-two-step′*)
apply (*rule check-mult-alt-def*)
apply *refine-rcg*
subgoal
by (*drule sorted-poly-rel-vars-llist*)+
(*fastforce simp*: *set-rel-def var-rel-def br-def*)
subgoal
by *auto*
subgoal
by *auto*
subgoal
by *auto*
subgoal
by *auto*
subgoal
by (*auto simp*: *weak-equality-l-def bind-RES-RETURN-eq*)
done
qed

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 −
obtain *s* **where**
rs: ‹(*r*, *s*) ∈ *unsorted-poly-rel*› **and**
st: ‹(*s*, *t*) ∈ *mset-poly-rel*›
using *assms* **by** *auto*
show *?thesis*
by (*rule normalize-poly-normalize-poly-p*[*THEN order-trans*, *OF rs*])
(*use st* **in** ‹*auto dest*!: *rtranclp-normalize-poly-p-poly-of-mset*
intro!: *ref-two-step′ RES-refine exI*[*of - t*]
simp: *normalize-poly-spec-def ideal.span-zero mset-poly-rel-def*
simp flip: *conc-fun-chain*›)
qed

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*›
by (*induction xs ys rule*: *list-rel-induct*)
(*auto simp*: *single-valued-def IS-LEFT-UNIQUE-def*)

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*›
apply (*induction xs ys rule*: *list-rel-induct*)
apply (*solves* ‹*simp* (*no-asm*)›)
by (*smt* (*verit*) *list-rel-simp*(*4*) *remove1.simps*(*2*))

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 −
have [*simp*]: ‹([*a*], {#*a*#}) ∈ *term-poly-list-rel*›
‹⋀*aa*. ([*a*], *aa*) ∈ *term-poly-list-rel* ⟷ *aa* = {#*a*#}›
by (*auto simp*: *term-poly-list-rel-def*)
have *H*:
‹⋀*aa*. ([*a*], *aa*) ∈ *term-poly-list-rel* ⟹ *aa* = {#*a*#}›
‹⋀*aa*. (*aa*, {#*a*#}) ∈ *term-poly-list-rel* ⟹ *aa* = [*a*]›
by (*auto simp*: *single-valued-def IS-LEFT-UNIQUE-def*
term-poly-list-rel-def)

have [*simp*]: ‹*Const* (*1* :: *int*) = (*1* :: *int mpoly*)›
by (*simp add*: *Const.abs-eq Const*$_0$*-one one-mpoly.abs-eq*)
have [*simp*]: ‹*sorted-wrt term-order* (*map fst r*) ⟹
sorted-wrt term-order (*map fst* (*remove1* ([*a*], *1*) *r*))›
by (*induction r*) *auto*
have [*intro*]: ‹*distinct* (*map fst r*) ⟹ *distinct* (*map fst* (*remove1 x r*))› **for** *x*
by (*induction r*) (*auto dest*: *notin-set-remove1*)
have [*simp*]: ‹(*r*, *ya*) ∈ ⟨*term-poly-list-rel* ×$_r$ *int-rel*⟩*list-rel* ⟹
polynomial-of-mset (*mset ya*) − *Var* (*φ a*) =
polynomial-of-mset (*remove1-mset* ({#*a*#}, *1*) (*mset ya*))› **for** *ya*
using *assms*
by (*auto simp*: *list-rel-append1 list-rel-split-right-iff*
dest!: *split-list*)

show *?thesis*
using *assms*
apply (*elim relcompEpair*)
apply (*rename-tac za*, *rule-tac b* = ‹*remove1-mset* ({#*a*#}, *1*) *za*› **in** *relcompI*)
apply (*auto simp*: *mset-poly-rel-def sorted-poly-list-rel-wrt-def Collect-eq-comp'*
intro!: *relcompI*[*of* - ‹*remove1* ({#*a*#}, *1*) *ya*›

```
      for ya :: ‹(string multiset × int) list›]  remove1-list-rel2 intro: H
    simp: list-mset-rel-def br-def
    dest: in-diffD)
  done
qed

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 (φ a))
        ∈ sorted-poly-rel O mset-poly-rel›
proof −
  have [simp]: ‹([a], {#a#}) ∈ term-poly-list-rel›
    ‹⋀aa. ([a], aa) ∈ term-poly-list-rel ⟷ aa = {#a#}›
    by (auto simp: term-poly-list-rel-def)
  have H:
    ‹⋀aa. ([a], aa) ∈ term-poly-list-rel ⟹ aa = {#a#}›
    ‹⋀aa. (aa, {#a#}) ∈ term-poly-list-rel ⟹ aa = [a]›
    by (auto simp: single-valued-def IS-LEFT-UNIQUE-def
      term-poly-list-rel-def)

  have [simp]: ‹Const (1 :: int) = (1 :: int mpoly)›
    by (simp add: Const.abs-eq Const0-one one-mpoly.abs-eq)
  have [simp]: ‹sorted-wrt term-order (map fst r) ⟹
      sorted-wrt term-order (map fst (remove1 ([a], −1) r))›
    by (induction r) auto
  have [intro]: ‹distinct (map fst r) ⟹ distinct (map fst (remove1 x r))› for x
    apply (induction r) apply auto
    by (meson img-fst in-set-remove1D)
  have [simp]: ‹(r, ya) ∈ ⟨term-poly-list-rel ×r int-rel⟩list-rel ⟹
      polynomial-of-mset (mset ya) +  Var (φ a) =
      polynomial-of-mset (remove1-mset ({#a#}, −1) (mset ya))› for ya
    using assms
    by (auto simp: list-rel-append1 list-rel-split-right-iff
      dest!: split-list)

  show ?thesis
    using assms
    apply (elim relcompEpair)
    apply (rename-tac za, rule-tac b = ‹remove1-mset ({#a#}, −1) za› in relcompI)
    by (auto simp: mset-poly-rel-def sorted-poly-list-rel-wrt-def Collect-eq-comp'
      dest: in-diffD
      intro!: relcompI[of - ‹remove1 ({#a#}, −1) ya›
        for ya :: ‹(string multiset × int) list›]  remove1-list-rel2 intro: H
      simp: list-mset-rel-def br-def)
qed


lemma insert-var-rel-set-rel:
  ‹(𝒱, 𝒱') ∈ ⟨var-rel⟩set-rel ⟹
  (yb, x2) ∈ var-rel ⟹
  (insert yb 𝒱, insert x2 𝒱') ∈ ⟨var-rel⟩set-rel›
  by (auto simp: var-rel-def set-rel-def)
```

lemma *var-rel-set-rel-iff*:
  ‹$(\mathcal{V}, \mathcal{V}') \in \langle$*var-rel*$\rangle$*set-rel* $\Longrightarrow$
  $(yb, x2) \in$ *var-rel* $\Longrightarrow$
  $yb \in \mathcal{V} \longleftrightarrow x2 \in \mathcal{V}'$›
  using φ*-inj inj-eq* **by** (*fastforce simp*: *var-rel-def set-rel-def br-def*)

lemma *check-extension-l-check-extension*:
  assumes ‹$(A, B) \in$ *fmap-polys-rel*› **and** ‹$(r, r') \in$ *sorted-poly-rel O mset-poly-rel*› **and**
    ‹$(i, i') \in$ *nat-rel*› ‹$(\mathcal{V}, \mathcal{V}') \in \langle$*var-rel*$\rangle$*set-rel*› ‹$(x, x') \in$ *var-rel*›
  shows
    ‹*check-extension-l spec A* $\mathcal{V}$ *i x r* $\leq$
      $\Downarrow\{((st), (b)).$
        $(\neg$*is-cfailed st* $\longleftrightarrow$ *b*$) \wedge$
       (*is-cfound st* $\longrightarrow$ *spec* $=$ *r*)$\}$ (*check-extension B* $\mathcal{V}'$ *i' x' r'*)›
proof −
  have ‹$x' = \varphi\ x$›
    using *assms*(5) **by** (*auto simp*: *var-rel-def br-def*)
  have [*refine*]:
    ‹*mult-poly-full p q* $\leq \Downarrow$ (*sorted-poly-rel O mset-poly-rel*) (*mult-poly-spec p' q'*)›
    if ‹$(p, p') \in$ *sorted-poly-rel O mset-poly-rel*›
      ‹$(q, q') \in$ *sorted-poly-rel O mset-poly-rel*›
    for *p p' q q'*
    using *that*
    by (*auto intro*!: *mult-poly-full-mult-poly-p'*[*THEN order-trans*] *ref-two-step'*
        mult-poly-p'-mult-poly-spec
      simp flip: *conc-fun-chain*)
  have [*refine*]:
    ‹*add-poly-l* $(p, q) \leq \Downarrow$ (*sorted-poly-rel O mset-poly-rel*) (*add-poly-spec p' q'*)›
    if ‹$(p, p') \in$ *sorted-poly-rel O mset-poly-rel*›
      ‹$(q, q') \in$ *sorted-poly-rel O mset-poly-rel*›
    for *p p' q q'*
    using *that*
    by (*auto intro*!: *add-poly-l-add-poly-p'*[*THEN order-trans*] *ref-two-step'*
        add-poly-p'-add-poly-spec
      simp flip: *conc-fun-chain*)

  have [*simp*]: ‹$(l, l') \in \langle$*term-poly-list-rel* $\times_r$ *int-rel*$\rangle$*list-rel* $\Longrightarrow$
      (*map* $(\lambda(a, b).\ (a, -\ b))\ l$, *map* $(\lambda(a, b).\ (a, -\ b))\ l'$)
      $\in \langle$*term-poly-list-rel* $\times_r$ *int-rel*$\rangle$*list-rel*› **for** *l l'*
    by (*induction l l' rule*: *list-rel-induct*)
      (*auto simp*: *list-mset-rel-def br-def*)

  have [*intro*!]:
    ‹$(x2c, za) \in \langle$*term-poly-list-rel* $\times_r$ *int-rel*$\rangle$*list-rel O list-mset-rel* $\Longrightarrow$
    (*map* $(\lambda(a, b).\ (a, -\ b))$ *x2c*,
      $\{\#$*case x of* $(a, b) \Rightarrow (a, -\ b).\ x \in\#$ *za*$\#\}$)
     $\in \langle$*term-poly-list-rel* $\times_r$ *int-rel*$\rangle$*list-rel O list-mset-rel*› **for** *x2c za*
    apply (*auto*)
    subgoal for *y*
      apply (*induction x2c y rule*: *list-rel-induct*)
      apply (*auto simp*: *list-mset-rel-def br-def*)
     apply (*rename-tac a ba aa l l', rule-tac b* $=$ ‹$(aa, -\ ba)$ $\#$ *map* $(\lambda(a, b).\ (a, -\ b))\ l'$› **in** *relcompI*)
      by *auto*

```
    done
  have [simp]: ‹(λx. fst (case x of (a, b) ⇒ (a, − b))) = fst›
    by (auto intro: ext)

  have uminus: ‹(x2c, x2a) ∈ sorted-poly-rel O mset-poly-rel ⟹
      (map (λ(a, b). (a, − b)) x2c,
       − x2a)
      ∈ sorted-poly-rel O mset-poly-rel› for x2c x2a x1c x1a
    apply (clarsimp simp: sorted-poly-list-rel-wrt-def
      mset-poly-rel-def)
   apply (rule-tac b = ‹(λ(a, b). (a, − b)) `# za› in relcompI)
   by (auto simp: sorted-poly-list-rel-wrt-def
     mset-poly-rel-def comp-def polynomial-of-mset-uminus)
  have [simp]: ‹([], 0) ∈ sorted-poly-rel O mset-poly-rel›
    by (auto simp: sorted-poly-list-rel-wrt-def
     mset-poly-rel-def list-mset-rel-def br-def
     intro!: relcompI[of - ‹{#}›])
  show ?thesis
    unfolding check-extension-l-def
      check-extension-l-dom-err-def
      check-extension-l-no-new-var-err-def
      check-extension-l-new-var-multiple-err-def
      check-extension-l-side-cond-err-def
     apply (rule order-trans)
     defer
     apply (rule ref-two-step')
     apply (rule check-extension-alt-def)
     apply (refine-vcg )
     subgoal using assms(1,3,4,5)
       by (auto simp: var-rel-set-rel-iff)
     subgoal using assms(1,3,4,5)
       by (auto simp: var-rel-set-rel-iff)
     subgoal by auto
     subgoal by auto
     apply (subst ‹x' = φ x›, rule remove1-sorted-poly-rel-mset-poly-rel-minus)
     subgoal using assms by auto
     subgoal using assms by auto
     subgoal using sorted-poly-rel-vars-llist[of ‹r› ‹r'›] assms
       by (force simp: set-rel-def var-rel-def br-def
         dest!: sorted-poly-rel-vars-llist)
     subgoal by auto
     subgoal by auto
     subgoal using assms by auto
     subgoal using assms by auto
     apply (rule uminus)
     subgoal using assms by auto
     subgoal using assms by auto
     subgoal using assms by auto
     subgoal using assms by auto
     subgoal using assms by auto
     done
qed


lemma full-normalize-poly-diff-ideal:
```

fixes *dom*
assumes ‹(*p*, *p'*) ∈ *fully-unsorted-poly-rel O mset-poly-rel*›
shows
‹*full-normalize-poly p*
≤ ⇓ (*sorted-poly-rel O mset-poly-rel*)
(*normalize-poly-spec p'*)›
proof −
obtain *q* **where**
pq: ‹(*p*, *q*) ∈ *fully-unsorted-poly-rel*› **and** *qp'*:‹(*q*, *p'*) ∈ *mset-poly-rel*›
using *assms* **by** *auto*
show *?thesis*
unfolding *normalize-poly-spec-def*
apply (*rule full-normalize-poly-normalize-poly-p*[*THEN order-trans*])
apply (*rule pq*)
unfolding *conc-fun-chain*[*symmetric*]
by (*rule ref-two-step'*, *rule RES-refine*)
(*use qp'* **in** ‹*auto dest*!: *rtranclp-normalize-poly-p-poly-of-mset*
simp: *mset-poly-rel-def ideal.span-zero*›)
qed

lemma *insort-key-rel-decomp*:
‹∃ *ys zs*. *xs* = *ys* @ *zs* ∧ *insort-key-rel R x xs* = *ys* @ *x* # *zs*›
apply (*induction xs*)
subgoal by *auto*
subgoal for *a xs*
by (*force intro*: *exI*[*of* - ‹*a* # -›])
done

