PAC Checker

Mathias Fleury and Daniela Kaufmann

December 14, 2021

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

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*:
‹$Const_0\ (a + b) = Const_0\ a + Const_0\ b$›
⟨*proof*⟩

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

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

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

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

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

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

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

lemma *degree-uminus*[*simp*]:
‹$degree\ (-A)\ x' = degree\ A\ x'$›
⟨*proof*⟩

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

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

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

lemma *keys-add'*:
  $p \in keys\ (f + g) \Longrightarrow p \in keys\ f \cup keys\ g$
  ⟨*proof*⟩

lemma *keys-mapping-sum-add*:
  $‹finite\ A \Longrightarrow keys\ (mapping\text{-}of\ (\sum v \in A.\ f\ v)) \subseteq \bigcup(keys\ `\ mapping\text{-}of\ `\ f\ `\ UNIV)›$
  ⟨*proof*⟩

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

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

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

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

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

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

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

$\langle proof \rangle$

term ‹$(x', u, lookup\ u\ x', A)$›

lemma *in-mapping-mult-single*:
‹$x \in (\lambda x.\ lookup\ x\ x')$ ‘ $keys\ (A * (Var_0\ x' :: (nat \Rightarrow_0 nat) \Rightarrow_0 'b :: \{monoid\text{-}mult, zero\text{-}neq\text{-}one, semiring\text{-}0\}))$
$\longleftrightarrow$
$x > 0 \wedge x - 1 \in (\lambda x.\ lookup\ x\ x')$ ‘ $keys\ (A)$›
$\langle proof \rangle$

lemma *Max-Suc-Suc-Max*:
‹$finite\ A \Longrightarrow A \neq \{\} \Longrightarrow Max\ (insert\ 0\ (Suc$ ‘ $A)) =$
$Suc\ (Max\ (insert\ 0\ A))$›
$\langle proof \rangle$

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

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

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

lemma *degree-times-le*:
‹$degree\ (A * B)\ x \leq degree\ A\ x + degree\ B\ x$›
$\langle proof \rangle$

lemma *monomial-inj*:
$monomial\ c\ s = monomial\ (d{::}'b{::}zero\text{-}neq\text{-}one)\ t \longleftrightarrow (c = 0 \wedge d = 0) \vee (c = d \wedge s = t)$
$\langle proof \rangle$

lemma *MPoly-monomial-power'*:
‹$MPoly\ (monomial\ 1\ x')\ \hat{}\ (n+1) = MPoly\ (monomial\ (1)\ (((\lambda x.\ x + x')\ \hat{}\hat{}\ n)\ x'))$›
$\langle proof \rangle$

lemma *MPoly-monomial-power*:
‹$n > 0 \Longrightarrow MPoly\ (monomial\ 1\ x')\ \hat{}\ (n) = MPoly\ (monomial\ (1)\ (((\lambda x.\ x + x')\ \hat{}\hat{}\ (n - 1))\ x'))$›
$\langle proof \rangle$

lemma *vars-uminus*[*simp*]:
‹$vars\ (-p) = vars\ p$›
$\langle proof \rangle$

lemma *coeff-uminus*[*simp*]:
‹$MPoly\text{-}Type.coeff\ (-p)\ x = -MPoly\text{-}Type.coeff\ p\ x$›
$\langle proof \rangle$

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

$f\ k)$

lemma *remove-key-lookup*:
$lookup\ (decrease\text{-}key\ k0\ f)\ k = (if\ k = k0 \wedge lookup\ f\ k \neq 0\ then\ lookup\ f\ k - 1\ else\ lookup\ f\ k)$
$\langle proof \rangle$

lemma *polynomial-split-on-var*:
fixes $p :: \langle 'a :: \{comm\text{-}monoid\text{-}add, cancel\text{-}comm\text{-}monoid\text{-}add, semiring\text{-}0, comm\text{-}semiring\text{-}1\}\ mpoly \rangle$
obtains $q\ r$ **where**
$\langle p = monom\ (monomial\ (Suc\ 0)\ x')\ 1 * q + r \rangle$ **and**
$\langle x' \notin vars\ r \rangle$
$\langle proof \rangle$

lemma *polynomial-split-on-var2*:
fixes $p :: \langle int\ mpoly \rangle$
assumes $\langle x' \notin vars\ s \rangle$
obtains $q\ r$ **where**
$\langle p = (monom\ (monomial\ (Suc\ 0)\ x')\ 1 - s) * q + r \rangle$ **and**
$\langle x' \notin vars\ r \rangle$
$\langle proof \rangle$

lemma *finit-whenI*[*intro*]:
$\langle finite\ \ \{x.\ (0 :: nat) < (y\ x)\} \Longrightarrow finite\ \{x.\ 0 < (y\ x\ when\ x \neq x')\} \rangle$
$\langle proof \rangle$

lemma *polynomial-split-on-var-diff-sq2*:
fixes $p :: \langle int\ mpoly \rangle$
obtains $q\ r\ s$ **where**
$\langle p = monom\ (monomial\ (Suc\ 0)\ x')\ 1 * q + r + s * (monom\ (monomial\ (Suc\ 0)\ x')\ 1\hat{}2 - monom\ (monomial\ (Suc\ 0)\ x')\ 1) \rangle$ **and**
$\langle x' \notin vars\ r \rangle$ **and**
$\langle x' \notin vars\ q \rangle$
$\langle proof \rangle$

lemma *polynomial-decomp-alien-var*:
fixes $q\ A\ b :: \langle int\ mpoly \rangle$
assumes
q: $\langle q = A * (monom\ (monomial\ (Suc\ 0)\ x')\ 1) + b \rangle$ **and**
x: $\langle x' \notin vars\ q \rangle\ \langle x' \notin vars\ b \rangle$
shows
$\langle A = 0 \rangle$ **and**
$\langle q = b \rangle$
$\langle proof \rangle$

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**
$\langle q = b \rangle$
$\langle proof \rangle$

lemma *vars-unE*: ‹$x \in$ *vars* $(a * b) \Longrightarrow (x \in$ *vars* $a \Longrightarrow$ *thesis*$) \Longrightarrow (x \in$ *vars* $b \Longrightarrow$ *thesis*$) \Longrightarrow$ *thesis*›
⟨*proof*⟩

lemma *in-keys-minusI1*:
assumes $t \in$ *keys* p **and** $t \notin$ *keys* q
shows $t \in$ *keys* $(p - q)$
⟨*proof*⟩

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

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

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

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

lemma *keys-mult-monomial*:
‹*keys* (*monomial* (n :: *int*) k * *mapping-of* a) = (*if* n = *0* *then* {} *else* ((+) k) ‘ *keys* (*mapping-of* a))›
⟨*proof*⟩

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

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

lemma *Const-1-eq-1*: ‹*Const* (*1* :: *int*) = (*1* :: *int mpoly*)›
⟨*proof*⟩

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

2.2 More Ideals

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

The following theorem is very close to *More-Modules.ideal* (*insert ?a ?S*) = {x. $\exists k$. $x - k$ *

$?a \in$ *More-Modules.ideal* $?S\}$, except that it is more useful if we need to take an element of *More-Modules.ideal* (*insert a S*).

lemma *ideal-insert'*:
‹*More-Modules.ideal* (*insert a S*) $= \{y.\ \exists\, x\ k.\ y = x + k * a \wedge x \in$ *More-Modules.ideal S*$\}$›
⟨*proof*⟩

lemma *ideal-mult-right-in*:
‹$a \in$ *ideal A* $\Longrightarrow a * b \in$ *More-Modules.ideal A*›
⟨*proof*⟩

lemma *ideal-mult-right-in2*:
‹$a \in$ *ideal A* $\Longrightarrow b * a \in$ *More-Modules.ideal A*›
⟨*proof*⟩

lemma [*simp*]: ‹*vars* (*Var x* :: *'a* :: {*zero-neq-one*} *mpoly*) = {*x*}›
⟨*proof*⟩

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* $\mathcal{V}$›
⟨*proof*⟩

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* $\mathcal{V}$›
⟨*proof*⟩

lemma *coeff-monomila-in-varsD*:
‹*coeff p* (*monomial* (*Suc 0*) *x*) $\neq 0 \Longrightarrow x \in$ *vars* (*p* :: *int mpoly*)›
⟨*proof*⟩

lemma *coeff-MPoly-monomial*[*simp*]:
‹(*MPoly-Type.coeff* (*MPoly* (*monomial a m*)) *m*) = *a*›
⟨*proof*⟩

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* ⇒ *'a multiset*› **where**
‹*mset-fset N* ≡ *mset-set* (*fset N*)›

definition *fset-mset* :: ‹*'a multiset* ⇒ *'a fset*› **where**
‹*fset-mset N* ≡ *Abs-fset* (*set-mset N*)›

lemma *fset-mset-mset-fset*: ‹*fset-mset* (*mset-fset N*) = *N*›

⟨*proof*⟩

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

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

lemma *in-fset-mset-mset*[*simp*]: ‹x |∈| *fset-mset* N ⟷ x ∈# N›
⟨*proof*⟩

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)›
⟨*proof*⟩

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

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

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

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

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

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

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

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

lemma *fset-fmdom-fmrestrict-set*:
‹$fset\ (fmdom\ (fmrestrict\text{-}set\ xs\ N)) = fset\ (fmdom\ N) \cap xs$›
⟨*proof*⟩

lemma *dom-m-fmrestrict-set*: ‹$dom\text{-}m\ (fmrestrict\text{-}set\ (set\ xs)\ N) = mset\ xs \cap\#\ dom\text{-}m\ N$›
⟨*proof*⟩

lemma *dom-m-fmrestrict-set'*: ‹$dom\text{-}m\ (fmrestrict\text{-}set\ xs\ N) = mset\text{-}set\ (xs \cap set\text{-}mset\ (dom\text{-}m\ N))$›
⟨*proof*⟩

lemma *indom-mI*: ‹$fmlookup\ m\ x = Some\ y \Longrightarrow x \in\#\ dom\text{-}m\ m$›
⟨*proof*⟩

lemma *fmupd-fmdrop-id*:
assumes ‹$k\ |\in|\ fmdom\ N'$›
shows ‹$fmupd\ k\ (the\ (fmlookup\ N'\ k))\ (fmdrop\ k\ N') = N'$›
⟨*proof*⟩

lemma *fm-member-split*: ‹$k\ |\in|\ fmdom\ N' \Longrightarrow \exists N''\ v.\ N' = fmupd\ k\ v\ N'' \wedge the\ (fmlookup\ N'\ k) = v$
$\wedge$
$k\ |\notin|\ fmdom\ N''$›
⟨*proof*⟩

lemma ‹$fmdrop\ k\ (fmupd\ k\ va\ N'') = fmdrop\ k\ N''$›
⟨*proof*⟩

lemma *fmap-ext-fmdom*:
‹$(fmdom\ N = fmdom\ N') \Longrightarrow (\bigwedge x.\ x\ |\in|\ fmdom\ N \Longrightarrow fmlookup\ N\ x = fmlookup\ N'\ x) \Longrightarrow$
$N = N'$›
⟨*proof*⟩

lemma *fmrestrict-set-insert-in*:
‹$xa\ \in fset\ (fmdom\ N) \Longrightarrow$
$fmrestrict\text{-}set\ (insert\ xa\ l1)\ N = fmupd\ xa\ (the\ (fmlookup\ N\ xa))\ (fmrestrict\text{-}set\ l1\ N)$›
⟨*proof*⟩

lemma *fmrestrict-set-insert-notin*:
‹$xa\ \notin fset\ (fmdom\ N) \Longrightarrow$
$fmrestrict\text{-}set\ (insert\ xa\ l1)\ N = fmrestrict\text{-}set\ l1\ N$›
⟨*proof*⟩

lemma *fmrestrict-set-insert-in-dom-m*[*simp*]:
‹$xa\ \in\#\ dom\text{-}m\ N \Longrightarrow$
$fmrestrict\text{-}set\ (insert\ xa\ l1)\ N = fmupd\ xa\ (the\ (fmlookup\ N\ xa))\ (fmrestrict\text{-}set\ l1\ N)$›
⟨*proof*⟩

lemma *fmrestrict-set-insert-notin-dom-m*[*simp*]:
‹$xa\ \notin\#\ dom\text{-}m\ N \Longrightarrow$
$fmrestrict\text{-}set\ (insert\ xa\ l1)\ N = fmrestrict\text{-}set\ l1\ N$›
⟨*proof*⟩

lemma *fmlookup-restrict-set-id*: ‹$fset\ (fmdom\ N) \subseteq A \Longrightarrow fmrestrict\text{-}set\ A\ N = N$›
⟨*proof*⟩

lemma *fmlookup-restrict-set-id'*: ‹$set\text{-}mset\ (dom\text{-}m\ N) \subseteq A \Longrightarrow fmrestrict\text{-}set\ A\ N = N$›

⟨*proof*⟩

lemma *ran-m-mapsto-upd*:
assumes
NC: ‹$C \in\# \ dom\text{-}m\ N$›
shows ‹$ran\text{-}m\ (fmupd\ C\ C'\ N) = add\text{-}mset\ C'\ (remove1\text{-}mset\ (the\ (fmlookup\ N\ C))\ (ran\text{-}m\ N))$›
⟨*proof*⟩

lemma *ran-m-mapsto-upd-notin*:
assumes *NC*: ‹$C \notin\# \ dom\text{-}m\ N$›
shows ‹$ran\text{-}m\ (fmupd\ C\ C'\ N) = add\text{-}mset\ C'\ (ran\text{-}m\ N)$›
⟨*proof*⟩

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

lemma *filter-mset-cong2*:
$(\bigwedge x.\ x \in\# \ M \Longrightarrow f\ x = g\ x) \Longrightarrow M = N \Longrightarrow filter\text{-}mset\ f\ M = filter\text{-}mset\ g\ N$
⟨*proof*⟩

lemma *ran-m-fmdrop*:
‹$C \in\# \ dom\text{-}m\ N \Longrightarrow \ ran\text{-}m\ (fmdrop\ C\ N) = remove1\text{-}mset\ (the\ (fmlookup\ N\ C))\ (ran\text{-}m\ N)$›
⟨*proof*⟩

lemma *ran-m-fmdrop-notin*:
‹$C \notin\# \ dom\text{-}m\ N \Longrightarrow ran\text{-}m\ (fmdrop\ C\ N) = ran\text{-}m\ N$›
⟨*proof*⟩

lemma *ran-m-fmdrop-If*:
‹$ran\text{-}m\ (fmdrop\ C\ N) = (if\ C \in\# \ dom\text{-}m\ N\ then\ remove1\text{-}mset\ (the\ (fmlookup\ N\ C))\ (ran\text{-}m\ N)\ else\ ran\text{-}m\ N)$›
⟨*proof*⟩

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

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* :: ‹$('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow nat \Rightarrow nat \Rightarrow nat\ nres$› **where**
‹$choose\text{-}pivot\ \text{-}\ \text{-}\ \text{-}\ lo\ hi = SPEC(\lambda k.\ k \geq lo \wedge k \leq hi)$›

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

definition *isPartition-wrt* :: ‹$('b \Rightarrow 'b \Rightarrow bool) \Rightarrow 'b\ list \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow bool$› **where**

‹$isPartition\text{-}wrt\ R\ xs\ lo\ hi\ p \equiv (\forall\ i.\ i \geq lo \wedge i < p \longrightarrow R\ (xs!i)\ (xs!p)) \wedge (\forall\ j.\ j > p \wedge j \leq hi \longrightarrow R\ (xs!p)\ (xs!j))$›

lemma *isPartition-wrtI*:
‹$(\bigwedge\ i.\ [\![i \geq lo;\ i < p]\!] \Longrightarrow R\ (xs!i)\ (xs!p)) \Longrightarrow (\bigwedge\ j.\ [\![j > p;\ j \leq hi]\!] \Longrightarrow R\ (xs!p)\ (xs!j)) \Longrightarrow isPartition\text{-}wrt\ R\ xs\ lo\ hi\ p$›
⟨*proof*⟩

definition *isPartition* :: ‹$'a :: order\ list \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow bool$› **where**
‹$isPartition\ xs\ lo\ hi\ p \equiv isPartition\text{-}wrt\ (\leq)\ xs\ lo\ hi\ p$›

abbreviation *isPartition-map* :: ‹$('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow bool$›
where
‹$isPartition\text{-}map\ R\ h\ xs\ i\ j\ k \equiv isPartition\text{-}wrt\ (\lambda a\ b.\ R\ (h\ a)\ (h\ b))\ xs\ i\ j\ k$›

lemma *isPartition-map-def'*:
‹$lo \leq p \Longrightarrow p \leq hi \Longrightarrow hi < length\ xs \Longrightarrow isPartition\text{-}map\ R\ h\ xs\ lo\ hi\ p = isPartition\text{-}wrt\ R\ (map\ h\ xs)\ lo\ hi\ p$›
⟨*proof*⟩

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

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

definition *sublist* :: ‹$'a\ list \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ list$› **where**
‹$sublist\ xs\ i\ j \equiv take\ (Suc\ j - i)\ (drop\ i\ xs)$›

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

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

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

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

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

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

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

definition *sorted-sublist-wrt* :: ‹$('b \Rightarrow 'b \Rightarrow bool) \Rightarrow 'b\ list \Rightarrow nat \Rightarrow nat \Rightarrow bool$› **where**
‹$sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ hi = sorted\text{-}wrt\ R\ (sublist\ xs\ lo\ hi)$›

definition *sorted-sublist* :: ‹$'a :: linorder\ list \Rightarrow nat \Rightarrow nat \Rightarrow bool$› **where**
‹$sorted\text{-}sublist\ xs\ lo\ hi = sorted\text{-}sublist\text{-}wrt\ (\leq)\ xs\ lo\ hi$›

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

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

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

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

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

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

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

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

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

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

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

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

Elements in a sorted sublists are actually sorted

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

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

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

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

lemma *sorted-sublist-map-le*: ‹*hi* ≤ *lo* ⟹ *hi* < *length xs* ⟹ *sorted-sublist-map R h xs lo hi*›
⟨*proof*⟩

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

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*›
⟨*proof*⟩

lemma *sorted-sublist-wrt-cons*:
assumes *trans*: ‹($\bigwedge$ *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*⟩

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

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

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

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

⟨proof⟩

lemma *sublist-split*: ‹$lo \leq hi \Longrightarrow lo < p \Longrightarrow p < hi \Longrightarrow hi < length\ xs \Longrightarrow sublist\ xs\ lo\ p\ @\ sublist\ xs\ (p{+}1)\ hi = sublist\ xs\ lo\ hi$›
⟨proof⟩

lemma *sublist-split-part*: ‹$lo \leq hi \Longrightarrow lo < p \Longrightarrow p < hi \Longrightarrow hi < length\ xs \Longrightarrow sublist\ xs\ lo\ (p{-}1)\ @\ xs!p\ \#\ sublist\ xs\ (p{+}1)\ hi = sublist\ xs\ lo\ hi$›
⟨proof⟩

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

lemma *isPartition-wrt-trans*:
‹$(\bigwedge x\ y\ z.\ [\![R\ x\ y;\ R\ y\ z]\!] \Longrightarrow R\ x\ z) \Longrightarrow$
$isPartition\text{-}wrt\ R\ xs\ lo\ hi\ p \Longrightarrow$
$(\forall i\ j.\ lo \leq i \wedge i < p \wedge p < j \wedge j \leq hi \longrightarrow R\ (xs!i)\ (xs!j))$›
⟨proof⟩

lemma *isPartition-map-trans*:
‹$(\bigwedge x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)) \Longrightarrow$
$hi < length\ xs \Longrightarrow$
$isPartition\text{-}map\ R\ h\ xs\ lo\ hi\ p \Longrightarrow$
$(\forall i\ j.\ lo \leq i \wedge i < p \wedge p < j \wedge j \leq hi \longrightarrow R\ (h\ (xs!i))\ (h\ (xs!j)))$›
⟨proof⟩

lemma *merge-sorted-wrt-partitions-between′*:
‹$lo \leq hi \Longrightarrow lo < p \Longrightarrow p < hi \Longrightarrow hi < length\ xs \Longrightarrow$
$isPartition\text{-}wrt\ R\ xs\ lo\ hi\ p \Longrightarrow$
$sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ (p{-}1) \Longrightarrow sorted\text{-}sublist\text{-}wrt\ R\ xs\ (p{+}1)\ hi \Longrightarrow$
$(\forall i\ j.\ lo \leq i \wedge i < p \wedge p < j \wedge j \leq hi \longrightarrow R\ (xs!i)\ (xs!j)) \Longrightarrow$
$sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ hi$›
⟨proof⟩

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

The main theorem to merge sorted lists

lemma *merge-sorted-wrt-partitions*:
‹$isPartition\text{-}wrt\ R\ xs\ lo\ hi\ p \Longrightarrow$
$sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ (p - Suc\ 0) \Longrightarrow sorted\text{-}sublist\text{-}wrt\ R\ xs\ (Suc\ p)\ hi \Longrightarrow$
$lo \leq hi \Longrightarrow lo \leq p \Longrightarrow p \leq hi \Longrightarrow hi < length\ xs \Longrightarrow$
$(\forall i\ j.\ lo \leq i \wedge i < p \wedge p < j \wedge j \leq hi \longrightarrow R\ (xs!i)\ (xs!j)) \Longrightarrow$
$sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ hi$›
⟨proof⟩

theorem *merge-sorted-map-partitions*:
‹$(\bigwedge x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)) \Longrightarrow$
$isPartition\text{-}map\ R\ h\ xs\ lo\ hi\ p \Longrightarrow$
$sorted\text{-}sublist\text{-}map\ R\ h\ xs\ lo\ (p - Suc\ 0) \Longrightarrow sorted\text{-}sublist\text{-}map\ R\ h\ xs\ (Suc\ p)\ hi \Longrightarrow$
$lo \leq hi \Longrightarrow lo \leq p \Longrightarrow p \leq hi \Longrightarrow hi < length\ xs \Longrightarrow$

sorted-sublist-map R h xs lo hi›
⟨*proof*⟩

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

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

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

lemma *take-ext*:
‹$(\forall\, i{<}k.\ xs'!i{=}xs!i) \Longrightarrow$
k < *length xs* $\Longrightarrow$ *k* < *length xs′* $\Longrightarrow$
take k xs′ = *take k xs*›
⟨*proof*⟩

lemma *drop-ext′*:
‹$(\forall\, i.\ i{\ge}k \wedge i{<}length\ xs \longrightarrow xs'!i{=}xs!i) \Longrightarrow$
$0{<}k \Longrightarrow xs{\ne}[] \Longrightarrow$ — These corner cases will be dealt with in the next lemma
length xs′=*length xs* $\Longrightarrow$
drop k xs′ = *drop k xs*›
⟨*proof*⟩

lemma *drop-ext*:
‹$(\forall\, i.\ i{\ge}k \wedge i{<}length\ xs \longrightarrow xs'!i{=}xs!i) \Longrightarrow$
length xs′=*length xs* $\Longrightarrow$
drop k xs′ = *drop k xs*›
⟨*proof*⟩

lemma *sublist-ext′*:
‹$(\forall\, i.\ lo{\le}i{\wedge}i{\le}hi \longrightarrow xs'!i{=}xs!i) \Longrightarrow$
length xs′ = *length xs* $\Longrightarrow$

