PAC Checker

Mathias Fleury and Daniela Kaufmann

February 10, 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 *EPAC-Specification*
imports *PAC-Checker.PAC-More-Poly*
PAC-Checker.PAC-Specification
begin

end

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

PAC-Checker.PAC-Checker-Specification
begin

1 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.

1.1 Algorithm

datatype ($'a$, $'b$, $'lbls$) *pac-step* =
  CL (*pac-srcs*: ‹($'a \times 'lbls$) *list*›) (*new-id*: $'lbls$) (*pac-res*: $'a$) |
  Extension (*new-id*: $'lbls$) (*new-var*: $'b$) (*pac-res*: $'a$) |
  Del (*pac-src1*: $'lbls$)

definition *check-linear-comb* :: ‹(*nat*, *int mpoly*) *fmap* ⇒ *nat set* ⇒ (*int mpoly* × *nat*) *list* ⇒ *nat* ⇒ *int mpoly* ⇒ *bool nres*› **where**
  ‹*check-linear-comb* $\mathcal{A}$ $\mathcal{V}$ *xs n r* = *SPEC*(λ*b*. *b* ⟶ (∀ *i* ∈ *set xs*. *snd i* ∈# *dom-m* $\mathcal{A}$ ∧ *vars* (*fst i*) ⊆ $\mathcal{V}$) ∧ *n* ∉# *dom-m* $\mathcal{A}$ ∧
  vars r ⊆ $\mathcal{V}$ ∧ *xs* ≠ [] ∧ (∑ (*p*,*n*) ∈#*mset xs*. *the* (*fmlookup* $\mathcal{A}$ *n*) ∗ *p*) − *r* ∈ *ideal polynomial-bool*)›

lemma *PAC-Format-LC*:
  assumes
    i: ‹(($\mathcal{V}$, *A*), $\mathcal{V}_B$, *B*) ∈ *polys-rel-full*› **and**
    st: ‹$PAC\text{-}Format^{**}$ ($\mathcal{V}_0$, A_0) ($\mathcal{V}$, *B*)› **and**
    vars: ‹∀ *i*∈#*x11*. *snd i* ∈# *dom-m A* ∧ *vars* (*fst i*) ⊆ $\mathcal{V}$› **and**
    AV: ‹⋃ (*vars* ‘ *set-mset* (*ran-m A*)) ⊆ $\mathcal{V}$› **and**
    fin: ‹*x11* ≠ {#}› **and**
   r: ‹(∑ *x*∈#*x11*. *case x of* (*p*, *n*) ⇒ *the* (*fmlookup A n*) ∗ *p*) − *r* ∈ *More-Modules.ideal polynomial-bool*›
    ‹*vars r* ⊆ $\mathcal{V}$›
  shows ‹$PAC\text{-}Format^{**}$ ($\mathcal{V}$, *B*) ($\mathcal{V}$, *add-mset r B*)›
⟨*proof*⟩

definition *PAC-checker-step-inv* **where**
  ‹*PAC-checker-step-inv spec stat* $\mathcal{V}$ *A* ⟷
  (∀ *i*∈#*dom-m A*. *vars* (*the* (*fmlookup A i*)) ⊆ $\mathcal{V}$) ∧
  vars spec ⊆ $\mathcal{V}$›

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

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*
     CL - - - ⇒
  do {

ASSERT(*PAC-checker-step-inv spec stat* $\mathcal{V}$ *A*);
r ← *normalize-poly-spec* (*pac-res st*);
eq ← *check-linear-comb A* $\mathcal{V}$ (*pac-srcs 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 {
ASSERT(*PAC-checker-step-inv spec stat* $\mathcal{V}$ *A*);
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*))
}
| *Extension* - - - ⇒
do {
ASSERT(*PAC-checker-step-inv spec stat* $\mathcal{V}$ *A*);
r ← *normalize-poly-spec* (*pac-res st*);
(*eq*) ← *check-extension-precalc A* $\mathcal{V}$ (*new-id st*) (*new-var st*) *r*;
if eq
then do {
r0 ← *SPEC*(λ*r0*. *r0* = (*r* − *Var* (*new-var st*)) ∧
vars r0 = *vars* (*r*) ∪ {*new-var st*});
RETURN (*stat*,
insert (*new-var st*) $\mathcal{V}$, *fmupd* (*new-id st*) (*r0*) *A*)}
else RETURN (*FAILED*, ($\mathcal{V}$, *A*))
}
)›

lemma *PAC-checker-step-PAC-checker-specification2*:
fixes *a* :: ‹*status*›
assumes *AB*: ‹(($\mathcal{V}$, *A*),($\mathcal{V}_B$, *B*)) ∈ *polys-rel-full*› **and**
‹¬*is-failed a*› **and**
[*simp*,*intro*]: ‹*a* = *FOUND* ⟹ *spec* ∈ *pac-ideal* (*set-mset* A_0)› **and**
A_0B: ‹*PAC-Format*** ($\mathcal{V}_0$, A_0) ($\mathcal{V}$, *B*)› **and**
$spec_0$: ‹*vars spec* ⊆ $\mathcal{V}_0$› **and**
vars-A_0: ‹⋃ (*vars* ‘ *set-mset* A_0) ⊆ $\mathcal{V}_0$›
shows ‹*PAC-checker-step spec* (*a*, ($\mathcal{V}$, *A*)) *st* ≤ ⇓ (*status-rel* $\times_r$ *polys-rel-full*) (*PAC-checker-specification-step2* ($\mathcal{V}_0$, A_0) *spec* ($\mathcal{V}$, *B*))›
⟨*proof*⟩

definition *PAC-checker*
:: ‹*int-poly* ⇒ *fpac-step* ⇒ *status* ⇒ (*int-poly*, *nat*, *nat*) *pac-step list* ⇒
(*status* × *fpac-step*) *nres*›
where
‹*PAC-checker spec A b st* = *do* {
(*S*, -) ← $WHILE_T$
(λ((*b* :: *status*, *A* :: *fpac-step*), *st*). ¬*is-failed b* ∧ *st* ≠ [])
(λ((*bA*), *st*). *do* {
ASSERT(*st* ≠ []);
S ← *PAC-checker-step spec* (*bA*) (*hd st*);
RETURN (*S*, *tl st*)

})
((*b*, *A*), *st*);
RETURN S
}›

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

1.2 Full Checker

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

lemma *full-checker-spec*:
assumes ‹(*A*, *A*′) ∈ *polys-rel*›
shows
‹*full-checker spec A pac* ≤ ⇓{((*st*, *G*), (*st*′, *G*′)). (*st*, *st*′) ∈ *status-rel* ∧
(*st* ≠ *FAILED* ⟶ (*G*, *G*′) ∈ *polys-rel-full*)}
(*PAC-checker-specification spec* (*A*′))›
⟨*proof*⟩

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

end

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

begin

hide-const (**open**) *PAC-Checker-Specification.PAC-checker-step PAC-Checker.PAC-checker-l PAC-Checker-Specification.PAC-checker*
hide-fact (**open**) *PAC-Checker-Specification.PAC-checker-step-def PAC-Checker.PAC-checker-l-def PAC-Checker-Specification.PAC-checker-def*

lemma *vars-llist*[*simp*]:
‹*vars-llist* [] = {}›
‹*vars-llist* (*xs* @ *ys*) = *vars-llist xs* ∪ *vars-llist ys*›
‹*vars-llist* (*x* # *ys*) = *set* (*fst x*) ∪ *vars-llist ys*›
⟨*proof*⟩

2 Executable Checker

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

2.1 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).