lemma *list-rel-append-same-length*:
‹*length xs* = *length xs'* ⟹ (*xs* @ *ys*, *xs'* @ *ys'*) ∈ ⟨*R*⟩*list-rel* ⟷ (*xs*, *xs'*) ∈ ⟨*R*⟩*list-rel* ∧ (*ys*, *ys'*) ∈ ⟨*R*⟩*list-rel*›
by (*auto simp*: *list-rel-def list-all2-append2 dest*: *list-all2-lengthD*)

lemma *term-poly-list-rel-list-relD*: ‹(*ys*, *cs*) ∈ ⟨*term-poly-list-rel* ×$_r$ *int-rel*⟩*list-rel* ⟹
cs = *map* (λ(*a*, *y*). (*mset a*, *y*)) *ys*›
by (*induction ys arbitrary*: *cs*)
(*auto simp*: *term-poly-list-rel-def list-rel-def list-all2-append list-all2-Cons1 list-all2-Cons2*)

lemma *term-poly-list-rel-single*: ‹([*x32*], {#*x32*#}) ∈ *term-poly-list-rel*›
by (*auto simp*: *term-poly-list-rel-def*)

lemma *unsorted-poly-rel-list-rel-list-rel-uminus*:
‹(*map* (λ(*a*, *b*). (*a*, − *b*)) *r*, *yc*)
∈ ⟨*unsorted-term-poly-list-rel* ×$_r$ *int-rel*⟩*list-rel* ⟹
(*r*, *map* (λ(*a*, *b*). (*a*, − *b*)) *yc*)
∈ ⟨*unsorted-term-poly-list-rel* ×$_r$ *int-rel*⟩*list-rel*›
by (*induction r arbitrary*: *yc*)
(*auto simp*: *elim*!: *list-relE3*)

lemma *mset-poly-rel-minus*: ‹({#(*a*, *b*)#}, *v'*) ∈ *mset-poly-rel* ⟹
(*mset yc*, *r'*) ∈ *mset-poly-rel* ⟹
(*r*, *yc*)
∈ ⟨*unsorted-term-poly-list-rel* ×$_r$ *int-rel*⟩*list-rel* ⟹
(*add-mset* (*a*, *b*) (*mset yc*),
v' + *r'*)

```
    ∈ mset-poly-rel›
  by (induction r arbitrary: r')
    (auto simp: mset-poly-rel-def polynomial-of-mset-uminus)

lemma fully-unsorted-poly-rel-diff:
   ‹([v], v') ∈ fully-unsorted-poly-rel O mset-poly-rel ⟹
   (r, r') ∈ fully-unsorted-poly-rel O mset-poly-rel ⟹
    (v # r,
     v' + r')
    ∈ fully-unsorted-poly-rel O mset-poly-rel›
  apply auto
  apply (rule-tac b = ‹y + ya› in relcompI)
  apply (auto simp: fully-unsorted-poly-list-rel-def list-mset-rel-def br-def)
  apply (rule-tac b = ‹yb @ yc› in relcompI)
  apply (auto elim!: list-relE3 simp: unsorted-poly-rel-list-rel-list-rel-uminus mset-poly-rel-minus)
  done

lemma PAC-checker-l-step-PAC-checker-step:
  assumes
    ‹(Ast, Bst) ∈ code-status-status-rel ×r ⟨var-rel⟩set-rel ×r fmap-polys-rel› and
    ‹(st, st') ∈ pac-step-rel› and
    spec: ‹(spec, spec') ∈ sorted-poly-rel O mset-poly-rel›
  shows
     ‹PAC-checker-l-step spec Ast st ≤ ⇓ (code-status-status-rel ×r ⟨var-rel⟩set-rel ×r fmap-polys-rel)
(PAC-checker-step spec' Bst st')›
proof −
  obtain A V cst B V' cst' where
    Ast: ‹Ast = (cst, V, A)› and
    Bst: ‹Bst = (cst', V', B)› and
    V[intro]: ‹(V, V') ∈ ⟨var-rel⟩set-rel› and
    AB: ‹(A, B) ∈ fmap-polys-rel›
     ‹(cst, cst') ∈ code-status-status-rel›
    using assms(1)
    by (cases Ast; cases Bst; auto)
  have [refine]: ‹(r, ra) ∈ sorted-poly-rel O mset-poly-rel ⟹
      (eqa, eqaa)
      ∈ {(st, b). (¬ is-cfailed st ⟷ b) ∧ (is-cfound st ⟶ spec = r)} ⟹
      RETURN eqa
      ≤ ⇓ code-status-status-rel
        (SPEC
          (λst'. (¬ is-failed st' ∧
                 is-found st' ⟶
                  ra − spec' ∈ More-Modules.ideal polynomial-bool)))›
    for r ra eqa eqaa
    using spec
    by (cases eqa)
      (auto intro!: RETURN-RES-refine dest!: sorted-poly-list-relD
        simp: mset-poly-rel-def ideal.span-zero)
  have [simp]: ‹(eqa, st'a) ∈ code-status-status-rel ⟹
      (merge-cstatus cst eqa, merge-status cst' st'a)
      ∈ code-status-status-rel› for eqa st'a
    using AB
    by (cases eqa; cases st'a)
      (auto simp: code-status-status-rel-def)
  have [simp]: ‹(merge-cstatus cst CSUCCESS, cst') ∈ code-status-status-rel›
```

using *AB*
by (*cases st*)
(*auto simp*: *code-status-status-rel-def*)
have [*simp*]: ‹(*x32*, *x32a*) ∈ *var-rel* ⟹
([([*x32*], −*1*::*int*)], −*Var x32a*) ∈ *fully-unsorted-poly-rel O mset-poly-rel*› **for** *x32 x32a*
by (*auto simp*: *mset-poly-rel-def fully-unsorted-poly-list-rel-def list-mset-rel-def br-def*
unsorted-term-poly-list-rel-def var-rel-def Const-1-eq-1
intro!: *relcompI*[*of* - ‹{#({#*x32*#}, −*1* :: *int*)#}›]
relcompI[*of* - ‹[({#*x32*#}, −*1*)]›])
have *H3*: ‹*p* − *Var a* = (−*Var a*) + *p*› **for** *p* :: ‹*int mpoly*› **and** *a*
by *auto*
show *?thesis*
using *assms*(*2*)
unfolding *PAC-checker-l-step-def PAC-checker-step-def Ast Bst prod.case*
apply (*cases st*; *cases st'*; *simp only*: *p2rel-def pac-step.case*
pac-step-rel-raw-def mem-Collect-eq prod.case pac-step-rel-raw.simps)
subgoal
apply (*refine-rcg normalize-poly-normalize-poly-spec*
check-mult-l-check-mult check-addition-l-check-add
full-normalize-poly-diff-ideal)
subgoal using *AB* **by** *auto*
subgoal using *AB* **by** *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*auto intro*: *V*)
apply *assumption*+
subgoal
by (*auto simp*: *code-status-status-rel-def*)
subgoal
by (*auto intro*!: *fmap-rel-fmupd-fmap-rel*
fmap-rel-fmdrop-fmap-rel AB)
subgoal using *AB* **by** *auto*
done
subgoal
apply (*refine-rcg normalize-poly-normalize-poly-spec*
check-mult-l-check-mult check-addition-l-check-add
full-normalize-poly-diff-ideal[*unfolded normalize-poly-spec-def*[*symmetric*]])
subgoal using *AB* **by** *auto*
subgoal using *AB* **by** *auto*
subgoal using *AB* **by** *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
apply *assumption*+
subgoal
by (*auto simp*: *code-status-status-rel-def*)
subgoal
by (*auto intro*!: *fmap-rel-fmupd-fmap-rel*
fmap-rel-fmdrop-fmap-rel AB)
subgoal using *AB* **by** *auto*
done
subgoal
apply (*refine-rcg full-normalize-poly-diff-ideal*
check-extension-l-check-extension)

subgoal using *AB* **by** (*auto intro*!: *fully-unsorted-poly-rel-diff*[*of* - ‹−*Var* - :: *int mpoly*›, *unfolded H3*[*symmetric*]] *simp*: *comp-def case-prod-beta*)
subgoal using *AB* **by** *auto*
subgoal using *AB* **by** *auto*
subgoal by *auto*
subgoal by *auto*
subgoal
by (*auto simp*: *code-status-status-rel-def*)
subgoal
by (*auto simp*: *AB*
intro!: *fmap-rel-fmupd-fmap-rel insert-var-rel-set-rel*)
subgoal
by (*auto simp*: *code-status-status-rel-def AB*
code-status.is-cfailed-def)
done
subgoal
apply (*refine-rcg normalize-poly-normalize-poly-spec*
check-del-l-check-del check-addition-l-check-add
full-normalize-poly-diff-ideal[*unfolded normalize-poly-spec-def*[*symmetric*]])
subgoal using *AB* **by** *auto*
subgoal using *AB* **by** *auto*
subgoal
by (*auto intro*!: *fmap-rel-fmupd-fmap-rel*
fmap-rel-fmdrop-fmap-rel code-status-status-rel-def AB)
subgoal
by (*auto intro*!: *fmap-rel-fmupd-fmap-rel*
fmap-rel-fmdrop-fmap-rel AB)
done
done
qed

lemma *code-status-status-rel-discrim-iff*:
‹(*x1a*, *x1c*) ∈ *code-status-status-rel* ⟹ *is-cfailed x1a* ⟷ *is-failed x1c*›
‹(*x1a*, *x1c*) ∈ *code-status-status-rel* ⟹ *is-cfound x1a* ⟷ *is-found x1c*›
by (*cases x1a*; *cases x1c*; *auto*; *fail*)+

lemma *PAC-checker-l-PAC-checker*:
assumes
‹(*A*, *B*) ∈ ⟨*var-rel*⟩*set-rel* ×$_r$ *fmap-polys-rel*› **and**
‹(*st*, *st′*) ∈ ⟨*pac-step-rel*⟩*list-rel*› **and**
‹(*spec*, *spec′*) ∈ *sorted-poly-rel O mset-poly-rel*› **and**
‹(*b*, *b′*) ∈ *code-status-status-rel*›
shows
‹*PAC-checker-l spec A b st* ≤ ⇓ (*code-status-status-rel* ×$_r$ ⟨*var-rel*⟩*set-rel* ×$_r$ *fmap-polys-rel*) (*PAC-checker spec′ B b′ st′*)›
proof −
have [*refine0*]: ‹(((*b*, *A*), *st*), (*b′*, *B*), *st′*) ∈ ((*code-status-status-rel* ×$_r$ ⟨*var-rel*⟩*set-rel* ×$_r$ *fmap-polys-rel*) ×$_r$ ⟨*pac-step-rel*⟩*list-rel*)›
using *assms* **by** (*auto simp*: *code-status-status-rel-def*)
show *?thesis*
using *assms*
unfolding *PAC-checker-l-def PAC-checker-def*
apply (*refine-rcg PAC-checker-l-step-PAC-checker-step*
WHILEIT-refine[**where** *R* = ‹((*bool-rel* ×$_r$ ⟨*var-rel*⟩*set-rel* ×$_r$ *fmap-polys-rel*) ×$_r$ ⟨*pac-step-rel*⟩*list-rel*)›])
subgoal by (*auto simp*: *code-status-status-rel-discrim-iff*)

subgoal by *auto*
subgoal by (*auto simp*: *neq-Nil-conv*)
subgoal by (*auto simp*: *neq-Nil-conv intro*!: *param-nth*)
subgoal by (*auto simp*: *neq-Nil-conv*)
subgoal by *auto*
done
qed

end

lemma *in-var-order-rel-iff* [*code-unfold*, *no-atp*]:
‹(*v1*, *v2*) ∈ *var-order-rel* ⟷ *List.lexordp* (λ*c1 c2*. (*of-char c1* :: *nat*) < *of-char c2*) *v1 v2*›
by (*simp add*: *lexordp-def var-order-rel-def less-char-def* [*abs-def*] *p2rel-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 −
have [*refine*]:
‹(*spec*, *spec′*) ∈ *sorted-poly-rel O mset-poly-rel* ⟹
(*V*, *V′*) ∈ ⟨*var-rel*⟩*set-rel* ⟹
remap-polys-l spec V A ≤ ⇓(*code-status-status-rel* ×$_r$ ⟨*var-rel*⟩*set-rel* ×$_r$ *fmap-polys-rel*)
(*remap-polys-change-all spec′ V′ B*)› **for** *spec spec′ V V′*

```
    apply (rule remap-polys-l-remap-polys[THEN order-trans, OF assms(1)])
    apply assumption+
    apply (rule ref-two-step[OF order.refl])
    apply(rule remap-polys-spec[THEN order-trans])
    by (rule remap-polys-polynomial-bool-remap-polys-change-all)

  show ?thesis
    unfolding full-checker-l-def full-checker-def
    apply (refine-rcg remap-polys-l-remap-polys
       full-normalize-poly-diff-ideal[unfolded normalize-poly-spec-def[symmetric]]
       PAC-checker-l-PAC-checker)
    subgoal
      using assms(3) .
    subgoal by auto
    subgoal by (auto simp: is-cfailed-def is-failed-def)
    subgoal by auto
    apply (rule fully-unsorted-poly-rel-extend-vars)
    subgoal using assms(3) .
    subgoal by auto
    subgoal by auto
    subgoal
      using assms(2) by (auto simp: p2rel-def)
    subgoal by auto
    done
qed


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›
  apply (intro frefI nres-relI)
  using full-checker-l-full-checker by force

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 ≥ 2^64);
   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)
      (λi (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 −
  have [refine]: ‹(A, A') ∈ Id ⟹ upper-bound-on-dom A
   ≤ ⇓ {(n, dom). dom = set [0..<n]} (SPEC (λdom. set-mset (dom-m A') ⊆ dom ∧ finite dom))› for
A A'
    unfolding upper-bound-on-dom-def
    apply (rule RES-refine)
    apply (auto simp: upper-bound-on-dom-def)
    done
  have 1: ‹inj-on id dom› for dom
    by auto
  have 2: ‹x ∈# dom-m A ⟹
      x' ∈# dom-m A' ⟹
      (x, x') ∈ nat-rel ⟹
      (A, A') ∈ Id ⟹
      full-normalize-poly (the (fmlookup A x))
      ≤ ⇓ Id
        (full-normalize-poly (the (fmlookup A' x')))›
      for A A' x x'
      by (auto)
  have 3: ‹(n, dom) ∈ {(n, dom). dom = set [0..<n]} ⟹
      ([0..<n], dom) ∈ ⟨nat-rel⟩list-set-rel› for n dom
  by (auto simp: list-set-rel-def br-def)
  have 4: ‹(p,q) ∈ Id ⟹
    weak-equality-l p spec ≤ ⇓Id (weak-equality-l q spec)› for p q spec
    by auto

  have 6: ‹a = b ⟹ (a, b) ∈ Id› for a b
    by auto
  show ?thesis
    unfolding remap-polys-l2-def remap-polys-l-def
    apply (refine-rcg LFO-refine[where R= ‹Id ×r ⟨Id⟩set-rel ×r Id›])
    subgoal by auto
    subgoal by auto
    subgoal by auto
    apply (rule 3)
    subgoal by auto
    subgoal by (simp add: in-dom-m-lookup-iff)
    subgoal by (simp add: in-dom-m-lookup-iff)
    apply (rule 2)
    subgoal by auto
    subgoal by auto
    subgoal by auto
    subgoal by auto
    apply (rule 4; assumption)
    apply (rule 6)
```

subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
done
qed

end

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

10 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
by *standard* (*simp add*: *def-hashmap-size-uint32-def*)
end

instantiation *uint64* :: *hashable*
begin

context
includes *bit-operations-syntax*
begin

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

end

definition *def-hashmap-size-uint64* :: ‹*uint64 itself* ⇒ *nat*› **where**
‹*def-hashmap-size-uint64* = (λ-. *16*)›
— same as *nat*
instance
by *standard* (*simp add*: *def-hashmap-size-uint64-def*)
end

lemma *word-nat-of-uint64-Rep-inject*[*simp*]: ‹*nat-of-uint64 ai = nat-of-uint64 bi* ⟷ *ai = bi*›
by *transfer* (*simp add*: *word-unat-eq-iff*)

instance *uint64* :: *heap*
by *standard* (*auto simp*: *inj-def exI*[*of - nat-of-uint64*])

instance *uint64* :: *semiring-numeral*
by *standard*

lemma *nat-of-uint64-012*[*simp*]: ‹*nat-of-uint64 0 = 0*› ‹*nat-of-uint64 2 = 2*› ‹*nat-of-uint64 1 = 1*›
by (*simp-all add*: *nat-of-uint64.rep-eq zero-uint64.rep-eq one-uint64.rep-eq*)

definition *uint64-of-nat-conv* **where**
[*simp*]: ‹*uint64-of-nat-conv* (*x* :: *nat*) = *x*›

lemma *less-upper-bintrunc-id*: ‹*n* < *2* ^*b* ⟹ *n* ≥ *0* ⟹ *take-bit b n* = *n*› **for** *n* :: *int*
by (*rule take-bit-int-eq-self*)

lemma *nat-of-uint64-uint64-of-nat-id*: ‹*n* < *2*^*64* ⟹ *nat-of-uint64* (*uint64-of-nat n*) = *n*›
by *transfer* (*simp add*: *take-bit-nat-eq-self unsigned-of-nat*)

lemma [*sepref-fr-rules*]:
‹(*return o uint64-of-nat*, *RETURN o uint64-of-nat-conv*) ∈ [λ*a*. *a* < *2* ^*64*]$_a$ *nat-assn*^{k} → *uint64-nat-assn*›
by *sepref-to-hoare*
(*sep-auto simp*: *uint64-nat-rel-def br-def nat-of-uint64-uint64-of-nat-id*)

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*›
by (*auto intro*!: *frefI simp*: *string-rel-def String.less-literal-def*
less-than-char-def rel2p-def literal.explode-inject)

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*›
by (*simp add*: *list-assn-pure-conv*)

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*›
by (*auto simp*: *pure-app-eq*)

lemma *single-valued-string-rel*:
‹*single-valued string-rel*›
by (*auto simp*: *single-valued-def string-rel-def*)

lemma *IS-LEFT-UNIQUE-string-rel*:
‹*IS-LEFT-UNIQUE string-rel*›
by (*auto simp*: *IS-LEFT-UNIQUE-def single-valued-def string-rel-def*
literal.explode-inject)

lemma *IS-RIGHT-UNIQUE-string-rel*:
‹*IS-RIGHT-UNIQUE string-rel*›
by (*auto simp*: *single-valued-def string-rel-def*
literal.explode-inject)

lemma *single-valued-monom-rel*: ‹*single-valued monom-rel*›
by (*rule list-rel-sv*)
(*auto intro*!: *frefI simp*: *string-rel-def*
rel2p-def single-valued-def p2rel-def)

lemma *single-valued-monomial-rel*:
‹*single-valued monomial-rel*›
using *single-valued-monom-rel*
by (*auto intro*!: *frefI simp*:
rel2p-def single-valued-def p2rel-def)

lemma *single-valued-monom-rel′*: ‹*IS-LEFT-UNIQUE monom-rel*›
unfolding *IS-LEFT-UNIQUE-def inv-list-rel-eq string2-rel-def*
by (*rule list-rel-sv*)+
(*auto intro*!: *frefI simp*: *string-rel-def*
rel2p-def single-valued-def p2rel-def literal.explode-inject)

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

using *single-valued-monom-rel'*
unfolding *IS-LEFT-UNIQUE-def inv-list-rel-eq*
by (*auto intro*!: *frefI simp*:
rel2p-def single-valued-def p2rel-def)

lemma [*safe-constraint-rules*]:
‹*Sepref-Constraints.CONSTRAINT single-valued string-rel*›
‹*Sepref-Constraints.CONSTRAINT IS-LEFT-UNIQUE string-rel*›
by (*auto simp*: *CONSTRAINT-def single-valued-def*
string-rel-def IS-LEFT-UNIQUE-def literal.explode-inject)

lemma *eq-string-monom-hnr*[*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (=)), *uncurry* (*RETURN oo* (=))) ∈ *monom-assn*^{k} $*_a$ *monom-assn*^{k} $\to_a$ *bool-assn*›
using *single-valued-monom-rel' single-valued-monom-rel*
unfolding *list-assn-pure-conv*
by *sepref-to-hoare*
(*sep-auto simp*: *list-assn-pure-conv string-rel-string-assn*
single-valued-def IS-LEFT-UNIQUE-def
dest!: *mod-starD*
simp flip: *inv-list-rel-eq*)

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*›
by *auto*

lemma *term-order-rel-alt-def*:
‹*term-order-rel* = *lexord* (*p2rel char.lexordp*)›
by (*auto simp*: *p2rel-def char.lexordp-conv-lexord var-order-rel-def intro*!: *arg-cong*[*of - - lexord*])

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
apply *standard*
using *char.linorder-axioms*
by (*auto simp*: *class.linorder-def class.order-def class.preorder-def*
less-eq-char-def less-than-char-def class.order-axioms-def
class.linorder-axioms-def p2rel-def less-char-def)
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 *standard*
have [*dest*]: ‹⋀*x y* :: *'a* :: *linorder list*. (*x*, *y*) ∈ *lexord* {(*x*, *y*). *x* < *y*} ⟹

```
        lexordp-eq y x ⟹ False›
    by (metis lexordp-antisym lexordp-conv-lexord lexordp-eq-conv-lexord)
  have [simp]: ‹⋀x y :: 'a :: linorder list. lexordp-eq x y ⟹
        ¬ lexordp-eq y x ⟹
        (x, y) ∈ lexord {(x, y). x < y}›
    using lexordp-conv-lexord lexordp-conv-lexordp-eq by blast
  show
   ‹(x < y) = Restricted-Predicates.strict (≤) x y›
   ‹x ≤ x›
   ‹x ≤ y ⟹ y ≤ z ⟹ x ≤ z›
   ‹x ≤ y ⟹ y ≤ x ⟹ x = y›
   ‹x ≤ y ∨ y ≤ x›
   for x y z :: ‹'a :: linorder list›
    by (auto simp: less-list-def less-eq-list-def List.lexordp-def
    lexordp-conv-lexord lexordp-into-lexordp-eq lexordp-antisym
    antisym-def lexordp-eq-refl lexordp-eq-linear intro: lexordp-eq-trans
    dest: lexordp-eq-antisym)
qed

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 −
  show
    ‹term-order-rel' x y = ord-class.lexordp x y›
    ‹term-order-rel' x y ⟷ x < y›
    by (auto simp: lexordp-conv-lexord less-eq-list-def
        less-list-def lexordp-def var-order-rel-def
        rel2p-def term-order-rel-alt-def p2rel-def)
qed

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
  have H: ‹(a, b) ∈ ⟨string-rel⟩list-rel ⟹
      (a, cs) ∈ ⟨string-rel⟩list-rel ⟹ b = cs› for cs
    using single-valued-monom-rel' IS-RIGHT-UNIQUE-string-rel
    unfolding string2-rel-def
    by (subst (asm)list-rel-sv-iff[symmetric])
      (auto simp: single-valued-def)
  assume ‹a < a'›
  then consider
    u u' where ‹a' = a @ u # u'› |
    u aa v w aaa where ‹a = u @ aa # v› ‹a' = u @ aaa # w› ‹aa < aaa›
    by (subst (asm) less-list-def)
     (auto simp: lexord-def List.lexordp-def
      list-rel-append1 list-rel-split-right-iff)
  then show ‹b < b'›
  proof cases
    case 1
```

```
    then show ‹b < b'›
      using assms
      by (subst less-list-def)
        (auto simp: lexord-def List.lexordp-def
        list-rel-append1 list-rel-split-right-iff dest: H)
  next
    case 2
    then obtain u' aa' v' w' aaa' where
      ‹b = u' @ aa' # v'› ‹b' = u' @ aaa' # w'›
      ‹(aa, aa') ∈ string-rel›
      ‹(aaa, aaa') ∈ string-rel›
      using assms
      by (smt (verit) list-rel-append1 list-rel-split-right-iff single-valued-def single-valued-monom-rel)
    with ‹aa < aaa› have ‹aa' < aaa'›
      by (auto simp: string-rel-def less-literal.rep-eq less-list-def
        lexordp-conv-lexord lexordp-def char.lexordp-conv-lexord
          simp flip: less-char-def PAC-Polynomials-Term.less-char-def)
    then show ‹b < b'›
      using ‹b = u' @ aa' # v'› ‹b' = u' @ aaa' # w'›
      by (subst less-list-def)
        (fastforce simp: lexord-def List.lexordp-def
        list-rel-append1 list-rel-split-right-iff)
  qed
next
  have H: ‹(a, b) ∈ ⟨string-rel⟩list-rel ⟹
      (a', b) ∈ ⟨string-rel⟩list-rel ⟹ a = a'› for a a' b
    using single-valued-monom-rel'
    by (auto simp: single-valued-def IS-LEFT-UNIQUE-def
      simp flip: inv-list-rel-eq)
  assume ‹b < b'›
  then consider
    u u' where ‹b' = b @ u # u'› |
    u aa v w aaa where ‹b = u @ aa # v› ‹b' = u @ aaa # w› ‹aa < aaa›
    by (subst (asm) less-list-def)
     (auto simp: lexord-def List.lexordp-def
      list-rel-append1 list-rel-split-right-iff)
  then show ‹a < a'›
  proof cases
    case 1
    then show ‹a < a'›
      using assms
      by (subst less-list-def)
        (auto simp: lexord-def List.lexordp-def
        list-rel-append2 list-rel-split-left-iff dest: H)
  next
    case 2
    then obtain u' aa' v' w' aaa' where
      ‹a = u' @ aa' # v'› ‹a' = u' @ aaa' # w'›
      ‹(aa', aa) ∈ string-rel›
      ‹(aaa', aaa) ∈ string-rel›
      using assms
      by (auto simp: lexord-def List.lexordp-def
        list-rel-append2 list-rel-split-left-iff dest: H)
    with ‹aa < aaa› have ‹aa' < aaa'›
      by (auto simp: string-rel-def less-literal.rep-eq less-list-def
```

```
      lexordp-conv-lexord lexordp-def char.lexordp-conv-lexord
        simp flip: less-char-def PAC-Polynomials-Term.less-char-def)
    then show ‹a < a'›
      using ‹a = u' @ aa' # v'› ‹a' = u' @ aaa' # w'›
      by (subst less-list-def)
        (fastforce simp: lexord-def List.lexordp-def
        list-rel-append1 list-rel-split-right-iff)
  qed
qed


lemma string-rel-le[sepref-import-param]:
  shows ‹((<), (<)) ∈ ⟨string-rel⟩list-rel → ⟨string-rel⟩list-rel → bool-rel›
  by (auto intro!: fun-relI simp: list-rel-list-rel-order-iff)


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›
  apply (intro fun-relI)
  subgoal premises p for x y xs ys
    using p(2) p(1) assms
    by (induction xs ys rule: list-rel-induct)
      (auto simp: IS-LEFT-UNIQUE-def single-valued-def)
  done

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

instance
proof standard
  obtain f :: ‹'a ⇒ nat› where
    f: ‹inj f›
    by blast
  obtain g :: ‹nat × nat × nat × nat × nat ⇒ nat› where
    g: ‹inj g›
    by blast
  obtain h :: ‹'b ⇒ nat› where
    h: ‹inj h›
    by blast
  obtain i :: ‹'c ⇒ nat› where
    i: ‹inj i›
    by blast
  have [iff]: ‹g a = g b ⟷ a = b›‹h a'' = h b'' ⟷ a'' = b''› ‹f a' = f b' ⟷ a' = b'›
    ‹i a''' = i b''' ⟷ a''' = b'''› for a b a' b' a'' b'' a''' b'''
    using f g h i unfolding inj-def by blast+
  let ?f = ‹λx :: ('a, 'b, 'c) pac-step.
      g (case x of
         Add a b c d ⇒      (0, i a,  i b,  i c, f d)
       | Del a  ⇒           (1, i a,    0,   0,   0)
       | Mult a b c d ⇒     (2, i a, f b, i c, f d)
       | Extension a b c ⇒ (3, i a, f c, 0, h b))›
   have ‹inj ?f›
     apply (auto simp: inj-def)
     apply (case-tac x; case-tac y)
```

apply *auto*
done
then show ‹∃*f* :: (*'a*, *'b*, *'c*) *pac-step* ⇒ *nat*. *inj f*›
by *blast*
qed