$lo \le hi \Longrightarrow Suc\ hi < length\ xs \Longrightarrow$
$sublist\ xs'\ lo\ hi = sublist\ xs\ lo\ hi$›
⟨*proof*⟩

lemma *lt-Suc*: ‹$(a < b) = (Suc\ a = b \lor Suc\ a < b)$›
⟨*proof*⟩

lemma *sublist-until-end-eq-drop*: ‹$Suc\ hi = length\ xs \Longrightarrow sublist\ xs\ lo\ hi = drop\ lo\ xs$›
⟨*proof*⟩

lemma *sublist-ext*:
‹$(\forall i.\ lo{\le}i{\land}i{\le}hi \longrightarrow xs'!i{=}xs!i) \Longrightarrow$
$length\ xs' = length\ xs \Longrightarrow$
$lo \le hi \Longrightarrow hi < length\ xs \Longrightarrow$
$sublist\ xs'\ lo\ hi = sublist\ xs\ lo\ hi$›
⟨*proof*⟩

lemma *sorted-wrt-lower-sublist-still-sorted*:
assumes ‹$sorted\text{-}sublist\text{-}wrt\ R\ xs\ lo\ (lo' - Suc\ 0)$› **and**
‹$lo \le lo'$› **and** ‹$lo' < length\ xs$› **and**
‹$(\forall\ i.\ lo{\le}i{\land}i{<}lo' \longrightarrow xs'!i{=}xs!i)$› **and** ‹$length\ xs' = length\ xs$›
shows ‹$sorted\text{-}sublist\text{-}wrt\ R\ xs'\ lo\ (lo' - Suc\ 0)$›
⟨*proof*⟩

lemma *sorted-map-lower-sublist-still-sorted*:
assumes ‹$sorted\text{-}sublist\text{-}map\ R\ h\ xs\ lo\ (lo' - Suc\ 0)$› **and**
‹$lo \le lo'$› **and** ‹$lo' < length\ xs$› **and**
‹$(\forall\ i.\ lo{\le}i{\land}i{<}lo' \longrightarrow xs'!i{=}xs!i)$› **and** ‹$length\ xs' = length\ xs$›
shows ‹$sorted\text{-}sublist\text{-}map\ R\ h\ xs'\ lo\ (lo' - Suc\ 0)$›
⟨*proof*⟩

lemma *sorted-wrt-upper-sublist-still-sorted*:
assumes ‹$sorted\text{-}sublist\text{-}wrt\ R\ xs\ (hi'{+}1)\ hi$› **and**
‹$lo \le lo'$› **and** ‹$hi < length\ xs$› **and**
‹$\forall\ j.\ hi'{<}j{\land}j{\le}hi \longrightarrow xs'!j{=}xs!j$› **and** ‹$length\ xs' = length\ xs$›
shows ‹$sorted\text{-}sublist\text{-}wrt\ R\ xs'\ (hi'{+}1)\ hi$›
⟨*proof*⟩

lemma *sorted-map-upper-sublist-still-sorted*:
assumes ‹$sorted\text{-}sublist\text{-}map\ R\ h\ xs\ (hi'{+}1)\ hi$› **and**
‹$lo \le lo'$› **and** ‹$hi < length\ xs$› **and**
‹$\forall\ j.\ hi'{<}j{\land}j{\le}hi \longrightarrow xs'!j{=}xs!j$› **and** ‹$length\ xs' = length\ xs$›
shows ‹$sorted\text{-}sublist\text{-}map\ R\ h\ xs'\ (hi'{+}1)\ hi$›
⟨*proof*⟩

The specification of the partition function

definition *partition-spec* :: ‹$('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ list \Rightarrow nat \Rightarrow bool$› **where**
‹$partition\text{-}spec\ R\ h\ xs\ lo\ hi\ xs'\ p \equiv$
$mset\ xs' = mset\ xs\ \land$ — The list is a permutation
$isPartition\text{-}map\ R\ h\ xs'\ lo\ hi\ p\ \land$ — We have a valid partition on the resulting list
$lo \le p \land p \le hi\ \land$ — The partition index is in bounds
$(\forall\ i.\ i{<}lo \longrightarrow xs'!i{=}xs!i) \land (\forall\ i.\ hi{<}i{\land}i{<}length\ xs' \longrightarrow xs'!i{=}xs!i)$› — Everything else is unchanged.

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

lemma *mset-drop-upto*: ‹$mset\ (drop\ a\ N) = \{\#N!i.\ i \in\#\ mset\text{-}set\ \{a..<length\ N\}\#\}$›
⟨*proof*⟩

lemma *mathias*:
assumes
Perm: ‹$mset\ xs' = mset\ xs$›
and *I*: ‹$lo \le i$› ‹$i \le hi$› ‹$xs'!i = x$›
and *Bounds*: ‹$hi < length\ xs$›
and *Fix*: ‹$\bigwedge\ i.\ i<lo \Longrightarrow xs'!i = xs!i$› ‹$\bigwedge\ j.\ [\![hi<j;\ j<length\ xs]\!] \Longrightarrow xs'!j = xs!j$›
shows ‹$\exists\, j.\ lo \le j \wedge j \le hi \wedge xs!j = x$›
⟨*proof*⟩

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

But we only need that the sets are equal.

lemma *mset-sublist-incl*:
assumes *Perm*: ‹$mset\ xs' = mset\ xs$›
and *Fix*: ‹$\bigwedge\ i.\ i<lo \Longrightarrow xs'!i = xs!i$› ‹$\bigwedge\ j.\ [\![hi<j;\ j<length\ xs]\!] \Longrightarrow xs'!j = xs!j$›
and *bounds*: ‹$lo \le hi$› ‹$hi < length\ xs$›
shows ‹$set\ (sublist\ xs'\ lo\ hi) \subseteq set\ (sublist\ xs\ lo\ hi)$›
⟨*proof*⟩

lemma *mset-sublist-eq*:
assumes ‹$mset\ xs' = mset\ xs$›
and ‹$\bigwedge\ i.\ i<lo \Longrightarrow xs'!i = xs!i$›
and ‹$\bigwedge\ j.\ [\![hi<j;\ j<length\ xs]\!] \Longrightarrow xs'!j = xs!j$›
and *bounds*: ‹$lo \le hi$› ‹$hi < length\ xs$›
shows ‹$set\ (sublist\ xs'\ lo\ hi) = set\ (sublist\ xs\ lo\ hi)$›
⟨*proof*⟩

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

definition *quicksort* :: ‹$('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow nat \times nat \times 'a\ list \Rightarrow 'a\ list\ nres$› **where**
‹*quicksort R h* = (λ(*lo*,*hi*,*xs0*). *do* {
  RECT (λ*f* (*lo*,*hi*,*xs*). *do* {
    ASSERT($lo \le hi \wedge hi < length\ xs \wedge mset\ xs = mset\ xs0$); — 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* $p-1 \le lo$ *then RETURN xs else f* (lo, $p-1$, xs));
    ASSERT($mset\ xs = mset\ xs0$);
    if $hi \le p+1$ *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* :: ‹$('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ list \Rightarrow bool$›
where
‹*quicksort-pre R h xs0 lo hi xs* ≡ $lo \le hi \wedge hi < length\ xs \wedge mset\ xs = mset\ xs0$›

definition *quicksort-post* :: ‹$('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ list \Rightarrow 'a\ list \Rightarrow bool$›
where

‹$\mathit{quicksort\text{-}post}\ R\ h\ lo\ hi\ xs\ xs' \equiv$
$\mathit{mset}\ xs' = \mathit{mset}\ xs\ \wedge$
$\mathit{sorted\text{-}sublist\text{-}map}\ R\ h\ xs'\ lo\ hi\ \wedge$
$(\forall\ i.\ i<lo \longrightarrow xs'!i = xs!i)\ \wedge$
$(\forall\ j.\ hi<j\wedge j<\mathit{length}\ xs \longrightarrow xs'!j = xs!j)$›

Convert Pure to HOL

lemma *quicksort-postI*:
‹$[\![\mathit{mset}\ xs' = \mathit{mset}\ xs;\ \mathit{sorted\text{-}sublist\text{-}map}\ R\ h\ xs'\ lo\ hi;\ (\bigwedge\ i.\ [\![i<lo]\!] \Longrightarrow xs'!i = xs!i);\ (\bigwedge\ j.\ [\![hi<j;\ j<\mathit{length}\ xs]\!] \Longrightarrow xs'!j = xs!j)]\!] \Longrightarrow \mathit{quicksort\text{-}post}\ R\ h\ lo\ hi\ xs\ xs'$›
⟨*proof*⟩

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

lemma *quicksort-correct-case1*:
assumes *trans*: ‹$\bigwedge\ x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)$› **and** *lin*: ‹$\bigwedge x\ y.\ x \neq y \Longrightarrow R\ (h\ x)\ (h\ y) \vee R\ (h\ y)\ (h\ x)$›
and *pre*: ‹$\mathit{quicksort\text{-}pre}\ R\ h\ xs0\ lo\ hi\ xs$›
and *part*: ‹$\mathit{partition\text{-}spec}\ R\ h\ xs\ lo\ hi\ xs'\ p$›
and *ifs*: ‹$p-1 \leq lo$› ‹$hi \leq p+1$›
shows ‹$\mathit{quicksort\text{-}post}\ R\ h\ lo\ hi\ xs\ xs'$›
⟨*proof*⟩

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

lemma *quicksort-correct-case2*:
assumes
pre: ‹$\mathit{quicksort\text{-}pre}\ R\ h\ xs0\ lo\ hi\ xs$›
and *part*: ‹$\mathit{partition\text{-}spec}\ R\ h\ xs\ lo\ hi\ xs'\ p$›
and *ifs*: ‹$\neg\ hi \leq p + 1$›
shows ‹$\mathit{quicksort\text{-}pre}\ R\ h\ xs0\ (\mathit{Suc}\ p)\ hi\ xs'$›
⟨*proof*⟩

lemma *quicksort-post-set*:
assumes ‹$\mathit{quicksort\text{-}post}\ R\ h\ lo\ hi\ xs\ xs'$›
and *bounds*: ‹$lo \leq hi$› ‹$hi < \mathit{length}\ xs$›
shows ‹$\mathit{set}\ (\mathit{sublist}\ xs'\ lo\ hi) = \mathit{set}\ (\mathit{sublist}\ xs\ lo\ hi)$›
⟨*proof*⟩

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

lemma *quicksort-correct-case3*:
assumes *trans*: ‹$\bigwedge\ x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)$› **and** *lin*: ‹$\bigwedge x\ y.\ x \neq y \Longrightarrow R\ (h\ x)\ (h\ y) \vee R\ (h\ y)\ (h\ x)$›
and *pre*: ‹$\mathit{quicksort\text{-}pre}\ R\ h\ xs0\ lo\ hi\ xs$›
and *part*: ‹$\mathit{partition\text{-}spec}\ R\ h\ xs\ lo\ hi\ xs'\ p$›
and *ifs*: ‹$p - \mathit{Suc}\ 0 \leq lo$› ‹$\neg\ hi \leq \mathit{Suc}\ p$›
and *IH1'*: ‹$\mathit{quicksort\text{-}post}\ R\ h\ (\mathit{Suc}\ p)\ hi\ xs'\ xs''$›
shows ‹$\mathit{quicksort\text{-}post}\ R\ h\ lo\ hi\ xs\ xs''$›
⟨*proof*⟩

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

Analogous to case 2.

lemma *quicksort-correct-case4*:
assumes
pre: ‹$quicksort\text{-}pre\ R\ h\ xs0\ lo\ hi\ xs$›
and *part*: ‹$partition\text{-}spec\ R\ h\ xs\ lo\ hi\ xs'\ p$›
and *ifs*: ‹$\neg\ p - Suc\ 0 \le lo$ ›
shows ‹$quicksort\text{-}pre\ R\ h\ xs0\ lo\ (p{-}Suc\ 0)\ xs'$›
⟨*proof*⟩

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

lemma *quicksort-correct-case5*:
assumes *trans*: ‹$\bigwedge\ x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)$› **and** *lin*: ‹$\bigwedge x\ y.\ x \ne y \Longrightarrow R\ (h\ x)\ (h\ y) \lor R\ (h\ y)\ (h\ x)$›
and *pre*: ‹$quicksort\text{-}pre\ R\ h\ xs0\ lo\ hi\ xs$›
and *part*: ‹$partition\text{-}spec\ R\ h\ xs\ lo\ hi\ xs'\ p$›
and *ifs*: ‹$\neg\ p - Suc\ 0 \le lo$› ‹$hi \le Suc\ p$›
and *IH1'*: ‹$quicksort\text{-}post\ R\ h\ lo\ (p - Suc\ 0)\ xs'\ xs''$›
shows ‹$quicksort\text{-}post\ R\ h\ lo\ hi\ xs\ xs''$›
⟨*proof*⟩

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

lemma *quicksort-correct-case6*:
assumes
pre: ‹$quicksort\text{-}pre\ R\ h\ xs0\ lo\ hi\ xs$›
and *part*: ‹$partition\text{-}spec\ R\ h\ xs\ lo\ hi\ xs'\ p$›
and *ifs*: ‹$\neg\ p - Suc\ 0 \le lo$› ‹$\neg\ hi \le Suc\ p$›
and *IH1*: ‹$quicksort\text{-}post\ R\ h\ lo\ (p - Suc\ 0)\ xs'\ xs''$›
shows ‹$quicksort\text{-}pre\ R\ h\ xs0\ (Suc\ p)\ hi\ xs''$›
⟨*proof*⟩

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

lemma *quicksort-correct-case7*:
assumes *trans*: ‹$\bigwedge\ x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)$› **and** *lin*: ‹$\bigwedge x\ y.\ x \ne y \Longrightarrow R\ (h\ x)\ (h\ y) \lor R\ (h\ y)\ (h\ x)$›
and *pre*: ‹$quicksort\text{-}pre\ R\ h\ xs0\ lo\ hi\ xs$›
and *part*: ‹$partition\text{-}spec\ R\ h\ xs\ lo\ hi\ xs'\ p$›
and *ifs*: ‹$\neg\ p - Suc\ 0 \le lo$› ‹$\neg\ hi \le Suc\ p$›
and *IH1'*: ‹$quicksort\text{-}post\ R\ h\ lo\ (p - Suc\ 0)\ xs'\ xs''$›
and *IH2'*: ‹$quicksort\text{-}post\ R\ h\ (Suc\ p)\ hi\ xs''\ xs'''$›
shows ‹$quicksort\text{-}post\ R\ h\ lo\ hi\ xs\ xs'''$›
⟨*proof*⟩

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

lemma *quicksort-correct*:
assumes *trans*: ‹$\bigwedge\ x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)$› **and** *lin*: ‹$\bigwedge x\ y.\ x \ne y \Longrightarrow R\ (h\ x)\ (h\ y) \lor R\ (h\ y)\ (h\ x)$›
and *Pre*: ‹$lo0 \le hi0$› ‹$hi0 < length\ xs0$›
shows ‹$quicksort\ R\ h\ (lo0,hi0,xs0) \le \Downarrow Id\ (SPEC(\lambda xs.\ quicksort\text{-}post\ R\ h\ lo0\ hi0\ xs0\ xs))$›
⟨*proof*⟩

definition *partition-main-inv* :: ‹$('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ list \Rightarrow (nat \times nat \times 'a\ list) \Rightarrow bool$› **where**
‹*partition-main-inv* $R\ h\ lo\ hi\ xs0\ p \equiv$
$case\ p\ of\ (i,j,xs) \Rightarrow$
$j < length\ xs \wedge j \leq hi \wedge i < length\ xs \wedge lo \leq i \wedge i \leq j \wedge mset\ xs = mset\ xs0 \wedge$
$(\forall k.\ k \geq lo \wedge k < i \longrightarrow R\ (h\ (xs!k))\ (h\ (xs!hi))) \wedge$ — All elements from lo to $i - (1::'c)$ are smaller than the pivot
$(\forall k.\ k \geq i \wedge k < j \longrightarrow R\ (h\ (xs!hi))\ (h\ (xs!k))) \wedge$ — All elements from i to $j - (1::'c)$ are greater than the pivot
$(\forall k.\ k < lo \longrightarrow xs!k = xs0!k) \wedge$ — Everything below lo is unchanged
$(\forall k.\ k \geq j \wedge k < length\ xs \longrightarrow xs!k = xs0!k)$ — 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* :: ‹$('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ list \Rightarrow ('a\ list \times nat)\ nres$› **where**
‹*partition-main* $R\ h\ lo\ hi\ xs0 = do\ \{$
$ASSERT(hi < length\ xs0);$
$pivot \leftarrow RETURN\ (h\ (xs0\ !\ hi));$
$(i,j,xs) \leftarrow WHILE_T{}^{partition\text{-}main\text{-}inv\ R\ h\ lo\ hi\ xs0}$ — We loop from $j = lo$ to $j = hi - (1::'c)$.
$(\lambda(i,j,xs).\ j < hi)$
$(\lambda(i,j,xs).\ do\ \{$
$ASSERT(i < length\ xs \wedge j < length\ xs);$
$if\ R\ (h\ (xs!j))\ pivot$
$then\ RETURN\ (i+1,\ j+1,\ swap\ xs\ i\ j)$
$else\ RETURN\ (i,\ \ j+1,\ xs)$
$\})$
$(lo,\ lo,\ xs0);$ — i and j are both initialized to lo
$ASSERT(i < length\ xs \wedge j = hi \wedge lo \leq i \wedge hi < length\ xs \wedge mset\ xs = mset\ xs0);$
$RETURN\ (swap\ xs\ i\ hi,\ i)$
$\}$›

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

definition *partition-between* :: ‹$('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ list \Rightarrow ('a\ list \times nat)\ nres$› **where**
‹*partition-between* $R\ h\ lo\ hi\ xs0 = do\ \{$
$ASSERT(hi < length\ xs0 \wedge lo \leq hi);$
$k \leftarrow choose\text{-}pivot\ R\ h\ xs0\ lo\ hi;$ — choice of pivot

ASSERT(*k* < *length xs0*);
xs ← *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*⟩

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*)›
⟨*proof*⟩

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*)›

⟨*proof*⟩

lemma *Down-id-eq*:
‹$\Downarrow$*Id x* = *x*›
⟨*proof*⟩

lemma *partition-between-ref-partition-between*:
‹*partition-between-ref R h lo hi xs* $\leq$ (*partition-between R h lo hi xs*)›
⟨*proof*⟩

Technical lemma for sepref

lemma *partition-between-ref-partition-between'*:
‹(*uncurry2* (*partition-between-ref R h*), *uncurry2* (*partition-between R h*)) $\in$
(*nat-rel* $\times_r$ *nat-rel*) $\times_r$ ⟨*Id*⟩*list-rel* $\to_f$ ⟨⟨*Id*⟩*list-rel* $\times_r$ *nat-rel*⟩*nres-rel*›
⟨*proof*⟩

Example instantiation for pivot

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

lemma *partition-between-ref-correct*:
assumes *trans*: ‹$\bigwedge$ *x y z*. $[\![$*R* (*h x*) (*h y*); *R* (*h y*) (*h z*)$]\!]$ $\Longrightarrow$ *R* (*h x*) (*h z*)› **and** *lin*: ‹$\bigwedge$*x y*. *R* (*h x*) (*h y*) $\vee$ *R* (*h y*) (*h x*)›
and *bounds*: ‹*hi* < *length xs*› ‹*lo* $\leq$ *hi*›
shows ‹*partition-between-ref R h lo hi xs* $\leq$ *SPEC* (*uncurry* (*partition-spec R h xs lo hi*))›
⟨*proof*⟩

Refined quicksort algorithm: We use the refined partition function.

definition *quicksort-ref* :: ‹- $\Rightarrow$ - $\Rightarrow$ *nat* $\times$ *nat* $\times$ *'a list* $\Rightarrow$ *'a list nres*› **where**
‹*quicksort-ref R h* = (λ(*lo*,*hi*,*xs0*).
do {
RECT (λ*f* (*lo*,*hi*,*xs*). *do* {
ASSERT(*lo* $\leq$ *hi* $\wedge$ *hi* < *length xs0* $\wedge$ *mset xs* = *mset xs0*);
(*xs*, *p*) $\leftarrow$ *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* $\wedge$ *p* $\geq$ *lo* $\wedge$ *p* < *length xs0*);
xs $\leftarrow$ (*if p*−*1*$\leq$*lo then RETURN xs else f* (*lo*, *p*−*1*, *xs*));
ASSERT(*mset xs* = *mset xs0*);
if hi$\leq$*p*+*1 then RETURN xs else f* (*p*+*1*, *hi*, *xs*)
}) (*lo*,*hi*,*xs0*)
})›

lemma *fref-to-Down-curry2*:
‹(*uncurry2 f*, *uncurry2 g*) $\in$ $[P]_f$ *A* $\to$ ⟨*B*⟩*nres-rel* $\Longrightarrow$
($\bigwedge$*x x' y y' z z'*. *P* ((*x'*, *y'*), *z'*) $\Longrightarrow$ (((*x*, *y*), *z*), ((*x'*, *y'*), *z'*)) $\in$ *A*$\Longrightarrow$
f x y z $\leq$ $\Downarrow$ *B* (*g x' y' z'*))›
⟨*proof*⟩

lemma *fref-to-Down-curry*:
‹(*f*, *g*) $\in$ $[P]_f$ *A* $\to$ ⟨*B*⟩*nres-rel* $\Longrightarrow$

$(\bigwedge x\ x'\ .\ P\ x' \Longrightarrow (x,\ x') \in A \Longrightarrow$
$f\ x\ \ \le \Downarrow B\ (g\ x'))$›
⟨*proof*⟩

lemma *quicksort-ref-quicksort*:
assumes *bounds*: ‹$hi < length\ xs$› ‹$lo \le hi$› **and**
trans: ‹$\bigwedge\ x\ y\ z.\ [\![R\ (h\ x)\ (h\ y);\ R\ (h\ y)\ (h\ z)]\!] \Longrightarrow R\ (h\ x)\ (h\ z)$› **and** *lin*: ‹$\bigwedge x\ y.\ R\ (h\ x)\ (h\ y) \vee R\ (h\ y)\ (h\ x)$›
shows ‹$quicksort\text{-}ref\ R\ h\ x0 \le \Downarrow Id\ (quicksort\ R\ h\ x0)$›
⟨*proof*⟩
definition *full-quicksort* **where**
‹$full\text{-}quicksort\ R\ h\ xs \equiv if\ xs = []\ then\ RETURN\ xs\ else\ quicksort\ R\ h\ (0,\ length\ xs - 1,\ xs)$›

definition *full-quicksort-ref* **where**
‹$full\text{-}quicksort\text{-}ref\ R\ h\ xs \equiv$
$if\ List.null\ xs\ then\ RETURN\ xs$
$else\ quicksort\text{-}ref\ R\ h\ (0,\ length\ xs - 1,\ xs)$›

definition *full-quicksort-impl* :: ‹$nat\ list \Rightarrow nat\ list\ nres$› **where**
‹$full\text{-}quicksort\text{-}impl\ xs = full\text{-}quicksort\text{-}ref\ (\le)\ id\ xs$›

lemma *full-quicksort-ref-full-quicksort*:
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 ‹$(full\text{-}quicksort\text{-}ref\ R\ h,\ full\text{-}quicksort\ R\ h) \in$
$\langle Id\rangle list\text{-}rel \rightarrow_f \langle\ \langle Id\rangle list\text{-}rel\rangle nres\text{-}rel$›
⟨*proof*⟩

lemma *sublist-entire*:
‹$sublist\ xs\ 0\ (length\ xs - 1) = xs$›
⟨*proof*⟩

lemma *sorted-sublist-wrt-entire*:
assumes ‹$sorted\text{-}sublist\text{-}wrt\ R\ xs\ 0\ (length\ xs - 1)$›
shows ‹$sorted\text{-}wrt\ R\ xs$›
⟨*proof*⟩

lemma *sorted-sublist-map-entire*:
assumes ‹$sorted\text{-}sublist\text{-}map\ R\ h\ xs\ 0\ (length\ xs - 1)$›
shows ‹$sorted\text{-}wrt\ (\lambda\ x\ y.\ R\ (h\ x)\ (h\ y))\ xs$›
⟨*proof*⟩