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* (*CL p i r*) (*CL p' i' r'*) ⟷
(*p*, *p'*) ∈ ⟨*R2* $\times_r$ *R1*⟩*list-rel* ∧ (*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* (*CL p i r*) (*CL p' i' r'*) =
list-assn (*R2* $\times_a$ *R1*) *p p'* ∗ *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*
(*CL p i r*, *CL p' i' r'*) ⇒
list-assn (*R2* $\times_a$ *R1*) *p p'* ∗ *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

Linear Combination **definition** *check-linear-combi-l-pre-err* :: ‹*nat* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ *string nres*› **where**
‹*check-linear-combi-l-pre-err r - - - = SPEC (λ-. True)*›

definition *check-linear-combi-l-dom-err* :: ‹*llist-polynomial* ⇒ *nat* ⇒ *string nres*› **where**
‹*check-linear-combi-l-dom-err p r = SPEC (λ-. True)*›

definition *check-linear-combi-l-mult-err* :: ‹*llist-polynomial* ⇒ *llist-polynomial* ⇒ *string nres*› **where**
‹*check-linear-combi-l-mult-err pq r = SPEC (λ-. True)*›
definition *linear-combi-l-pre* **where**
‹*linear-combi-l-pre i A 𝒱 xs* ⟷
(∀ *i*∈#*dom-m A*. *vars-llist* (*the* (*fmlookup A i*)) ⊆ 𝒱)›

definition *linear-combi-l* **where**

```
‹linear-combi-l i A 𝒱 xs = do {
    ASSERT(linear-combi-l-pre i A 𝒱 xs);
    WHILE_T
      (λ(p, xs, err). xs ≠ [] ∧ ¬is-cfailed err)
      (λ(p, xs, -). do {
         ASSERT(xs ≠ []);
         ASSERT(vars-llist p ⊆ 𝒱);
         let (q_0 :: llist-polynomial, i) = hd xs;
         if (i ∉# dom-m A ∨ ¬(vars-llist q_0 ⊆ 𝒱))
         then do {
           err ← check-linear-combi-l-dom-err q_0 i;
           RETURN (p, xs, error-msg i err)
         } else do {
           ASSERT(fmlookup A i ≠ None);
           let r = the (fmlookup A i);
           ASSERT(vars-llist r ⊆ 𝒱);
           if q_0 = [([], 1)]
           then do {
             pq ← add-poly-l (p, r);
             RETURN (pq, tl xs, CSUCCESS)
           }
           else do {
             q ← full-normalize-poly (q_0);
             ASSERT(vars-llist q ⊆ 𝒱);
             pq ← mult-poly-full q r;
             ASSERT(vars-llist pq ⊆ 𝒱);
             pq ← add-poly-l (p, pq);
             RETURN (pq, tl xs, CSUCCESS)
          }
         }
      })
     ([], xs, CSUCCESS)
  }›
```

definition *check-linear-combi-l* **where**

```
‹check-linear-combi-l spec A 𝒱 i xs r = do{
  b← RES(UNIV::bool set);
  if b ∨ i ∈# dom-m A ∨ xs = [] ∨ ¬(vars-llist r ⊆ 𝒱)
  then do {
```

```
    err ← check-linear-combi-l-pre-err i (i ∈# dom-m A) (xs = []) (¬(vars-llist r ⊆ 𝒱));
    RETURN (error-msg i err)
  }
  else do {
    (p, -, err) ← linear-combi-l i A 𝒱 xs;
    if (is-cfailed err)
    then do {
      RETURN err
    }
    else do {
      b ← weak-equality-l p r;
      b′ ← weak-equality-l r spec;
      if b then (if b′ then RETURN CFOUND else RETURN CSUCCESS) else do {
        c ← check-linear-combi-l-mult-err p r;
        RETURN (error-msg i c)
      }
    }
  }}›
```

Deletion checking **definition** *check-extension-l-side-cond-err*

```
  :: ‹string ⇒ llist-polynomial ⇒ llist-polynomial ⇒ string nres›
where
  ‹check-extension-l-side-cond-err v p′ q = SPEC (λ-. True)›
```

definition (**in** −)*check-extension-l2*

```
  :: ‹- ⇒ - ⇒ string set ⇒ nat ⇒ string ⇒ llist-polynomial ⇒ (string code-status) nres›
where
‹check-extension-l2 spec A 𝒱 i v p′ = do {
  b ← SPEC(λb. b ⟶ i ∉# dom-m A ∧ v ∉ 𝒱);
  if ¬b
  then do {
    c ← check-extension-l-dom-err i;
    RETURN (error-msg i c)
  } else do {
      let p′ = 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 {
         ASSERT(vars-llist p′ ⊆ 𝒱);
         p2 ← mult-poly-full p′ p′;
         ASSERT(vars-llist p2 ⊆ 𝒱);
         let p′ = map (λ(a,b). (a, −b)) p′;
         ASSERT(vars-llist p′ ⊆ 𝒱);
         q ← add-poly-l (p2, p′);
         ASSERT(vars-llist q ⊆ 𝒱);
         eq ← weak-equality-l q [];
         if eq then do {
           RETURN (CSUCCESS)
         } else do {
          c ← check-extension-l-side-cond-err v p′ q;
          RETURN (error-msg i c)
```

```
        }
      }
    }
          }›
```

Extension checking

Step checking **definition** *PAC-checker-l-step-inv* **where**

```
‹PAC-checker-l-step-inv spec st′ 𝒱 A ⟷
(∀ i∈#dom-m A. vars-llist (the (fmlookup A i)) ⊆ 𝒱)›
```

definition *PAC-checker-l-step* :: ‹- ⇒ string code-status × string set × - ⇒ (llist-polynomial, string, nat) pac-step ⇒ -› **where**

```
‹PAC-checker-l-step = (λspec (st′, 𝒱, A) st. do {
   ASSERT(¬is-cfailed st′);
   ASSERT(PAC-checker-l-step-inv spec st′ 𝒱 A);
   case st of
    CL - - - ⇒
      do {
       ASSERT (PAC-checker-l-step-inv spec st′ 𝒱 A);
        r ← full-normalize-poly (pac-res st);
       eq ← check-linear-combi-l spec A 𝒱 (new-id st) (pac-srcs st) r;
       let - = eq;
       if ¬is-cfailed eq
       then RETURN (merge-cstatus st′ eq,
         𝒱, fmupd (new-id st) r A)
       else RETURN (eq, 𝒱, A)
  }
  | Del - ⇒
      do {
       ASSERT (PAC-checker-l-step-inv spec st′ 𝒱 A);
       eq ← check-del-l spec A (pac-src1 st);
       let - = eq;
       if ¬is-cfailed eq
       then RETURN (merge-cstatus st′ eq, 𝒱, fmdrop (pac-src1 st) A)
       else RETURN (eq, 𝒱, A)
  }
  | Extension - - - ⇒
      do {
       ASSERT (PAC-checker-l-step-inv spec st′ 𝒱 A);
        r ← full-normalize-poly (pac-res st);
       eq ← check-extension-l2 spec A 𝒱 (new-id st) (new-var st) r;
       if ¬is-cfailed eq
       then do {
         ASSERT(new-var st ∉ vars-llist r ∧ vars-llist r ⊆ 𝒱);
         r′ ← add-poly-l ([([new-var st], −1)], r);
         RETURN (st′,
           insert (new-var st) 𝒱, fmupd (new-id st) r′ A)}
       else RETURN (eq, 𝒱, A)
  }}
)›
```

lemma *pac-step-rel-raw-def*:

```
‹⟨K, V, R⟩ pac-step-rel-raw = pac-step-rel-raw K V R›
⟨proof⟩
```

2.2 Correctness

We now enter the locale to reason about polynomials directly.

context *poly-embed*
begin

lemma (**in** −) *vars-llist-merge-coeffsD*:
‹$x \in$ *vars-llist* (*merge-coeffs pa*) $\Longrightarrow x \in$ *vars-llist pa*›
⟨*proof*⟩
lemma (**in** −) *add-nset-list-rel-add-mset-iff*:
‹(*pa*, *add-mset* (*aa*) (*ys*)) $\in \langle R\rangle$*list-rel O* $\{(c, a).\ a =$ *mset c*$\}$ $\longleftrightarrow$
$(\exists\, pa_1\ pa_2\ x.\ pa = pa_1\ @\ x\ \#\ pa_2 \wedge (pa_1\ @\ pa_2,\ ys) \in \langle R\rangle$*list-rel O* $\{(c, a).\ a =$ *mset c*$\} \wedge$
$(x, aa) \in R)$›
⟨*proof*⟩

lemma (**in** −) *sorted-poly-rel-vars-llist2*:
‹(*pa*, *r*) $\in$ *sorted-poly-rel* $\Longrightarrow$ (*vars-llist pa*) $= \bigcup$ (*set-mset* ‘ *fst* ‘ *set-mset r*)›
⟨*proof*⟩
lemma (**in** −)*normalize-poly-p-vars*: ‹*normalize-poly-p p q* $\Longrightarrow \bigcup$ (*set-mset* ‘ *fst* ‘ *set-mset q*) $\subseteq \bigcup$ (*set-mset* ‘ *fst* ‘ *set-mset p*)›
⟨*proof*⟩