end

end
theory *PAC-Assoc-Map-Rel*
imports *PAC-Map-Rel*
begin

11 Hash Map as association list

type-synonym (*'k*, *'v*) *hash-assoc* = ‹(*'k* × *'v*) *list*›

definition *hassoc-map-rel-raw* :: ‹((*'k*, *'v*) *hash-assoc* × -) *set*› **where**
‹*hassoc-map-rel-raw* = *br map-of* (λ-. *True*)›

abbreviation *hassoc-map-assn* :: ‹(*'k* ⇒ *'v option*) ⇒ (*'k*, *'v*) *hash-assoc* ⇒ *assn*› **where**
‹*hassoc-map-assn* ≡ *pure* (*hassoc-map-rel-raw*)›

lemma *hassoc-map-rel-raw-empty*[*simp*]:
‹([], *m*) ∈ *hassoc-map-rel-raw* ⟷ *m* = *Map.empty*›
‹(*p*, *Map.empty*) ∈ *hassoc-map-rel-raw* ⟷ *p* = []›
‹*hassoc-map-assn Map.empty* [] = *emp*›
by (*auto simp*: *hassoc-map-rel-raw-def br-def pure-def*)

definition *hassoc-new* :: ‹(*'k*, *'v*) *hash-assoc Heap*›**where**
‹*hassoc-new* = *return* []›

lemma *precise-hassoc-map-assn*: ‹*precise hassoc-map-assn*›
by (*auto intro*!: *precise-pure*)
(*auto simp*: *single-valued-def hassoc-map-rel-raw-def*
br-def)

definition *hassoc-isEmpty* :: (*'k* × *'v*) *list* ⇒ *bool Heap* **where**
hassoc-isEmpty ht ≡ *return* (*length ht* = *0*)

interpretation *hassoc*: *bind-map-empty hassoc-map-assn hassoc-new*
by *unfold-locales*
(*auto intro*: *precise-hassoc-map-assn*
simp: *ent-refl-true hassoc-new-def*
intro!: *return-cons-rule*)

interpretation *hassoc*: *bind-map-is-empty hassoc-map-assn hassoc-isEmpty*
by *unfold-locales*
(*auto simp*: *precise-hassoc-map-assn hassoc-isEmpty-def ent-refl-true*
intro!: *precise-pure return-cons-rule*)

definition *op-assoc-empty* ≡ *IICF-Map.op-map-empty*

interpretation *hassoc*: *map-custom-empty op-assoc-empty*
by *unfold-locales* (*simp add*: *op-assoc-empty-def*)

lemmas [*sepref-fr-rules*] = *hassoc.empty-hnr*[*folded op-assoc-empty-def*]

definition *hassoc-update* :: *'k* ⇒ *'v* ⇒ (*'k*, *'v*) *hash-assoc* ⇒ (*'k*, *'v*) *hash-assoc Heap* **where**
hassoc-update k v ht = *return* ((*k*, *v*) # *ht*)

lemma *hassoc-map-assn-Cons*:
‹*hassoc-map-assn* (*m*) (*p*) ⟹$_A$ *hassoc-map-assn* (*m*(*k* ↦ *v*)) ((*k*, *v*) # *p*) ∗ *true*›
by (*auto simp*: *hassoc-map-rel-raw-def pure-def br-def*)

interpretation *hassoc*: *bind-map-update hassoc-map-assn hassoc-update*
by *unfold-locales*
(*auto intro*!: *return-cons-rule*
simp: *hassoc-update-def hassoc-map-assn-Cons*)

definition *hassoc-delete* :: ‹*'k* ⇒ (*'k*, *'v*) *hash-assoc* ⇒ (*'k*, *'v*) *hash-assoc Heap*› **where**
‹*hassoc-delete k ht* = *return* (*filter* (λ(*a*, *b*). *a* ≠ *k*) *ht*)›

lemma *hassoc-map-of-filter-all*:
‹*map-of p* |‘ (− {*k*}) = *map-of* (*filter* (λ(*a*, *b*). *a* ≠ *k*) *p*)›
apply (*induction p*)
apply (*auto simp*: *restrict-map-def fun-eq-iff split*: *if-split*)
apply *presburger*+
done

lemma *hassoc-map-assn-hassoc-delete*: ‹<*hassoc-map-assn m p*> *hassoc-delete k p* <*hassoc-map-assn* (*m* |‘ (− {*k*}))>$_t$›
by (*auto simp*: *hassoc-delete-def hassoc-map-rel-raw-def pure-def br-def*
hassoc-map-of-filter-all
intro!: *return-cons-rule*)

interpretation *hassoc*: *bind-map-delete hassoc-map-assn hassoc-delete*
by *unfold-locales*
(*auto intro*: *hassoc-map-assn-hassoc-delete*)

definition *hassoc-lookup* :: ‹*'k* ⇒ (*'k*, *'v*) *hash-assoc* ⇒ *'v option Heap*› **where**
‹*hassoc-lookup k ht* = *return* (*map-of ht k*)›

lemma *hassoc-map-assn-hassoc-lookup*:
‹<*hassoc-map-assn m p*> *hassoc-lookup k p* <λ*r*. *hassoc-map-assn m p* ∗ ↑ (*r* = *m k*)>$_t$›
by (*auto simp*: *hassoc-lookup-def hassoc-map-rel-raw-def pure-def br-def*
hassoc-map-of-filter-all
intro!: *return-cons-rule*)

interpretation *hassoc*: *bind-map-lookup hassoc-map-assn hassoc-lookup*
by *unfold-locales*
(*rule hassoc-map-assn-hassoc-lookup*)

setup *Locale-Code.open-block*
interpretation *hassoc*: *gen-contains-key-by-lookup hassoc-map-assn hassoc-lookup*

by *unfold-locales*
setup *Locale-Code.close-block*

interpretation *hassoc*: *bind-map-contains-key hassoc-map-assn hassoc.contains-key*
by *unfold-locales*

11.1 Conversion from assoc to other map

definition *hash-of-assoc-map* **where**
‹*hash-of-assoc-map xs* = *fold* (λ(*k*, *v*) *m*. *if m k* ≠ *None then m else m*(*k* ↦ *v*)) *xs Map.empty*›

lemma *map-upd-map-add-left*:
‹*m*(*a* ↦ *b*) ++ *m*′ = *m* ++ (*if a* ∉ *dom m*′ *then m*′(*a* ↦ *b*) *else m*′)›
proof −
have ‹*m*′ *a* = *Some y* ⟹ *m*(*a* ↦ *b*) ++ *m*′ = *m* ++ *m*′› **for** *y*
by (*metis* (*no-types*) *fun-upd-triv fun-upd-upd map-add-assoc map-add-empty map-add-upd map-le-iff-map-add-commute*)
then show *?thesis*
by *auto*
qed

lemma *fold-map-of-alt*:
‹*fold* (λ(*k*, *v*) *m*. *if m k* ≠ *None then m else m*(*k* ↦ *v*)) *xs m*′ = *map-of xs* ++ *m*′›
by (*induction xs arbitrary*: *m*′)
(*auto simp*: *map-upd-map-add-left*)

lemma *map-of-alt-def*:
‹*map-of xs* = *hash-of-assoc-map xs*›
using *fold-map-of-alt*[*of xs Map.empty*]
unfolding *hash-of-assoc-map-def*
by *auto*

definition *hashmap-conv* **where**
[*simp*]: ‹*hashmap-conv x* = *x*›

lemma *hash-of-assoc-map-id*:
‹(*hash-of-assoc-map*, *hashmap-conv*) ∈ *hassoc-map-rel-raw* → *Id*›
by (*auto intro*!: *fun-relI simp*: *hassoc-map-rel-raw-def br-def map-of-alt-def*)

definition *hassoc-map-rel* **where**
hassoc-map-rel-internal-def:
‹*hassoc-map-rel K V* = *hassoc-map-rel-raw O* ⟨*K*, *V*⟩*map-rel*›

lemma *hassoc-map-rel-def*:
‹⟨*K*, *V*⟩ *hassoc-map-rel* = *hassoc-map-rel-raw O* ⟨*K*, *V*⟩*map-rel*›
unfolding *relAPP-def hassoc-map-rel-internal-def*
by *auto*

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 Nipkow. 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
  by (induct f xs ys rule: merge.induct) (simp-all add: ac-simps)

lemma set-merge [simp]:
  set (merge f xs ys) = set xs ∪ set ys
  by (induct f xs ys rule: merge.induct) auto

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
  apply (induct f xs ys rule: merge.induct)
  apply (auto simp add: ball-Un not-le less-le dest: transpD)
  apply blast
  apply (blast dest: transpD)
  done

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))

lemma msort2-code [code]:
  ‹msort2 f [] = []›
  ‹msort2 f [x] = [x]›
  ‹msort2 f [x, y] = (if f x y then [x, y] else [y, x])›
  ‹msort2 f (v # vb # va # vd) =
    PAC-Checker-Init.merge f (msort f (v # take (Suc (length vd) div 2) (vb # va # vd)))
     (msort f (drop (Suc (length vd) div 2) (vb # va # vd)))›
  by simp-all

lemma msort-msort2:
  fixes xs :: ‹'a :: linorder list›
  shows ‹msort (≤) xs = msort2 (≤) xs›
  apply (induction ‹(≤) :: 'a ⇒ 'a ⇒ bool› xs rule: msort2.induct)
  apply (auto dest: transpD)
  done

lemma sorted-msort:
  transp f ⟹ (⋀x y. f x y ∨ f y x) ⟹
   sorted-wrt f (msort f xs)
  by (induct f xs rule: msort.induct) (simp-all add: sorted-merge)

lemma mset-msort[simp]:
  mset (msort f xs) = mset xs
  by (induction f xs rule: msort.induct)
    (simp-all add: union-code)
```

12.2 Sorting applied to monomials

```
lemma merge-coeffs-alt-def:
  ‹(RETURN o merge-coeffs) p =
   REC_T(λf p.
     (case p of
       [] ⇒ RETURN []
     | [-] => 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 do {p ← f ((ys, m) # p); RETURN ((xs, n) # p)})))
    p›
  apply (induction p rule: merge-coeffs.induct)
  subgoal by (subst RECT-unfold, refine-mono) auto
  subgoal by (subst RECT-unfold, refine-mono) auto
  subgoal for x p y q
    by (subst RECT-unfold, refine-mono)
     (smt case-prod-conv list.simps(5) merge-coeffs.simps(3) nres-monad1
      push-in-let-conv(2))
  done

lemma hn-invalid-recover:
  ‹is-pure R ⟹ hn-invalid R = (λx y. R x y * true)›
  ‹is-pure R ⟹ invalid-assn R = (λx y. R x y * true)›
```

by (*auto simp*: *is-pure-conv invalid-pure-recover hn-ctxt-def intro*!: *ext*)

lemma *safe-poly-vars*:
shows
[*safe-constraint-rules*]:
is-pure (*poly-assn*) **and**
[*safe-constraint-rules*]:
is-pure (*monom-assn*) **and**
[*safe-constraint-rules*]:
is-pure (*monomial-assn*) **and**
[*safe-constraint-rules*]:
is-pure string-assn
by (*auto intro*!: *pure-prod list-assn-pure simp*: *prod-assn-pure-conv*)

lemma *invalid-assn-distrib*:
‹*invalid-assn monom-assn* $\times_a$ *invalid-assn int-assn* = *invalid-assn* (*monom-assn* $\times_a$ *int-assn*)›
apply (*simp add*: *invalid-pure-recover hn-invalid-recover*
safe-constraint-rules)
apply (*subst hn-invalid-recover*)
apply (*rule safe-poly-vars*(*2*))
apply (*subst hn-invalid-recover*)
apply (*rule safe-poly-vars*)
apply (*auto intro*!: *ext*)
done

lemma *WTF-RF-recover*:
‹*hn-ctxt* (*invalid-assn monom-assn* $\times_a$ *invalid-assn int-assn*) *xb*
x'a $\vee_A$
hn-ctxt monomial-assn xb x'a $\Longrightarrow_t$
hn-ctxt (*monomial-assn*) *xb x'a*›
by (*smt assn-aci*(*5*) *hn-ctxt-def invalid-assn-distrib invalid-pure-recover is-pure-conv*
merge-thms(*4*) *merge-true-star reorder-enttI safe-poly-vars*(*3*) *star-aci*(*2*) *star-aci*(*3*))

lemma *WTF-RF*:
‹*hn-ctxt* (*invalid-assn monom-assn* $\times_a$ *invalid-assn int-assn*) *xb x'a* $*$
(*hn-invalid poly-assn la l'a* $*$ *hn-invalid int-assn a2' a2* $*$
hn-invalid monom-assn a1' a1 $*$
hn-invalid poly-assn l l' $*$
hn-invalid monomial-assn xa x' $*$
hn-invalid poly-assn ax px) $\Longrightarrow_t$
hn-ctxt (*monomial-assn*) *xb x'a* $*$
hn-ctxt poly-assn
la l'a $*$
hn-ctxt poly-assn l l' $*$
(*hn-invalid int-assn a2' a2* $*$
hn-invalid monom-assn a1' a1 $*$
hn-invalid monomial-assn xa x' $*$
hn-invalid poly-assn ax px)›
‹*hn-ctxt* (*invalid-assn monom-assn* $\times_a$ *invalid-assn int-assn*) *xa x'* $*$
(*hn-ctxt poly-assn l l'* $*$ *hn-invalid poly-assn ax px*) $\Longrightarrow_t$
hn-ctxt (*monomial-assn*) *xa x'* $*$
hn-ctxt poly-assn l l' $*$
hn-ctxt poly-assn ax px $*$
emp›
by *sepref-dbg-trans-step+*

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*›
  supply [[*goals-limit=1*]]
  unfolding *merge-coeffs-alt-def*
    HOL-list.fold-custom-empty poly-assn-alt-def
  apply (*rewrite* **in** ‹-› *annotate-assn*[**where** *A*=‹*poly-assn*›])
  apply *sepref-dbg-preproc*
  apply *sepref-dbg-cons-init*
  apply *sepref-dbg-id*
  apply *sepref-dbg-monadify*
  apply *sepref-dbg-opt-init*
  apply (*rule WTF-RF* | *sepref-dbg-trans-step*)+
  apply *sepref-dbg-opt*
  apply *sepref-dbg-cons-solve*
  apply *sepref-dbg-cons-solve*
  apply *sepref-dbg-constraints*
  done

definition *full-quicksort-poly* **where**
  ‹*full-quicksort-poly* = *full-quicksort-ref* (λ*x y*. *x* = *y* ∨ (*x*, *y*) ∈ *term-order-rel*) *fst*›

lemma *down-eq-id-list-rel*: ‹⇓(⟨*Id*⟩*list-rel*) *x* = *x*›
  by *auto*

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*})›
  by (*smt* (*verit*) *less-char-def char.less-trans less-than-char-def lexord-partial-trans p2rel-def*)

lemma *full-quicksort-sort-poly-spec*:
  ‹(*full-quicksort-poly*, *sort-poly-spec*) ∈ ⟨*Id*⟩*list-rel* →$_f$ ⟨⟨*Id*⟩*list-rel*⟩*nres-rel*›
proof −
  have *xs*: ‹(*xs*, *xs*) ∈ ⟨*Id*⟩*list-rel*› **and** ‹⇓(⟨*Id*⟩*list-rel*) *x* = *x*› **for** *x xs*
    by *auto*
  show *?thesis*
    apply (*intro frefI nres-relI*)
    unfolding *full-quicksort-poly-def*
    apply (*rule full-quicksort-ref-full-quicksort*[*THEN fref-to-Down-curry*, *THEN order-trans*])
    subgoal