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*: ‹$\bigwedge x\ y.\ x \ne y \Longrightarrow R\ (h\ x)\ (h\ y) \vee R\ (h\ y)\ (h\ x)$›
shows ‹$full\text{-}quicksort\ R\ h\ xs \le \Downarrow Id\ (SPEC(\lambda xs'.\ mset\ xs' = mset\ xs \wedge sorted\text{-}wrt\ (\lambda\ x\ y.\ R\ (h\ x)\ (h\ y))\ xs'))$›
⟨*proof*⟩

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*))›
  ⟨*proof*⟩

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*:
  ‹$\Downarrow$*Id x* = *x*›
  ⟨*proof*⟩

lemma *while-upt-while-direct1*:
  $b \geq a \Longrightarrow$
  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) ← $WHILE_T$ ($\lambda(i, x)$. $i < b \wedge c\ x$) ($\lambda(i, x)$. *do* {*ASSERT* ($i < b$); σ'←*f i x*; *RETURN* (i+*1*,σ') }) (*a*,σ);
    RETURN σ
  }
  ⟨*proof*⟩

lemma *while-upt-while-direct2*:
  $b \geq a \Longrightarrow$
  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) ← $WHILE_T$ ($\lambda(i, x)$. $i < b \wedge c\ x$) ($\lambda(i, x)$. *do* {*ASSERT* ($i < b$); σ'←*f i x*; *RETURN* (i+*1*,σ') }) (*a*,σ);
    RETURN σ
  }
  ⟨*proof*⟩

lemma *while-upt-while-direct*:
  $b \geq a \Longrightarrow$
  do {

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

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

lemma *while-eq-nfoldli*: *do* {
(-,σ) ← *WHILE*$_T$ (*FOREACH-cond c*) (λx. *do* {*ASSERT* (*FOREACH-cond c x*); *FOREACH-body f x*}) (*l*,σ);
RETURN σ
} = *nfoldli l c f* σ
⟨*proof*⟩

end

theory *PAC-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* = (λc. *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*›
⟨*proof*⟩

lemma *pac-idealI1*[*intro*]:

‹$p \in A \Longrightarrow p \in$ *pac-ideal* A›
⟨*proof*⟩

lemma *pac-idealI2*[*intro*]:
‹$p \in$ *ideal* $A \Longrightarrow p \in$ *pac-ideal* A›
⟨*proof*⟩

lemma *pac-idealI3*[*intro*]:
‹$p \in$ *ideal* $A \Longrightarrow p*q \in$ *pac-ideal* A›
⟨*proof*⟩

lemma *pac-ideal-Xsq2-iff*:
‹*Var* c ^ $2 \in$ *pac-ideal* $A \longleftrightarrow$ *Var* $c \in$ *pac-ideal* A›
⟨*proof*⟩

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

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›
⟨*proof*⟩

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

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

definition *restricted-ideal-to* **where**
‹*restricted-ideal-to* B $A = \{p \in A.$ *vars* $p \subseteq B\}$›

abbreviation *restricted-ideal-to*$_I$ **where**
‹*restricted-ideal-to*$_I$ B $A \equiv$ *restricted-ideal-to* B (*pac-ideal* (*set-mset* A))›

abbreviation *restricted-ideal-to*$_V$ **where**
‹*restricted-ideal-to*$_V$ $B \equiv$ *restricted-ideal-to* $(\bigcup($*vars* ‘ *set-mset* $B))$›

abbreviation *restricted-ideal-to*$_{V\,I}$ **where**
‹*restricted-ideal-to*$_{V\,I}$ B $A \equiv$ *restricted-ideal-to* $(\bigcup($*vars* ‘ *set-mset* $B))$ (*pac-ideal* (*set-mset* A))›

lemma *restricted-idealI*:
‹$p \in$ *pac-ideal* (*set-mset* A) $\Longrightarrow$ *vars* $p \subseteq C \Longrightarrow p \in$ *restricted-ideal-to*$_I$ C A›
⟨*proof*⟩

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

lemma *pac-ideal-add*:

‹$p \in\# A \Longrightarrow q \in\# A \Longrightarrow p + q \in$ *pac-ideal* (*set-mset* A)›
⟨*proof*⟩
lemma *pac-ideal-mult*:
‹$p \in\# A \Longrightarrow p * q \in$ *pac-ideal* (*set-mset* A)›
⟨*proof*⟩

lemma *pac-ideal-mono*:
‹$A \subseteq B \Longrightarrow$ *pac-ideal* $A \subseteq$ *pac-ideal* B›
⟨*proof*⟩

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
‹$p \in\# A$› ‹$q \in\# A$›
‹$p+q - p' \in$ *ideal polynomial-bool*›
‹*vars* $p' \subseteq \mathcal{V}$› |
mult:
‹*PAC-Format* ($\mathcal{V}$, A) ($\mathcal{V}$, *add-mset* p' A)›
if
‹$p \in\# A$›
‹$p*q - p' \in$ *ideal polynomial-bool*›
‹*vars* $p' \subseteq \mathcal{V}$›
‹*vars* $q \subseteq \mathcal{V}$› |
del:
‹$p \in\# A \Longrightarrow$ *PAC-Format* ($\mathcal{V}$, A) ($\mathcal{V}$, $A - \{\#p\#\}$)› |
extend-pos:
‹*PAC-Format* ($\mathcal{V}$, A) ($\mathcal{V} \cup \{x' \in$ *vars* ($-$*Var* $x + p'$). $x' \notin \mathcal{V}\}$, *add-mset* ($-$*Var* $x + p'$) A)›
if
‹$(p')^2 - p' \in$ *ideal polynomial-bool*›
‹*vars* $p' \subseteq \mathcal{V}$›
‹$x \notin \mathcal{V}$›

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:
‹$\bigwedge p\ q\ p'\ A\ \mathcal{V}.\ p \in\#\ A \Longrightarrow q \in\#\ A \Longrightarrow p{+}q - p' \in$ *ideal polynomial-bool* $\Longrightarrow$ *vars* $p' \subseteq \mathcal{V} \Longrightarrow P\ \mathcal{V}\ A\ \mathcal{V}$ (*add-mset* p' *A*)›
‹$\bigwedge p\ q\ p'\ A\ \mathcal{V}.\ p \in\#\ A \Longrightarrow p{*}q - p' \in$ *ideal polynomial-bool* $\Longrightarrow$ *vars* $p' \subseteq \mathcal{V} \Longrightarrow$ *vars* $q \subseteq \mathcal{V} \Longrightarrow P\ \mathcal{V}\ A\ \mathcal{V}$ (*add-mset* p' *A*)›
‹$\bigwedge p\ A\ \mathcal{V}.\ p \in\#\ A \Longrightarrow P\ \mathcal{V}\ A\ \mathcal{V}\ (A - \{\#p\#\})$›
‹$\bigwedge p'\ x\ r.$
$(p')\hat{\ }2 - (p') \in$ *ideal polynomial-bool* $\Longrightarrow$ *vars* $p' \subseteq \mathcal{V} \Longrightarrow$
$x \notin \mathcal{V} \Longrightarrow P\ \mathcal{V}\ A\ (\mathcal{V} \cup \{x' \in$ *vars* $(p' -$ *Var* $x).\ x' \notin \mathcal{V}\})$ (*add-mset* (p' − *Var* x) *A*)›
shows
‹$P\ \mathcal{V}\ A\ \mathcal{V}'\ A'$›
⟨*proof*⟩

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

lemma *extensions-are-safe*:
assumes ‹$x' \in$ *vars* p› **and**
x': ‹$x' \notin \mathcal{V}$› **and**
‹$\bigcup$ (*vars* ‘ *set-mset* A) $\subseteq \mathcal{V}$› **and**
p-x-coeff: ‹*coeff* p (*monomial* (*Suc 0*) x') = *1*› **and**
vars-q: ‹*vars* $q \subseteq \mathcal{V}$› **and**
q: ‹$q \in$ *More-Modules.ideal* (*insert* p (*set-mset* A ∪ *polynomial-bool*))› **and**
leading: ‹$x' \notin$ *vars* (p − *Var* x')› **and**
diff: ‹(*Var* x' − p)^2 − (*Var* x' − p) ∈ *More-Modules.ideal polynomial-bool*›
shows
‹$q \in$ *More-Modules.ideal* (*set-mset* A ∪ *polynomial-bool*)›
⟨*proof*⟩

lemma *extensions-are-safe-uminus*:
assumes ‹$x' \in$ *vars* p› **and**
x': ‹$x' \notin \mathcal{V}$› **and**
‹$\bigcup$ (*vars* ‘ *set-mset* A) $\subseteq \mathcal{V}$› **and**
p-x-coeff: ‹*coeff* p (*monomial* (*Suc 0*) x') = −*1*› **and**
vars-q: ‹*vars* $q \subseteq \mathcal{V}$› **and**
q: ‹$q \in$ *More-Modules.ideal* (*insert* p (*set-mset* A ∪ *polynomial-bool*))› **and**
leading: ‹$x' \notin$ *vars* (p + *Var* x')› **and**
diff: ‹(*Var* x' + p)^*2* − (*Var* x' + p) ∈ *More-Modules.ideal polynomial-bool*›
shows
‹$q \in$ *More-Modules.ideal* (*set-mset* A ∪ *polynomial-bool*)›
⟨*proof*⟩

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

lemma *vars-subst-in-left-only*:
‹$x \notin$ *vars* $p \Longrightarrow x \in$ *vars* (p − *Var* x)› **for** p :: ‹*int mpoly*›
⟨*proof*⟩

lemma *vars-subst-in-left-only-diff-iff*:
fixes p :: ‹*int mpoly*›
assumes ‹$x \notin$ *vars* p›
shows ‹*vars* (p − *Var* x) = *insert* x (*vars* p)›
⟨*proof*⟩

lemma *vars-subst-in-left-only-iff*:
‹$x \notin$ *vars* $p \Longrightarrow$ *vars* $(p + Var\ x)$ = *insert* x (*vars* p)› **for** p :: ‹*int mpoly*›
⟨*proof*⟩

lemma *coeff-add-right-notin*:
‹$x \notin$ *vars* $p \Longrightarrow$ *MPoly-Type.coeff* $(Var\ x - p)$ (*monomial* (*Suc 0*) x) = *1*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*))›
⟨*proof*⟩

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) $\subseteq \mathcal{V}'$›
⟨*proof*⟩

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

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

lemma *rtranclp-PAC-Format-subset-ideal*:
‹*rtranclp PAC-Format* $(\mathcal{V}, A)$ $(\mathcal{V}', B) \Longrightarrow \bigcup$(*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) $\subseteq \mathcal{V}'$›
⟨*proof*⟩

end

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

5 Hash-Map for finite mappings

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

definition *fmap-rel* **where**
[*to-relAPP*]:
fmap-rel K V $\equiv$ {(*m1*, *m2*).
($\forall$ *i j*. *i* |∈| *fmdom m2* $\longrightarrow$ $(j, i) \in K \longrightarrow$ (*the* (*fmlookup m1 j*), *the* (*fmlookup m2 i*)) $\in V$) $\wedge$
fset (*fmdom m1*) $\subseteq$ *Domain* K $\wedge$ *fset* (*fmdom m2*) $\subseteq$ *Range* K $\wedge$
($\forall$ *i j*. $(i, j) \in K \longrightarrow$ *j* |∈| *fmdom m2* $\longleftrightarrow$ *i* |∈| *fmdom m1*)}

lemma *fmap-rel-alt-def*:

‹$\langle K,\ V\rangle$*fmap-rel* ≡
{(*m1*, *m2*).
($\forall$ *i j*. *i* ∈# *dom-m m2* ⟶
(*j*, *i*) ∈ *K* ⟶ (*the* (*fmlookup m1 j*), *the* (*fmlookup m2 i*)) ∈ *V*) ∧
fset (*fmdom m1*) ⊆ *Domain K* ∧
fset (*fmdom m2*) ⊆ *Range K* ∧
($\forall$ *i j*. (*i*, *j*) ∈ *K* ⟶ (*j* ∈# *dom-m m2*) = (*i* ∈# *dom-m m1*))}
›
⟨*proof*⟩

lemma *fmdom-empty-fmempty-iff*[*simp*]: ‹*fmdom m* = {||} ⟷ *m* = *fmempty*›
⟨*proof*⟩

lemma *fmap-rel-empty1-simp*[*simp*]:
(*fmempty*,*m*)∈$\langle K,V\rangle$*fmap-rel* ⟷ *m*=*fmempty*
⟨*proof*⟩

lemma *fmap-rel-empty2-simp*[*simp*]:
(*m*,*fmempty*)∈$\langle K,V\rangle$*fmap-rel* ⟷ *m*=*fmempty*
⟨*proof*⟩

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* ($\langle K,V\rangle$*fmap-rel*) *TYPE*((′*k*,′*v*) *f-map*) ⟨*proof*⟩

5.1 Operations

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

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

lemma *fmap-rel-fmupd-fmap-rel*:
‹(*A*, *B*) ∈ $\langle K,\ R\rangle$*fmap-rel* ⟹ (*p*, *p*′) ∈ *K* ⟹ (*q*, *q*′) ∈ *R* ⟹
(*fmupd p q A*, *fmupd p*′ *q*′ *B*) ∈ $\langle K,\ R\rangle$*fmap-rel*›
if *single-valued K single-valued* (K^{-1})
⟨*proof*⟩

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

lemma *remove1-mset-eq-add-mset-iff*:
‹*remove1-mset a A* = *add-mset a A*′ ⟷ *A* = *add-mset a* (*add-mset a A*′)›
⟨*proof*⟩

lemma *fmap-rel-fmdrop-fmap-rel*:
‹(*fmdrop p A*, *fmdrop p*′ *B*) ∈ $\langle K,\ R\rangle$*fmap-rel*›
if *single*: *single-valued K single-valued* (K^{-1}) **and**
H0: ‹(*A*, *B*) ∈ $\langle K,\ R\rangle$*fmap-rel*› ‹(*p*, *p*′) ∈ *K*›
⟨*proof*⟩

sepref-decl-op *fmap-delete*: *fmdrop* :: *K* → $\langle K,V\rangle$*fmap-rel* → $\langle K,V\rangle$*fmap-rel*

where *single-valued K single-valued* (K^{-1})
⟨*proof*⟩

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

lemma *fmap-rel-in-dom-iff*:
‹$(aa, a'a) \in \langle K, V\rangle$*fmap-rel* $\Longrightarrow$
$(a, a') \in K \Longrightarrow$
$a' \in\#$ *dom-m* $a'a \longleftrightarrow$
$a \in\#$ *dom-m* aa›
⟨*proof*⟩

lemma *fmap-rel-fmlookup-rel*:
‹$(a, a') \in K \Longrightarrow (aa, a'a) \in \langle K, V\rangle$*fmap-rel* $\Longrightarrow$
(*fmlookup* aa a, *fmlookup* $a'a$ a') $\in \langle V\rangle$*option-rel*›
⟨*proof*⟩

sepref-decl-op *fmap-lookup*: *fmlookup* :: $\langle K,V\rangle$*fmap-rel* $\to K \to \langle V\rangle$*option-rel*
⟨*proof*⟩

lemma *in-fdom-alt*: $k\in\#$*dom-m* $m \longleftrightarrow \neg$*is-None* (*fmlookup* m k)
⟨*proof*⟩

sepref-decl-op *fmap-contains-key*: $\lambda k\ m.\ k\in\#$*dom-m* m :: $K \to \langle K,V\rangle$*fmap-rel* $\to$ *bool-rel*
⟨*proof*⟩

5.2 Patterns

lemma *pat-fmap-empty*[*pat-rules*]: *fmempty* $\equiv$ *op-fmap-empty* ⟨*proof*⟩

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

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

5.3 Mapping to Normal Hashmaps

abbreviation *map-of-fmap* :: ‹($'k \Rightarrow$ $'v$ *option*) $\Rightarrow$ ($'k$, $'v$) *fmap*› **where**
‹*map-of-fmap* h $\equiv$ *Abs-fmap* h›

definition *map-fmap-rel* **where**
‹*map-fmap-rel* = *br map-of-fmap* ($\lambda a.$ *finite* (*dom* a))›

lemma *fmdrop-set-None*:
‹(*op-map-delete*, *fmdrop*) $\in Id \to$ *map-fmap-rel* $\to$ *map-fmap-rel*›
⟨*proof*⟩

lemma *map-upd-fmupd*:
‹$(op\text{-}map\text{-}update, fmupd) \in Id \to Id \to map\text{-}fmap\text{-}rel \to map\text{-}fmap\text{-}rel$›
⟨*proof*⟩

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

definition *fmlookup'* **where**
[*simp*]: ‹$fmlookup'\ A\ k = fmlookup\ k\ A$›

lemma [*def-pat-rules*]:
‹$((\in\#)\$k\$(dom\text{-}m\$A)) \equiv Not\$(is\text{-}None\$(fmlookup'\$k\$A))$›
⟨*proof*⟩

lemma *op-map-lookup-fmlookup*:
‹$(op\text{-}map\text{-}lookup, fmlookup') \in Id \to map\text{-}fmap\text{-}rel \to \langle Id\rangle option\text{-}rel$›
⟨*proof*⟩

abbreviation *hm-fmap-assn* **where**
‹$hm\text{-}fmap\text{-}assn\ K\ V \equiv hr\text{-}comp\ (hm.assn\ K\ V)\ map\text{-}fmap\text{-}rel$›

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*:
‹$(uncurry0\ (RETURN\ op\text{-}map\text{-}empty), uncurry0\ (RETURN\ fmempty)) \in unit\text{-}rel \to_f \langle map\text{-}fmap\text{-}rel\rangle nres\text{-}rel$›
⟨*proof*⟩

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

abbreviation *iam-fmap-assn* **where**
‹$iam\text{-}fmap\text{-}assn\ K\ V \equiv hr\text{-}comp\ (iam.assn\ K\ V)\ map\text{-}fmap\text{-}rel$›

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**
‹$op\text{-}iam\text{-}fmap\text{-}empty = fmempty$›

lemma *iam-fmempty-empty*:
‹$(uncurry0\ (RETURN\ op\text{-}map\text{-}empty),\ uncurry0\ (RETURN\ op\text{-}iam\text{-}fmap\text{-}empty)) \in unit\text{-}rel \rightarrow_f \langle map\text{-}fmap\text{-}rel\rangle nres\text{-}rel$›
⟨*proof*⟩

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

definition *upper-bound-on-dom* **where**
‹$upper\text{-}bound\text{-}on\text{-}dom\ A = SPEC(\lambda n.\ \forall i \in\#(dom\text{-}m\ A).\ i < n)$›

lemma [*sepref-fr-rules*]:
‹$((Array.len),\ upper\text{-}bound\text{-}on\text{-}dom) \in (iam\text{-}fmap\text{-}assn\ nat\text{-}assn\ V)^k \rightarrow_a nat\text{-}assn$›
⟨*proof*⟩

lemma *fmap-rel-nat-rel-dom-m*[*simp*]:
‹$(A,\ B) \in \langle nat\text{-}rel,\ R\rangle fmap\text{-}rel \Longrightarrow dom\text{-}m\ A = dom\text{-}m\ B$›
⟨*proof*⟩

lemma *ref-two-step'*:
‹$A \leq B \Longrightarrow\ \Downarrow R\ A \leq \Downarrow R\ B$›
⟨*proof*⟩

end

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

6 Checker Algorithm

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

6.1 Specification

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

lemma *is-success-alt-def*:
‹$is\text{-}success\ a \longleftrightarrow a = SUCCESS$›
⟨*proof*⟩

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

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

definition *PAC-checker-specification*
:: ‹*int-poly* ⇒ *int-poly multiset* ⇒ (*status* × *nat set* × *int-poly multiset*) *nres*›
where
‹*PAC-checker-specification spec A* = *SPEC*(λ(*b*, $\mathcal{V}$, *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($\mathcal{V}$, *A*) (*b*, *B*). (¬*is-failed b* ⟶ ⋃(*vars* ‘ *set-mset A*) ⊆ $\mathcal{V}$) ∧
(*is-success b* ⟶ *PAC-Format*** ($\mathcal{V}$, *A*) *B*) ∧
(*is-found b* ⟶ *PAC-Format*** ($\mathcal{V}$, *A*) *B* ∧ *spec* ∈ *pac-ideal* (*set-mset A*)))›

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

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

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

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

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

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

definition *check-add* :: ‹(*nat*, *int mpoly*) *fmap* ⇒ *nat set* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *int mpoly* ⇒ *bool nres*› **where**
‹*check-add A* $\mathcal{V}$ *p q i r* =
SPEC(λ*b*. *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(λ*b*. *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(λ*b*. *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(λ*b*. *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* = (λ*spec* (*stat*, ($\mathcal{V}$, A)) *st*. *case st of*
  Add - - - - ⇒
    do {
      r ← *normalize-poly-spec* (*pac-res st*);
      eq ← *check-add* A $\mathcal{V}$ (*pac-src1 st*) (*pac-src2 st*) (*new-id st*) r;
      st′ ← *SPEC*(λ*st*′. (¬*is-failed st*′ ∧ *is-found st*′ ⟶ r − *spec* ∈ *ideal polynomial-bool*));
      if eq
      then RETURN (*merge-status stat st*′,
        $\mathcal{V}$, *fmupd* (*new-id st*) r A)
      else RETURN (*FAILED*, ($\mathcal{V}$, A))
  }
  | *Del* - ⇒
    do {
      eq ← *check-del* A (*pac-src1 st*);
      if eq
      then RETURN (*stat*, ($\mathcal{V}$, *fmdrop* (*pac-src1 st*) A))
      else RETURN (*FAILED*, ($\mathcal{V}$, A))
  }
  | *Mult* - - - - ⇒
    do {
      r ← *normalize-poly-spec* (*pac-res st*);

$q \leftarrow$ *normalize-poly-spec* (*pac-mult st*);
eq $\leftarrow$ *check-mult A* $\mathcal{V}$ (*pac-src1 st*) *q* (*new-id st*) *r*;
$st' \leftarrow SPEC(\lambda st'.\ (\neg$*is-failed st'* $\wedge$ *is-found st'* $\longrightarrow r - spec \in$ *ideal polynomial-bool*));
if eq
then RETURN (*merge-status stat st'*,
$\mathcal{V}$, *fmupd* (*new-id st*) *r A*)
else RETURN (*FAILED*, ($\mathcal{V}$, *A*))
}
| *Extension* - - - $\Rightarrow$
do {
r $\leftarrow$ *normalize-poly-spec* (*pac-res st* $-$ *Var* (*new-var st*));
(*eq*) $\leftarrow$ *check-extension A* $\mathcal{V}$ (*new-id st*) (*new-var st*) *r*;
if eq
then do {
RETURN (*stat*,
insert (*new-var st*) $\mathcal{V}$, *fmupd* (*new-id st*) (*r*) *A*)}
else RETURN (*FAILED*, ($\mathcal{V}$, *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* = {(($\mathcal{V}$, *A*), ($\mathcal{V}'$, *B*)). (*A*, *B*) ∈ *polys-rel* ∧ $\mathcal{V} = \mathcal{V}'$}›