lemma (**in** −)*rtranclp-normalize-poly-p-vars*: ‹*normalize-poly-p*** *p q* $\Longrightarrow \bigcup$ (*set-mset* ‘ *fst* ‘ *set-mset q*) $\subseteq \bigcup$ (*set-mset* ‘ *fst* ‘ *set-mset p*)›
⟨*proof*⟩

lemma *normalize-poly-normalize-poly-p2*:
assumes ‹(*p*, p') $\in$ *unsorted-poly-rel*›
shows ‹*normalize-poly p* $\le \Downarrow\{$(*xs*,*ys*). (*xs*,*ys*)$\in$*sorted-poly-rel* $\wedge$ *vars-llist xs* $\subseteq$ *vars-llist p*$\}$ (*SPEC* (λr. *normalize-poly-p*** p' *r*))›
⟨*proof*⟩

lemma (**in** −)*vars-llist-mult-poly-raw*: ‹*vars-llist* (*mult-poly-raw p q*) $\subseteq$ *vars-llist p* $\cup$ *vars-llist q*›
⟨*proof*⟩

lemma *mult-poly-full-mult-poly-p′2*:
assumes ‹(*p*, p') $\in$ *sorted-poly-rel*› ‹(*q*, q') $\in$ *sorted-poly-rel*›
shows ‹*mult-poly-full p q* $\le \Downarrow$ $\{$(*xs*,*ys*). (*xs*,*ys*)$\in$*sorted-poly-rel* $\wedge$ *vars-llist xs* $\subseteq$ *vars-llist p* $\cup$ *vars-llist q*$\}$ (*mult-poly-p′* p' q')›
⟨*proof*⟩

lemma *mult-poly-full-spec2*:
assumes
‹(*p*, p'') $\in$ *sorted-poly-rel O mset-poly-rel*› **and**
‹(*q*, q'') $\in$ *sorted-poly-rel O mset-poly-rel*›
shows
‹*mult-poly-full p q* $\le \Downarrow\{$(*xs*,*ys*). (*xs*,*ys*)$\in$*sorted-poly-rel O mset-poly-rel* $\wedge$ *vars-llist xs* $\subseteq$ *vars-llist p* $\cup$ *vars-llist q*$\}$
(*SPEC* ($\lambda s.\ s - p'' * q'' \in$ *ideal polynomial-bool*))›
⟨*proof*⟩

lemma *mult-poly-full-mult-poly-spec*:
assumes ‹(*p*, p') $\in$ *sorted-poly-rel O mset-poly-rel*› ‹(*q*, q') $\in$ *sorted-poly-rel O mset-poly-rel*›
shows ‹*mult-poly-full p q* $\le \Downarrow$ $\{$(*xs*,*ys*). (*xs*,*ys*)$\in$*sorted-poly-rel O mset-poly-rel* $\wedge$ *vars-llist xs* $\subseteq$ *vars-llist p* $\cup$ *vars-llist q*$\}$ (*mult-poly-spec* p' q')›

⟨*proof*⟩

lemma *vars-llist-merge-coeff0*: ‹*vars-llist* (*merge-coeffs0 paa*) ⊆ *vars-llist paa*›
⟨*proof*⟩

lemma *sort-poly-spec-id′2*:
assumes ‹(p, p') ∈ *unsorted-poly-rel-with0*›
shows ‹*sort-poly-spec p* ≤ ⇓ {(*xs*, *ys*). (*xs*, *ys*) ∈ *sorted-repeat-poly-rel-with0* ∧ *vars-llist xs* ⊆ *vars-llist p*} (*RETURN* p')›
⟨*proof*⟩

lemma *sort-all-coeffs-unsorted-poly-rel-with02*:
assumes ‹(p, p') ∈ *fully-unsorted-poly-rel*›
shows ‹*sort-all-coeffs p* ≤ ⇓ {(*xs*, *ys*). (*xs*, *ys*) ∈ *unsorted-poly-rel-with0* ∧ *vars-llist xs* ⊆ *vars-llist p*} (*RETURN* p')›
⟨*proof*⟩

lemma *full-normalize-poly-normalize-poly-p2*:
assumes ‹(p, p') ∈ *fully-unsorted-poly-rel*›
shows ‹*full-normalize-poly p* ≤ ⇓ {(*xs*, *ys*). (*xs*, *ys*) ∈ *sorted-poly-rel* ∧ *vars-llist xs* ⊆ *vars-llist p*} (*SPEC* (λ*r*. *normalize-poly-p*** p' *r*))›
(**is** ‹*?A* ≤ ⇓ *?R ?B*›)
⟨*proof*⟩

lemma *add-poly-full-spec*:
assumes
‹(p, p'') ∈ *sorted-poly-rel O mset-poly-rel*› **and**
‹(q, q'') ∈ *sorted-poly-rel O mset-poly-rel*›
shows
‹*add-poly-l* (*p*, *q*) ≤ ⇓(*sorted-poly-rel O mset-poly-rel*)
(*SPEC* (λ*s*. *s* − (p'' + q'')∈ *ideal polynomial-bool*))›
⟨*proof*⟩
lemma (**in** −)*add-poly-l-simps*:

```
‹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)
        }))›
```

⟨*proof*⟩
lemma *nat-less-induct-useful*:

assumes ‹$P\ 0$›‹($\bigwedge m.\ (\forall n < Suc\ m.\ P\ n) \Longrightarrow P\ (Suc\ m)$)›
shows ‹$P\ m$›
⟨*proof*⟩
lemma *add-poly-l-vars*: ‹$add\text{-}poly\text{-}l\ (p,\ q) \leq SPEC(\lambda xa.\ vars\text{-}llist\ xa \subseteq vars\text{-}llist\ p \cup vars\text{-}llist\ q)$›
⟨*proof*⟩
lemma *pw-le-SPEC-merge*: ‹$f \leq \Downarrow R\ g \Longrightarrow f \leq RES\ \Phi \Longrightarrow f \leq \Downarrow\{(x,y).\ (x,y)\in R \wedge x \in \Phi\}\ g$›
⟨*proof*⟩
lemma *add-poly-l-add-poly-p'2*:
assumes ‹$(p,\ p') \in sorted\text{-}poly\text{-}rel$› ‹$(q,\ q') \in sorted\text{-}poly\text{-}rel$›
shows ‹$add\text{-}poly\text{-}l\ (p,\ q) \leq \Downarrow \{(xs,ys).\ (xs,ys) \in sorted\text{-}poly\text{-}rel \wedge vars\text{-}llist\ xs \subseteq vars\text{-}llist\ p \cup vars\text{-}llist\ q\}\ (add\text{-}poly\text{-}p'\ p'\ q')$›
⟨*proof*⟩

lemma *add-poly-full-spec2*:
assumes
‹$(p,\ p'') \in sorted\text{-}poly\text{-}rel\ O\ mset\text{-}poly\text{-}rel$› **and**
‹$(q,\ q'') \in sorted\text{-}poly\text{-}rel\ O\ mset\text{-}poly\text{-}rel$›
shows
‹$add\text{-}poly\text{-}l\ (p,\ q) \leq \Downarrow \{(xs,ys).\ (xs,ys) \in sorted\text{-}poly\text{-}rel\ \ O\ mset\text{-}poly\text{-}rel \wedge vars\text{-}llist\ xs \subseteq vars\text{-}llist\ p \cup vars\text{-}llist\ q\}$
$(SPEC\ (\lambda s.\ \ s - (p'' + q'')\in ideal\ polynomial\text{-}bool))$›
⟨*proof*⟩

lemma *add-poly-full-spec3*:
assumes
‹$(p,\ p'') \in sorted\text{-}poly\text{-}rel\ O\ mset\text{-}poly\text{-}rel$› **and**
‹$(q,\ q'') \in sorted\text{-}poly\text{-}rel\ O\ mset\text{-}poly\text{-}rel$›
shows
‹$add\text{-}poly\text{-}l\ (p,\ q) \leq \Downarrow \{(xs,ys).\ (xs,ys) \in sorted\text{-}poly\text{-}rel\ \ O\ mset\text{-}poly\text{-}rel \wedge vars\text{-}llist\ xs \subseteq vars\text{-}llist\ p \cup vars\text{-}llist\ q\}$
$(add\text{-}poly\text{-}spec\ p''\ q'')$›
⟨*proof*⟩