```
    by (auto simp: rel2p-def var-order-rel-def p2rel-def Relation.total-on-def
      dest: string-list-trans)
   subgoal
    using total-on-lexord-less-than-char-linear[unfolded var-order-rel-def]
    apply (auto simp: rel2p-def var-order-rel-def p2rel-def Relation.total-on-def less-char-def)
    done
   subgoal by fast
   apply (rule xs)
   apply (subst down-eq-id-list-rel)
   unfolding sorted-wrt-map sort-poly-spec-def
   apply (rule full-quicksort-correct-sorted[where R = ‹(λx y. x = y ∨ (x, y) ∈ term-order-rel)› and
h = ‹fst›,
     THEN order-trans])
   subgoal
    by (auto simp: rel2p-def var-order-rel-def p2rel-def Relation.total-on-def dest: string-list-trans)
   subgoal for x y
    using total-on-lexord-less-than-char-linear[unfolded var-order-rel-def]
    apply (auto simp: rel2p-def var-order-rel-def p2rel-def Relation.total-on-def
      less-char-def)
    done
  subgoal
   by (auto simp: rel2p-def p2rel-def)
  done
qed
```

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 (λa 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)))›
  unfolding msort-poly-impl-def
 apply (auto split: list.splits simp: merge-poly-def)
```

done

lemma *le-term-order-rel′*:
‹(≤) = (λx y. x = y ∨ term-order-rel′ x y)›
apply (*intro ext*)
apply (*auto simp add*: *less-list-def less-eq-list-def*
lexordp-eq-conv-lexord lexordp-def)
using *term-order-rel′-alt-def-lexord term-order-rel′-def* **apply** *blast*
using *term-order-rel′-alt-def-lexord term-order-rel′-def* **apply** *blast*
done

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*›
apply *auto*
done

lemma *var-order-rel′*:
‹(≤) = (λx y. x = y ∨ (x,y) ∈ *var-order-rel*)›
by (*intro ext*)
(*auto simp add*: *less-list-def less-eq-list-def*
lexordp-eq-conv-lexord lexordp-def var-order-rel-def
lexordp-conv-lexord p2rel-def)

lemma *var-order-rel″*:
‹(*x*,*y*) ∈ *var-order-rel* ⟷ *x* < *y*›
by (*metis leD less-than-char-linear lexord-linear neq-iff var-order-rel′ var-order-rel-antisym*
var-order-rel-def)

lemma *lexord-eq-alt-def1*:
‹*a* ≤ *b* = *lexord-eq a b*› **for** *a b* :: ‹*String.literal list*›
unfolding *le-term-order-rel′*
apply (*induction a b rule*: *lexord-eq.induct*)
apply (*auto simp*: *var-order-rel″ less-eq-list-def*)
done

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*)›
apply (*subst eq-commute*)
apply (*induction xs ys rule*: *lexord-eq.induct*)

subgoal by (*subst RECT-unfold*, *refine-mono*) *auto*
subgoal by (*subst RECT-unfold*, *refine-mono*) *auto*
subgoal by (*subst RECT-unfold*, *refine-mono*) *auto*
done

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*›
by (*auto simp*: *p2rel-def rel2p-def*)

lemma *var-order-rel-alt-def*:
‹*var-order-rel* = *p2rel char.lexordp*›
apply (*auto simp*: *p2rel-def char.lexordp-conv-lexord var-order-rel-def*)
using *char.lexordp-conv-lexord* **apply** *auto*
done

lemma *var-order-rel-var-order*:
‹(*x*, *y*) ∈ *var-order-rel* ⟷ *var-order x y*›
by (*auto simp*: *rel2p-def*)

lemma *var-order-string-le*[*sepref-import-param*]:
‹((<), *var-order′*) ∈ *string-rel* → *string-rel* → *bool-rel*›
apply (*auto intro*!: *frefI simp*: *string-rel-def String.less-literal-def*
rel2p-def linorder.lexordp-conv-lexord[*OF char.linorder-axioms*,
unfolded less-eq-char-def] *var-order-rel-def*
p2rel-def
simp flip: *PAC-Polynomials-Term.less-char-def*)
using *char.lexordp-conv-lexord* **apply** *auto*
done

lemma [*sepref-import-param*]:
‹((≤), (≤)) ∈ *monom-rel* → *monom-rel* →*bool-rel*›
apply (*intro fun-relI*)
using *list-rel-list-rel-order-iff* **by** *fastforce*

lemma [*sepref-import-param*]:
‹((<), (<)) ∈ *string-rel* → *string-rel* →*bool-rel*›
proof −
have [*iff*]: ‹*ord.lexordp* (<) (*literal.explode a*) (*literal.explode aa*) ⟷
List.lexordp (<) (*literal.explode a*) (*literal.explode aa*)› **for** *a aa*
apply (*rule iffI*)
apply (*metis PAC-Checker-Relation.less-char-def char.lexordp-conv-lexord less-list-def*
p2rel-def var-order-rel′′ var-order-rel-def)
apply (*metis PAC-Checker-Relation.less-char-def char.lexordp-conv-lexord less-list-def*
p2rel-def var-order-rel′′ var-order-rel-def)
done
show *?thesis*
unfolding *string-rel-def less-literal.rep-eq less-than-char-def*
less-eq-list-def PAC-Polynomials-Term.less-char-def[*symmetric*]
by (*intro fun-relI*)
(*auto simp*: *string-rel-def less-literal.rep-eq*
less-list-def char.lexordp-conv-lexord lexordp-eq-refl

lexordp-eq-conv-lexord)
qed

lemma *lexordp-char-char*: ‹*ord-class.lexordp* = *char.lexordp*›
unfolding *char.lexordp-def ord-class.lexordp-def*
by (*rule arg-cong*[*of* - - *lfp*])
(*auto intro*!: *ext*)

lemma [*sepref-import-param*]:
‹((≤), (≤)) ∈ *string-rel* → *string-rel* →*bool-rel*›
unfolding *string-rel-def less-eq-literal.rep-eq less-than-char-def*
less-eq-list-def PAC-Polynomials-Term.less-char-def[*symmetric*]
by (*intro fun-relI*)
(*auto simp*: *string-rel-def less-eq-literal.rep-eq less-than-char-def*
less-eq-list-def char.lexordp-eq-conv-lexord lexordp-eq-refl
lexordp-eq-conv-lexord lexordp-char-char
simp flip: *less-char-def*[*abs-def*])

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*›
supply[[*goals-limit*=*1*]]
unfolding *lexord-eq-alt-def2*
by *sepref*

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*›
by (*metis PAC-Checker-Relation.less-char-def p2rel-def string-list-trans var-order-rel-def*)

lemma *merge-sort-poly-sort-poly-spec*:
‹(*RETURN o merge-sort-poly*, *sort-poly-spec*) ∈ ⟨*Id*⟩*list-rel* →*f* ⟨⟨*Id*⟩*list-rel*⟩*nres-rel*›
unfolding *sort-poly-spec-def merge-sort-poly-def*
apply (*intro frefI nres-relI*)
using *total-on-lexord-less-than-char-linear var-order-rel-def*
by (*auto intro*!: *sorted-msort simp*: *sorted-wrt-map rel2p-def*
le-term-order-rel′ transp-def dest: *term-order-rel-trans*)

lemma *msort-alt-def*:
‹*RETURN o* (*msort f*) =
RECT (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)})›
  apply (intro ext)
  unfolding comp-def
  apply (induct-tac f x rule: msort.induct)
  subgoal by (subst RECT-unfold, refine-mono) auto
  subgoal by (subst RECT-unfold, refine-mono) auto
  subgoal
    by (subst RECT-unfold, refine-mono)
     (smt (verit) let-to-bind-conv list.simps(5) msort.simps(3))
  done

lemma monomial-rel-order-map:
  ‹(x, a, b) ∈ monomial-rel ⟹
      (y, aa, bb) ∈ monomial-rel ⟹
      fst x ≤ fst y ⟷ a ≤ aa›
  apply (cases x; cases y)
  apply auto
  using list-rel-list-rel-order-iff by fastforce+


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)›
  subgoal
    apply (intro ext)
    apply (case-tac x; case-tac xa)
    apply (auto simp: relAPP-def p2rel-def pure-def)
    done
  subgoal H
    apply (intro ext)
    apply (case-tac x; case-tac xa)
    by (simp add: list-assn-pure-conv)
  subgoal
    unfolding H
    by (simp add: list-assn-pure-conv relAPP-def)
  done

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)
  by (auto simp: step-rewrite-pure(1)[symmetric] is-pure-conv)


lemma merge-poly-merge-poly:
  ‹(merge-poly, merge-poly)
   ∈ poly-rel → poly-rel → poly-rel›
   unfolding merge-poly-def
  apply (intro fun-relI)
  subgoal for a a' aa a'a
    apply (induction ‹(λ(a :: String.literal list × int)
```

```
    (b :: String.literal list × int). fst a ≤ fst b)› a aa
    arbitrary: a' a'a
    rule: merge.induct)
  subgoal
    by (auto elim!: list-relE3 list-relE4 list-relE list-relE2
      simp: monomial-rel-order-map)
  subgoal
    by (auto elim!: list-relE3 list-relE)
  subgoal
    by (auto elim!: list-relE3 list-relE4 list-relE list-relE2)
  done
done

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›
  using merge-poly-merge-poly
  unfolding fun-rel-def
  by auto

lemma list-rel-takeD:
  ‹(a, b) ∈ ⟨R⟩list-rel ⟹ (n, n')∈ Id ⟹ (take n a, take n' b) ∈ ⟨R⟩list-rel›
  by (simp add: list-rel-eq-listrel listrel-iff-nth relAPP-def)

lemma list-rel-dropD:
  ‹(a, b) ∈ ⟨R⟩list-rel ⟹ (n, n')∈ Id ⟹ (drop n a, drop n' b) ∈ ⟨R⟩list-rel›
  by (simp add: list-rel-eq-listrel listrel-iff-nth relAPP-def)