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

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

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

‹*r* $-$ *x14* $\in$ *More-Modules.ideal polynomial-bool* $\Longrightarrow$
(*A*, *B*) $\in$ *polys-rel* $\Longrightarrow$
the (*fmlookup A x11*) + *the* (*fmlookup A x12*) $-$ *r* $\in$ *More-Modules.ideal polynomial-bool* $\Longrightarrow$
x11 $\in\#$ *dom-m A* $\Longrightarrow$
x12 $\in\#$ *dom-m A* $\Longrightarrow$
vars r $\subseteq \mathcal{V}$ $\Longrightarrow$
PAC-Format** ($\mathcal{V}$, *B*) ($\mathcal{V}$, *add-mset r B*)›
‹*r* $-$ *x14* $\in$ *More-Modules.ideal polynomial-bool* $\Longrightarrow$
(*A*, *B*) $\in$ *polys-rel* $\Longrightarrow$
x11 $\in\#$ *dom-m A* $\Longrightarrow$
x12 $\in\#$ *dom-m A* $\Longrightarrow$
the (*fmlookup A x11*) + *the* (*fmlookup A x12*) $-$ *r* $\in$ *More-Modules.ideal polynomial-bool* $\Longrightarrow$
vars r $\subseteq \mathcal{V}$ $\Longrightarrow$
x11 $\neq$ *x12* $\Longrightarrow$
PAC-Format** ($\mathcal{V}$, *B*)
($\mathcal{V}$, *add-mset r B* $-$ {#*the* (*fmlookup A x11*), *the* (*fmlookup A x12*)#})›
‹(*A*, *B*) $\in$ *polys-rel* $\Longrightarrow$
r $-$ *x34* $\in$ *More-Modules.ideal polynomial-bool* $\Longrightarrow$
x11 $\in\#$ *dom-m A* $\Longrightarrow$
the (*fmlookup A x11*) $*$ *x32* $-$ *r* $\in$ *More-Modules.ideal polynomial-bool* $\Longrightarrow$
vars x32 $\subseteq \mathcal{V}$ $\Longrightarrow$
vars r $\subseteq \mathcal{V}$ $\Longrightarrow$
PAC-Format** ($\mathcal{V}$, *B*) ($\mathcal{V}$, *add-mset r B*)›
‹(*A*, *B*) $\in$ *polys-rel* $\Longrightarrow$
r $-$ *x34* $\in$ *More-Modules.ideal polynomial-bool* $\Longrightarrow$
x11 $\in\#$ *dom-m A* $\Longrightarrow$
the (*fmlookup A x11*) $*$ *x32* $-$ *r* $\in$ *More-Modules.ideal polynomial-bool* $\Longrightarrow$
vars x32 $\subseteq \mathcal{V}$ $\Longrightarrow$
vars r $\subseteq \mathcal{V}$ $\Longrightarrow$
PAC-Format** ($\mathcal{V}$, *B*) ($\mathcal{V}$, *remove1-mset* (*the* (*fmlookup A x11*)) (*add-mset r B*))›
‹(*A*, *B*) $\in$ *polys-rel* $\Longrightarrow$
x12 $\in\#$ *dom-m A* $\Longrightarrow$
PAC-Format** ($\mathcal{V}$, *B*) ($\mathcal{V}$, *remove1-mset* (*the* (*fmlookup A x12*)) *B*)›
‹(*A*, *B*) $\in$ *polys-rel* $\Longrightarrow$
$(p' + Var\ x)^2 - (p' + Var\ x) \in$ *ideal polynomial-bool* $\Longrightarrow$
$x \notin \mathcal{V}$ $\Longrightarrow$
$x \notin vars(p' + Var\ x)$ $\Longrightarrow$
$vars(p' + Var\ x) \subseteq \mathcal{V}$ $\Longrightarrow$
PAC-Format** ($\mathcal{V}$, *B*)
(*insert x* $\mathcal{V}$, *add-mset* p' *B*)›
⟨*proof*⟩

abbreviation *status-rel* :: ‹(*status* $\times$ *status*) *set*› **where**
‹*status-rel* $\equiv$ *Id*›

lemma *is-merge-status*[*simp*]:
‹*is-failed* (*merge-status a st*$'$) $\longleftrightarrow$ *is-failed a* $\vee$ *is-failed st*$'$›
‹*is-found* (*merge-status a st*$'$) $\longleftrightarrow$ $\neg$*is-failed a* $\wedge$ $\neg$*is-failed st*$'$ $\wedge$ (*is-found a* $\vee$ *is-found st*$'$)›
‹*is-success* (*merge-status a st*$'$) $\longleftrightarrow$ (*is-success a* $\wedge$ *is-success st*$'$)›
⟨*proof*⟩

lemma *status-rel-merge-status*:
‹(*merge-status a b*, *SUCCESS*) $\notin$ *status-rel* $\longleftrightarrow$
(*a* = *FAILED*) $\vee$ (*b* = *FAILED*) $\vee$

$a = FOUND \lor (b = FOUND)$›
⟨*proof*⟩

lemma *Ex-status-iff*:
‹$(\exists a.\ P\ a) \longleftrightarrow P\ SUCCESS \lor P\ FOUND \lor (P\ (FAILED))$›
⟨*proof*⟩

lemma *is-failed-alt-def*:
‹$is\text{-}failed\ st' \longleftrightarrow \neg is\text{-}success\ st' \land \neg is\text{-}found\ st'$›
⟨*proof*⟩

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

lemma *fmdrop-irrelevant*: ‹$x11 \notin\#\ dom\text{-}m\ A \Longrightarrow fmdrop\ x11\ A = A$›
⟨*proof*⟩

lemma *PAC-checker-step-PAC-checker-specification2*:
fixes a :: ‹*status*›
assumes AB: ‹$((\mathcal{V}, A),(\mathcal{V}_B, B)) \in polys\text{-}rel\text{-}full$› **and**
‹$\neg is\text{-}failed\ a$› **and**
[*simp*,*intro*]: ‹$a = FOUND \Longrightarrow spec \in pac\text{-}ideal\ (set\text{-}mset\ A_0)$› **and**
A_0B: ‹$PAC\text{-}Format^{**}\ (\mathcal{V}_0, A_0)\ (\mathcal{V}, B)$› **and**
$spec_0$: ‹$vars\ spec \subseteq \mathcal{V}_0$› **and**
$vars\text{-}A_0$: ‹$\bigcup\ (vars\ `\ set\text{-}mset\ A_0) \subseteq \mathcal{V}_0$›
shows ‹$PAC\text{-}checker\text{-}step\ spec\ (a, (\mathcal{V}, A))\ st \leq \Downarrow (status\text{-}rel \times_r polys\text{-}rel\text{-}full)\ (PAC\text{-}checker\text{-}specification\text{-}step2\ (\mathcal{V}_0, A_0)\ spec\ (\mathcal{V}, B))$›
⟨*proof*⟩

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

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

lemma *RES-SPEC-eq*:
‹*RES* Φ = *SPEC*(λP. $P \in \Phi$)›
⟨*proof*⟩

lemma *is-failed-is-success-completeD*:
‹$\neg$ *is-failed x* $\Longrightarrow$ $\neg$*is-success x* $\Longrightarrow$ *is-found x*›
⟨*proof*⟩

lemma *PAC-checker-PAC-checker-specification2*:
‹(*A*, *B*) $\in$ *polys-rel-full* $\Longrightarrow$
$\neg$*is-failed a* $\Longrightarrow$
(*a* = *FOUND* $\Longrightarrow$ *spec* $\in$ *pac-ideal* (*set-mset* (*snd B*))) $\Longrightarrow$
$\bigcup$(*vars* ‘ *set-mset* (*ran-m* (*snd A*))) $\subseteq$ *fst B* $\Longrightarrow$
vars spec $\subseteq$ *fst B* $\Longrightarrow$
PAC-checker spec A a st $\le$ $\Downarrow$ (*status-rel* $\times_r$ *polys-rel-full*) (*PAC-checker-specification2 spec B*)›
⟨*proof*⟩

definition *remap-polys-polynomial-bool* :: ‹*int mpoly* $\Rightarrow$ *nat set* $\Rightarrow$ (*nat*, *int-poly*) *fmap* $\Rightarrow$ (*status* $\times$ *fpac-step*) *nres*› **where**
‹*remap-polys-polynomial-bool spec* = ($\lambda\mathcal{V}$ *A*.
SPEC(λ(*st*, $\mathcal{V}'$, *A'*). ($\neg$*is-failed st* $\longrightarrow$
dom-m A = *dom-m A'* $\wedge$
($\forall$ *i* $\in\#$ *dom-m A*. *the* (*fmlookup A i*) $-$ *the* (*fmlookup A' i*) $\in$ *ideal polynomial-bool*) $\wedge$
$\bigcup$(*vars* ‘ *set-mset* (*ran-m A*)) $\subseteq \mathcal{V}'$ $\wedge$
$\bigcup$(*vars* ‘ *set-mset* (*ran-m A'*)) $\subseteq \mathcal{V}'$) $\wedge$
(*st* = *FOUND* $\longrightarrow$ *spec* $\in\#$ *ran-m A'*)))›

definition *remap-polys-change-all* :: ‹*int mpoly* $\Rightarrow$ *nat set* $\Rightarrow$ (*nat*, *int-poly*) *fmap* $\Rightarrow$ (*status* $\times$ *fpac-step*) *nres*› **where**
‹*remap-polys-change-all spec* = ($\lambda\mathcal{V}$ *A*. *SPEC* (λ(*st*, $\mathcal{V}'$, *A'*).
($\neg$*is-failed st* $\longrightarrow$
pac-ideal (*set-mset* (*ran-m A*)) = *pac-ideal* (*set-mset* (*ran-m A'*)) $\wedge$
$\bigcup$(*vars* ‘ *set-mset* (*ran-m A*)) $\subseteq \mathcal{V}'$ $\wedge$
$\bigcup$(*vars* ‘ *set-mset* (*ran-m A'*)) $\subseteq \mathcal{V}'$) $\wedge$
(*st* = *FOUND* $\longrightarrow$ *spec* $\in\#$ *ran-m A'*)))›

lemma *fmap-eq-dom-iff*:
‹*A* = *A'* $\longleftrightarrow$ *dom-m A* = *dom-m A'* $\wedge$ ($\forall$ *i* $\in\#$ *dom-m A*. *the* (*fmlookup A i*) = *the* (*fmlookup A' i*))›
⟨*proof*⟩

lemma *ideal-remap-incl*:
‹*finite A'* $\Longrightarrow$ ($\forall a' \in A'$. $\exists a \in A$. $a - a' \in B$) $\Longrightarrow$ *ideal* ($A' \cup B$) $\subseteq$ *ideal* ($A \cup B$)›
⟨*proof*⟩

lemma *pac-ideal-remap-eq*:
‹*dom-m b* = *dom-m ba* $\Longrightarrow$
$\forall$ *i*$\in\#$*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*))›
⟨*proof*⟩

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*›
⟨*proof*⟩

definition *remap-polys* :: ‹*int mpoly* ⇒ *nat set* ⇒ (*nat*, *int-poly*) *fmap* ⇒ (*status* × *fpac-step*) *nres*›
where
‹*remap-polys 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 {
RETURN (*FAILED*, $\mathcal{V}$, *fmempty*)
}
else do {
(*b*, *N*) ← *FOREACH dom*
(λ*i* (*b*, $\mathcal{V}$, *A'*).
if *i* ∈# *dom-m* *A*
then do {
p ← *SPEC*(λ*p*. *the* (*fmlookup* *A* *i*) − *p* ∈ *ideal polynomial-bool* ∧ *vars* *p* ⊆ *vars* (*the* (*fmlookup* *A* *i*)));
eq ← *SPEC*(λ*eq*. *eq* ⟶ *p* = *spec*);
$\mathcal{V}$ ← *SPEC*(λ$\mathcal{V}'$. $\mathcal{V}$ ∪ *vars* (*the* (*fmlookup* *A* *i*)) ⊆ $\mathcal{V}'$);
RETURN(*b* ∨ *eq*, $\mathcal{V}$, *fmupd* *i* *p* *A'*)
} *else RETURN* (*b*, $\mathcal{V}$, *A'*))
(*False*, $\mathcal{V}$, *fmempty*);
RETURN (*if b then FOUND else SUCCESS*, *N*)
}
})›

lemma *remap-polys-spec*:
‹*remap-polys spec* $\mathcal{V}$ *A* ≤ *remap-polys-polynomial-bool spec* $\mathcal{V}$ *A*›
⟨*proof*⟩

6.3 Full Checker

definition *full-checker*
:: ‹*int-poly* ⇒ (*nat*, *int-poly*) *fmap* ⇒ (*int-poly*, *nat*,*nat*) *pac-step list* ⇒ (*status* × -) *nres*›
where
‹*full-checker spec0 A pac* = *do* {
spec ← *normalize-poly-spec spec0*;
(*st*, $\mathcal{V}$, *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 $\subseteq$ *restricted-ideal-to*$_I$ $\mathcal{V}'$ J $\Longrightarrow$
  $\mathcal{U} \subseteq \mathcal{V} \Longrightarrow$
  restricted-ideal-to$_I$ $\mathcal{U}$ I $\subseteq$ *restricted-ideal-to*$_I$ $\mathcal{U}$ J›
  ⟨*proof*⟩

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

lemma *full-checker-spec*:
  assumes ‹$(A,\ A') \in$ *polys-rel*›
  shows
    ‹*full-checker spec A pac* $\le \Downarrow\{((st,\ G),\ (st',\ G')).\ (st,\ st') \in$ *status-rel* $\wedge$
      $(st \neq$ *FAILED* $\longrightarrow (G,\ G') \in$ *polys-rel-full*$)\}$
    (*PAC-checker-specification spec* (A'))›
⟨*proof*⟩

lemma *full-checker-spec'*:
  shows
   ‹(*uncurry2 full-checker*, *uncurry2* (λ*spec A -. PAC-checker-specification spec A*)) $\in$
    (*Id* $\times_r$ *polys-rel*) $\times_r$ *Id* $\rightarrow_f$ ⟨$\{((st,\ G),\ (st',\ G')).\ (st,\ st') \in$ *status-rel* $\wedge$
      $(st \neq$ *FAILED* $\longrightarrow (G,\ G') \in$ *polys-rel-full*$)\}$⟩*nres-rel*›
  ⟨*proof*⟩

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

7 Polynomials of strings

Isabelle's definition of polynomials only work with variables of type *nat*. Therefore, we introduce a version that uses strings 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*:
  ‹$\exists\,\varphi$. *bij* (φ :: *string* $\Rightarrow$ *nat*)›
  ⟨*proof*⟩

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

type-synonym *term-poly* = ‹*string multiset*›
type-synonym *mset-polynomial* =
  ‹(*term-poly* $*$ *int*) *multiset*›

definition *normalized-poly* :: ‹*mset-polynomial* $\Rightarrow$ *bool*› **where**

‹*normalized-poly p* ⟷
distinct-mset (*fst* ‘# *p*) ∧
0 ∉# *snd* ‘# *p*›

lemma *normalized-poly-simps*[*simp*]:
‹*normalized-poly* {#}›
‹*normalized-poly* (*add-mset t p*) ⟷ *snd t* ≠ *0* ∧
fst t ∉# *fst* ‘# *p* ∧ *normalized-poly p*›
⟨*proof*⟩

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

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

definition *mult-poly-raw* :: ‹*mset-polynomial* ⇒ *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* = $\sum_{\#}$ (*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*#}))#})›
⟨*proof*⟩

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

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

7.2 Addition

inductive *add-poly-p* :: ‹*mset-polynomial* × *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*:
‹$add\text{-}poly\text{-}p^{**}$ (*p*, *q*, *r*) ({#}, *q*, *p* + *r*)›
⟨*proof*⟩

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

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

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

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

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

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

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

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

7.3 Normalisation

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

7.4 Correctness

This locales maps string polynomials to real polynomials.

locale *poly-embed* =
  fixes *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*⟩

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

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

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

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

⟨proof⟩

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

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

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

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

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

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

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

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

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

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

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

lemma *rtranclp-add-poly-p-polynomial-of-mset-full*:

‹*add-poly-p*** $(p, q, \{\#\})$ $(\{\#\}, \{\#\}, r')$ $\Longrightarrow$
polynomial-of-mset r' = (*polynomial-of-mset* p + *polynomial-of-mset* q)›
⟨*proof*⟩

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

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

lemma *mult-poly-p-mult-ideal*:
‹*mult-poly-p* q (p, r) (p', r') $\Longrightarrow$
(*polynomial-of-mset* p' ∗ *polynomial-of-mset* q + *polynomial-of-mset* r') − (*polynomial-of-mset* p ∗ *polynomial-of-mset* q + *polynomial-of-mset* r)
∈ *ideal polynomial-bool*›
⟨*proof*⟩

lemma *rtranclp-mult-poly-p-mult-ideal*:
‹(*mult-poly-p* q)** (p, r) (p', r') $\Longrightarrow$
(*polynomial-of-mset* p' ∗ *polynomial-of-mset* q + *polynomial-of-mset* r') − (*polynomial-of-mset* p ∗ *polynomial-of-mset* q + *polynomial-of-mset* r)
∈ *ideal polynomial-bool*›
⟨*proof*⟩

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

lemma *normalize-poly-p-poly-of-mset*:
‹*normalize-poly-p* p q $\Longrightarrow$ *polynomial-of-mset* p = *polynomial-of-mset* q›
⟨*proof*⟩

lemma *rtranclp-normalize-poly-p-poly-of-mset*:
‹*normalize-poly-p*** p q $\Longrightarrow$ *polynomial-of-mset* p = *polynomial-of-mset* q›
⟨*proof*⟩

end

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

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

definition φ' :: ‹*nat* ⇒ *string*› **where**
‹φ' = *the-inv-into V* φ›

lemma φ'-φ[*simp*]:
‹$x \in V \Longrightarrow \varphi'\ (\varphi\ x) = x$›
⟨*proof*⟩

lemma φ-φ'[*simp*]:
‹$x \in N \Longrightarrow \varphi\ (\varphi'\ x) = x$›
⟨*proof*⟩

end

end

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

8 Terms

We define some helper functions.

8.1 Ordering

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

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

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

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

definition *less-eq-char* :: ‹*char* ⇒ *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*

⟨*proof*⟩

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*›
⟨*proof*⟩

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

8.2 Polynomials

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

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

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

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

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

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

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

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

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

abbreviation *unsorted-poly-rel* **where**
‹*unsorted-poly-rel* ≡ *poly-list-rel term-poly-list-rel*›

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

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

abbreviation *fully-unsorted-poly-rel* **where**
‹*fully-unsorted-poly-rel* ≡ *fully-unsorted-poly-list-rel unsorted-term-poly-list-rel*›

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

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

abbreviation *unsorted-poly-rel-with0* **where**
‹*unsorted-poly-rel-with0* ≡ *fully-unsorted-poly-list-rel term-poly-list-rel*›

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

definition *sorted-repeat-poly-list-rel-with0-wrt* :: ‹($'a \Rightarrow 'a \Rightarrow$ *bool*)
$\Rightarrow$ ($'a \times$ *string multiset*) *set* $\Rightarrow$ (($'a \times$ *int*) *list* $\times$ *mset-polynomial*) *set*› **where**

‹*sorted-repeat-poly-list-rel-with0-wrt S R* = {(*xs*, *ys*).
  (*xs*, *ys*) ∈ ⟨*R* $\times_r$ *int-rel*⟩*list-rel O list-mset-rel* ∧
  sorted-wrt S (*map fst xs*)}›

abbreviation *sorted-repeat-poly-list-rel-with0* **where**
‹*sorted-repeat-poly-list-rel-with0 R* ≡ *sorted-repeat-poly-list-rel-with0-wrt R term-poly-list-rel*›

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

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

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

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
  (λ*add-poly-l* (*p*, *q*).
    case (*p*,*q*) *of*
      (*p*, []) ⇒ *RETURN p*
    | ([], *q*) ⇒ *RETURN q*
    | ((*xs*, *n*) # *p*, (*ys*, *m*) # *q*) ⇒
      (*if xs* = *ys then if n* + *m* = *0 then add-poly-l* (*p*, *q*) *else*
        do {

        pq ← *add-poly-l* (*p*, *q*);
        RETURN ((*xs*, *n* + *m*) # *pq*)
    }
    else if (*xs*, *ys*) ∈ *term-order-rel*
      then do {
        pq ← *add-poly-l* (*p*, (*ys*, *m*) # *q*);
        RETURN ((*xs*, *n*) # *pq*)
    }
    else do {
        pq ← *add-poly-l* ((*xs*, *n*) # *p*, *q*);
        RETURN ((*ys*, *m*) # *pq*)
    }))›

definition *nonzero-coeffs* **where**
‹*nonzero-coeffs a* ⟷ *0* ∉# *snd* ‘# *a*›

lemma *nonzero-coeffs-simps*[*simp*]:
‹*nonzero-coeffs* {#}›
‹*nonzero-coeffs* (*add-mset* (*xs*, *n*) *a*) ⟷ *nonzero-coeffs a* ∧ *n* ≠ *0*›
⟨*proof*⟩

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

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

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

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

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

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

lemma *monoms-add-poly-l′D*:
‹$(aa,\ ba) \in$ *set* (*add-poly-l′* x) $\Longrightarrow aa \in$ *fst* ‘ *set* (*fst* x) $\vee\ aa \in$ *fst* ‘ *set* (*snd* x)›
⟨*proof*⟩

lemma *add-poly-p-add-to-result*:
‹*add-poly-p*** $(A,\ B,\ r)\ (A',\ B',\ r') \Longrightarrow$
add-poly-p**
$(A,\ B,\ p + r)\ (A',\ B',\ p + r')$›
⟨*proof*⟩

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

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

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

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

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

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

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

$(ys, xs) \in lexord\ (lexord\ less\text{-}than\text{-}char)$›
⟨*proof*⟩

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

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

lemma *add-poly-l-add-poly*:
‹$add\text{-}poly\text{-}l\ x = RETURN\ (add\text{-}poly\text{-}l'\ x)$›
⟨*proof*⟩

lemma *add-poly-l-spec*:
‹$(add\text{-}poly\text{-}l, uncurry\ (\lambda p\ q.\ SPEC(\lambda r.\ add\text{-}poly\text{-}p^{**}\ (p, q, \{\#\})\ (\{\#\}, \{\#\}, r)))) \in sorted\text{-}poly\text{-}rel \times_r sorted\text{-}poly\text{-}rel \rightarrow_f \langle sorted\text{-}poly\text{-}rel\rangle nres\text{-}rel$›
⟨*proof*⟩

definition *sort-poly-spec* :: ‹$llist\text{-}polynomial \Rightarrow llist\text{-}polynomial\ nres$› **where**
‹$sort\text{-}poly\text{-}spec\ p = SPEC(\lambda p'.\ mset\ p = mset\ p' \wedge sorted\text{-}wrt\ (rel2p\ (Id \cup term\text{-}order\text{-}rel))\ (map\ fst\ p'))$›

lemma *sort-poly-spec-id*:
assumes ‹$(p, p') \in unsorted\text{-}poly\text{-}rel$›
shows ‹$sort\text{-}poly\text{-}spec\ p \le \Downarrow (sorted\text{-}repeat\text{-}poly\text{-}rel)\ (RETURN\ p')$›
⟨*proof*⟩

8.4 Multiplication

fun *mult-monoms* :: ‹$term\text{-}poly\text{-}list \Rightarrow term\text{-}poly\text{-}list \Rightarrow term\text{-}poly\text{-}list$› **where**
‹$mult\text{-}monoms\ p\ [] = p$› |
‹$mult\text{-}monoms\ []\ p = p$› |
‹$mult\text{-}monoms\ (x \# p)\ (y \# q) =$
$(if\ x = y\ then\ x \# mult\text{-}monoms\ p\ q$
$else\ if\ (x, y) \in var\text{-}order\text{-}rel\ then\ x \# mult\text{-}monoms\ p\ (y \# q)$
$else\ y \# mult\text{-}monoms\ (x \# p)\ q)$›

lemma *term-poly-list-rel-empty-iff* [*simp*]:
‹$([], q') \in term\text{-}poly\text{-}list\text{-}rel \longleftrightarrow q' = \{\#\}$›
⟨*proof*⟩

lemma *mset-eqD-set-mset*: ‹$mset\ xs = A \Longrightarrow set\ xs = set\text{-}mset\ A$›
⟨*proof*⟩

lemma *term-poly-list-rel-Cons-iff*:
‹$(y \# p, p') \in term\text{-}poly\text{-}list\text{-}rel \longleftrightarrow$
$(p, remove1\text{-}mset\ y\ p') \in term\text{-}poly\text{-}list\text{-}rel \wedge$