lemma *full-normalize-poly-full-spec2*:
assumes
‹$(p,\ p'') \in fully\text{-}unsorted\text{-}poly\text{-}rel\ O\ mset\text{-}poly\text{-}rel$›
shows
‹$full\text{-}normalize\text{-}poly\ p \leq \Downarrow\{(xs,\ ys).\ (xs,\ ys) \in sorted\text{-}poly\text{-}rel\ O\ mset\text{-}poly\text{-}rel \wedge vars\text{-}llist\ xs \subseteq vars\text{-}llist\ p\}$
$(SPEC\ (\lambda s.\ \ s - (p'')\in ideal\ polynomial\text{-}bool \wedge vars\ s \subseteq vars\ p''))$›
⟨*proof*⟩
lemma (**in** −) *add-poly-l-simps-empty*[*simp*]: ‹$add\text{-}poly\text{-}l\ ([],\ a) = RETURN\ a$›
⟨*proof*⟩

definition *term-rel* :: ‹-› **where**
‹$term\text{-}rel = sorted\text{-}poly\text{-}rel\ O\ mset\text{-}poly\text{-}rel$›
definition *raw-term-rel* **where**
‹$raw\text{-}term\text{-}rel = fully\text{-}unsorted\text{-}poly\text{-}rel\ O\ mset\text{-}poly\text{-}rel$›

fun (**in** −)*insort-wrt* :: ‹$('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$› **where**
‹$insort\text{-}wrt$ - - $a\ [] = [a]$› |
‹$insort\text{-}wrt\ f\ P\ a\ (x\ \#\ xs) =$
$(if\ P\ (f\ a)\ (f\ x)\ then\ a\ \#\ x\ \#\ xs\ else\ x\ \#\ insort\text{-}wrt\ f\ P\ a\ xs)$›

lemma (**in** −)*set-insort-wrt* [*simp*]: ‹$set\ (insort\text{-}wrt\ P\ f\ a\ xs) = insert\ a\ (set\ xs)$›

⟨proof⟩

lemma (**in** −)*sorted-insort-wrt*:
‹*transp P* ⟹ *total* (*p2rel P*) ⟹ *sorted-wrt* (λ*a b*. *P* (*f a*) (*f b*)) *xs* ⟹ *reflp-on P* (*f* ‘ *set* (*a* # *xs*)) ⟹
sorted-wrt (λ*a b*. *P* (*f a*) (*f b*)) (*insort-wrt f P a xs*)›
⟨proof⟩

lemma (**in** −)*sorted-insort-wrt3*:
‹*transp P* ⟹ *total* (*p2rel P*) ⟹ *sorted-wrt* (λ*a b*. *P* (*f a*) (*f b*)) *xs* ⟹ *f a*∉*f* ‘ *set xs* ⟹
sorted-wrt (λ*a b*. *P* (*f a*) (*f b*)) (*insort-wrt f P a xs*)›
⟨proof⟩
lemma (**in** −)*sorted-insort-wrt4*:
‹*transp P* ⟹ *total* (*p2rel P*) ⟹ *f a*∉*f* ‘ *set xs* ⟹ *sorted-wrt* (λ*a b*. *P* (*f a*) (*f b*)) *xs* ⟹ *f*′=(λ*a b*. *P* (*f a*) (*f b*)) ⟹
sorted-wrt f′ (*insort-wrt f P a xs*)›
⟨proof⟩

When a is empty, constants are added up.

lemma *add-poly-p-insort*:
‹*fst a* ≠ [] ⟹ *vars-llist* [*a*] ∩ *vars-llist b* = {} ⟹ *add-poly-l* ([*a*],*b*) = *RETURN* (*insort-wrt fst term-order a b*)›
⟨proof⟩

lemma (**in** −) *map-insort-wrt*: ‹*map f* (*insort-wrt f P x xs*) = *insort-wrt id P* (*f x*) (*map f xs*)›
⟨proof⟩

lemma (**in**−) *distinct-insort-wrt*[*simp*]: ‹*distinct* (*insort-wrt f P x xs*) ⟷ *distinct* (*x* # *xs*)›
⟨proof⟩
lemma (**in** −) *mset-insort-wrt*[*simp*]: ‹*mset* (*insort-wrt f P x xs*) = *add-mset x* (*mset xs*)›
⟨proof⟩
lemma (**in** −) *transp-term-order-rel*: ‹*transp* (λ*x y*. (*fst x*, *fst y*) ∈ *term-order-rel*)›
⟨proof⟩

lemma (**in** −) *transp-term-order*: ‹*transp term-order*›
⟨proof⟩

lemma *total-term-order-rel*: ‹*total* (*term-order-rel*)›
⟨proof⟩

lemma *monomom-rel-mapI*: ‹*sorted-wrt* (λ*x y*. (*fst x*, *fst y*) ∈ *term-order-rel*) *r* ⟹
distinct (*map fst r*) ⟹
(∀ *x*∈*set r*. *distinct* (*fst x*) ∧ *sorted-wrt var-order* (*fst x*)) ⟹
(*r*, *map* (λ(*a*, *y*). (*mset a*, *y*)) *r*) ∈ ⟨*term-poly-list-rel* $\times_r$ *int-rel*⟩*list-rel*›
⟨proof⟩

lemma *add-poly-l-single-new-var*:
assumes ‹(*r*, *ra*) ∈ *sorted-poly-rel O mset-poly-rel*› **and**
‹*v* ∉ *vars-llist r*› **and**
v: ‹(*v*, *v*′) ∈ *var-rel*›
shows
‹*add-poly-l* ([([*v*], −*1*)], *r*)
≤ ⇓ {(*a*,*b*). (*a*,*b*)∈*sorted-poly-rel O mset-poly-rel* ∧ *vars-llist a* ⊆ *insert v* (*vars-llist r*)}
(*SPEC*

$(\lambda r0.\ r0 = ra - Var\ v' \wedge$
$vars\ r0 = vars\ ra \cup \{v'\}))$›
⟨*proof*⟩

lemma *empty-sorted-poly-rel*[*simp*,*intro*]: ‹ ([], *0*) ∈ *sorted-poly-rel O mset-poly-rel*›
⟨*proof*⟩

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

lemma *single-valued-monomials*: ‹*single-valued* (⟨*term-poly-list-rel* $\times_r$ *int-rel*⟩*list-rel*)›
⟨*proof*⟩
lemma *single-valued-term*: ‹*single-valued* (*sorted-poly-rel O mset-poly-rel*) ›
⟨*proof*⟩

lemma *single-valued-poly*:
‹(*ysa*, *cs*) ∈ ⟨*sorted-poly-rel O mset-poly-rel* $\times_r$ *nat-rel*⟩*list-rel* ⟹
(*ysa*, *csa*) ∈ ⟨*sorted-poly-rel O mset-poly-rel* $\times_r$ *nat-rel*⟩*list-rel* ⟹
cs = *csa*›
⟨*proof*⟩

lemma *check-linear-combi-l-check-linear-comb*:
assumes ‹(*A*, *B*) ∈ *fmap-polys-rel*› **and** ‹(*r*, *r'*) ∈ *sorted-poly-rel O mset-poly-rel*›
‹(*i*, *i'*) ∈ *nat-rel*›
‹($\mathcal{V}'$, $\mathcal{V}$) ∈ ⟨*var-rel*⟩*set-rel*› **and**
xs: ‹(*xs*, *xs'*) ∈ ⟨(*fully-unsorted-poly-rel O mset-poly-rel*) $\times_r$ *nat-rel*⟩*list-rel*› **and**
A: ‹⋀*i*. *i* ∈# *dom-m A* ⟹ *vars-llist* (*the* (*fmlookup A i*)) ⊆ $\mathcal{V}'$›
shows
‹*check-linear-combi-l spec A* $\mathcal{V}'$ *i xs r* ≤ ⇓ {(*st*, *b*). (¬*is-cfailed st* ⟷ *b*) ∧
(*is-cfound st* ⟶ *spec* = *r*) ∧ (*b* ⟶ *vars-llist r* ⊆ $\mathcal{V}'$ ∧ *i* ∉# *dom-m A*)} (*check-linear-comb B* $\mathcal{V}$ *xs' i' r'*)›
⟨*proof*⟩