lemma merge-sort-poly[sepref-import-param]:
  ‹(msort-poly-impl, merge-sort-poly)
   ∈ poly-rel → poly-rel›
   unfolding merge-sort-poly-def msort-poly-impl-def
  apply (intro fun-relI)
  subgoal for a a'
    apply (induction ‹(λ(a :: String.literal list × int)
      (b :: String.literal list × int). fst a ≤ fst b)› a
      arbitrary: a'
      rule: msort.induct)
    subgoal
      by auto
    subgoal
      by (auto elim!: list-relE3 list-relE)
    subgoal premises p
      using p
      by (auto elim!: list-relE3 list-relE4 list-relE list-relE2
        simp: merge-poly-def[symmetric]
        intro!: list-rel-takeD list-rel-dropD
        intro!: merge-poly-merge-poly2 p(1)[simplified] p(2)[simplified],
        auto simp: list-rel-imp-same-length)
    done
  done
```

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*›
  :: ‹$nat\text{-}assn^k *_a nat\text{-}assn^k *_a poly\text{-}assn^k \rightarrow_a prod\text{-}assn\ poly\text{-}assn\ nat\text{-}assn$ ›
  unfolding *partition-main-poly-def partition-main-def*
    term-order-rel'-def[*symmetric*]
    term-order-rel'-alt-def
    le-term-order-rel'
  by *sepref*

declare *partition-main-poly-impl.refine*[*sepref-fr-rules*]

sepref-definition *partition-between-poly-impl*
  is ‹*uncurry2 partition-between-poly*›
  :: ‹$nat\text{-}assn^k *_a nat\text{-}assn^k *_a poly\text{-}assn^k \rightarrow_a prod\text{-}assn\ poly\text{-}assn\ nat\text{-}assn$ ›
  unfolding *partition-between-poly-def partition-between-ref-def*
    partition-main-poly-def[*symmetric*]
  unfolding *choose-pivot3-def*
    term-order-rel'-def[*symmetric*]
    term-order-rel'-alt-def choose-pivot-def
    lexord-eq-alt-def1
  by *sepref*

declare *partition-between-poly-impl.refine*[*sepref-fr-rules*]

sepref-definition *quicksort-poly-impl*
  is ‹*uncurry2 quicksort-poly*›
  :: ‹$nat\text{-}assn^k *_a nat\text{-}assn^k *_a poly\text{-}assn^k \rightarrow_a poly\text{-}assn$›
  unfolding *partition-main-poly-def quicksort-ref-def quicksort-poly-def*
    partition-between-poly-def[*symmetric*]
  by *sepref*

lemmas [*sepref-fr-rules*] = *quicksort-poly-impl.refine*

sepref-register *quicksort-poly*
sepref-definition *full-quicksort-poly-impl*
  is ‹*full-quicksort-poly*›
  :: ‹$poly\text{-}assn^k \rightarrow_a poly\text{-}assn$›
  unfolding *full-quicksort-poly-def full-quicksort-ref-def*
    quicksort-poly-def[*symmetric*]
    le-term-order-rel'[*symmetric*]
    term-order-rel'-def[*symmetric*]
    List.null-iff
  by *sepref*

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} $\rightarrow_a$ *poly-assn*›
supply [[*goals-limit=1*]]
unfolding *normalize-poly-def*
by *sepref*

declare *normalize-poly-impl.refine*[*sepref-fr-rules*]

definition *full-quicksort-vars* **where**
‹*full-quicksort-vars* = *full-quicksort-ref* ($\lambda x\ y.\ x = y \vee (x, y) \in$ *var-order-rel*) *id*›

definition *quicksort-vars*:: ‹*nat* ⇒ *nat* ⇒ *string list* ⇒ (*string list*) *nres*› **where**
‹*quicksort-vars x y z* = *quicksort-ref* (≤) *id* (*x*, *y*, *z*)›

definition *partition-between-vars* :: ‹*nat* ⇒ *nat* ⇒ *string list* ⇒ (*string list* × *nat*) *nres*› **where**
‹*partition-between-vars* = *partition-between-ref* (≤) *id*›

definition *partition-main-vars* :: ‹*nat* ⇒ *nat* ⇒ *string list* ⇒ (*string list* × *nat*) *nres*› **where**
‹*partition-main-vars* = *partition-main* (≤) *id*›

lemma *total-on-lexord-less-than-char-linear2*:
‹*xs* ≠ *ys* ⟹ (*xs*, *ys*) ∉ *lexord* (*less-than-char*) ⟷
(*ys*, *xs*) ∈ *lexord less-than-char*›
using *lexord-linear*[*of* ‹*less-than-char*› *xs ys*]
using *lexord-linear*[*of* ‹*less-than-char*›] *less-than-char-linear*
apply (*auto simp*: *Relation.total-on-def*)
using *lexord-irrefl*[*OF irrefl-less-than-char*]
antisym-lexord[*OF antisym-less-than-char irrefl-less-than-char*]
apply (*auto simp*: *antisym-def*)
done

lemma *string-trans*:
‹(*xa*, *ya*) ∈ *lexord* {(*x*::*char*, *y*::*char*). *x* < *y*} ⟹
(*ya*, *z*) ∈ *lexord* {(*x*::*char*, *y*::*char*). *x* < *y*} ⟹
(*xa*, *z*) ∈ *lexord* {(*x*::*char*, *y*::*char*). *x* < *y*}›
by (*smt* (*verit*) *less-char-def char.less-trans less-than-char-def lexord-partial-trans p2rel-def*)

lemma *full-quicksort-sort-vars-spec*:
‹(*full-quicksort-vars*, *sort-coeff*) ∈ ⟨*Id*⟩*list-rel* $\rightarrow_f$ ⟨⟨*Id*⟩*list-rel*⟩*nres-rel*›
proof −
have *xs*: ‹(*xs*, *xs*) ∈ ⟨*Id*⟩*list-rel*› **and** ‹⇓(⟨*Id*⟩*list-rel*) *x* = *x*› **for** *x xs*
by *auto*
show *?thesis*
apply (*intro frefI nres-relI*)
unfolding *full-quicksort-vars-def*
apply (*rule full-quicksort-ref-full-quicksort*[*THEN fref-to-Down-curry*, *THEN order-trans*])
subgoal
by (*auto simp*: *rel2p-def var-order-rel-def p2rel-def Relation.total-on-def*
dest: *string-trans*)
subgoal
using *total-on-lexord-less-than-char-linear2*[*unfolded var-order-rel-def*]
apply (*auto simp*: *rel2p-def var-order-rel-def p2rel-def Relation.total-on-def less-char-def*)

```
    done
  subgoal by fast
  apply (rule xs)
  apply (subst down-eq-id-list-rel)
  unfolding sorted-wrt-map sort-coeff-def
  apply (rule full-quicksort-correct-sorted[where R = ‹(λx y. x = y ∨ (x, y) ∈ var-order-rel)› and h
= ‹id›,
      THEN order-trans])
  subgoal
    by (auto simp: rel2p-def var-order-rel-def p2rel-def Relation.total-on-def dest: string-trans)
  subgoal for x y
    using total-on-lexord-less-than-char-linear2[unfolded var-order-rel-def]
    by (auto simp: rel2p-def var-order-rel-def p2rel-def Relation.total-on-def
      less-char-def)
 subgoal
  by (auto simp: rel2p-def p2rel-def rel2p-def[abs-def])
 done
qed


sepref-definition partition-main-vars-impl
  is ‹uncurry2 partition-main-vars›
  :: ‹nat-assn^k *_a nat-assn^k *_a (monom-assn)^k →_a prod-assn (monom-assn) nat-assn›
  unfolding partition-main-vars-def partition-main-def
    var-order-rel-var-order
    var-order'-def[symmetric]
    term-order-rel'-alt-def
    le-term-order-rel'
    id-apply
    by sepref

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 →_a prod-assn monom-assn nat-assn ›
  unfolding partition-between-vars-def partition-between-ref-def
    partition-main-vars-def[symmetric]
  unfolding choose-pivot3-def
    term-order-rel'-def[symmetric]
    term-order-rel'-alt-def choose-pivot-def
    le-term-order-rel' id-apply
  by sepref

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 →_a monom-assn›
  unfolding partition-main-vars-def quicksort-ref-def quicksort-vars-def
    partition-between-vars-def[symmetric]
  by sepref

lemmas [sepref-fr-rules] = quicksort-vars-impl.refine
```

sepref-register *quicksort-vars*

lemma *le-var-order-rel*:
‹(≤) = (λx y. x = y ∨ (x, y) ∈ var-order-rel)›
by (*intro ext*)
(*auto simp add*: *less-list-def less-eq-list-def rel2p-def*
p2rel-def lexordp-conv-lexord p2rel-def var-order-rel-def
lexordp-eq-conv-lexord lexordp-def)

sepref-definition *full-quicksort-vars-impl*
is ‹*full-quicksort-vars*›
:: ‹*monom-assn*^{k} →$_a$ *monom-assn*›
unfolding *full-quicksort-vars-def full-quicksort-ref-def*
quicksort-vars-def[*symmetric*]
le-var-order-rel[*symmetric*]
term-order-rel'-def[*symmetric*]
List.null-iff
by *sepref*

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*›
unfolding *string-rel-def less-eq-literal.rep-eq less-than-char-def*
less-eq-list-def PAC-Polynomials-Term.less-char-def[*symmetric*]
by (*auto simp*: *string-rel-def less-eq-literal.rep-eq less-than-char-def*
less-eq-list-def char.lexordp-eq-conv-lexord lexordp-eq-refl
lexordp-char-char lexordp-eq-conv-lexord
simp flip: *less-char-def*[*abs-def*])

lemma *merge-monoms-merge-monoms*:
‹(*merge-monoms*, *merge-monoms*) ∈ *monom-rel* → *monom-rel* → *monom-rel*›
unfolding *merge-monoms-def*
apply (*intro fun-relI*)
subgoal for *a a' aa a'a*
apply (*induction* ‹(λ(*a* :: *String.literal*)
(*b* :: *String.literal*). *a* ≤ *b*)› *a aa*
arbitrary: *a' a'a*
rule: *merge.induct*)
subgoal
by (*auto elim*!: *list-relE3 list-relE4 list-relE list-relE2*
simp: *string-rel-order-map*)
subgoal
by (*auto elim*!: *list-relE3 list-relE*)
subgoal
by (*auto elim*!: *list-relE3 list-relE4 list-relE list-relE2*)
done
done

lemma *merge-monoms-merge-monoms2*:
‹(*a*, *b*) ∈ *monom-rel* ⟹ (*a'*, *b'*) ∈ *monom-rel* ⟹

```
    (merge-monoms a a', merge-monoms b b') ∈ monom-rel›
  using merge-monoms-merge-monoms
  unfolding fun-rel-def merge-monoms-def
  by auto


lemma msort-monoms-impl:
  ‹(msort-monoms-impl, merge-monoms-poly)
   ∈ monom-rel → monom-rel›
   unfolding msort-monoms-impl-def merge-monoms-poly-def
  apply (intro fun-relI)
  subgoal for a a'
    apply (induction ‹(λ(a :: String.literal)
      (b :: String.literal). a ≤ b)› a
      arbitrary: a'
      rule: msort.induct)
    subgoal
      by auto
    subgoal
      by (auto elim!: list-relE3 list-relE)
    subgoal premises p
      using p
      by (auto elim!: list-relE3 list-relE4 list-relE list-relE2
        simp: merge-monoms-def[symmetric] intro!: list-rel-takeD list-rel-dropD
        intro!: merge-monoms-merge-monoms2 p(1)[simplified] p(2)[simplified])
        (simp-all add: list-rel-imp-same-length)
    done
  done

lemma merge-sort-monoms-sort-monoms-spec:
  ‹(RETURN o merge-monoms-poly, sort-coeff) ∈ ⟨Id⟩list-rel →f ⟨⟨Id⟩list-rel⟩nres-rel›
  unfolding merge-monoms-poly-def sort-coeff-def
  by (intro frefI nres-relI)
    (auto intro!: sorted-msort simp: sorted-wrt-map rel2p-def
     le-term-order-rel' transp-def rel2p-def[abs-def]
     simp flip: le-var-order-rel)

sepref-register sort-coeff
lemma [sepref-fr-rules]:
  ‹(return o msort-monoms-impl, sort-coeff) ∈ monom-assn^k →a monom-assn›
  using msort-monoms-impl[sepref-param, FCOMP merge-sort-monoms-sort-monoms-spec]
  by auto

sepref-definition sort-all-coeffs-impl
  is ‹sort-all-coeffs›
  :: ‹poly-assn^k →a poly-assn›
  unfolding sort-all-coeffs-def
    HOL-list.fold-custom-empty
  by sepref

declare sort-all-coeffs-impl.refine[sepref-fr-rules]

lemma merge-coeffs0-alt-def:
  ‹(RETURN o merge-coeffs0) p =
   REC_T(λf 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›
  apply (subst eq-commute)
  apply (induction p rule: merge-coeffs0.induct)
  subgoal by (subst RECT-unfold, refine-mono) auto
  subgoal by (subst RECT-unfold, refine-mono) auto
  subgoal by (subst RECT-unfold, refine-mono) (auto simp: let-to-bind-conv)
  done
```

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›
  supply [[goals-limit=1]]
  unfolding merge-coeffs0-alt-def
    HOL-list.fold-custom-empty
  apply sepref-dbg-preproc
  apply sepref-dbg-cons-init
  apply sepref-dbg-id
  apply sepref-dbg-monadify
  apply sepref-dbg-opt-init
  apply (rule WTF-RF | sepref-dbg-trans-step)+
  apply sepref-dbg-opt
  apply sepref-dbg-cons-solve
  apply sepref-dbg-cons-solve
  apply sepref-dbg-constraints
  done


declare merge-coeffs0-impl.refine[sepref-fr-rules]

sepref-definition fully-normalize-poly-impl
  is ‹full-normalize-poly›
  :: ‹poly-assn^k →_a poly-assn›
  supply [[goals-limit=1]]
  unfolding full-normalize-poly-def
  by sepref

declare fully-normalize-poly-impl.refine[sepref-fr-rules]


end

theory PAC-Version
  imports Main
```

begin

This code was taken from IsaFoR. However, for the AFP, we use the version name *AFP*, instead of a mercurial version.

local-setup ‹
  let
    val version = AFP
  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 and adding more efficient data structures (mostly replacing the set of variables by a more efficient hash map).

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*) $\mathcal{V}$ ⟷ *x* ∈ $\mathcal{V}$ ∧ *vars-of-monom-in xs* $\mathcal{V}$›

fun *vars-of-poly-in* **where**
  ‹*vars-of-poly-in* [] - = *True*› |
  ‹*vars-of-poly-in* ((*x*, -) # *xs*) $\mathcal{V}$ ⟷ *vars-of-monom-in x* $\mathcal{V}$ ∧ *vars-of-poly-in xs* $\mathcal{V}$›

lemma *vars-of-monom-in-alt-def*:
  ‹*vars-of-monom-in xs* $\mathcal{V}$ ⟷ *set xs* ⊆ $\mathcal{V}$›
  by (*induction xs*)
   auto

lemma *vars-llist-alt-def*:
  ‹*vars-llist xs* ⊆ $\mathcal{V}$ ⟷ *vars-of-poly-in xs* $\mathcal{V}$›
  by (*induction xs*)
   (*auto simp*: *vars-llist-def vars-of-monom-in-alt-def*)

lemma *vars-of-monom-in-alt-def2*:
  ‹*vars-of-monom-in xs* $\mathcal{V}$ ⟷ *fold* (λ*x b*. *b* ∧ *x* ∈ $\mathcal{V}$) *xs True*›
  apply (*subst foldr-fold*[*symmetric*])
  subgoal by *auto*
  subgoal by (*induction xs*) *auto*

done

sepref-definition *vars-of-monom-in-impl*
is ‹*uncurry* (*RETURN oo vars-of-monom-in*)›
:: ‹(*list-assn string-assn*)^k $*_a$ *vars-assn*^{k} $\rightarrow_a$ *bool-assn*›
unfolding *vars-of-monom-in-alt-def2*
by *sepref*

declare *vars-of-monom-in-impl.refine*[*sepref-fr-rules*]

lemma *vars-of-poly-in-alt-def2*:
‹*vars-of-poly-in xs* $\mathcal{V}$ $\longleftrightarrow$ *fold* (λ(*x*, -) *b*. *b* $\wedge$ *vars-of-monom-in x* $\mathcal{V}$) *xs True*›
apply (*subst foldr-fold*[*symmetric*])
subgoal by *auto*
subgoal by (*induction xs*) *auto*
done

sepref-definition *vars-of-poly-in-impl*
is ‹*uncurry* (*RETURN oo vars-of-poly-in*)›
:: ‹(*poly-assn*)^k $*_a$ *vars-assn*^{k} $\rightarrow_a$ *bool-assn*›
unfolding *vars-of-poly-in-alt-def2*
by *sepref*

declare *vars-of-poly-in-impl.refine*[*sepref-fr-rules*]

definition *union-vars-monom* :: ‹*string list* $\Rightarrow$ *string set* $\Rightarrow$ *string set*› **where**
‹*union-vars-monom xs* $\mathcal{V}$ = *fold insert xs* $\mathcal{V}$›

definition *union-vars-poly* :: ‹*llist-polynomial* $\Rightarrow$ *string set* $\Rightarrow$ *string set*› **where**
‹*union-vars-poly xs* $\mathcal{V}$ = *fold* (λ(*xs*, -) $\mathcal{V}$. *union-vars-monom xs* $\mathcal{V}$) *xs* $\mathcal{V}$›

lemma *union-vars-monom-alt-def*:
‹*union-vars-monom xs* $\mathcal{V}$ = $\mathcal{V}$ $\cup$ *set xs*›
unfolding *union-vars-monom-def*
apply (*subst foldr-fold*[*symmetric*])
subgoal for *x y*
by (*cases x*; *cases y*) *auto*
subgoal
by (*induction xs*) *auto*
done

lemma *union-vars-poly-alt-def*:
‹*union-vars-poly xs* $\mathcal{V}$ = $\mathcal{V}$ $\cup$ *vars-llist xs*›
unfolding *union-vars-poly-def*
apply (*subst foldr-fold*[*symmetric*])
subgoal for *x y*
by (*cases x*; *cases y*)
(*auto simp*: *union-vars-monom-alt-def*)
subgoal
by (*induction xs*)
(*auto simp*: *vars-llist-def union-vars-monom-alt-def*)
done

sepref-definition *union-vars-monom-impl*
is ‹*uncurry* (*RETURN oo union-vars-monom*)›
:: ‹*monom-assn*^{k} $*_a$ *vars-assn*^{d} $\rightarrow_a$ *vars-assn*›
unfolding *union-vars-monom-def*
by *sepref*