$y \in\# p' \wedge y \notin set\ p \wedge y \notin\#\ remove1\text{-}mset\ y\ p' \wedge$
$(\forall x{\in}\#mset\ p.\ (y,\ x) \in var\text{-}order\text{-}rel)$›
⟨proof⟩

lemma *var-order-rel-antisym*[*simp*]:
‹$(y,\ y) \notin var\text{-}order\text{-}rel$›
⟨proof⟩

lemma *term-poly-list-rel-remdups-mset*:
‹$(p,\ p') \in term\text{-}poly\text{-}list\text{-}rel \Longrightarrow$
$(p,\ remdups\text{-}mset\ p') \in term\text{-}poly\text{-}list\text{-}rel$›
⟨proof⟩

lemma *var-notin-notin-mult-monomsD*:
‹$y \in set\ (mult\text{-}monoms\ p\ q) \Longrightarrow y \in set\ p \vee y \in set\ q$›
⟨proof⟩

lemma *term-poly-list-rel-set-mset*:
‹$(p,\ q) \in term\text{-}poly\text{-}list\text{-}rel \Longrightarrow set\ p = set\text{-}mset\ q$›
⟨proof⟩

lemma *mult-monoms-spec*:
‹$(mult\text{-}monoms,\ (\lambda a\ b.\ remdups\text{-}mset\ (a + b))) \in term\text{-}poly\text{-}list\text{-}rel \to term\text{-}poly\text{-}list\text{-}rel \to term\text{-}poly\text{-}list\text{-}rel$›
⟨proof⟩

definition *mult-monomials* :: ‹$term\text{-}poly\text{-}list \times int \Rightarrow term\text{-}poly\text{-}list \times int \Rightarrow term\text{-}poly\text{-}list \times int$›
where
‹$mult\text{-}monomials = (\lambda(x,\ a)\ (y,\ b).\ (mult\text{-}monoms\ x\ y,\ a * b))$›

definition *mult-poly-raw* :: ‹$llist\text{-}polynomial \Rightarrow llist\text{-}polynomial \Rightarrow llist\text{-}polynomial$› **where**
‹$mult\text{-}poly\text{-}raw\ p\ q = foldl\ (\lambda b\ x.\ map\ (mult\text{-}monomials\ x)\ q\ @\ b)\ []\ p$›

fun *map-append* **where**
‹$map\text{-}append\ f\ b\ [] = b$› |
‹$map\text{-}append\ f\ b\ (x\ \#\ xs) = f\ x\ \#\ map\text{-}append\ f\ b\ xs$›

lemma *map-append-alt-def*:
‹$map\text{-}append\ f\ b\ xs = map\ f\ xs\ @\ b$›
⟨proof⟩

lemma *foldl-append-empty*:
‹$NO\text{-}MATCH\ []\ xs \Longrightarrow foldl\ (\lambda b\ x.\ f\ x\ @\ b)\ xs\ p = foldl\ (\lambda b\ x.\ f\ x\ @\ b)\ []\ p\ @\ xs$›
⟨proof⟩

lemma *poly-list-rel-empty-iff*[*simp*]:
‹$([],\ r) \in poly\text{-}list\text{-}rel\ R \longleftrightarrow r = \{\#\}$›
⟨proof⟩

lemma *mult-poly-raw-simp*[*simp*]:
‹$mult\text{-}poly\text{-}raw\ []\ q = []$›
‹$mult\text{-}poly\text{-}raw\ (x\ \#\ p)\ q = mult\text{-}poly\text{-}raw\ p\ q\ @\ map\ (mult\text{-}monomials\ x)\ q$›
⟨proof⟩

lemma *sorted-poly-list-relD*:
‹$(q, q') \in$ *sorted-poly-list-rel* $R \Longrightarrow q' = (\lambda(a, b).\ (mset\ a, b))$ ‘# *mset q*›
⟨*proof*⟩

lemma *list-all2-in-set-ExD*:
‹*list-all2 R p q* $\Longrightarrow x \in$ *set p* $\Longrightarrow \exists\, y \in$ *set q*. *R x y*›
⟨*proof*⟩

inductive-cases *mult-poly-p-elim*: ‹*mult-poly-p q* $(A, r)\ (B, r')$›

lemma *mult-poly-p-add-mset-same*:
‹$(mult\text{-}poly\text{-}p\ q')^{**}\ (A, r)\ (B, r') \Longrightarrow (mult\text{-}poly\text{-}p\ q')^{**}\ (add\text{-}mset\ x\ A, r)\ (add\text{-}mset\ x\ B, r')$›
⟨*proof*⟩

lemma *mult-poly-raw-mult-poly-p*:
assumes ‹$(p, p') \in$ *sorted-poly-rel*› **and** ‹$(q, q') \in$ *sorted-poly-rel*›
shows ‹$\exists\, r.\ (mult\text{-}poly\text{-}raw\ p\ q, r) \in unsorted\text{-}poly\text{-}rel \wedge (mult\text{-}poly\text{-}p\ q')^{**}\ (p', \{\#\})\ (\{\#\}, r)$›
⟨*proof*⟩

fun *merge-coeffs* :: ‹*llist-polynomial* ⇒ *llist-polynomial*› **where**
‹*merge-coeffs* [] = []› |
‹*merge-coeffs* [(*xs*, *n*)] = [(*xs*, *n*)]› |
‹*merge-coeffs* ((*xs*, *n*) # (*ys*, *m*) # *p*) =
(*if xs* = *ys*
then if $n + m \neq 0$ *then merge-coeffs* ((*xs*, *n* + *m*) # *p*) *else merge-coeffs p*
else (*xs*, *n*) # *merge-coeffs* ((*ys*, *m*) # *p*))›

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

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

lemma *fst-normalize-polynomial-subsetD*:
‹$(a, b) \in$ *set* (*merge-coeffs p*) $\Longrightarrow a \in$ *mononoms p*›
⟨*proof*⟩

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

lemma *in-set-merge-coeffsD*:
‹$(a, b) \in$ *set* (*merge-coeffs p*) $\Longrightarrow \exists\, b.\ (a, b) \in$ *set p*›
⟨*proof*⟩

lemma *rtranclp-normalize-poly-add-mset*:
‹$normalize\text{-}poly\text{-}p^{**}\ A\ r \Longrightarrow normalize\text{-}poly\text{-}p^{**}\ (add\text{-}mset\ x\ A)\ (add\text{-}mset\ x\ r)$›
⟨*proof*⟩

lemma *nonzero-coeffs-diff*:

‹*nonzero-coeffs A* $\Longrightarrow$ *nonzero-coeffs* (*A* − *B*)›
⟨*proof*⟩

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

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* $\Longrightarrow$ *sorted-wrt* (*rel2p* (*Id* ∪ *var-order-rel*)) *a* $\Longrightarrow$
sorted-wrt var-order a›
⟨*proof*⟩

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

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

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

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

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

lemma *RETURN-map-alt-def*:
‹*RETURN o* (*map f*) =
REC_T (λ*g xs*.
case xs of
[] ⇒ *RETURN* []

$|\ x\ \#\ xs \Rightarrow do\ \{xs \leftarrow g\ xs;\ RETURN\ (f\ x\ \#\ xs)\}\})$›
⟨*proof*⟩

lemma *fully-unsorted-poly-rel-Cons-iff*:
‹$((ys,\ n)\ \#\ p,\ a) \in$ *fully-unsorted-poly-rel* $\longleftrightarrow$
$(p,\ remove1\text{-}mset\ (mset\ ys,\ n)\ a) \in$ *fully-unsorted-poly-rel* $\wedge$
$(mset\ ys,\ n) \in\#\ a \wedge distinct\ ys$›
⟨*proof*⟩

lemma *map-mset-unsorted-term-poly-list-rel*:
‹$(\bigwedge a.\ a \in mononoms\ s \Longrightarrow distinct\ a) \Longrightarrow \forall x \in mononoms\ s.\ distinct\ x \Longrightarrow$
$(\forall xs \in mononoms\ s.\ sorted\text{-}wrt\ (rel2p\ (Id \cup var\text{-}order\text{-}rel))\ xs) \Longrightarrow$
$(s,\ map\ (\lambda(a,\ y).\ (mset\ a,\ y))\ s)$
$\in \langle term\text{-}poly\text{-}list\text{-}rel \times_r int\text{-}rel\rangle list\text{-}rel$›
⟨*proof*⟩

lemma *list-rel-unsorted-term-poly-list-relD*:
‹$(p,\ y) \in \langle unsorted\text{-}term\text{-}poly\text{-}list\text{-}rel \times_r int\text{-}rel\rangle list\text{-}rel \Longrightarrow$
$mset\ y = (\lambda(a,\ y).\ (mset\ a,\ y))\ \text{`}\#\ mset\ p \wedge (\forall x \in mononoms\ p.\ distinct\ x)$›
⟨*proof*⟩

lemma *shuffle-terms-distinct-iff*:
assumes ‹$map\ (\lambda(a,\ y).\ (mset\ a,\ y))\ p = map\ (\lambda(a,\ y).\ (mset\ a,\ y))\ s$›
shows ‹$(\forall x{\in}set\ p.\ distinct\ (fst\ x)) \longleftrightarrow (\forall x{\in}set\ s.\ distinct\ (fst\ x))$›
⟨*proof*⟩

lemma
‹$(p,\ y) \in \langle unsorted\text{-}term\text{-}poly\text{-}list\text{-}rel \times_r int\text{-}rel\rangle list\text{-}rel \Longrightarrow$
$(a,\ b) \in set\ p \Longrightarrow distinct\ a$›
⟨*proof*⟩

lemma *sort-all-coeffs-unsorted-poly-rel-with0*:
assumes ‹$(p,\ p') \in$ *fully-unsorted-poly-rel*›
shows ‹$sort\text{-}all\text{-}coeffs\ p \leq \Downarrow (unsorted\text{-}poly\text{-}rel\text{-}with0)\ (RETURN\ p')$›
⟨*proof*⟩

lemma *sort-poly-spec-id'*:
assumes ‹$(p,\ p') \in unsorted\text{-}poly\text{-}rel\text{-}with0$›
shows ‹$sort\text{-}poly\text{-}spec\ p \leq \Downarrow (sorted\text{-}repeat\text{-}poly\text{-}rel\text{-}with0)\ (RETURN\ p')$›
⟨*proof*⟩

fun *merge-coeffs0* :: ‹*llist-polynomial* ⇒ *llist-polynomial*› **where**
‹$merge\text{-}coeffs0\ [] = []$› |
‹$merge\text{-}coeffs0\ [(xs,\ n)] = (if\ n = 0\ then\ []\ else\ [(xs,\ n)])$› |
‹$merge\text{-}coeffs0\ ((xs,\ n)\ \#\ (ys,\ m)\ \#\ p) =$
$(if\ xs = ys$
$then\ if\ n + m \neq 0\ then\ merge\text{-}coeffs0\ ((xs,\ n + m)\ \#\ p)\ else\ merge\text{-}coeffs0\ p$
$else\ if\ n = 0\ then\ merge\text{-}coeffs0\ ((ys,\ m)\ \#\ p)$
$else(xs,\ n)\ \#\ merge\text{-}coeffs0\ ((ys,\ m)\ \#\ p))$›

lemma *sorted-repeat-poly-list-rel-with0-wrt-ConsD*:
‹$((ys,\ n)\ \#\ p,\ a) \in sorted\text{-}repeat\text{-}poly\text{-}list\text{-}rel\text{-}with0\text{-}wrt\ S\ term\text{-}poly\text{-}list\text{-}rel \Longrightarrow$

$(p,\ remove1\text{-}mset\ (mset\ ys,\ n)\ a) \in sorted\text{-}repeat\text{-}poly\text{-}list\text{-}rel\text{-}with0\text{-}wrt\ S\ term\text{-}poly\text{-}list\text{-}rel\ \wedge$
$(mset\ ys,\ n) \in\#\ a \wedge (\forall x \in set\ p.\ S\ ys\ (fst\ x)) \wedge sorted\text{-}wrt\ (rel2p\ var\text{-}order\text{-}rel)\ ys \wedge$
$distinct\ ys$›
⟨*proof*⟩

lemma *sorted-repeat-poly-list-rel-with0-wrtl-Cons-iff*:
‹$((ys,\ n)\ \#\ p,\ a) \in sorted\text{-}repeat\text{-}poly\text{-}list\text{-}rel\text{-}with0\text{-}wrt\ S\ term\text{-}poly\text{-}list\text{-}rel \longleftrightarrow$
$(p,\ remove1\text{-}mset\ (mset\ ys,\ n)\ a) \in sorted\text{-}repeat\text{-}poly\text{-}list\text{-}rel\text{-}with0\text{-}wrt\ S\ term\text{-}poly\text{-}list\text{-}rel\ \wedge$
$(mset\ ys,\ n) \in\#\ a \wedge (\forall x \in set\ p.\ S\ ys\ (fst\ x)) \wedge sorted\text{-}wrt\ (rel2p\ var\text{-}order\text{-}rel)\ ys \wedge$
$distinct\ ys$›
⟨*proof*⟩

lemma *fst-normalize0-polynomial-subsetD*:
‹$(a,\ b) \in set\ (merge\text{-}coeffs0\ p) \Longrightarrow a \in mononoms\ p$›
⟨*proof*⟩

lemma *in-set-merge-coeffs0D*:
‹$(a,\ b) \in set\ (merge\text{-}coeffs0\ p) \Longrightarrow \exists b.\ (a,\ b) \in set\ p$›
⟨*proof*⟩

lemma *merge-coeffs0-is-normalize-poly-p*:
‹$(xs,\ ys) \in sorted\text{-}repeat\text{-}poly\text{-}rel\text{-}with0 \Longrightarrow \exists r.\ (merge\text{-}coeffs0\ xs,\ r) \in sorted\text{-}poly\text{-}rel\ \wedge\ normalize\text{-}poly\text{-}p^{**}\ ys\ r$›
⟨*proof*⟩

definition *full-normalize-poly* **where**
‹$full\text{-}normalize\text{-}poly\ p = do\ \{$
$p \leftarrow sort\text{-}all\text{-}coeffs\ p;$
$p \leftarrow sort\text{-}poly\text{-}spec\ p;$
$RETURN\ (merge\text{-}coeffs0\ p)$
$\}$›

fun *sorted-remdups* **where**
‹$sorted\text{-}remdups\ (x\ \#\ y\ \#\ zs) =$
$(if\ x = y\ then\ sorted\text{-}remdups\ (y\ \#\ zs)\ else\ x\ \#\ sorted\text{-}remdups\ (y\ \#\ zs))$› |
‹$sorted\text{-}remdups\ zs = zs$›

lemma *set-sorted-remdups*[*simp*]:
‹$set\ (sorted\text{-}remdups\ xs) = set\ xs$›
⟨*proof*⟩

lemma *distinct-sorted-remdups*:
‹$sorted\text{-}wrt\ R\ xs \Longrightarrow transp\ R \Longrightarrow Restricted\text{-}Predicates.total\text{-}on\ R\ UNIV \Longrightarrow$
$antisymp\ R \Longrightarrow distinct\ (sorted\text{-}remdups\ xs)$›
⟨*proof*⟩

lemma *full-normalize-poly-normalize-poly-p*:
assumes ‹$(p,\ p') \in fully\text{-}unsorted\text{-}poly\text{-}rel$›
shows ‹$full\text{-}normalize\text{-}poly\ p \le \Downarrow (sorted\text{-}poly\text{-}rel)\ (SPEC\ (\lambda r.\ normalize\text{-}poly\text{-}p^{**}\ p'\ r))$›
(**is** ‹$?A \le \Downarrow\ ?R\ ?B$›)
⟨*proof*⟩

definition *mult-poly-full* :: ‹-› **where**
‹$mult\text{-}poly\text{-}full\ p\ q = do\ \{$

let pq = *mult-poly-raw p q*;
normalize-poly pq
}›

lemma *normalize-poly-normalize-poly-p*:
assumes ‹$(p, p') \in$ *unsorted-poly-rel*›
shows ‹*normalize-poly p* $\leq \Downarrow$ (*sorted-poly-rel*) (*SPEC* ($\lambda r.$ *normalize-poly-p*** p' r))›
⟨*proof*⟩

8.6 Multiplication and normalisation

definition *mult-poly-p'* :: ‹-› **where**
‹*mult-poly-p'* p' q' = *do* {
$pq \leftarrow$ *SPEC*($\lambda r.$ (*mult-poly-p* q')** (p', {#}) ({#}, r));
SPEC ($\lambda r.$ *normalize-poly-p*** *pq r*)
}›

lemma *unsorted-poly-rel-fully-unsorted-poly-rel*:
‹*unsorted-poly-rel* $\subseteq$ *fully-unsorted-poly-rel*›
⟨*proof*⟩

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')›
⟨*proof*⟩

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')›
⟨*proof*⟩

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')›
⟨*proof*⟩

lemma *mult-poly-p'-mult-poly-spec*:
‹$(p, p') \in$ *mset-poly-rel* $\Longrightarrow$ $(q, q') \in$ *mset-poly-rel* $\Longrightarrow$

$mult\text{-}poly\text{-}p'\ p\ q \le \Downarrow mset\text{-}poly\text{-}rel\ (mult\text{-}poly\text{-}spec\ p'\ q')$›
⟨*proof*⟩

lemma *add-poly-p′-add-poly-spec*:
‹$(p,\ p') \in mset\text{-}poly\text{-}rel \Longrightarrow (q,\ q') \in mset\text{-}poly\text{-}rel \Longrightarrow$
$add\text{-}poly\text{-}p'\ p\ q \le \Downarrow mset\text{-}poly\text{-}rel\ (add\text{-}poly\text{-}spec\ p'\ q')$›
⟨*proof*⟩

end

definition *weak-equality-l* :: ‹$llist\text{-}polynomial \Rightarrow llist\text{-}polynomial \Rightarrow bool\ nres$› **where**
‹$weak\text{-}equality\text{-}l\ p\ q = RETURN\ (p = q)$›

definition *weak-equality* :: ‹$int\ mpoly \Rightarrow int\ mpoly \Rightarrow bool\ nres$› **where**
‹$weak\text{-}equality\ p\ q = SPEC\ (\lambda r.\ r \longrightarrow p = q)$›

definition *weak-equality-spec* :: ‹$mset\text{-}polynomial \Rightarrow mset\text{-}polynomial \Rightarrow bool\ nres$› **where**
‹$weak\text{-}equality\text{-}spec\ p\ q = SPEC\ (\lambda r.\ r \longrightarrow p = q)$›

lemma *term-poly-list-rel-same-rightD*:
‹$(a,\ aa) \in term\text{-}poly\text{-}list\text{-}rel \Longrightarrow (a,\ ab) \in term\text{-}poly\text{-}list\text{-}rel \Longrightarrow aa = ab$›
⟨*proof*⟩

lemma *list-rel-term-poly-list-rel-same-rightD*:
‹$(xa,\ y) \in \langle term\text{-}poly\text{-}list\text{-}rel \times_r int\text{-}rel\rangle list\text{-}rel \Longrightarrow$
$(xa,\ ya) \in \langle term\text{-}poly\text{-}list\text{-}rel \times_r int\text{-}rel\rangle list\text{-}rel \Longrightarrow$
$y = ya$›
⟨*proof*⟩

lemma *weak-equality-l-weak-equality-spec*:
‹$(uncurry\ weak\text{-}equality\text{-}l,\ uncurry\ weak\text{-}equality\text{-}spec) \in$
$sorted\text{-}poly\text{-}rel \times_r sorted\text{-}poly\text{-}rel \rightarrow_f \langle bool\text{-}rel\rangle nres\text{-}rel$›
⟨*proof*⟩

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

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

lemma *Collect-eq-comp′*:
$\{(x,\ y).\ P\ x\ y\}\ O\ \{(c,\ a).\ c = f\ a\} = \{(x,\ a).\ P\ x\ (f\ a)\}$
⟨*proof*⟩

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

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

lemma *in-set-remove1D*:

$a \in set\ (remove1\ x\ xs) \Longrightarrow a \in set\ xs$
⟨*proof*⟩

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*›
⟨*proof*⟩

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

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

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

Addition checking **definition** *error-msg* **where**
  ‹error-msg i msg = CFAILED (''s CHECKING failed at line '' @ show i @ '' with error '' @ msg)›

definition *error-msg-notin-dom-err* **where**
  ‹error-msg-notin-dom-err = '' notin domain''›

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

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

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

@ *show r*›

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

definition *vars-llist* :: ‹*llist-polynomial* ⇒ *string set*› **where**
‹*vars-llist xs* = $\bigcup$(*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* $\mathcal{V}$ *p q i r* = *do* {
  let b = *p* ∈# *dom-m A* ∧ *q* ∈# *dom-m A* ∧ *i* ∉# *dom-m A* ∧ *vars-llist r* ⊆ $\mathcal{V}$;
  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* $\mathcal{V}$ *p q i r* = *do* {
   let b = *p* ∈# *dom-m A* ∧ *i* ∉# *dom-m A* ∧ *vars-llist q* ⊆ $\mathcal{V}$∧ *vars-llist r* ⊆ $\mathcal{V}$;
   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;

$RETURN\ (error\text{-}msg\ i\ c)$
}
}
}
}›

lemma *check-extension-alt-def*:
‹$check\text{-}extension\ A\ \mathcal{V}\ i\ v\ p \geq do$ {
$b \leftarrow SPEC(\lambda b.\ b \longrightarrow i \notin\#\ dom\text{-}m\ A \wedge v \notin \mathcal{V})$;
$if\ \neg b$
$then\ RETURN\ (False)$
$else\ do$ {
$p' \leftarrow RETURN\ (p + Var\ v)$;
$b \leftarrow SPEC(\lambda b.\ b \longrightarrow vars\ p' \subseteq \mathcal{V})$;
$if\ \neg b$
$then\ RETURN\ (False)$
$else\ do$ {
$pq \leftarrow mult\text{-}poly\text{-}spec\ p'\ p'$;
$let\ p' = -\ p'$;
$p \leftarrow add\text{-}poly\text{-}spec\ pq\ p'$;
$eq \leftarrow weak\text{-}equality\ p\ 0$;
$if\ eq\ then\ RETURN(True)$
$else\ RETURN\ (False)$
}
}
}›
⟨*proof*⟩

lemma *RES-RES-RETURN-RES*: ‹$RES\ A \gg= (\lambda T.\ RES\ (f\ T)) = RES\ (\bigcup (f\ `\ A))$›
⟨*proof*⟩

lemma *check-add-alt-def*:
‹$check\text{-}add\ A\ \mathcal{V}\ p\ q\ i\ r \geq$
do {
$b \leftarrow SPEC(\lambda b.\ b \longrightarrow p \in\#\ dom\text{-}m\ A \wedge q \in\#\ dom\text{-}m\ A \wedge i \notin\#\ dom\text{-}m\ A \wedge vars\ r \subseteq \mathcal{V})$;
$if\ \neg b$
$then\ RETURN\ False$
$else\ do$ {
$ASSERT\ (p \in\#\ dom\text{-}m\ A)$;
$let\ p = the\ (fmlookup\ A\ p)$;
$ASSERT\ (q \in\#\ dom\text{-}m\ A)$;
$let\ q = the\ (fmlookup\ A\ q)$;
$pq \leftarrow add\text{-}poly\text{-}spec\ p\ q$;
$eq \leftarrow weak\text{-}equality\ pq\ r$;
$RETURN\ eq$
}
}› (**is** ‹- ≥ *?A*›)
⟨*proof*⟩

lemma *check-mult-alt-def*:
‹$check\text{-}mult\ A\ \mathcal{V}\ p\ q\ i\ r \geq$
do {