definition *remap-polys-with-err* :: ‹*int mpoly* ⇒ *int mpoly* ⇒ *nat set* ⇒ (*nat*, *int-poly*) *fmap* ⇒ (*status* × *fpac-step*) *nres*› **where**
‹*remap-polys-with-err spec spec0* = (λ$\mathcal{V}$ *A*. *do*{
dom ← *SPEC*(λ*dom*. *set-mset* (*dom-m A*) ⊆ *dom* ∧ *finite dom*);
$\mathcal{V}$ ← *SPEC*(λ$\mathcal{V}'$. $\mathcal{V}$ ∪ *vars spec0* ⊆ $\mathcal{V}'$);
failed ← *SPEC*(λ-::*bool*. *True*);
if failed
then do {
SPEC (λ(*mem*, -, -). *mem* = *FAILED*)
}
else do {
(*b*, *N*) ← $FOREACH_C$ *dom* (λ(*b*, $\mathcal{V}$, *A'*). ¬*is-failed b*)
(λ*i* (*b*, $\mathcal{V}$, *A'*).
if i ∈# *dom-m A*
then do {
ASSERT(¬*is-failed b*);
err ← *RES* {*FAILED*,*SUCCESS*};
if is-failed err then SPEC(λ(*err'*, $\mathcal{V}$, *A'*). *err* = *err'*)
else do {

p ← SPEC(λp. the (fmlookup A i) − p ∈ ideal polynomial-bool ∧ vars p ⊆ vars (the (fmlookup A i)));
eq ← SPEC(λeq. eq ≠ FAILED ∧ (eq = FOUND ⟶ p = spec));
𝒱 ← SPEC(λ𝒱′. 𝒱 ∪ vars (the (fmlookup A i)) ⊆ 𝒱′);
RETURN(merge-status eq err, 𝒱, fmupd i p A′)
}
}
else RETURN (b, 𝒱, A′))
(SUCCESS, 𝒱, fmempty);
RETURN (b, N)
}
})›

lemma *remap-polys-with-err-spec*:
‹*remap-polys-with-err spec spec0 𝒱 A ≤ ⇓{(a,(err, 𝒱′, A)). a = (err, 𝒱′, A) ∧ (¬is-failed err ⟶ vars spec0 ⊆ 𝒱′)} (remap-polys-polynomial-bool spec 𝒱 A)*›
⟨*proof*⟩

definition (**in** −) *remap-polys-l-with-err-pre*
:: ‹*llist-polynomial ⇒ llist-polynomial ⇒ string set ⇒ (nat, llist-polynomial) fmap ⇒ bool*›
where
‹*remap-polys-l-with-err-pre spec spec0 𝒱 A ⟷ vars-llist spec ⊆ vars-llist spec0*›

definition (**in** −) *remap-polys-l-with-err* :: ‹*llist-polynomial ⇒ llist-polynomial ⇒ string set ⇒ (nat, llist-polynomial) fmap ⇒*
(- code-status × string set × (nat, llist-polynomial) fmap) nres› **where**
‹*remap-polys-l-with-err spec spec0 = (λ𝒱 A. do{*
ASSERT(remap-polys-l-with-err-pre spec spec0 𝒱 A);
dom ← SPEC(λdom. set-mset (dom-m A) ⊆ dom ∧ finite dom);
𝒱 ← RETURN(𝒱 ∪ vars-llist spec0);
failed ← SPEC(λ-::bool. True);
if failed
then do {
c ← remap-polys-l-dom-err;
SPEC (λ(mem, -, -). mem = error-msg (0 :: nat) c)
}
else do {
(err, 𝒱, A) ← $FOREACH_C$ *dom (λ(err, 𝒱, A′). ¬is-cfailed err)*
(λi (err, 𝒱, A′).
if i ∈# dom-m A
then do {
err′ ← SPEC(λerr. err ≠ CFOUND);
if is-cfailed err′ then RETURN((err′, 𝒱, A′))
else do {
p ← full-normalize-poly (the (fmlookup A i));
eq ← weak-equality-l p spec;
𝒱 ← RETURN(𝒱 ∪ vars-llist (the (fmlookup A i)));
RETURN((if eq then CFOUND else CSUCCESS), 𝒱, fmupd i p A′)
}
} *else RETURN (err, 𝒱, A′))*
(CSUCCESS, 𝒱, fmempty);
RETURN (err, 𝒱, A)
}})›

lemma *sorted-poly-rel-extend-vars*:

‹(A, B) ∈ sorted-poly-rel O mset-poly-rel ⟹
(x1c, x1a) ∈ ⟨var-rel⟩set-rel ⟹
RETURN (x1c ∪ vars-llist A)
≤ ⇓ (⟨var-rel⟩set-rel)
(SPEC ((⊆) (x1a ∪ vars (B))))›
⟨proof⟩

lemma *remap-polys-l-remap-polys*:
assumes
AB: ‹(A, B) ∈ ⟨nat-rel, fully-unsorted-poly-rel O mset-poly-rel⟩fmap-rel› **and**
spec: ‹(spec, spec′) ∈ sorted-poly-rel O mset-poly-rel› **and**
V: ‹(𝒱, 𝒱′) ∈ ⟨var-rel⟩set-rel› **and**
‹(spec0, spec0′) ∈ fully-unsorted-poly-rel O mset-poly-rel›
‹remap-polys-l-with-err-pre spec spec0 𝒱 A›
shows ‹remap-polys-l-with-err spec spec0 𝒱 A ≤
⇓{(a,b). ¬is-cfailed (fst a) ⟶ (a,b)∈ code-status-status-rel $\times_r$ ⟨var-rel⟩set-rel $\times_r$ fmap-polys-rel}
(remap-polys-with-err spec′ spec0′ 𝒱′ B)›
(**is** ‹- ≤ ⇓ ?R -›)
⟨proof⟩

end

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

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
}›

lemma (**in** −) *keys-mult-monomial2*:
‹keys (monomial (n::int) (k::′a ⇒$_0$ nat) ∗ a) = (if n = 0 then {} else ((+) k) ‘ keys (a))›
⟨proof⟩

lemma *keys-Const$_0$-mult-left*:
‹keys (Const$_0$ (b::int) ∗ aa) = (if b = 0 then {} else keys aa)› **for** *aa* :: ‹(′a :: {cancel-semigroup-add,monoid-add} ⇒$_0$ nat) ⇒$_0$ -›
⟨proof⟩

hide-fact (**open**) *poly-embed.PAC-checker-l-PAC-checker*
context *poly-embed*
begin

definition *fmap-polys-rel2* **where**
‹fmap-polys-rel2 err 𝒱 ≡ {(xs, ys). ¬ is-cfailed err ⟶ ((xs, ys) ∈ fmap-polys-rel ∧ (∀ i∈#dom-m xs. vars-llist (the (fmlookup xs i)) ⊆ 𝒱))}›