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*›
unfolding *union-vars-poly-def*
by *sepref*

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*›
by (*sepref-to-hoare*)
sep-auto

lemma *FOUND-hnr*[*sepref-fr-rules*]:
‹(*uncurry0* (*return CFOUND*), *uncurry0* (*RETURN CFOUND*)) ∈ *unit-assn*^{k} $\rightarrow_a$ *status-assn R*›
by (*sepref-to-hoare*)
sep-auto

lemma *is-success-hnr*[*sepref-fr-rules*]:
‹*CONSTRAINT is-pure R* ⟹
((*return o is-cfound*), (*RETURN o is-cfound*)) ∈ (*status-assn R*)^k $\rightarrow_a$ *bool-assn*›
apply (*sepref-to-hoare*)
apply (*rename-tac xi x*; *case-tac xi*; *case-tac x*)
apply *sep-auto*+
done

lemma *is-cfailed-hnr*[*sepref-fr-rules*]:
‹*CONSTRAINT is-pure R* ⟹
((*return o is-cfailed*), (*RETURN o is-cfailed*)) ∈ (*status-assn R*)^k $\rightarrow_a$ *bool-assn*›
apply (*sepref-to-hoare*)
apply (*rename-tac xi x*; *case-tac xi*; *case-tac x*)
apply *sep-auto*+
done

lemma *merge-cstatus-hnr*[*sepref-fr-rules*]:
‹*CONSTRAINT is-pure R* ⟹
(*uncurry* (*return oo merge-cstatus*), *uncurry* (*RETURN oo merge-cstatus*)) ∈
(*status-assn R*)^k $*_a$ (*status-assn R*)^k $\rightarrow_a$ *status-assn R*›

apply (*sepref-to-hoare*)
by (*case-tac b*; *case-tac bi*; *case-tac a*; *case-tac ai*; *sep-auto simp*: *is-pure-conv pure-app-eq*)

sepref-definition *add-poly-impl*
is ‹*add-poly-l*›
:: ‹(*poly-assn* $\times_a$ *poly-assn*)^k $\rightarrow_a$ *poly-assn*›
supply [[*goals-limit=1*]]
unfolding *add-poly-l-def*
HOL-list.fold-custom-empty
term-order-rel'-def[*symmetric*]
term-order-rel'-alt-def
by *sepref*

declare *add-poly-impl.refine*[*sepref-fr-rules*]

sepref-register *mult-monomials*
lemma *mult-monoms-alt-def*:
‹(*RETURN oo mult-monoms*) *x y* = REC_T
(λ*f* (*p*, *q*).
case (*p*, *q*) *of*
([], -) ⇒ *RETURN q*
| (-, []) ⇒ *RETURN p*
| (*x* # *p*, *y* # *q*) ⇒
(*if x* = *y then do* {
pq ← *f* (*p*, *q*);
RETURN (*x* # *pq*)}
else if (*x*, *y*) ∈ *var-order-rel*
then do {
pq ← *f* (*p*, *y* # *q*);
RETURN (*x* # *pq*)}
else do {
pq ← *f* (*x* # *p*, *q*);
RETURN (*y* # *pq*)}))
(*x*, *y*)›
apply (*subst eq-commute*)
apply (*induction x y rule*: *mult-monoms.induct*)
subgoal for *p*
by (*subst RECT-unfold*, *refine-mono*) (*auto split*: *list.splits*)
subgoal for *p*
by (*subst RECT-unfold*, *refine-mono*) (*auto split*: *list.splits*)
subgoal for *x p y q*
by (*subst RECT-unfold*, *refine-mono*) (*auto split*: *list.splits simp*: *let-to-bind-conv*)
done

sepref-definition *mult-monoms-impl*
is ‹*uncurry* (*RETURN oo mult-monoms*)›
:: ‹(*monom-assn*)^k $*_a$ (*monom-assn*)^k $\rightarrow_a$ (*monom-assn*)›
supply [[*goals-limit=1*]]
unfolding *mult-poly-raw-def*
HOL-list.fold-custom-empty
var-order'-def[*symmetric*]
term-order-rel'-alt-def

mult-monoms-alt-def
var-order-rel-var-order
by *sepref*

declare *mult-monoms-impl.refine*[*sepref-fr-rules*]

sepref-definition *mult-monomials-impl*
is ‹*uncurry* (*RETURN oo mult-monomials*)›
:: ‹(*monomial-assn*)^k $*_a$ (*monomial-assn*)^k $\rightarrow_a$ (*monomial-assn*)›
supply [[*goals-limit=1*]]
unfolding *mult-monomials-def*
HOL-list.fold-custom-empty
term-order-rel'-def[*symmetric*]
term-order-rel'-alt-def
by *sepref*

lemma *map-append-alt-def2*:
‹(*RETURN o* (*map-append f b*)) *xs* = *REC*$_T$
(λ*g xs*. *case xs of* [] ⇒ *RETURN b*
| *x* # *xs* ⇒ *do* {
y ← *g xs*;
RETURN (*f x* # *y*)
}) *xs*›
apply (*subst eq-commute*)
apply (*induction f b xs rule*: *map-append.induct*)
subgoal by (*subst RECT-unfold*, *refine-mono*) *auto*
subgoal by (*subst RECT-unfold*, *refine-mono*) *auto*
done

definition *map-append-poly-mult* **where**
‹*map-append-poly-mult x* = *map-append* (*mult-monomials x*)›

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*›
unfolding *map-append-poly-mult-def*
map-append-alt-def2
by *sepref*

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*›
supply [[*goals-limit=1*]]
supply [[*eta-contract* = *false*, *show-abbrevs=false*]]
unfolding *mult-poly-raw-def*
HOL-list.fold-custom-empty
term-order-rel'-def[*symmetric*]
term-order-rel'-alt-def

foldl-conv-fold
fold-eq-nfoldli
map-append-poly-mult-def[*symmetric*]
map-append-alt-def[*symmetric*]
by *sepref*

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*›
supply [[*goals-limit=1*]]
unfolding *mult-poly-full-def*
HOL-list.fold-custom-empty
term-order-rel'-def[*symmetric*]
term-order-rel'-alt-def
by *sepref*

declare *mult-poly-impl.refine*[*sepref-fr-rules*]

lemma *inverse-monomial*:
‹*monom-rel*$^{-1}$ $\times_r$ *int-rel* = (*monom-rel* $\times_r$ *int-rel*)$^{-1}$›
by (*auto*)

lemma *eq-poly-rel-eq*[*sepref-import-param*]:
‹((=), (=)) ∈ *poly-rel* → *poly-rel* → *bool-rel*›
using *list-rel-sv*[*of* ‹*monomial-rel*›, *OF single-valued-monomial-rel*]
using *list-rel-sv*[*OF single-valued-monomial-rel'*[*unfolded IS-LEFT-UNIQUE-def inv-list-rel-eq*]]
unfolding *inv-list-rel-eq*[*symmetric*]
by (*auto intro*!: *frefI simp*:
rel2p-def single-valued-def p2rel-def
simp del: *inv-list-rel-eq*)

sepref-definition *weak-equality-l-impl*
is ‹*uncurry weak-equality-l*›
:: ‹*poly-assn*^{k} $*_a$ *poly-assn*^{k} $\rightarrow_a$ *bool-assn*›
supply [[*goals-limit=1*]]
unfolding *weak-equality-l-def*
by *sepref*

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*›
by (*auto intro*: *fun-relI*)

lemma *status-assn-pure-conv*:

‹*status-assn* (*id-assn*) *a b* = *id-assn a b*›
by (*cases a*; *cases b*)
(*auto simp*: *pure-def*)

lemma [*sepref-fr-rules*]:
‹(*uncurry3* (λ*x 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} $\rightarrow_a$ *raw-string-assn*›
unfolding *show-nat-def*[*symmetric*] *list-assn-pure-conv*
prod-assn-pure-conv check-not-equal-dom-err-def
by (*sepref-to-hoare*; *sep-auto simp*: *error-msg-not-equal-dom-def*)

lemma [*sepref-fr-rules*]:
‹(*return o* (*error-msg-notin-dom o nat-of-uint64*), *RETURN o error-msg-notin-dom*)
∈ *uint64-nat-assn*^{k} $\rightarrow_a$ *raw-string-assn*›
‹(*return o* (*error-msg-reused-dom o nat-of-uint64*), *RETURN o error-msg-reused-dom*)
∈ *uint64-nat-assn*^{k} $\rightarrow_a$ *raw-string-assn*›
‹(*uncurry* (*return oo* (λ*i*. *error-msg* (*nat-of-uint64 i*))), *uncurry* (*RETURN oo error-msg*))
∈ *uint64-nat-assn*^{k} $*_a$ *raw-string-assn*^{k} $\rightarrow_a$ *status-assn raw-string-assn*›
‹(*uncurry* (*return oo error-msg*), *uncurry* (*RETURN oo error-msg*))
∈ *nat-assn*^{k} $*_a$ *raw-string-assn*^{k} $\rightarrow_a$ *status-assn raw-string-assn*›
unfolding *error-msg-notin-dom-def list-assn-pure-conv list-rel-id-simp*
unfolding *status-assn-pure-conv*
unfolding *show-nat-def*[*symmetric*]
by (*sepref-to-hoare*; *sep-auto simp*: *uint64-nat-rel-def br-def*; *fail*)+

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} $\rightarrow_a$ *status-assn raw-string-assn*›
supply [[*goals-limit=1*]]
unfolding *mult-poly-full-def*
HOL-list.fold-custom-empty
term-order-rel'-def[*symmetric*]
term-order-rel'-alt-def
check-addition-l-def
in-dom-m-lookup-iff
fmlookup'-def[*symmetric*]
vars-llist-alt-def
by *sepref*

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* ((λ*x 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*›
unfolding *check-mult-l-dom-err-def check-mult-l-dom-err-impl-def list-assn-pure-conv*
apply *sepref-to-hoare*
apply *sep-auto*
done

lemma [*sepref-fr-rules*]:
‹(*uncurry3* ((λ*x 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*›
unfolding *check-mult-l-mult-err-def check-mult-l-mult-err-impl-def list-assn-pure-conv*
apply *sepref-to-hoare*
apply *sep-auto*
done

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*›
supply [[*goals-limit=1*]]
unfolding *check-mult-l-def*
HOL-list.fold-custom-empty
term-order-rel'-def[*symmetric*]
term-order-rel'-alt-def
in-dom-m-lookup-iff
fmlookup'-def[*symmetric*]
vars-llist-alt-def
by *sepref*

declare *check-mult-l-impl.refine*[*sepref-fr-rules*]

definition *check-ext-l-dom-err-impl* :: ‹*uint64* ⇒ -› **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*)) ∈ *uint64-nat-assn*^{k} →$_a$ *raw-string-assn*›
unfolding *check-extension-l-dom-err-def check-ext-l-dom-err-impl-def list-assn-pure-conv*
apply *sepref-to-hoare*
apply *sep-auto*
done

definition *check-extension-l-no-new-var-err-impl* :: ‹- ⇒ -› **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*)) ∈ *poly-assn*^{k} →$_a$ *raw-string-assn*›
unfolding *check-extension-l-no-new-var-err-impl-def check-extension-l-no-new-var-err-def*

list-assn-pure-conv
apply *sepref-to-hoare*
apply *sep-auto*
done

definition *check-extension-l-side-cond-err-impl* :: ‹- ⇒ -› **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* (λ*x y*. *return oo* (*check-extension-l-side-cond-err-impl x y*))),
uncurry3 (*check-extension-l-side-cond-err*)) ∈ *string-assn*^{k} ∗$_a$ *poly-assn*^{k} ∗$_a$ *poly-assn*^{k} ∗$_a$ *poly-assn*^{k} →$_a$ *raw-string-assn*›
unfolding *check-extension-l-side-cond-err-impl-def check-extension-l-side-cond-err-def*
list-assn-pure-conv
apply *sepref-to-hoare*
apply *sep-auto*
done

definition *check-extension-l-new-var-multiple-err-impl* :: ‹- ⇒ -› **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*)) ∈ *string-assn*^{k} ∗$_a$ *poly-assn*^{k} →$_a$ *raw-string-assn*›
unfolding *check-extension-l-new-var-multiple-err-impl-def*
check-extension-l-new-var-multiple-err-def
list-assn-pure-conv
apply *sepref-to-hoare*
apply *sep-auto*
done

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* ⇒ *llist-polynomial*› **where**
‹*uminus-poly p′* = *map* (λ(*a*, *b*). (*a*, − *b*)) *p′*›

sepref-register *uminus-poly*
lemma [*sepref-import-param*]:
‹(*map* (λ(*a*, *b*). (*a*, − *b*)), *uminus-poly*) ∈ *poly-rel* → *poly-rel*›
unfolding *uminus-poly-def*
apply (*intro fun-relI*)
subgoal for *p p′*
by (*induction p p′ rule*: *list-rel-induct*)
auto
done