$b \leftarrow SPEC(\lambda b.\ b \longrightarrow p \in\#\ dom\text{-}m\ A \land i \notin\#\ dom\text{-}m\ A \land vars\ q \subseteq \mathcal{V}\ \land vars\ r \subseteq \mathcal{V});$
if $\neg b$
then RETURN False
else do {
ASSERT ($p \in\#$ *dom-m A*);
let p = *the* (*fmlookup A p*);
pq ← *mult-poly-spec p q*;
p ← *weak-equality pq r*;
RETURN p
}
}›
⟨*proof*⟩

primrec *insort-key-rel* :: ($'b \Rightarrow 'b \Rightarrow bool) \Rightarrow 'b \Rightarrow 'b\ list \Rightarrow 'b\ list$ **where**
insort-key-rel f x [] = [*x*] |
insort-key-rel f x (*y#ys*) =
(*if f x y then* (*x#y#ys*) *else y#*(*insort-key-rel f x ys*))

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

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

lemma *sorted-wrt-insort-key-rel2*:
‹*total-on R* (*insert x* (*set xs*)) ⟹ *transp R* ⟹ $x \notin$ *set xs* ⟹
sorted-wrt R xs ⟹ *sorted-wrt R* (*insort-key-rel R x xs*)›
⟨*proof*⟩

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

$q \leftarrow$ *full-normalize-poly* (*pac-mult st*);
eq $\leftarrow$ *check-mult-l spec A* $\mathcal{V}$ (*pac-src1 st*) *q* (*new-id st*) *r*;
let - = *eq*;
if $\neg$*is-cfailed eq*
then RETURN (*merge-cstatus st' eq*,
$\mathcal{V}$, *fmupd* (*new-id st*) *r A*)
else RETURN (*eq*, $\mathcal{V}$, *A*)
}
| *Extension* - - - $\Rightarrow$
do {
r $\leftarrow$ *full-normalize-poly* (([*new-var st*], -1) # (*pac-res st*));
(*eq*) $\leftarrow$ *check-extension-l spec A* $\mathcal{V}$ (*new-id st*) (*new-var st*) *r*;
if $\neg$*is-cfailed eq*
then do {
RETURN (*st'*,
insert (*new-var st*) $\mathcal{V}$, *fmupd* (*new-id st*) *r A*)}
else RETURN (*eq*, $\mathcal{V}$, *A*)
}
)›

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

definition *mononoms-equal-up-to-reorder* **where**
‹*mononoms-equal-up-to-reorder xs ys* $\longleftrightarrow$
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* ($\lambda p'$.
normalize-poly-p** ((λ(*a*, *b*). (*mset a*, *b*)) ‘# *mset p*) ((λ(*a*, *b*). (*mset a*, *b*)) ‘# *mset p'*) $\wedge$
0 $\notin$# *snd* ‘# *mset p'* $\wedge$
sorted-wrt (*rel2p* (*term-order-rel* $\times_r$ *int-rel*)) *p'* $\wedge$
($\forall$ *x* $\in$ *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* $\Rightarrow$ *string set* $\Rightarrow$ (*nat*, *llist-polynomial*) *fmap* $\Rightarrow$
(- *code-status* $\times$ *string set* $\times$ (*nat*, *llist-polynomial*) *fmap*) *nres*› **where**
‹*remap-polys-l spec* = ($\lambda\mathcal{V}$ *A*. *do*{
dom $\leftarrow$ *SPEC*(λ*dom*. *set-mset* (*dom-m A*) $\subseteq$ *dom* $\wedge$ *finite dom*);
failed $\leftarrow$ *SPEC*(λ-::*bool*. *True*);
if failed
then do {
c $\leftarrow$ *remap-polys-l-dom-err*;
RETURN (*error-msg* (*0* :: *nat*) *c*, $\mathcal{V}$, *fmempty*)
}
else do {
(*b*, $\mathcal{V}$, *A*) $\leftarrow$ *FOREACH dom*
(λ*i* (*b*, $\mathcal{V}$, *A'*).
if i $\in$# *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›
⟨proof⟩

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

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

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

lemma *fully-unsorted-poly-rel-extend-vars*:
‹(A, B) ∈ fully-unsorted-poly-rel O mset-poly-rel ⟹

$(x1c,\ x1a) \in \langle var\text{-}rel\rangle set\text{-}rel \Longrightarrow$
$RETURN\ (x1c \cup vars\text{-}llist\ A)$
$\le \Downarrow (\langle var\text{-}rel\rangle set\text{-}rel)$
$(SPEC\ ((\subseteq)\ (x1a \cup vars\ (B))))$›
⟨*proof*⟩

lemma *remap-polys-l-remap-polys*:
assumes
AB: ‹$(A,\ B) \in \langle nat\text{-}rel,\ fully\text{-}unsorted\text{-}poly\text{-}rel\ O\ mset\text{-}poly\text{-}rel\rangle fmap\text{-}rel$› **and**
spec: ‹$(spec,\ spec') \in sorted\text{-}poly\text{-}rel\ O\ mset\text{-}poly\text{-}rel$› **and**
V: ‹$(\mathcal{V},\ \mathcal{V}') \in \langle var\text{-}rel\rangle set\text{-}rel$›
shows ‹$remap\text{-}polys\text{-}l\ spec\ \mathcal{V}\ A \le$
$\Downarrow(code\text{-}status\text{-}status\text{-}rel \times_r \langle var\text{-}rel\rangle set\text{-}rel \times_r fmap\text{-}polys\text{-}rel)\ (remap\text{-}polys\ spec'\ \mathcal{V}'\ B)$›
(**is** ‹- $\le \Downarrow$ *?R* -›)
⟨*proof*⟩

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

lemma *weak-equality-spec-weak-equality*:
‹$(p,\ p') \in mset\text{-}poly\text{-}rel \Longrightarrow$
$(r,\ r') \in mset\text{-}poly\text{-}rel \Longrightarrow$
$weak\text{-}equality\text{-}spec\ p\ r \le weak\text{-}equality\ p'\ r'$›
⟨*proof*⟩

lemma *weak-equality-l-weak-equality-l'*[*refine*]:
‹$weak\text{-}equality\text{-}l\ p\ q \le \Downarrow bool\text{-}rel\ (weak\text{-}equality\ p'\ q')$›
if ‹$(p,\ p') \in sorted\text{-}poly\text{-}rel\ O\ mset\text{-}poly\text{-}rel$›
‹$(q,\ q') \in sorted\text{-}poly\text{-}rel\ O\ mset\text{-}poly\text{-}rel$›
for $p\ p'\ q\ q'$
⟨*proof*⟩

lemma *error-msg-ne-SUCCES*[*iff*]:
‹$error\text{-}msg\ i\ m \ne CSUCCESS$›
‹$error\text{-}msg\ i\ m \ne CFOUND$›
‹$is\text{-}cfailed\ (error\text{-}msg\ i\ m)$›
‹$\neg is\text{-}cfound\ (error\text{-}msg\ i\ m)$›
⟨*proof*⟩

lemma *sorted-poly-rel-vars-llist*:
‹$(r,\ r') \in sorted\text{-}poly\text{-}rel\ O\ mset\text{-}poly\text{-}rel \Longrightarrow$
$vars\ r' \subseteq \varphi\ `\ vars\text{-}llist\ r$›
⟨*proof*⟩

lemma *check-addition-l-check-add*:
assumes ‹$(A,\ B) \in fmap\text{-}polys\text{-}rel$› **and** ‹$(r,\ r') \in sorted\text{-}poly\text{-}rel\ O\ mset\text{-}poly\text{-}rel$›
‹$(p,\ p') \in Id$› ‹$(q,\ q') \in Id$› ‹$(i,\ i') \in nat\text{-}rel$›
‹$(\mathcal{V}',\ \mathcal{V}) \in \langle var\text{-}rel\rangle set\text{-}rel$›
shows
‹$check\text{-}addition\text{-}l\ spec\ A\ \mathcal{V}'\ p\ q\ i\ r \le \Downarrow \{(st,\ b).\ (\neg is\text{-}cfailed\ st \longleftrightarrow b) \wedge$

$(is\text{-}cfound\ st \longrightarrow spec = r)\}$ $(check\text{-}add\ B\ \mathcal{V}\ p'\ q'\ i'\ r')$›
⟨*proof*⟩

lemma *check-del-l-check-del*:
‹$(A, B) \in fmap\text{-}polys\text{-}rel \Longrightarrow (x3, x3a) \in Id \Longrightarrow check\text{-}del\text{-}l\ spec\ A\ (pac\text{-}src1\ (Del\ x3))$
$\leq \Downarrow \{(st, b).\ (\neg is\text{-}cfailed\ st \longleftrightarrow b) \wedge (b \longrightarrow st = CSUCCESS)\}\ (check\text{-}del\ B\ (pac\text{-}src1\ (Del\ x3a)))$›
⟨*proof*⟩

lemma *check-mult-l-check-mult*:
assumes ‹$(A, B) \in fmap\text{-}polys\text{-}rel$› **and** ‹$(r, r') \in sorted\text{-}poly\text{-}rel\ O\ mset\text{-}poly\text{-}rel$› **and**
‹$(q, q') \in sorted\text{-}poly\text{-}rel\ O\ mset\text{-}poly\text{-}rel$›
‹$(p, p') \in Id$› ‹$(i, i') \in nat\text{-}rel$› ‹$(\mathcal{V}, \mathcal{V}') \in \langle var\text{-}rel\rangle set\text{-}rel$›
shows
‹$check\text{-}mult\text{-}l\ spec\ A\ \mathcal{V}\ p\ q\ i\ r \leq \Downarrow\ \{(st, b).\ (\neg is\text{-}cfailed\ st \longleftrightarrow b) \wedge$
$(is\text{-}cfound\ st \longrightarrow spec = r)\}\ (check\text{-}mult\ B\ \mathcal{V}'\ p'\ q'\ i'\ r')$›
⟨*proof*⟩

lemma *normalize-poly-normalize-poly-spec*:
assumes ‹$(r, t) \in unsorted\text{-}poly\text{-}rel\ O\ mset\text{-}poly\text{-}rel$›
shows
‹$normalize\text{-}poly\ r \leq \Downarrow(sorted\text{-}poly\text{-}rel\ O\ mset\text{-}poly\text{-}rel)\ (normalize\text{-}poly\text{-}spec\ t)$›
⟨*proof*⟩

lemma *remove1-list-rel*:
‹$(xs, ys) \in \langle R\rangle\ list\text{-}rel \Longrightarrow$
$(a, b) \in R \Longrightarrow$
$IS\text{-}RIGHT\text{-}UNIQUE\ R \Longrightarrow$
$IS\text{-}LEFT\text{-}UNIQUE\ R \Longrightarrow$
$(remove1\ a\ xs, remove1\ b\ ys) \in \langle R\rangle list\text{-}rel$›
⟨*proof*⟩

lemma *remove1-list-rel2*:
‹$(xs, ys) \in \langle R\rangle\ list\text{-}rel \Longrightarrow$
$(a, b) \in R \Longrightarrow$
$(\bigwedge c.\ (a, c) \in R \Longrightarrow c = b) \Longrightarrow$
$(\bigwedge c.\ (c, b) \in R \Longrightarrow c = a) \Longrightarrow$
$(remove1\ a\ xs, remove1\ b\ ys) \in \langle R\rangle list\text{-}rel$›
⟨*proof*⟩

lemma *remove1-sorted-poly-rel-mset-poly-rel*:
assumes
‹$(r, r') \in sorted\text{-}poly\text{-}rel\ O\ mset\text{-}poly\text{-}rel$› **and**
‹$([a], 1) \in set\ r$›
shows
‹$(remove1\ ([a], 1)\ r, r' - Var\ (\varphi\ a))$
$\in sorted\text{-}poly\text{-}rel\ O\ mset\text{-}poly\text{-}rel$›
⟨*proof*⟩

lemma *remove1-sorted-poly-rel-mset-poly-rel-minus*:
assumes
‹$(r, r') \in sorted\text{-}poly\text{-}rel\ O\ mset\text{-}poly\text{-}rel$› **and**
‹$([a], -1) \in set\ r$›
shows
‹$(remove1\ ([a], -1)\ r, r' + Var\ (\varphi\ a))$

$\in$ *sorted-poly-rel O mset-poly-rel*›
⟨*proof*⟩

lemma *insert-var-rel-set-rel*:
‹$(\mathcal{V}, \mathcal{V}') \in \langle$*var-rel*$\rangle$*set-rel* $\Longrightarrow$
(*yb*, *x2*) $\in$ *var-rel* $\Longrightarrow$
(*insert yb* $\mathcal{V}$, *insert x2* $\mathcal{V}'$) $\in \langle$*var-rel*$\rangle$*set-rel*›
⟨*proof*⟩

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* $\in \mathcal{V}'$›
⟨*proof*⟩

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* $\le$
$\Downarrow$\{((*st*), (*b*)).
($\neg$*is-cfailed st* $\longleftrightarrow$ *b*) $\wedge$
(*is-cfound st* $\longrightarrow$ *spec* = *r*)\} (*check-extension B* $\mathcal{V}'$ *i' x' r'*)›
⟨*proof*⟩

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

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

lemma *list-rel-append-same-length*:
‹*length xs* = *length xs'* $\Longrightarrow$ (*xs* @ *ys*, *xs'* @ *ys'*) $\in \langle R\rangle$*list-rel* $\longleftrightarrow$ (*xs*, *xs'*) $\in \langle R\rangle$*list-rel* $\wedge$ (*ys*, *ys'*) $\in$ $\langle R\rangle$*list-rel*›
⟨*proof*⟩

lemma *term-poly-list-rel-list-relD*: ‹(*ys*, *cs*) $\in \langle$*term-poly-list-rel* $\times_r$ *int-rel*$\rangle$*list-rel* $\Longrightarrow$
cs = *map* (λ(*a*, *y*). (*mset a*, *y*)) *ys*›
⟨*proof*⟩

lemma *term-poly-list-rel-single*: ‹([*x32*], \{#*x32*#\}) $\in$ *term-poly-list-rel*›
⟨*proof*⟩

lemma *unsorted-poly-rel-list-rel-list-rel-uminus*:
‹(*map* (λ(*a*, *b*). (*a*, − *b*)) *r*, *yc*)

$\in \langle$*unsorted-term-poly-list-rel* $\times_r$ *int-rel*$\rangle$*list-rel* $\Longrightarrow$
$(r,\ map\ (\lambda(a,\ b).\ (a,\ -\ b))\ yc)$
$\in \langle$*unsorted-term-poly-list-rel* $\times_r$ *int-rel*$\rangle$*list-rel*›
⟨*proof*⟩

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

lemma *fully-unsorted-poly-rel-diff*:
‹$([v],\ v') \in$ *fully-unsorted-poly-rel O mset-poly-rel* $\Longrightarrow$
$(r,\ r') \in$ *fully-unsorted-poly-rel O mset-poly-rel* $\Longrightarrow$
$(v\ \#\ r,$
$v' + r')$
$\in$ *fully-unsorted-poly-rel O mset-poly-rel*›
⟨*proof*⟩

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

lemma *code-status-status-rel-discrim-iff*:
‹$(x1a,\ x1c) \in$ *code-status-status-rel* $\Longrightarrow$ *is-cfailed x1a* $\longleftrightarrow$ *is-failed x1c*›
‹$(x1a,\ x1c) \in$ *code-status-status-rel* $\Longrightarrow$ *is-cfound x1a* $\longleftrightarrow$ *is-found x1c*›
⟨*proof*⟩

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

end

lemma *less-than-char-of-char*[*code-unfold*]:
‹$(x,\ y) \in$ *less-than-char* $\longleftrightarrow$ (*of-char x* :: *nat*) $<$ *of-char y*›
⟨*proof*⟩

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

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

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

lemma *full-checker-l-full-checker′*:
‹(*uncurry2 full-checker-l*, *uncurry2 full-checker*) ∈
((*fully-unsorted-poly-rel O mset-poly-rel*) $\times_r$ *unsorted-fmap-polys-rel*) $\times_r$ ⟨*pac-step-rel*⟩*list-rel* $\rightarrow_f$
⟨(*code-status-status-rel* $\times_r$ ⟨*var-rel*⟩*set-rel* $\times_r$ *fmap-polys-rel*)⟩*nres-rel*›
⟨*proof*⟩

end

definition *remap-polys-l2* :: ‹*llist-polynomial* ⇒ *string set* ⇒ (*nat*, *llist-polynomial*) *fmap* ⇒ - *nres*›
where
‹*remap-polys-l2 spec* = (λ$\mathcal{V}$ *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*, $\mathcal{V}$, *fmempty*)
}
else do {
  (*b*, $\mathcal{V}$, *A*) ← *nfoldli* ([*0*..<*n*]) (λ-. *True*)
  (λ*i* (*b*, $\mathcal{V}$, *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*;
      $\mathcal{V}$ ← *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*)
}
})›

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

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
⟨*proof*⟩
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
  ⟨*proof*⟩
end

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

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

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

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

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

lemma *less-upper-bintrunc-id*: ‹*n* < *2* ^*b* ⟹ *n* ≥ *0* ⟹ *take-bit b n* = *n*› **for** *n* :: *int*
  ⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

abbreviation *monomial-rel* **where**
‹*monomial-rel* ≡ *monom-rel* $\times_r$ *int-rel*›

abbreviation *monomial-assn* **where**
‹*monomial-assn* ≡ *monom-assn* $\times_a$ *int-assn*›

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

instance
⟨*proof*⟩

end

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

lemma *list-rel-list-rel-order-iff*:
  assumes ‹$(a,\ b) \in \langle$*string-rel*$\rangle$*list-rel*› ‹$(a',\ b') \in \langle$*string-rel*$\rangle$*list-rel*›
  shows ‹$a < a' \longleftrightarrow b < b'$›
$\langle$*proof*$\rangle$

lemma *string-rel-le*[*sepref-import-param*]:
  shows ‹$((<),\ (<)) \in \langle$*string-rel*$\rangle$*list-rel* $\rightarrow$ $\langle$*string-rel*$\rangle$*list-rel* $\rightarrow$ *bool-rel*›
  $\langle$*proof*$\rangle$

lemma [*sepref-import-param*]:
  assumes ‹*CONSTRAINT IS-LEFT-UNIQUE R*› ‹*CONSTRAINT IS-RIGHT-UNIQUE R*›
  shows ‹(*remove1*, *remove1*) $\in R \rightarrow \langle R\rangle$*list-rel* $\rightarrow \langle R\rangle$*list-rel*›
  $\langle$*proof*$\rangle$

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

instance
$\langle$*proof*$\rangle$

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*›
  $\langle$*proof*$\rangle$

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

  lemma *precise-hassoc-map-assn*: ‹*precise hassoc-map-assn*›
    $\langle$*proof*$\rangle$

  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*
⟨*proof*⟩

interpretation *hassoc*: *bind-map-is-empty hassoc-map-assn hassoc-isEmpty*
⟨*proof*⟩

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

interpretation *hassoc*: *map-custom-empty op-assoc-empty*
⟨*proof*⟩

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)* $\Longrightarrow_A$ *hassoc-map-assn (m(k ↦ v)) ((k, v) # p) ∗ true*›
⟨*proof*⟩

interpretation *hassoc*: *bind-map-update hassoc-map-assn hassoc-update*
⟨*proof*⟩

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)*›
⟨*proof*⟩

lemma *hassoc-map-assn-hassoc-delete*: ‹<*hassoc-map-assn m p*> *hassoc-delete k p* <*hassoc-map-assn (m* |‘ (− {*k*}))>$_t$›
⟨*proof*⟩

interpretation *hassoc*: *bind-map-delete hassoc-map-assn hassoc-delete*
⟨*proof*⟩

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*)>$_t$›
⟨*proof*⟩

interpretation *hassoc*: *bind-map-lookup hassoc-map-assn hassoc-lookup*
⟨*proof*⟩

⟨*ML*⟩
interpretation *hassoc*: *gen-contains-key-by-lookup hassoc-map-assn hassoc-lookup*
⟨*proof*⟩

$\langle ML \rangle$

interpretation *hassoc*: *bind-map-contains-key hassoc-map-assn hassoc.contains-key*
$\langle proof \rangle$

11.1 Conversion from assoc to other map

definition *hash-of-assoc-map* **where**
‹*hash-of-assoc-map xs* = *fold* ($\lambda(k,\ v)\ m.$ *if m k* $\neq$ *None then m else m*($k \mapsto v$)) *xs Map.empty*›

lemma *map-upd-map-add-left*:
‹$m(a \ \mapsto b)$ ++ m' = m ++ (*if* $a \notin$ *dom* m' *then* $m'(a \ \mapsto b)$ *else* m')›
$\langle proof \rangle$

lemma *fold-map-of-alt*:
‹*fold* ($\lambda(k,\ v)\ m.$ *if m k* $\neq$ *None then m else m*($k \mapsto v$)) *xs* m' = *map-of xs* ++ m'›
$\langle proof \rangle$

lemma *map-of-alt-def*:
‹*map-of xs* = *hash-of-assoc-map xs*›
$\langle proof \rangle$

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

lemma *hash-of-assoc-map-id*:
‹(*hash-of-assoc-map*, *hashmap-conv*) $\in$ *hassoc-map-rel-raw* $\rightarrow$ *Id*›
$\langle proof \rangle$

definition *hassoc-map-rel* **where**
hassoc-map-rel-internal-def:
‹*hassoc-map-rel K V* = *hassoc-map-rel-raw O* $\langle K, V \rangle$*map-rel*›

lemma *hassoc-map-rel-def*:
‹$\langle K, V \rangle$ *hassoc-map-rel* = *hassoc-map-rel-raw O* $\langle K, V \rangle$*map-rel*›
$\langle proof \rangle$

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* :: - $\Rightarrow$ *'a list* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list*
where
merge f (*x*#*xs*) (*y*#*ys*) =
(*if f x y then x* # *merge f xs* (*y*#*ys*) *else y* # *merge f* (*x*#*xs*) *ys*)
| *merge f xs* [] = *xs*

| *merge f* [] *ys* = *ys*

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

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

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

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

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

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

lemmas [*code del*] =
msort2.simps

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

declare *msort2.simps*[*simp*]

lemma *msort-msort2*:

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

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

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

12.2 Sorting applied to monomials

lemma *merge-coeffs-alt-def*:
‹(*RETURN o merge-coeffs*) *p* =
REC_T(λ*f 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›
⟨*proof*⟩

lemma *hn-invalid-recover*:
‹*is-pure R* ⟹ *hn-invalid R* = (λ*x y*. *R x y* ∗ *true*)›
‹*is-pure R* ⟹ *invalid-assn R* = (λ*x y*. *R x y* ∗ *true*)›
⟨*proof*⟩

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
⟨*proof*⟩

lemma *invalid-assn-distrib*:
‹*invalid-assn monom-assn* $\times_a$ *invalid-assn int-assn* = *invalid-assn* (*monom-assn* $\times_a$ *int-assn*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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›
⟨*proof*⟩

The refinement frameword is completely lost here when synthesizing the constants – it does not understant what is pure (actually everything) and what must be destroyed.