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

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

lemma *check-extension-l2-check-extension*:
assumes ‹(A, B) ∈ fmap-polys-rel› **and** ‹(r, r′) ∈ sorted-poly-rel O mset-poly-rel› **and**
‹(i, i′) ∈ nat-rel› ‹(𝒱, 𝒱′) ∈ ⟨var-rel⟩set-rel› ‹(x, x′) ∈ var-rel›
shows
‹check-extension-l2 spec A 𝒱 i x r ≤
  ⇓{((st), (b)).
    (¬is-cfailed st ⟷ b) ∧
  (is-cfound st ⟶ spec = r) ∧
  (b ⟶ vars-llist r ⊆ 𝒱 ∧ x ∉ 𝒱)} (check-extension-precalc B 𝒱′ i′ x′ r′)›
⟨proof⟩

lemma *PAC-checker-l-step-PAC-checker-step*:
assumes
‹(Ast, Bst) ∈{((err, 𝒱, A), (err′, 𝒱′, A′)). ((err, 𝒱, A), (err′, 𝒱′, A′)) ∈ (code-status-status-rel ×$_r$ ⟨var-rel⟩set-rel ×$_r$ fmap-polys-rel2 err 𝒱)}› **and**
‹(st, st′) ∈ epac-step-rel› **and**
spec: ‹(spec, spec′) ∈ sorted-poly-rel O mset-poly-rel› **and**
fail: ‹¬is-cfailed (fst Ast)›
shows
‹PAC-checker-l-step spec Ast st ≤
⇓{((err, 𝒱, A), (err′, 𝒱′, A′)). ((err, 𝒱, A), (err′, 𝒱′, A′)) ∈ (code-status-status-rel ×$_r$ ⟨var-rel⟩set-rel ×$_r$ fmap-polys-rel2 err 𝒱)}
  (PAC-checker-step spec′ Bst st′)›
⟨proof⟩

lemma *PAC-checker-l-PAC-checker*:
assumes
‹(A, B) ∈{((𝒱, A), (𝒱′, A′)). ((𝒱, A), (𝒱′, A′)) ∈ (⟨var-rel⟩set-rel $\times_r$ fmap-polys-rel2 b 𝒱)}›
(**is** ‹- ∈ ?A›) **and**
‹(st, st′) ∈ ⟨epac-step-rel⟩list-rel› **and**
‹(spec, spec′) ∈ sorted-poly-rel O mset-poly-rel› **and**
‹(b, b′) ∈ code-status-status-rel›
shows
‹PAC-checker-l spec A b st ≤
⇓ {((err, 𝒱, A), (err′, 𝒱′, A′)). ((err, 𝒱, A), (err′, 𝒱′, A′)) ∈ (code-status-status-rel $\times_r$ ⟨var-rel⟩set-rel $\times_r$ fmap-polys-rel2 err 𝒱)} (PAC-checker spec′ B b′ st′)›
⟨*proof*⟩

lemma *sorted-poly-rel-extend-vars2*:
‹(A, B) ∈ sorted-poly-rel O mset-poly-rel ⟹
(x1c, x1a) ∈ ⟨var-rel⟩set-rel ⟹
RETURN (x1c ∪ vars-llist A)
≤ ⇓ {(a,b). (a,b) ∈ ⟨var-rel⟩set-rel ∧ a = x1c ∪ vars-llist A}
(SPEC ((⊆) (x1a ∪ vars (B))))›
⟨*proof*⟩

lemma *fully-unsorted-poly-rel-extend-vars2*:
‹(A, B) ∈ fully-unsorted-poly-rel O mset-poly-rel ⟹
(x1c, x1a) ∈ ⟨var-rel⟩set-rel ⟹
RETURN (x1c ∪ vars-llist A)
≤ ⇓ {(a,b). (a,b) ∈ ⟨var-rel⟩set-rel ∧ a = x1c ∪ vars-llist A}
(SPEC ((⊆) (x1a ∪ vars (B))))›
⟨*proof*⟩

lemma *remap-polys-l-with-err-remap-polys-with-err*:
assumes
AB: ‹(A, B) ∈ ⟨nat-rel, fully-unsorted-poly-rel O mset-poly-rel⟩fmap-rel› **and**
spec: ‹(spec, spec′) ∈ sorted-poly-rel O mset-poly-rel› **and**
V: ‹(𝒱, 𝒱′) ∈ ⟨var-rel⟩set-rel› **and**
spec0: ‹(spec0, spec0′) ∈ fully-unsorted-poly-rel O mset-poly-rel› **and**
pre: ‹remap-polys-l-with-err-pre spec spec0 𝒱 A›
shows ‹remap-polys-l-with-err spec spec0 𝒱 A ≤
⇓{((err, 𝒱, A), (err′, 𝒱′, A′)). (err, err′) ∈ code-status-status-rel ∧
(¬is-cfailed err ⟶ ((err, 𝒱, A), (err′, 𝒱′, A′)) ∈ (code-status-status-rel $\times_r$ ⟨var-rel⟩set-rel $\times_r$ fmap-polys-rel2 err 𝒱))}
(remap-polys-with-err spec′ spec0′ 𝒱′ B)›
(**is** ‹- ≤ ⇓ ?R -›)
⟨*proof*⟩

definition (**in** −) *full-checker-l*
:: ‹llist-polynomial ⇒ (nat, llist-polynomial) fmap ⇒ (-, string, nat) pac-step list ⇒
(string code-status × -) nres›
where
‹full-checker-l spec A st = do {
spec′ ← full-normalize-poly spec;
(b, 𝒱, A) ← remap-polys-l-with-err spec′ spec {} A;
if is-cfailed b
then RETURN (b, 𝒱, A)

else do {
  let $\mathcal{V}$ = $\mathcal{V}$;
  PAC-checker-l spec′ ($\mathcal{V}$, *A*) *b st*
 }
}›

lemma (**in** −)*RES-RES-RETURN-RES3*: ‹*RES A* ≫= (λ(*a*,*b*,*c*). *RES* (*f a b c*)) = *RES* (⋃((λ(*a*,*b*,*c*). *f a b c*) ‘ *A*))› **for** *A f*
⟨*proof*⟩

definition *vars-rel2* :: ‹-› **where**
‹*vars-rel2 err* = {(*A*,*B*). ¬*is-cfailed err* ⟶ (*A*,*B*)∈ ⟨*var-rel*⟩*set-rel*} ›

lemma *full-normalize-poly-normalize-poly-spec-vars2*: ‹(*p3*, *p1*) ∈ *fully-unsorted-poly-rel O mset-poly-rel* ⟹
full-normalize-poly p3
≤ ⇓ ({(*xs*, *ys*). (*xs*, *ys*) ∈ *sorted-poly-rel* ∧ *vars-llist xs* ⊆ *vars-llist p3*} *O*
mset-poly-rel)
(*normalize-poly-spec p1*)
›
⟨*proof*⟩

lemma *full-checker-l-full-checker*:
assumes
‹(*A*, *B*) ∈ *unsorted-fmap-polys-rel*› **and**
st: ‹(*st*, *st′*) ∈ ⟨*epac-step-rel*⟩*list-rel*› **and**
spec: ‹(*spec*, *spec′*) ∈ *fully-unsorted-poly-rel O mset-poly-rel*›
shows
‹*full-checker-l spec A st* ≤ ⇓ {((*err*, $\mathcal{V}$, *A*), *err′*, $\mathcal{V}'$, *A′*).
((*err*, $\mathcal{V}$, *A*), *err′*, $\mathcal{V}'$, *A′*) ∈ *code-status-status-rel* $\times_r$ *vars-rel2 err* $\times_r$ *fmap-polys-rel2 err* $\mathcal{V}$}
(*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*) $\times_r$ ⟨*epac-step-rel*⟩*list-rel* $\to_f$
⟨{((*err*, $\mathcal{V}$, *A*), *err′*, $\mathcal{V}'$, *A′*).
((*err*, $\mathcal{V}$, *A*), *err′*, $\mathcal{V}'$, *A′*)
∈ *code-status-status-rel* $\times_r$ *vars-rel2 err* $\times_r$ {(*xs*, *ys*).
(¬ *is-cfailed err* ⟶ (*xs*, *ys*) ∈ ⟨*nat-rel*, *sorted-poly-rel O mset-poly-rel*⟩*fmap-rel* ∧
(∀ *i*∈#*dom-m xs*. *vars-llist* (*the* (*fmlookup xs i*)) ⊆ $\mathcal{V}$))}}⟩*nres-rel*›
⟨*proof*⟩

end

end

theory *EPAC-Checker-Init*
imports *EPAC-Checker PAC-Checker.WB-Sort PAC-Checker.PAC-Checker-Relation*
begin

3 Initial Normalisation of Polynomials

3.1 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.