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*)$^{-1}$)›
unfolding *IS-LEFT-UNIQUE-def*[*symmetric*]
by (*auto simp*: *step-rewrite-pure single-valued-monomial-rel single-valued-monomial-rel' Sepref-Constraints.CONSTRAI*

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›
supply *option.splits*[*split*] *single-valued-the-monomial-assn*[*simp*]
supply [[*goals-limit=1*]]
unfolding
HOL-list.fold-custom-empty
term-order-rel'-def[*symmetric*]
term-order-rel'-alt-def
in-dom-m-lookup-iff
fmlookup'-def[*symmetric*]
vars-llist-alt-def
check-extension-l-def
not-not
option.case-eq-if
uminus-poly-def[*symmetric*]
HOL-list.fold-custom-empty
by *sepref*

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*›
supply [[*goals-limit=1*]]
unfolding *check-del-l-def*
in-dom-m-lookup-iff
fmlookup'-def[*symmetric*]
by *sepref*

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›
unfolding *poly-assn-list hn-ctxt-def*
by (*induction nat-assn poly-assn ‹id-assn :: string ⇒ -› b bi rule*: *pac-step-rel-assn.induct*)
(*auto simp*: *p2rel-def hn-val-unfold pac-step-rel-raw.simps relAPP-def*
pure-app-eq)

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*) ∈ [λ*x*. *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*) ∈ [λ*x*. *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*) ∈ [λ*x*. *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 →$_a$ *bool-assn*›
‹(*return o is-Mult*, *RETURN o is-Mult*) ∈ (*pac-step-rel-assn K V R*)^k →$_a$ *bool-assn*›
‹(*return o is-Del*, *RETURN o is-Del*) ∈ (*pac-step-rel-assn K V R*)^k →$_a$ *bool-assn*›
‹(*return o is-Extension*, *RETURN o is-Extension*) ∈ (*pac-step-rel-assn K V R*)^k →$_a$ *bool-assn*›
using *assms*
by (*sepref-to-hoare*; *sep-auto simp*: *pac-step-rel-assn-alt-def is-pure-conv ent-true-drop pure-app-eq*
split: *pac-step.splits*; *fail*)+

lemma [*sepref-fr-rules*]:
‹*CONSTRAINT is-pure K* ⟹
(*return o pac-src2*, *RETURN o pac-src2*) ∈ [λ*x*. *is-Add x*]$_a$ (*pac-step-rel-assn K V R*)^k → *K*›
‹*CONSTRAINT is-pure V* ⟹
(*return o pac-mult*, *RETURN o pac-mult*) ∈ [λ*x*. *is-Mult x*]$_a$ (*pac-step-rel-assn K V R*)^k → *V*›
‹*CONSTRAINT is-pure R* ⟹
(*return o new-var*, *RETURN o new-var*) ∈ [λ*x*. *is-Extension x*]$_a$ (*pac-step-rel-assn K V R*)^k → *R*›
by (*sepref-to-hoare*; *sep-auto simp*: *pac-step-rel-assn-alt-def is-pure-conv ent-true-drop pure-app-eq*
split: *pac-step.splits*; *fail*)+

lemma *is-Mult-lastI*:
‹¬ *is-Add b* ⟹ ¬*is-Mult b* ⟹ ¬*is-Extension b* ⟹ *is-Del b*›
by (*cases b*) *auto*

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*)›
unfolding *PAC-checker-l-step′-def* **by** *auto*

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*›
unfolding *poly-assn-list*
by *auto*

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*›
by (*auto simp*: *IS-LEFT-UNIQUE-def single-valued-def uint64-nat-rel-def br-def*)

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* ⇒ -))^d $\rightarrow_a$
status-assn raw-string-assn $\times_a$ *vars-assn* $\times_a$ *polys-assn*›
supply [[*goals-limit=1*]] *is-Mult-lastI*[*intro*] *single-valued-uint64-nat-rel*[*simp*]
unfolding *PAC-checker-l-step-def PAC-checker-l-step'-def*
pac-step.case-eq-if Let-def
is-success-alt-def[*symmetric*]
uminus-poly-def[*symmetric*]
HOL-list.fold-custom-empty
by *sepref*

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*)›
unfolding *PAC-checker-l'-def* **by** *auto*

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 $\rightarrow_a$
status-assn raw-string-assn $\times_a$ *vars-assn* $\times_a$ *polys-assn*›
supply [[*goals-limit=1*]] *is-Mult-lastI*[*intro*]
unfolding *PAC-checker-l-def is-success-alt-def*[*symmetric*] *PAC-checker-l-step-alt-def*
nres-bind-let-law[*symmetric*] *PAC-checker-l'-def*
apply (*subst nres-bind-let-law*)
by *sepref*

declare *PAC-checker-l-impl.refine*[*sepref-fr-rules*]

abbreviation *polys-assn-input* **where**
‹*polys-assn-input* ≡ *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*)) ∈ *unit-assn*^{k} →$_a$ *raw-string-assn*›
unfolding *remap-polys-l-dom-err-def*
remap-polys-l-dom-err-def
list-assn-pure-conv
by *sepref-to-hoare sep-auto*

MLton is not able to optimise the calls to pow.

lemma *pow-2-64*: ‹(2::*nat*) ^ 64 = 18446744073709551616›
by *auto*

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} →$_a$
status-assn raw-string-assn ×$_a$ *vars-assn* ×$_a$ *polys-assn*›
supply [[*goals-limit=1*]] *is-Mult-lastI*[*intro*] *indom-mI*[*dest*]
unfolding *remap-polys-l2-def op-fmap-empty-def*[*symmetric*] *while-eq-nfoldli*[*symmetric*]
while-upt-while-direct pow-2-64
in-dom-m-lookup-iff
fmlookup'-def[*symmetric*]
union-vars-poly-alt-def[*symmetric*]
apply (*rewrite at* ‹*fmupd* ⌑› *uint64-of-nat-conv-def*[*symmetric*])
apply (*subst while-upt-while-direct*)
apply *simp*
apply (*rewrite at* ‹*op-fmap-empty*› *annotate-assn*[**where** *A*=‹*polys-assn*›])
by *sepref*

lemma *remap-polys-l2-remap-polys-l*:
‹(*uncurry2 remap-polys-l2*, *uncurry2 remap-polys-l*) ∈ (*Id* ×$_r$ ⟨*Id*⟩*set-rel*) ×$_r$ *Id* →$_f$ ⟨*Id*⟩*nres-rel*›
apply (*intro frefI fun-relI nres-relI*)
using *remap-polys-l2-remap-polys-l* **by** *auto*

lemma [*sepref-fr-rules*]:
‹(*uncurry2 remap-polys-l-impl*,
uncurry2 remap-polys-l) ∈ *poly-assn*^{k} *$*_a$ *vars-assn*^{d} *$*_a$ *polys-assn-input*^{d} →$_a$
status-assn raw-string-assn ×$_a$ *vars-assn* ×$_a$ *polys-assn*›
using *hfcomp-tcomp-pre*[*OF remap-polys-l2-remap-polys-l remap-polys-l-impl.refine*]
by (*auto simp*: *hrp-comp-def hfprod-def*)

sepref-register *remap-polys-l*

sepref-definition *full-checker-l-impl*
is ‹*uncurry2 full-checker-l*›
:: ‹*poly-assn*^{k} *$*_a$ *polys-assn-input*^{d} *$*_a$ (*list-assn* (*pac-step-rel-assn* (*uint64-nat-assn*) *poly-assn string-assn*))^k
→$_a$
status-assn raw-string-assn ×$_a$ *vars-assn* ×$_a$ *polys-assn*›
supply [[*goals-limit=1*]] *is-Mult-lastI*[*intro*]
unfolding *full-checker-l-def hs.fold-custom-empty*

union-vars-poly-alt-def[*symmetric*]
PAC-checker-l-alt-def
by *sepref*

sepref-definition *PAC-update-impl*
is ‹*uncurry2* (*RETURN ooo fmupd*)›
:: ‹*nat-assn*^{k} $*_a$ *poly-assn*^{k} $*_a$ (*polys-assn-input*)^d $\rightarrow_a$ *polys-assn-input*›
unfolding *comp-def*
by *sepref*

sepref-definition *PAC-empty-impl*
is ‹*uncurry0* (*RETURN fmempty*)›
:: ‹*unit-assn*^{k} $\rightarrow_a$ *polys-assn-input*›
unfolding *op-iam-fmap-empty-def*[*symmetric*] *pat-fmap-empty*
by *sepref*

sepref-definition *empty-vars-impl*
is ‹*uncurry0* (*RETURN* {})›
:: ‹*unit-assn*^{k} $\rightarrow_a$ *vars-assn*›
unfolding *hs.fold-custom-empty*
by *sepref*

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* :: ‹-› **where**
‹*unsafe-asciis-of-literal xs* = *String.asciis-of-literal xs*›

definition *unsafe-asciis-of-literal′* :: ‹-› **where**
[*simp*, *symmetric*, *code*]: ‹*unsafe-asciis-of-literal′* = *unsafe-asciis-of-literal*›

code-printing
constant *unsafe-asciis-of-literal′* ⇀
(*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*]: ‹*raw-explode* = *String.explode*›
code-printing
constant *raw-explode* ⇀
(*SML*) *String.explode*

definition ‹*hashcode-literal′ s* ≡
foldl (λ*h x*. *h* ∗ *33* + *uint32-of-int* (*of-char x*)) *5381*
(*raw-explode s*)›

definition *uint32-of-char* :: ‹*char* ⇒ *uint32*›
where [*code-abbrev*]: ‹*uint32-of-char x* = *uint32-of-int* (*int-of-char x*)›

code-printing

constant *uint32-of-char* ⇀
(*SML*) !(*Word32.fromInt* ∘ *Char.ord*)

lemma [*code*]: ‹*hashcode s* = *hashcode-literal′ s*›
unfolding *hashcode-literal-def hashcode-list-def*
apply (*auto simp*: *unsafe-asciis-of-literal-def hashcode-list-def*
String.asciis-of-literal-def hashcode-literal-def hashcode-literal′-def)
done

We compile Pastèque in `PAC_Checker_MLton.thy`.

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*

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-poly-rel, *var-rel*⟩*pac-step-rel-raw*))))›

definition *code-status-assn* **where**
‹*code-status-assn* = *hr-comp* (*status-assn raw-string-assn*)
code-status-status-rel›

definition *full-vars-assn* **where**
‹*full-vars-assn* = *hr-comp* (*hs.assn string-assn*)
(⟨*var-rel*⟩*set-rel*)›

lemma *polys-rel-full-polys-rel*:
‹*polys-rel-full* = *Id* ×$_r$ *polys-rel*›
by (*auto simp*: *polys-rel-full-def*)

definition *full-polys-assn* :: ‹-› **where**
‹*full-polys-assn* = *hr-comp* (*hr-comp polys-assn*
(⟨*nat-rel*,
sorted-poly-rel O mset-poly-rel⟩*fmap-rel*))

polys-rel›

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*))
∈ (*full-poly-assn*)^k ∗$_a$ (*full-poly-input-assn*)^d ∗$_a$ (*fully-pac-assn*)^k →$_a$ *hr-comp*
(*code-status-assn* ×$_a$ *full-vars-assn* ×$_a$ *hr-comp polys-assn*
(⟨*nat-rel*, *sorted-poly-rel O mset-poly-rel*⟩*fmap-rel*))
{((*st*, *G*), *st*′, *G*′).
st = *st*′ ∧ (*st* ≠ *FAILED* ⟶ (*G*, *G*′) ∈ *Id* ×$_r$ *polys-rel*)}›
using
full-checker-l-impl.refine[*FCOMP full-checker-l-full-checker*′,
FCOMP full-checker-spec′,
unfolded full-poly-assn-def[*symmetric*]
full-poly-input-assn-def[*symmetric*]
fully-pac-assn-def[*symmetric*]
code-status-assn-def[*symmetric*]
full-vars-assn-def[*symmetric*]
polys-rel-full-polys-rel
hr-comp-prod-conv
full-polys-assn-def[*symmetric*]]
hr-comp-Id2
by *auto*

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* ⇒ *nat*› **where**
‹φ = (*SOME* φ. *bij* φ)›

lemma *bij-φ*: ‹*bij* φ›
using *someI*[*of* ‹$\lambda\varphi$ :: *string* ⇒ *nat*. *bij* φ›]
unfolding *φ-def*[*symmetric*]
using *poly-embed-EX*
by *auto*

global-interpretation *PAC*: *poly-embed* **where**
φ = φ
apply *standard*
apply (*use bij-φ* **in** ‹*auto simp*: *bij-def*›)
done

The full correctness theorem is (*uncurry2 full-checker-l-impl*, *uncurry2* (λ*spec A -. PAC-checker-specification spec A*)) ∈ $PAC.full\text{-}poly\text{-}assn^k *_a PAC.full\text{-}poly\text{-}input\text{-}assn^d *_a PAC.fully\text{-}pac\text{-}assn^k \rightarrow_a$ *hr-comp* (*PAC.code-status-assn* ×$_a$ *PAC.full-vars-assn* ×$_a$ *hr-comp* (*hm-fmap-assn uint64-nat-assn* (*list-assn* (*monom-assn* ×$_a$ *id-assn*))) (⟨*nat-rel*, *sorted-poly-rel O PAC.mset-poly-rel*⟩*fmap-rel*)) {((*st*, *G*), *st*′, *G*′). *st* = *st*′ ∧ (*st* ≠ *FAILED* ⟶ (*G*, *G*′) ∈ *Id* ×$_r$ *polys-rel*)}.

end

theory *PAC-Checker-MLton*
imports *PAC-Checker-Synthesis*
begin

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*

Here is how to compile it:

compile-generated-files -
external-files
‹*code/parser.sml*›
‹*code/pasteque.sml*›
‹*code/pasteque.mlb*›

```
  where ‹fn dir =>
    let
      val exec = Generated-Files.execute (dir + Path.basic code);
      val - =
        exec ‹Compilation›
          (verbatim ‹$ISABELLE-MLTON $ISABELLE-MLTON-OPTIONS › ^
            −const 'MLton.safe false' −verbose 1 −default−type int64 −output pasteque ^
            −codegen native −inline 700 −cc−opt −O3 pasteque.mlb);
    in () end›

end
```

Acknowledgment

This work is supported by Austrian Science Fund (FWF), NFN S11408-N23 (RiSE), and LIT AI Lab funded by the State of Upper Austria.

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.