sepref-definition *merge-coeffs-impl*
is ‹*RETURN o merge-coeffs*›
:: ‹*poly-assn*^{d} $\rightarrow_a$ *poly-assn*›
⟨*proof*⟩

definition *full-quicksort-poly* **where**
‹*full-quicksort-poly* = *full-quicksort-ref* ($\lambda x\ y.\ x = y \lor (x, y) \in$ *term-order-rel*) *fst*›

lemma *down-eq-id-list-rel*: ‹⇓(⟨*Id*⟩*list-rel*) *x* = *x*›
⟨*proof*⟩

definition *quicksort-poly*:: ‹*nat* ⇒ *nat* ⇒ *llist-polynomial* ⇒ (*llist-polynomial*) *nres*› **where**
‹*quicksort-poly x y z* = *quicksort-ref* (≤) *fst* (*x*, *y*, *z*)›

term *partition-between-ref*

definition *partition-between-poly* :: ‹*nat* ⇒ *nat* ⇒ *llist-polynomial* ⇒ (*llist-polynomial* × *nat*) *nres*›
where
‹*partition-between-poly* = *partition-between-ref* (≤) *fst*›

definition *partition-main-poly* :: ‹*nat* ⇒ *nat* ⇒ *llist-polynomial* ⇒ (*llist-polynomial* × *nat*) *nres*› **where**
‹*partition-main-poly* = *partition-main* (≤) *fst*›

lemma *string-list-trans*:
‹(*xa* ::*char list list*, *ya*) ∈ *lexord* (*lexord* {(*x*, *y*). *x* < *y*}) ⟹
(*ya*, *z*) ∈ *lexord* (*lexord* {(*x*, *y*). *x* < *y*}) ⟹
(*xa*, *z*) ∈ *lexord* (*lexord* {(*x*, *y*). *x* < *y*})›
⟨*proof*⟩

lemma *full-quicksort-sort-poly-spec*:

‹(*full-quicksort-poly*, *sort-poly-spec*) ∈ ⟨*Id*⟩*list-rel* $\rightarrow_f$ ⟨⟨*Id*⟩*list-rel*⟩*nres-rel*›
⟨*proof*⟩

12.3 Lifting to polynomials

definition *merge-sort-poly* :: ‹-› **where**
‹*merge-sort-poly* = *msort* (λ*a b*. *fst a* ≤ *fst b*)›

definition *merge-monoms-poly* :: ‹-› **where**
‹*merge-monoms-poly* = *msort* (≤)›

definition *merge-poly* :: ‹-› **where**
‹*merge-poly* = *merge* (λ*a b*. *fst a* ≤ *fst b*)›

definition *merge-monoms* :: ‹-› **where**
‹*merge-monoms* = *merge* (≤)›

definition *msort-poly-impl* :: ‹(*String.literal list* × *int*) *list* ⇒ -› **where**
‹*msort-poly-impl* = *msort* (λ*a b*. *fst a* ≤ *fst b*)›

definition *msort-monoms-impl* :: ‹(*String.literal list*) ⇒ -› **where**
‹*msort-monoms-impl* = *msort* (≤)›

lemma *msort-poly-impl-alt-def*:
‹*msort-poly-impl xs* =
(*case xs of*
[] ⇒ []
| [*a*] ⇒ [*a*]
| [*a*,*b*] ⇒ *if fst a* ≤ *fst b then* [*a*,*b*]*else* [*b*,*a*]
| *xs* ⇒ *merge-poly*
(*msort-poly-impl* (*take* ((*length xs*) *div 2*) *xs*))
(*msort-poly-impl* (*drop* ((*length xs*) *div 2*) *xs*)))›
⟨*proof*⟩

lemma *le-term-order-rel'*:
‹(≤) = (λ*x y*. *x* = *y* ∨ *term-order-rel' x y*)›
⟨*proof*⟩

fun *lexord-eq* **where**
‹*lexord-eq* [] - = *True*› |
‹*lexord-eq* (*x* # *xs*) (*y* # *ys*) = (*x* < *y* ∨ (*x* = *y* ∧ *lexord-eq xs ys*))› |
‹*lexord-eq* - - = *False*›

lemma [*simp*]:
‹*lexord-eq* [] [] = *True*›
‹*lexord-eq* (*a* # *b*)[] = *False*›
‹*lexord-eq* [] (*a* # *b*) = *True*›
⟨*proof*⟩

lemma *var-order-rel'*:
‹(≤) = (λ*x y*. *x* = *y* ∨ (*x*,*y*) ∈ *var-order-rel*)›
⟨*proof*⟩

lemma *var-order-rel''*:
‹(*x*,*y*) ∈ *var-order-rel* ⟷ *x* < *y*›

⟨proof⟩

lemma *lexord-eq-alt-def1*:
‹$a \leq b$ = *lexord-eq a b*› **for** *a b* :: ‹*String.literal list*›
⟨proof⟩

lemma *lexord-eq-alt-def2*:
‹(*RETURN oo lexord-eq*) *xs ys* =
  REC_T (λ*f* (*xs*, *ys*).
    case (*xs*, *ys*) *of*
      ([], -) ⇒ *RETURN True*
    | (*x* # *xs*, *y* # *ys*) ⇒
      if x < *y then RETURN True*
      else if x = *y then f* (*xs*, *ys*) *else RETURN False*
    | - ⇒ *RETURN False*)
    (*xs*, *ys*)›
⟨proof⟩

definition *var-order′* **where**
[*simp*]: ‹*var-order′* = *var-order*›

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

lemma *var-order-rel-alt-def*:
‹*var-order-rel* = *p2rel char.lexordp*›
⟨proof⟩

lemma *var-order-rel-var-order*:
‹(*x*, *y*) ∈ *var-order-rel* ⟷ *var-order x y*›
⟨proof⟩

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

lemma [*sepref-import-param*]:
‹((≤), (≤)) ∈ *monom-rel* → *monom-rel* →*bool-rel*›
⟨proof⟩

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

lemma *lexordp-char-char*: ‹*ord-class.lexordp* = *char.lexordp*›
⟨proof⟩

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

sepref-register *lexord-eq*

sepref-definition *lexord-eq-term*
is ‹*uncurry* (*RETURN oo lexord-eq*)›
:: ‹$monom\text{-}assn^k *_a monom\text{-}assn^k \rightarrow_a bool\text{-}assn$›
⟨*proof*⟩

declare *lexord-eq-term.refine*[*sepref-fr-rules*]

lemmas [*code del*] = *msort-poly-impl-def msort-monoms-impl-def*
lemmas [*code*] =
msort-poly-impl-def[*unfolded lexord-eq-alt-def1*[*abs-def*]]
msort-monoms-impl-def[*unfolded msort-msort2*]

lemma *term-order-rel-trans*:
‹(*a*, *aa*) ∈ *term-order-rel* ⟹
(*aa*, *ab*) ∈ *term-order-rel* ⟹ (*a*, *ab*) ∈ *term-order-rel*›
⟨*proof*⟩

lemma *merge-sort-poly-sort-poly-spec*:
‹(*RETURN o merge-sort-poly*, *sort-poly-spec*) ∈ ⟨*Id*⟩*list-rel* $\rightarrow_f$ ⟨⟨*Id*⟩*list-rel*⟩*nres-rel*›
⟨*proof*⟩

lemma *msort-alt-def*:
‹*RETURN o* (*msort f*) =
REC_T (λ*g xs*.
case xs of
[] ⇒ *RETURN* []
| [*x*] ⇒ *RETURN* [*x*]
| - ⇒ *do* {
a ← *g* (*take* (*size xs div 2*) *xs*);
b ← *g* (*drop* (*size xs div 2*) *xs*);
RETURN (*merge f a b*)})›
⟨*proof*⟩

lemma *monomial-rel-order-map*:
‹(*x*, *a*, *b*) ∈ *monomial-rel* ⟹
(*y*, *aa*, *bb*) ∈ *monomial-rel* ⟹
fst x ≤ *fst y* ⟷ *a* ≤ *aa*›
⟨*proof*⟩

lemma *step-rewrite-pure*:
fixes *K* :: ‹('*olbl* × '*lbl*) *set*›
shows
‹*pure* (*p2rel* (⟨*K*, *V*, *R*⟩*pac-step-rel-raw*)) = *pac-step-rel-assn* (*pure K*) (*pure V*) (*pure R*)›
‹*monomial-assn* = *pure* (*monom-rel* $\times_r$ *int-rel*)› **and**
poly-assn-list:
‹*poly-assn* = *pure* (⟨*monom-rel* $\times_r$ *int-rel*⟩*list-rel*)›
⟨*proof*⟩

lemma *safe-pac-step-rel-assn*[*safe-constraint-rules*]:
is-pure K ⟹ *is-pure V* ⟹ *is-pure R* ⟹ *is-pure* (*pac-step-rel-assn K V R*)
⟨*proof*⟩

lemma *merge-poly-merge-poly*:
‹(*merge-poly*, *merge-poly*)
 $\in$ *poly-rel* $\to$ *poly-rel* $\to$ *poly-rel*›
 ⟨*proof*⟩

lemmas [*fcomp-norm-unfold*] =
poly-assn-list[*symmetric*]
step-rewrite-pure(*1*)

lemma *merge-poly-merge-poly2*:
‹$(a, b) \in$ *poly-rel* $\Longrightarrow (a', b') \in$ *poly-rel* $\Longrightarrow$
 (*merge-poly* a a', *merge-poly* b b') $\in$ *poly-rel*›
⟨*proof*⟩

lemma *list-rel-takeD*:
‹$(a, b) \in \langle R\rangle$*list-rel* $\Longrightarrow (n, n') \in Id \Longrightarrow$ (*take* n a, *take* n' b) $\in \langle R\rangle$*list-rel*›
⟨*proof*⟩

lemma *list-rel-dropD*:
‹$(a, b) \in \langle R\rangle$*list-rel* $\Longrightarrow (n, n') \in Id \Longrightarrow$ (*drop* n a, *drop* n' b) $\in \langle R\rangle$*list-rel*›
⟨*proof*⟩

lemma *merge-sort-poly*[*sepref-import-param*]:
‹(*msort-poly-impl*, *merge-sort-poly*)
 $\in$ *poly-rel* $\to$ *poly-rel*›
 ⟨*proof*⟩

lemmas [*sepref-fr-rules*] = *merge-sort-poly*[*FCOMP merge-sort-poly-sort-poly-spec*]

sepref-definition *partition-main-poly-impl*
is ‹*uncurry2 partition-main-poly*›
:: ‹*nat-assn*^{k} $*_a$ *nat-assn*^{k} $*_a$ *poly-assn*^{k} $\to_a$ *prod-assn poly-assn nat-assn* ›
⟨*proof*⟩

declare *partition-main-poly-impl.refine*[*sepref-fr-rules*]

sepref-definition *partition-between-poly-impl*
is ‹*uncurry2 partition-between-poly*›
:: ‹*nat-assn*^{k} $*_a$ *nat-assn*^{k} $*_a$ *poly-assn*^{k} $\to_a$ *prod-assn poly-assn nat-assn* ›
⟨*proof*⟩

declare *partition-between-poly-impl.refine*[*sepref-fr-rules*]

sepref-definition *quicksort-poly-impl*
is ‹*uncurry2 quicksort-poly*›
:: ‹*nat-assn*^{k} $*_a$ *nat-assn*^{k} $*_a$ *poly-assn*^{k} $\to_a$ *poly-assn*›
⟨*proof*⟩

lemmas [*sepref-fr-rules*] = *quicksort-poly-impl.refine*

sepref-register *quicksort-poly*
sepref-definition *full-quicksort-poly-impl*
is ‹*full-quicksort-poly*›

:: ‹$poly\text{-}assn^k \rightarrow_a poly\text{-}assn$›
⟨*proof*⟩

lemmas *sort-poly-spec-hnr* =
full-quicksort-poly-impl.refine[*FCOMP full-quicksort-sort-poly-spec*]

declare *merge-coeffs-impl.refine*[*sepref-fr-rules*]

sepref-definition *normalize-poly-impl*
is ‹*normalize-poly*›
:: ‹$poly\text{-}assn^k \rightarrow_a poly\text{-}assn$›
⟨*proof*⟩

declare *normalize-poly-impl.refine*[*sepref-fr-rules*]

definition *full-quicksort-vars* **where**
‹$full\text{-}quicksort\text{-}vars = full\text{-}quicksort\text{-}ref\ (\lambda x\ y.\ x = y \vee (x, y) \in var\text{-}order\text{-}rel)\ id$›

definition *quicksort-vars*:: ‹$nat \Rightarrow nat \Rightarrow string\ list \Rightarrow (string\ list)\ nres$› **where**
‹$quicksort\text{-}vars\ x\ y\ \ z = quicksort\text{-}ref\ (\leq)\ id\ (x, y, z)$›

definition *partition-between-vars* :: ‹$nat \Rightarrow nat \Rightarrow string\ list \Rightarrow (string\ list \times nat)\ nres$› **where**
‹$partition\text{-}between\text{-}vars = partition\text{-}between\text{-}ref\ (\leq)\ id$›

definition *partition-main-vars* :: ‹$nat \Rightarrow nat \Rightarrow string\ list \Rightarrow (string\ list \times nat)\ nres$› **where**
‹$partition\text{-}main\text{-}vars = partition\text{-}main\ (\leq)\ id$›

lemma *total-on-lexord-less-than-char-linear2*:
‹$xs \neq ys \Longrightarrow (xs, ys) \notin lexord\ (less\text{-}than\text{-}char) \longleftrightarrow$
$(ys, xs) \in lexord\ less\text{-}than\text{-}char$›
⟨*proof*⟩

lemma *string-trans*:
‹$(xa, ya) \in lexord\ \{(x::char, y::char).\ x < y\} \Longrightarrow$
$(ya, z) \in lexord\ \{(x::char, y::char).\ x < y\} \Longrightarrow$
$(xa, z) \in lexord\ \{(x::char, y::char).\ x < y\}$›
⟨*proof*⟩

lemma *full-quicksort-sort-vars-spec*:
‹$(full\text{-}quicksort\text{-}vars, sort\text{-}coeff) \in \langle Id\rangle list\text{-}rel \rightarrow_f \langle\langle Id\rangle list\text{-}rel\rangle nres\text{-}rel$›
⟨*proof*⟩

sepref-definition *partition-main-vars-impl*
is ‹*uncurry2 partition-main-vars*›
:: ‹$nat\text{-}assn^k *_a nat\text{-}assn^k *_a (monom\text{-}assn)^k \rightarrow_a prod\text{-}assn\ (monom\text{-}assn)\ nat\text{-}assn$›
⟨*proof*⟩

declare *partition-main-vars-impl.refine*[*sepref-fr-rules*]

sepref-definition *partition-between-vars-impl*

is ‹*uncurry2 partition-between-vars*›
:: ‹$nat\text{-}assn^k *_a nat\text{-}assn^k *_a monom\text{-}assn^k \to_a prod\text{-}assn\ monom\text{-}assn\ nat\text{-}assn$ ›
⟨*proof*⟩

declare *partition-between-vars-impl.refine*[*sepref-fr-rules*]

sepref-definition *quicksort-vars-impl*
is ‹*uncurry2 quicksort-vars*›
:: ‹$nat\text{-}assn^k *_a nat\text{-}assn^k *_a monom\text{-}assn^k \to_a monom\text{-}assn$›
⟨*proof*⟩

lemmas [*sepref-fr-rules*] = *quicksort-vars-impl.refine*

sepref-register *quicksort-vars*

lemma *le-var-order-rel*:
‹$(\le) = (\lambda x\ y.\ x = y \lor (x, y) \in var\text{-}order\text{-}rel)$›
⟨*proof*⟩

sepref-definition *full-quicksort-vars-impl*
is ‹*full-quicksort-vars*›
:: ‹$monom\text{-}assn^k \to_a monom\text{-}assn$›
⟨*proof*⟩

lemmas *sort-vars-spec-hnr* =
full-quicksort-vars-impl.refine[*FCOMP full-quicksort-sort-vars-spec*]

lemma *string-rel-order-map*:
‹$(x, a) \in string\text{-}rel \Longrightarrow$
$(y, aa) \in string\text{-}rel \Longrightarrow$
$x \le y \longleftrightarrow a \le aa$›
⟨*proof*⟩

lemma *merge-monoms-merge-monoms*:
‹$(merge\text{-}monoms, merge\text{-}monoms) \in monom\text{-}rel \to monom\text{-}rel \to monom\text{-}rel$›
⟨*proof*⟩

lemma *merge-monoms-merge-monoms2*:
‹$(a, b) \in monom\text{-}rel \Longrightarrow (a', b') \in monom\text{-}rel \Longrightarrow$
$(merge\text{-}monoms\ a\ a', merge\text{-}monoms\ b\ b') \in monom\text{-}rel$›
⟨*proof*⟩

lemma *msort-monoms-impl*:
‹$(msort\text{-}monoms\text{-}impl, merge\text{-}monoms\text{-}poly)$
$\in monom\text{-}rel \to monom\text{-}rel$›
⟨*proof*⟩

lemma *merge-sort-monoms-sort-monoms-spec*:
‹$(RETURN\ o\ merge\text{-}monoms\text{-}poly, sort\text{-}coeff) \in \langle Id\rangle list\text{-}rel \to_f \langle\langle Id\rangle list\text{-}rel\rangle nres\text{-}rel$›
⟨*proof*⟩

sepref-register *sort-coeff*

lemma [*sepref-fr-rules*]:
‹(*return o msort-monoms-impl*, *sort-coeff*) ∈ *monom-assn*^{k} $\rightarrow_a$ *monom-assn*›
⟨*proof*⟩

sepref-definition *sort-all-coeffs-impl*
is ‹*sort-all-coeffs*›
:: ‹*poly-assn*^{k} $\rightarrow_a$ *poly-assn*›
⟨*proof*⟩

declare *sort-all-coeffs-impl.refine*[*sepref-fr-rules*]

lemma *merge-coeffs0-alt-def*:
‹(*RETURN o merge-coeffs0*) *p* =
REC$_T$(λ*f p*.
(*case p of*
[] ⇒ *RETURN* []
| [*p*] => *if snd p* = *0 then RETURN* [] *else RETURN* [*p*]
| ((*xs*, *n*) # (*ys*, *m*) # *p*) ⇒
(*if xs* = *ys*
then if n + *m* ≠ *0 then f* ((*xs*, *n* + *m*) # *p*) *else f p*
else if n = *0 then*
do {*p* ← *f* ((*ys*, *m*) # *p*);
RETURN p}
else do {*p* ← *f* ((*ys*, *m*) # *p*);
RETURN ((*xs*, *n*) # *p*)})))
p›
⟨*proof*⟩

Again, Sepref does not understand what is going here.

sepref-definition *merge-coeffs0-impl*
is ‹*RETURN o merge-coeffs0*›
:: ‹*poly-assn*^{k} $\rightarrow_a$ *poly-assn*›
⟨*proof*⟩

declare *merge-coeffs0-impl.refine*[*sepref-fr-rules*]

sepref-definition *fully-normalize-poly-impl*
is ‹*full-normalize-poly*›
:: ‹*poly-assn*^{k} $\rightarrow_a$ *poly-assn*›
⟨*proof*⟩

declare *fully-normalize-poly-impl.refine*[*sepref-fr-rules*]

end

theory *PAC-Version*
imports *Main*
begin

This code was taken from IsaFoR. However, for the AFP, we use the version name *AFP*, instead of a mercurial version.

⟨*ML*⟩

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**
‹$vars\text{-}assn \equiv hs.assn\ string\text{-}assn$›

fun *vars-of-monom-in* **where**
‹$vars\text{-}of\text{-}monom\text{-}in\ []\ \text{-} = True$› |
‹$vars\text{-}of\text{-}monom\text{-}in\ (x\ \#\ xs)\ \mathcal{V} \longleftrightarrow x \in \mathcal{V} \wedge vars\text{-}of\text{-}monom\text{-}in\ xs\ \mathcal{V}$›

fun *vars-of-poly-in* **where**
‹$vars\text{-}of\text{-}poly\text{-}in\ []\ \text{-} = True$› |
‹$vars\text{-}of\text{-}poly\text{-}in\ ((x, \text{-})\ \#\ xs)\ \mathcal{V} \longleftrightarrow vars\text{-}of\text{-}monom\text{-}in\ x\ \mathcal{V} \wedge vars\text{-}of\text{-}poly\text{-}in\ xs\ \mathcal{V}$›

lemma *vars-of-monom-in-alt-def*:
‹$vars\text{-}of\text{-}monom\text{-}in\ xs\ \mathcal{V} \longleftrightarrow set\ xs \subseteq \mathcal{V}$›
⟨*proof*⟩

lemma *vars-llist-alt-def*:
‹$vars\text{-}llist\ xs \subseteq \mathcal{V} \longleftrightarrow vars\text{-}of\text{-}poly\text{-}in\ xs\ \mathcal{V}$›
⟨*proof*⟩

lemma *vars-of-monom-in-alt-def2*:
‹$vars\text{-}of\text{-}monom\text{-}in\ xs\ \mathcal{V} \longleftrightarrow fold\ (\lambda x\ b.\ b \wedge x \in \mathcal{V})\ xs\ True$›
⟨*proof*⟩

sepref-definition *vars-of-monom-in-impl*
is ‹$uncurry\ (RETURN\ oo\ vars\text{-}of\text{-}monom\text{-}in)$›
:: ‹$(list\text{-}assn\ string\text{-}assn)^k *_a vars\text{-}assn^k \rightarrow_a bool\text{-}assn$›
⟨*proof*⟩

declare *vars-of-monom-in-impl.refine*[*sepref-fr-rules*]

lemma *vars-of-poly-in-alt-def2*:
‹$vars\text{-}of\text{-}poly\text{-}in\ xs\ \mathcal{V} \longleftrightarrow fold\ (\lambda(x, \text{-})\ b.\ b \wedge vars\text{-}of\text{-}monom\text{-}in\ x\ \mathcal{V})\ xs\ True$›
⟨*proof*⟩

sepref-definition *vars-of-poly-in-impl*
is ‹$uncurry\ (RETURN\ oo\ vars\text{-}of\text{-}poly\text{-}in)$›
:: ‹$(poly\text{-}assn)^k *_a vars\text{-}assn^k \rightarrow_a bool\text{-}assn$›
⟨*proof*⟩

declare *vars-of-poly-in-impl.refine*[*sepref-fr-rules*]

definition *union-vars-monom* :: ‹*string list* ⇒ *string set* ⇒ *string set*› **where**
‹*union-vars-monom xs* $\mathcal{V}$ = *fold insert xs* $\mathcal{V}$›

definition *union-vars-poly* :: ‹*llist-polynomial* ⇒ *string set* ⇒ *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* $\mathcal{V}$ = $\mathcal{V}$ ∪ *set xs*›
⟨*proof*⟩

lemma *union-vars-poly-alt-def*:
‹*union-vars-poly xs* $\mathcal{V}$ = $\mathcal{V}$ ∪ *vars-llist xs*›
⟨*proof*⟩

sepref-definition *union-vars-monom-impl*
is ‹*uncurry* (*RETURN oo union-vars-monom*)›
:: ‹$monom\text{-}assn^k *_a vars\text{-}assn^d \rightarrow_a vars\text{-}assn$›
⟨*proof*⟩

declare *union-vars-monom-impl.refine*[*sepref-fr-rules*]

sepref-definition *union-vars-poly-impl*
is ‹*uncurry* (*RETURN oo union-vars-poly*)›
:: ‹$poly\text{-}assn^k *_a vars\text{-}assn^d \rightarrow_a vars\text{-}assn$›
⟨*proof*⟩

declare *union-vars-poly-impl.refine*[*sepref-fr-rules*]

hide-const (**open**) *Autoref-Fix-Rel.CONSTRAINT*

fun *status-assn* **where**
‹*status-assn - CSUCCESS CSUCCESS = emp*› |
‹*status-assn - CFOUND CFOUND = emp*› |
‹*status-assn R* (*CFAILED a*) (*CFAILED b*) = *R a b*› |
‹*status-assn - - - = false*›

lemma *SUCCESS-hnr*[*sepref-fr-rules*]:
‹(*uncurry0* (*return CSUCCESS*), *uncurry0* (*RETURN CSUCCESS*)) ∈ $unit\text{-}assn^k \rightarrow_a status\text{-}assn\ R$›
⟨*proof*⟩

lemma *FOUND-hnr*[*sepref-fr-rules*]:
‹(*uncurry0* (*return CFOUND*), *uncurry0* (*RETURN CFOUND*)) ∈ $unit\text{-}assn^k \rightarrow_a status\text{-}assn\ R$›
⟨*proof*⟩

lemma *is-success-hnr*[*sepref-fr-rules*]:
‹*CONSTRAINT is-pure R* ⟹
((*return o is-cfound*), (*RETURN o is-cfound*)) ∈ $(status\text{-}assn\ R)^k \rightarrow_a bool\text{-}assn$›
⟨*proof*⟩

lemma *is-cfailed-hnr*[*sepref-fr-rules*]:
‹*CONSTRAINT is-pure R* ⟹

‹((return o is-cfailed), (RETURN o is-cfailed)) ∈ (status-assn R)^k $\to_a$ bool-assn›
⟨proof⟩