end

```
theory EPAC-Version
  imports Main
begin
```

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

```
local-setup ‹
  let
    val version =
       trim-line (#1 (Isabelle-System.bash-output (cd $ISAFOL/ && git rev−parse −−short HEAD ||
echo unknown)))
  in
    Local-Theory.define
      ((binding‹version›, NoSyn),
        ((binding‹version-def›, []), HOLogic.mk-literal version)) #> #2
  end
›

declare version-def [code]

end
theory EPAC-Steps-Refine
  imports EPAC-Checker
begin


lemma is-CL-import[sepref-fr-rules]:
  assumes ‹CONSTRAINT is-pure K› ‹CONSTRAINT is-pure V› ‹CONSTRAINT is-pure R›
  shows
    ‹(return o pac-res, RETURN o pac-res) ∈ [λx. is-Extension x ∨ is-CL x]a
       (pac-step-rel-assn K V R)^k → V›
    ‹(return o pac-src1, RETURN o pac-src1) ∈ [λx. is-Del x]a (pac-step-rel-assn K V R)^k → K›
    ‹(return o new-id, RETURN o new-id) ∈ [λx. is-Extension x ∨ is-CL x]a (pac-step-rel-assn K V R)^k
→ K›
    ‹(return o is-CL, RETURN o is-CL) ∈ (pac-step-rel-assn K V R)^k →a bool-assn›
    ‹(return o is-Del, RETURN o is-Del) ∈ (pac-step-rel-assn K V R)^k →a bool-assn›
    ‹(return o new-var, RETURN o new-var) ∈ [λx. is-Extension x]a (pac-step-rel-assn K V R)^k → R›
    ‹(return o is-Extension, RETURN o is-Extension) ∈ (pac-step-rel-assn K V R)^k →a bool-assn›
  ⟨proof⟩


lemma is-CL-import2[sepref-fr-rules]:
  assumes ‹CONSTRAINT is-pure K› ‹CONSTRAINT is-pure V›
  shows
    ‹(return o pac-srcs, RETURN o pac-srcs) ∈ [λx. is-CL x]a (pac-step-rel-assn K V R)^k → list-assn
(V ×a K)›
  ⟨proof⟩
```

lemma *is-Mult-lastI*:
‹¬ *is-CL b* ⟹ ¬*is-Extension b* ⟹ *is-Del b*›
⟨*proof*⟩

end

theory *EPAC-Checker-Synthesis*
imports *EPAC-Checker EPAC-Version*
EPAC-Checker-Init
EPAC-Steps-Refine
PAC-Checker.More-Loops
PAC-Checker.WB-Sort PAC-Checker.PAC-Checker-Relation
PAC-Checker.PAC-Checker-Synthesis
begin
hide-fact (**open**) *PAC-Checker.PAC-checker-l-def*
hide-const (**open**) *PAC-Checker.PAC-checker-l*

4 Code Synthesis of the Complete Checker

definition *check-linear-combi-l-pre-err-impl* :: ‹*uint64* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ *string*› **where**
‹*check-linear-combi-l-pre-err-impl i adom emptyl ivars* =
''*Precondition for* '%' *failed* '' @ *show* (*nat-of-uint64 i*) @
''(*already in domain*: '' @ *show adom* @
''; *empty CL*'' @ *show emptyl* @
''; *new vars*: '' @ *show ivars* @ '')''›

abbreviation *comp4* (**infixl** *oooo 55*) **where** *f oooo g* ≡ λ*x*. *f ooo* (*g x*)

lemma [*sepref-fr-rules*]:
‹(*uncurry3* (*return oooo check-linear-combi-l-pre-err-impl*),
uncurry3 check-linear-combi-l-pre-err) ∈ *uint64-nat-assn*^{k} $*_a$ *bool-assn*^{k} $*_a$ *bool-assn*^{k} $*_a$ *bool-assn*^{k} $\rightarrow_a$ *raw-string-assn*›
⟨*proof*⟩

definition *check-linear-combi-l-dom-err-impl* :: ‹ - ⇒ *uint64* ⇒ *string*› **where**
‹*check-linear-combi-l-dom-err-impl xs i* =
''*Invalid polynomial* '' @ *show* (*nat-of-uint64 i*)›

lemma [*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (*check-linear-combi-l-dom-err-impl*)),
uncurry (*check-linear-combi-l-dom-err*)) ∈ *poly-assn*^{k} $*_a$ *uint64-nat-assn*^{k} $\rightarrow_a$ *raw-string-assn*›
⟨*proof*⟩

definition *check-linear-combi-l-mult-err-impl* :: ‹ - ⇒ - ⇒ *string*› **where**
‹*check-linear-combi-l-mult-err-impl xs ys* =
''*Invalid calculation*, *found*'' @ *show xs* @ '' *instead of* '' @ *show ys*›

lemma [*sepref-fr-rules*]:
‹(*uncurry* (*return oo check-linear-combi-l-mult-err-impl*),
uncurry check-linear-combi-l-mult-err) ∈ *poly-assn*^{k} $*_a$ *poly-assn*^{k} $\rightarrow_a$ *raw-string-assn*›
⟨*proof*⟩

sepref-definition *linear-combi-l-impl*
is ‹*uncurry3 linear-combi-l*›

:: ‹uint64-nat-assn^k $*_a$ polys-assn^k $*_a$ vars-assn^k $*_a$ (list-assn (poly-assn $\times_a$ uint64-nat-assn))^k $\rightarrow_a$ poly-assn $\times_a$ (list-assn (poly-assn $\times_a$ uint64-nat-assn)) $\times_a$ status-assn raw-string-assn›
⟨proof⟩

definition *has-failed* :: ‹bool nres› **where**
‹has-failed = RES UNIV›

lemma [*sepref-fr-rules*]:
‹(uncurry0 (return False), uncurry0 has-failed)∈unit-assn^k $\rightarrow_a$ bool-assn›
⟨proof⟩

declare *linear-combi-l-impl.refine*[*sepref-fr-rules*]
sepref-register *check-linear-combi-l-pre-err*
sepref-definition *check-linear-combi-l-impl*
is ‹uncurry5 check-linear-combi-l›
:: ‹poly-assn^k $*_a$ polys-assn^k $*_a$ vars-assn^k $*_a$ uint64-nat-assn^k $*_a$
(list-assn (poly-assn $\times_a$ uint64-nat-assn))^k $*_a$ poly-assn^k $\rightarrow_a$ status-assn raw-string-assn›
⟨proof⟩

declare *check-linear-combi-l-impl.refine*[*sepref-fr-rules*]

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*
find-theorems *is-CL RETURN*

sepref-register *check-linear-combi-l check-extension-l2*
term *check-extension-l2*

definition *check-extension-l2-cond* :: ‹nat ⇒ -› **where**
‹check-extension-l2-cond i A $\mathcal{V}$ v = SPEC (λb. b ⟶ fmlookup′ i A = None ∧ v ∉ $\mathcal{V}$)›

definition *check-extension-l2-cond2* :: ‹nat ⇒ -› **where**
‹check-extension-l2-cond2 i A $\mathcal{V}$ v = RETURN (fmlookup′ i A = None ∧ v ∉ $\mathcal{V}$)›

sepref-definition *check-extension-l2-cond2-impl*
is ‹uncurry3 check-extension-l2-cond2›
:: ‹uint64-nat-assn^k $*_a$ polys-assn^k $*_a$ vars-assn^k $*_a$ string-assn^k $\rightarrow_a$ bool-assn›
⟨proof⟩

lemma *check-extension-l2-cond2-check-extension-l2-cond*:
‹(uncurry3 check-extension-l2-cond2, uncurry3 check-extension-l2-cond) ∈
(((nat-rel $\times_r$ Id) $\times_r$ Id) $\times_r$ Id) $\rightarrow_f$ ⟨bool-rel⟩nres-rel›
⟨proof⟩

lemmas [*sepref-fr-rules*] =
check-extension-l2-cond2-impl.refine[*FCOMP check-extension-l2-cond2-check-extension-l2-cond*]

definition *check-extension-l-side-cond-err-impl* :: ‹- ⇒ -› **where**
‹*check-extension-l-side-cond-err-impl v r s* =
''*Error while checking side conditions of extensions polynow, var is* '' @ *show v* @
''*side condition p*∗*p* − *p* = '' @ *show s* @ '' *and should be 0*''›
term *check-extension-l-side-cond-err*
lemma [*sepref-fr-rules*]:
‹(*uncurry2* (*return ooo* (*check-extension-l-side-cond-err-impl*)),
uncurry2 (*check-extension-l-side-cond-err*)) ∈ *string-assn*^{k} $*_a$ *poly-assn*^{k} $*_a$ *poly-assn*^{k} $\rightarrow_a$ *raw-string-assn*›
⟨*proof*⟩