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 $\to_a$ status-assn R›
⟨proof⟩

sepref-definition *add-poly-impl*
is ‹add-poly-l›
:: ‹(poly-assn $\times_a$ poly-assn)^k $\to_a$ poly-assn›
⟨proof⟩

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
(λf (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)›
⟨proof⟩

sepref-definition *mult-monoms-impl*
is ‹uncurry (RETURN oo mult-monoms)›
:: ‹(monom-assn)^k $*_a$ (monom-assn)^k $\to_a$ (monom-assn)›
⟨proof⟩

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 $\to_a$ (monomial-assn)›
⟨proof⟩

lemma *map-append-alt-def2*:
‹(RETURN o (map-append f b)) xs = REC_T

$(\lambda g\ xs.\ case\ xs\ of\ []\ \Rightarrow RETURN\ b$
$\ \ |\ x\ \#\ xs \Rightarrow do\ \{$
$\ \ \ \ \ \ y \leftarrow g\ xs;$
$\ \ \ \ \ \ RETURN\ (f\ x\ \#\ y)$
$\ \ \})\ xs$›
⟨*proof*⟩

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*›
⟨*proof*⟩

declare *map-append-poly-mult-impl.refine*[*sepref-fr-rules*]

TODO *foldl* ($\lambda l\ x.\ l$ @ [*?f x*]) [] *?l* = *map ?f ?l* is the worst possible implementation of map!

sepref-definition *mult-poly-raw-impl*
is ‹*uncurry* (*RETURN oo mult-poly-raw*)›
:: ‹*poly-assn*^{k} $*_a$ *poly-assn*^{k} $\rightarrow_a$ *poly-assn*›
⟨*proof*⟩

declare *mult-poly-raw-impl.refine*[*sepref-fr-rules*]

sepref-definition *mult-poly-impl*
is ‹*uncurry mult-poly-full*›
:: ‹*poly-assn*^{k} $*_a$ *poly-assn*^{k} $\rightarrow_a$ *poly-assn*›
⟨*proof*⟩

declare *mult-poly-impl.refine*[*sepref-fr-rules*]

lemma *inverse-monomial*:
‹*monom-rel*$^{-1}$ $\times_r$ *int-rel* = (*monom-rel* $\times_r$ *int-rel*)$^{-1}$›
⟨*proof*⟩

lemma *eq-poly-rel-eq*[*sepref-import-param*]:
‹((=), (=)) ∈ *poly-rel* → *poly-rel* → *bool-rel*›
⟨*proof*⟩

sepref-definition *weak-equality-l-impl*
is ‹*uncurry weak-equality-l*›
:: ‹*poly-assn*^{k} $*_a$ *poly-assn*^{k} $\rightarrow_a$ *bool-assn*›
⟨*proof*⟩

declare *weak-equality-l-impl.refine*[*sepref-fr-rules*]
sepref-register *add-poly-l mult-poly-full*

abbreviation *raw-string-assn* :: ‹*string* ⇒ *string* ⇒ *assn*› **where**
‹*raw-string-assn* ≡ *list-assn id-assn*›

definition *show-nat* :: ‹*nat* ⇒ *string*› **where**
‹*show-nat i* = *show i*›

lemma [*sepref-import-param*]:
‹(*show-nat*, *show-nat*) ∈ *nat-rel* → ⟨*Id*⟩*list-rel*›
⟨*proof*⟩

lemma *status-assn-pure-conv*:
‹*status-assn* (*id-assn*) *a b* = *id-assn a b*›
⟨*proof*⟩

lemma [*sepref-fr-rules*]:
‹(*uncurry3* (λ*x y*. *return oo* (*error-msg-not-equal-dom x y*)), *uncurry3 check-not-equal-dom-err*) ∈ $poly\text{-}assn^k *_a poly\text{-}assn^k *_a poly\text{-}assn^k *_a poly\text{-}assn^k \rightarrow_a raw\text{-}string\text{-}assn$›
⟨*proof*⟩

lemma [*sepref-fr-rules*]:
‹(*return o* (*error-msg-notin-dom o nat-of-uint64*), *RETURN o error-msg-notin-dom*)
∈ $uint64\text{-}nat\text{-}assn^k \rightarrow_a raw\text{-}string\text{-}assn$›
‹(*return o* (*error-msg-reused-dom o nat-of-uint64*), *RETURN o error-msg-reused-dom*)
∈ $uint64\text{-}nat\text{-}assn^k \rightarrow_a raw\text{-}string\text{-}assn$›
‹(*uncurry* (*return oo* (λ*i*. *error-msg* (*nat-of-uint64 i*))), *uncurry* (*RETURN oo error-msg*))
∈ $uint64\text{-}nat\text{-}assn^k *_a raw\text{-}string\text{-}assn^k \rightarrow_a status\text{-}assn\ raw\text{-}string\text{-}assn$›
‹(*uncurry* (*return oo error-msg*), *uncurry* (*RETURN oo error-msg*))
∈ $nat\text{-}assn^k *_a raw\text{-}string\text{-}assn^k \rightarrow_a status\text{-}assn\ raw\text{-}string\text{-}assn$›
⟨*proof*⟩

sepref-definition *check-addition-l-impl*
is ‹*uncurry6 check-addition-l*›
:: ‹$poly\text{-}assn^k *_a polys\text{-}assn^k *_a vars\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k *_a poly\text{-}assn^k \rightarrow_a status\text{-}assn\ raw\text{-}string\text{-}assn$›
⟨*proof*⟩

declare *check-addition-l-impl.refine*[*sepref-fr-rules*]

sepref-register *check-mult-l-dom-err*

definition *check-mult-l-dom-err-impl* **where**
‹*check-mult-l-dom-err-impl pd p ia i* =
(*if pd then* ''*The polynomial with id* '' @ *show* (*nat-of-uint64 p*) @ '' *was not found*'' *else* '''') @
(*if ia then* ''*The id of the resulting id* '' @ *show* (*nat-of-uint64 i*) @ '' *was already given*'' *else* '''')›

definition *check-mult-l-mult-err-impl* **where**
‹*check-mult-l-mult-err-impl p q pq r* =
''*Multiplying* '' @ *show p* @ '' *by* '' @ *show q* @ '' *gives* '' @ *show pq* @ '' *and not* '' @ *show r*›

lemma [*sepref-fr-rules*]:
‹(*uncurry3* ((λ*x y*. *return oo* (*check-mult-l-dom-err-impl x y*))),
uncurry3 (*check-mult-l-dom-err*)) ∈ $bool\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k *_a bool\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow_a raw\text{-}string\text{-}assn$›
⟨*proof*⟩

lemma [*sepref-fr-rules*]:
‹(*uncurry3* ((λ*x y*. *return oo* (*check-mult-l-mult-err-impl x y*))),
uncurry3 (*check-mult-l-mult-err*)) ∈ $poly\text{-}assn^k *_a poly\text{-}assn^k *_a poly\text{-}assn^k *_a poly\text{-}assn^k \rightarrow_a raw\text{-}string\text{-}assn$›
⟨*proof*⟩

sepref-definition *check-mult-l-impl*
is ‹*uncurry6 check-mult-l*›
:: ‹$poly\text{-}assn^k *_a polys\text{-}assn^k *_a vars\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k *_a poly\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k *_a poly\text{-}assn^k \rightarrow_a status\text{-}assn\ raw\text{-}string\text{-}assn$›
⟨*proof*⟩

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*)) ∈ $uint64\text{-}nat\text{-}assn^k \rightarrow_a raw\text{-}string\text{-}assn$›
⟨*proof*⟩

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*)) ∈ $poly\text{-}assn^k \rightarrow_a raw\text{-}string\text{-}assn$›
⟨*proof*⟩

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*)) ∈ $string\text{-}assn^k *_a poly\text{-}assn^k *_a poly\text{-}assn^k *_a poly\text{-}assn^k \rightarrow_a raw\text{-}string\text{-}assn$›
⟨*proof*⟩

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*)) ∈ $string\text{-}assn^k *_a poly\text{-}assn^k \rightarrow_a raw\text{-}string\text{-}assn$›
⟨*proof*⟩

sepref-register *check-extension-l-dom-err fmlookup′*
check-extension-l-side-cond-err check-extension-l-no-new-var-err
check-extension-l-new-var-multiple-err

definition *uminus-poly* :: ‹*llist-polynomial* ⇒ *llist-polynomial*› **where**
‹*uminus-poly* $p' = map\ (\lambda(a,\ b).\ (a,\ -\ b))\ p'$›

sepref-register *uminus-poly*
lemma [*sepref-import-param*]:
‹(*map* (λ(*a*, *b*). (*a*, − *b*)), *uminus-poly*) ∈ *poly-rel* → *poly-rel*›
⟨*proof*⟩

sepref-register *vars-of-poly-in*
weak-equality-l

lemma [*safe-constraint-rules*]:
‹*Sepref-Constraints.CONSTRAINT single-valued* (*the-pure monomial-assn*)› **and**
single-valued-the-monomial-assn:
‹*single-valued* (*the-pure monomial-assn*)›
‹*single-valued* $((\textit{the-pure monomial-assn})^{-1})$›
⟨*proof*⟩

sepref-definition *check-extension-l-impl*
is ‹*uncurry5 check-extension-l*›
:: ‹$\textit{poly-assn}^k *_a \textit{polys-assn}^k *_a \textit{vars-assn}^k *_a \textit{uint64-nat-assn}^k *_a \textit{string-assn}^k *_a \textit{poly-assn}^k \rightarrow_a$
status-assn raw-string-assn›
⟨*proof*⟩

declare *check-extension-l-impl.refine*[*sepref-fr-rules*]

sepref-definition *check-del-l-impl*
is ‹*uncurry2 check-del-l*›
:: ‹$\textit{poly-assn}^k *_a \textit{polys-assn}^k *_a \textit{uint64-nat-assn}^k \rightarrow_a$ *status-assn raw-string-assn*›
⟨*proof*⟩

lemmas [*sepref-fr-rules*] = *check-del-l-impl.refine*

abbreviation *pac-step-rel* **where**
‹*pac-step-rel* ≡ *p2rel* (⟨*Id*, ⟨*monomial-rel*⟩*list-rel*, *Id*⟩ *pac-step-rel-raw*)›

sepref-register *PAC-Polynomials-Operations.normalize-poly*
pac-src1 pac-src2 new-id pac-mult case-pac-step check-mult-l
check-addition-l check-del-l check-extension-l

lemma *pac-step-rel-assn-alt-def2*:
‹*hn-ctxt* (*pac-step-rel-assn nat-assn poly-assn id-assn*) *b bi* =
hn-val
(*p2rel*
(⟨*nat-rel*, *poly-rel*, *Id* :: (*string* × -) *set*⟩*pac-step-rel-raw*)) *b bi*›
⟨*proof*⟩

lemma *is-AddD-import*[*sepref-fr-rules*]:
assumes ‹*CONSTRAINT is-pure K*› ‹*CONSTRAINT is-pure V*›

shows
‹(*return o pac-res*, *RETURN o pac-res*) $\in$ [λx. *is-Add x* $\vee$ *is-Mult x* $\vee$ *is-Extension x*]$_a$ (*pac-step-rel-assn K V R*)^k $\rightarrow$ *V*›
‹(*return o pac-src1*, *RETURN o pac-src1*) $\in$ [λx. *is-Add x* $\vee$ *is-Mult x* $\vee$ *is-Del x*]$_a$ (*pac-step-rel-assn K V R*)^k $\rightarrow$ *K*›
‹(*return o new-id*, *RETURN o new-id*) $\in$ [λx. *is-Add x* $\vee$ *is-Mult x* $\vee$ *is-Extension x*]$_a$ (*pac-step-rel-assn K V R*)^k $\rightarrow$ *K*›
‹(*return o is-Add*, *RETURN o is-Add*) $\in$ (*pac-step-rel-assn K V R*)^k $\rightarrow_a$ *bool-assn*›
‹(*return o is-Mult*, *RETURN o is-Mult*) $\in$ (*pac-step-rel-assn K V R*)^k $\rightarrow_a$ *bool-assn*›
‹(*return o is-Del*, *RETURN o is-Del*) $\in$ (*pac-step-rel-assn K V R*)^k $\rightarrow_a$ *bool-assn*›
‹(*return o is-Extension*, *RETURN o is-Extension*) $\in$ (*pac-step-rel-assn K V R*)^k $\rightarrow_a$ *bool-assn*›
⟨*proof*⟩

lemma [*sepref-fr-rules*]:
‹*CONSTRAINT is-pure K* $\Longrightarrow$
(*return o pac-src2*, *RETURN o pac-src2*) $\in$ [λx. *is-Add x*]$_a$ (*pac-step-rel-assn K V R*)^k $\rightarrow$ *K*›
‹*CONSTRAINT is-pure V* $\Longrightarrow$
(*return o pac-mult*, *RETURN o pac-mult*) $\in$ [λx. *is-Mult x*]$_a$ (*pac-step-rel-assn K V R*)^k $\rightarrow$ *V*›
‹*CONSTRAINT is-pure R* $\Longrightarrow$
(*return o new-var*, *RETURN o new-var*) $\in$ [λx. *is-Extension x*]$_a$ (*pac-step-rel-assn K V R*)^k $\rightarrow$ *R*›
⟨*proof*⟩

lemma *is-Mult-lastI*:
‹$\neg$ *is-Add b* $\Longrightarrow$ $\neg$*is-Mult b* $\Longrightarrow$ $\neg$*is-Extension b* $\Longrightarrow$ *is-Del b*›
⟨*proof*⟩

sepref-register *is-cfailed is-Del*

definition *PAC-checker-l-step'* :: - **where**
‹*PAC-checker-l-step' a b c d* = *PAC-checker-l-step a* (*b*, *c*, *d*)›

lemma *PAC-checker-l-step-alt-def*:
‹*PAC-checker-l-step a bcd e* = (*let* (*b*,*c*,*d*) = *bcd in PAC-checker-l-step' a b c d e*)›
⟨*proof*⟩

sepref-decl-intf (*'k*) *acode-status* **is** (*'k*) *code-status*
sepref-decl-intf (*'k*, *'b*, *'lbl*) *apac-step* **is** (*'k*, *'b*, *'lbl*) *pac-step*

sepref-register *merge-cstatus full-normalize-poly new-var is-Add*

lemma *poly-rel-the-pure*:
‹*poly-rel* = *the-pure poly-assn*› **and**
nat-rel-the-pure:
‹*nat-rel* = *the-pure nat-assn*› **and**
WTF-RF: ‹*pure* (*the-pure nat-assn*) = *nat-assn*›
⟨*proof*⟩

lemma [*safe-constraint-rules*]:
‹*CONSTRAINT IS-LEFT-UNIQUE uint64-nat-rel*› **and**
single-valued-uint64-nat-rel[*safe-constraint-rules*]:
‹*CONSTRAINT single-valued uint64-nat-rel*›
⟨*proof*⟩

sepref-definition *check-step-impl*
is ‹*uncurry4 PAC-checker-l-step'*›

:: ‹*poly-assn*^{k} $*_a$ (*status-assn raw-string-assn*)^d $*_a$ *vars-assn*^{d} $*_a$ *polys-assn*^{d} $*_a$ (*pac-step-rel-assn* (*uint64-nat-assn*) *poly-assn* (*string-assn* :: *string* ⇒ -))^d $\rightarrow_a$
status-assn raw-string-assn $\times_a$ *vars-assn* $\times_a$ *polys-assn*›
⟨*proof*⟩

declare *check-step-impl.refine*[*sepref-fr-rules*]

sepref-register *PAC-checker-l-step PAC-checker-l-step′ fully-normalize-poly-impl*

definition *PAC-checker-l′* **where**
‹*PAC-checker-l′ p* $\mathcal{V}$ *A status steps* = *PAC-checker-l p* ($\mathcal{V}$, *A*) *status steps*›

lemma *PAC-checker-l-alt-def*:
‹*PAC-checker-l p* $\mathcal{V}$*A status steps* =
(*let* ($\mathcal{V}$, *A*) = $\mathcal{V}$*A in PAC-checker-l′ p* $\mathcal{V}$ *A status steps*)›
⟨*proof*⟩

sepref-definition *PAC-checker-l-impl*
is ‹*uncurry4 PAC-checker-l′*›
:: ‹*poly-assn*^{k} $*_a$ *vars-assn*^{d} $*_a$ *polys-assn*^{d} $*_a$ (*status-assn raw-string-assn*)^d $*_a$
(*list-assn* (*pac-step-rel-assn* (*uint64-nat-assn*) *poly-assn string-assn*))^k $\rightarrow_a$
status-assn raw-string-assn $\times_a$ *vars-assn* $\times_a$ *polys-assn*›
⟨*proof*⟩

declare *PAC-checker-l-impl.refine*[*sepref-fr-rules*]

abbreviation *polys-assn-input* **where**
‹*polys-assn-input* ≡ *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} $\rightarrow_a$ *raw-string-assn*›
⟨*proof*⟩

MLton is not able to optimise the calls to pow.

lemma *pow-2-64*: ‹(*2*::*nat*) ^ *64* = *18446744073709551616*›
⟨*proof*⟩

sepref-register *upper-bound-on-dom op-fmap-empty*

sepref-definition *remap-polys-l-impl*
is ‹*uncurry2 remap-polys-l2*›
:: ‹*poly-assn*^{k} $*_a$ *vars-assn*^{d} $*_a$ *polys-assn-input*^{d} $\rightarrow_a$
status-assn raw-string-assn $\times_a$ *vars-assn* $\times_a$ *polys-assn*›
⟨*proof*⟩

lemma *remap-polys-l2-remap-polys-l*:
‹(*uncurry2 remap-polys-l2*, *uncurry2 remap-polys-l*) ∈ (*Id* $\times_r$ ⟨*Id*⟩*set-rel*) $\times_r$ *Id* $\rightarrow_f$ ⟨*Id*⟩*nres-rel*›

⟨*proof*⟩

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} $\rightarrow_a$
status-assn raw-string-assn $\times_a$ *vars-assn* $\times_a$ *polys-assn*›
⟨*proof*⟩

sepref-register *remap-polys-l*

sepref-definition *full-checker-l-impl*
is ‹*uncurry2 full-checker-l*›
:: ‹*poly-assn*^{k} $*_a$ *polys-assn-input*^{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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

sepref-definition *PAC-empty-impl*
is ‹*uncurry0* (*RETURN fmempty*)›
:: ‹*unit-assn*^{k} $\rightarrow_a$ *polys-assn-input*›
⟨*proof*⟩

sepref-definition *empty-vars-impl*
is ‹*uncurry0* (*RETURN* {})›
:: ‹*unit-assn*^{k} $\rightarrow_a$ *vars-assn*›
⟨*proof*⟩

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 (*λh x. h ∗ 33 + uint32-of-int* (*of-char x*)) *5381*
(*raw-explode s*)›

lemmas [*code*] =
hashcode-literal-def[*unfolded String.explode-code*
unsafe-asciis-of-literal-def[*symmetric*]]

definition *uint32-of-char* **where**
[*symmetric*, *code-unfold*]: ‹*uint32-of-char x* = *uint32-of-int* (*int-of-char x*)›

code-printing
constant *uint32-of-char* ⇀
(*SML*) !(*Word32.fromInt* /*o* (*Char.ord*))

lemma [*code*]: ‹*hashcode s* = *hashcode-literal′ s*›
⟨*proof*⟩

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* $\times_r$ *polys-rel*›
⟨*proof*⟩

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*))
$\in$ (*full-poly-assn*)^k $*_a$ (*full-poly-input-assn*)^d $*_a$ (*fully-pac-assn*)^k $\rightarrow_a$ *hr-comp*
  (*code-status-assn* $\times_a$ *full-vars-assn* $\times_a$ *hr-comp polys-assn*
  (⟨*nat-rel*, *sorted-poly-rel O mset-poly-rel*⟩*fmap-rel*))
  $\{((st, G), st', G').$
  $st = st' \wedge (st \neq FAILED \longrightarrow (G, G') \in Id \times_r polys\text{-}rel)\}$›
⟨*proof*⟩

It would be more efficient to move the parsing to Isabelle, as this would be more memory efficient (and also reduce the TCB). But now comes the fun part: It cannot work. A stream (of a file) is consumed by side effects. Assume that this would work. The code could look like:
Let (*read-file file*) *f*
This code is equal to (in the HOL sense of equality): *let - = read-file file in Let* (*read-file file*) *f*
However, as an hypothetical *read-file* changes the underlying stream, we would get the next token. Remark that this is already a weird point of ML compilers. Anyway, I see currently two solutions to this problem:

1. The meta-argument: use it only in the Refinement Framework in a setup where copies are disallowed. Basically, this works because we can express the non-duplication constraints on the type level. However, we cannot forbid people from expressing things directly at the HOL level.

2. On the target language side, model the stream as the stream and the position. Reading takes two arguments. First, the position to read. Second, the stream (and the current position) to read. If the position to read does not match the current position, return an error. This would fit the correctness theorem of the code generation (roughly "if it terminates without exception, the answer is the same"), but it is still unsatisfactory.

end

definition φ :: ‹*string* ⇒ *nat*› **where**
‹φ = (*SOME* φ. *bij* φ)›

lemma *bij*-φ: ‹*bij* φ›
⟨*proof*⟩

global-interpretation *PAC*: *poly-embed* **where**
$\varphi = \varphi$
⟨*proof*⟩

The full correctness theorem is $($*uncurry2 full-checker-l-impl*, *uncurry2* $(\lambda$*spec A -. PAC-checker-specification spec A*$)) \in$ *PAC.full-poly-assn*$^k *_a$ *PAC.full-poly-input-assn*$^d *_a$ *PAC.fully-pac-assn*$^k \rightarrow_a$ *hr-comp* (*PAC.code-status-assn* $\times_a$ *PAC.full-vars-assn* $\times_a$ *hr-comp polys-assn* (⟨*nat-rel*, *sorted-poly-rel O PAC.mset-poly-rel*⟩*fmap-rel*)) $\{((st, G), st', G').\ st = st' \wedge (st \neq FAILED \longrightarrow (G, G') \in Id \times_r polys\text{-}rel)\}$.

end

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.