definition *check-extension-l-new-var-multiple-err-impl* :: ‹- ⇒ -› **where**
‹*check-extension-l-new-var-multiple-err-impl v p* =
''*Error while checking side conditions of extensions polynow, var is* '' @ *show v* @
'' *but it either appears at least once in the polynomial or another new variable is created* '' @
show p @ '' *but should not.*''›

lemma [*sepref-fr-rules*]:
‹((*uncurry* (*return oo* (*check-extension-l-new-var-multiple-err-impl*))),
uncurry (*check-extension-l-new-var-multiple-err*)) ∈ *string-assn*^{k} $*_a$ *poly-assn*^{k} $\rightarrow_a$ *raw-string-assn*›
⟨*proof*⟩

sepref-definition *check-extension-l-impl*
is ‹*uncurry5 check-extension-l2*›
:: ‹*poly-assn*^{k} $*_a$ *polys-assn*^{k} $*_a$ *vars-assn*^{k} $*_a$ *uint64-nat-assn*^{k} $*_a$
string-assn^k $*_a$ *poly-assn*^{k} $\rightarrow_a$ *status-assn raw-string-assn*›
⟨*proof*⟩

lemmas [*sepref-fr-rules*] =
check-extension-l-impl.refine

lemma *is-Mult-lastI*:
‹¬ *is-CL b* ⟹ ¬*is-Extension b* ⟹ *is-Del b*›
⟨*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 𝒱A status steps =
(let (𝒱, A) = 𝒱A in PAC-checker-l′ p 𝒱 A status steps)›
⟨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)›
⟨proof⟩

lemma *safe-epac-step-rel-assn*[*safe-constraint-rules*]:
‹CONSTRAINT is-pure K ⟹ CONSTRAINT is-pure V ⟹ CONSTRAINT is-pure R ⟹
CONSTRAINT is-pure (EPAC-Checker.pac-step-rel-assn K V R)›
⟨proof⟩

sepref-definition *PAC-checker-l-impl*
is ‹uncurry4 PAC-checker-l′›
:: ‹$poly\text{-}assn^k *_a vars\text{-}assn^d *_a polys\text{-}assn^d *_a (status\text{-}assn\ raw\text{-}string\text{-}assn)^d *_a$
$(list\text{-}assn\ (pac\text{-}step\text{-}rel\text{-}assn\ (uint64\text{-}nat\text{-}assn)\ poly\text{-}assn\ string\text{-}assn))^k \rightarrow_a$
$status\text{-}assn\ raw\text{-}string\text{-}assn \times_a vars\text{-}assn \times_a polys\text{-}assn$›
⟨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)) ∈ $unit\text{-}assn^k \rightarrow_a raw\text{-}string\text{-}assn$›
⟨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*

definition *full-checker-l2*
:: ‹llist-polynomial ⇒ (nat, llist-polynomial) fmap ⇒ (-, string, nat) pac-step list ⇒
(string code-status × -) nres›
where
‹full-checker-l2 spec A st = do {
spec′ ← full-normalize-poly spec;
(b, 𝒱, A) ← remap-polys-l spec {} A;
if is-cfailed b
then RETURN (b, 𝒱, A)
else do {

    PAC-checker-l spec′ (𝒱, *A*) *b st*
  }
}›

sepref-register *remap-polys-l*
find-theorems *full-checker-l2*
sepref-definition *full-checker-l-impl*
  is ‹*uncurry2 full-checker-l2*›
  :: ‹$poly\text{-}assn^k *_a polys\text{-}assn\text{-}input^d *_a (list\text{-}assn\ (pac\text{-}step\text{-}rel\text{-}assn\ (uint64\text{-}nat\text{-}assn)\ poly\text{-}assn\ string\text{-}assn))^k \rightarrow_a$
    $status\text{-}assn\ raw\text{-}string\text{-}assn \times_a vars\text{-}assn \times_a polys\text{-}assn$›
  ⟨*proof*⟩

sepref-definition *PAC-empty-impl*
  is ‹*uncurry0* (*RETURN fmempty*)›
  :: ‹$unit\text{-}assn^k \rightarrow_a polys\text{-}assn\text{-}input$›
  ⟨*proof*⟩

sepref-definition *empty-vars-impl*
  is ‹*uncurry0* (*RETURN* {})›
  :: ‹$unit\text{-}assn^k \rightarrow_a vars\text{-}assn$›
  ⟨*proof*⟩

end
theory *EPAC-Perfectly-Shared*
  imports *EPAC-Checker-Specification*
    PAC-Checker.PAC-Checker
    EPAC-Checker
begin

We now introduce sharing of variables to make a more efficient representation possible.

5 Perfectly sharing of elements

5.1 Definition

type-synonym (*′nat*, *′string*) *shared-vars* = ‹*′string multiset* × (*′nat*, *′string*) *fmap* × (*′string*, *′nat*) *fmap*›

definition *perfectly-shared-vars*
  :: ‹*′string multiset* ⇒ (*′nat*, *′string*) *shared-vars* ⇒ *bool*›
where
  ‹*perfectly-shared-vars* 𝒱 = (λ(𝒟, *V*, *V′*).
  set-mset (*dom-m V′*) = *set-mset* 𝒱 ∧ 𝒟 = 𝒱 ∧
  (∀ *i* ∈#*dom-m V*. *fmlookup V′* (*the* (*fmlookup V i*)) = *Some i*) ∧
  (∀ *str* ∈#*dom-m V′*. *fmlookup V* (*the* (*fmlookup V′ str*)) = *Some str*) ∧
  (∀ *i j*. *i*∈#*dom-m V* ⟶ *j*∈#*dom-m V* ⟶ (*fmlookup V i* = *fmlookup V j* ⟷ *i* = *j*)))›

abbreviation *fmlookup-direct* :: ‹(*′a*, *′b*) *fmap* ⇒ *′a* ⇒ *′b*› (**infix** ∝ *70*) **where**
  ‹*fmlookup-direct A b* ≡ *the* (*fmlookup A b*)›

lemma *perfectly-shared-vars-simps*:
  assumes ‹*perfectly-shared-vars* 𝒱 (*VV′*)›
  shows ‹*str* ∈# 𝒱 ⟷ *str* ∈# *dom-m* (*snd* (*snd VV′*))›
  ⟨*proof*⟩

lemma *perfectly-shared-add-new-var*:
  fixes V :: ‹('nat, 'string) fmap› **and**
    v :: ‹'string›
  assumes ‹perfectly-shared-vars $\mathcal{V}$ $(D,\ V,\ V')$› **and**
    ‹$v \notin\# \mathcal{V}$› **and**
    k-notin[*simp*]: ‹$k \notin\#$ dom-m V›
  shows ‹perfectly-shared-vars (add-mset v $\mathcal{V}$) (add-mset v D, fmupd k v V, fmupd v k V')›
⟨*proof*⟩

lemma *perfectly-shared-vars-remove-update*:
  assumes ‹perfectly-shared-vars (add-mset v $\mathcal{V}$) $(D,\ V,\ V')$› **and**
    ‹$v \notin\# \mathcal{V}$›
  shows ‹perfectly-shared-vars $\mathcal{V}$ (remove1-mset v D, fmdrop $(V' \propto v)$ V, fmdrop v V')›
  ⟨*proof*⟩

6 Refinement

datatype *memory-allocation* =
  Allocated | *alloc-failed*: *Mem-Out*

type-synonym ('nat, 'string) vars = ‹'string multiset›

definition *perfectly-shared-var-rel* :: ‹('nat,'string) shared-vars ⇒ ('nat × 'string) set› **where**
  ‹perfectly-shared-var-rel = $(\lambda(\mathcal{D},\ \mathcal{V},\ \mathcal{V}').$ br $(\lambda i.\ \mathcal{V} \propto i)$ $(\lambda i.\ i \in\#$ dom-m $\mathcal{V}))$›

definition *perfectly-shared-vars-rel* :: ‹(('nat,'string) shared-vars × ('nat, 'string) vars) set›
where
  ‹perfectly-shared-vars-rel = $\{(\mathcal{A},\ \mathcal{V}).$ perfectly-shared-vars $\mathcal{V}$ $\mathcal{A}\}$›

definition *find-new-idx* :: ‹('nat,'string) shared-vars ⇒ -› **where**
  ‹find-new-idx = $(\lambda(\text{-},\ \mathcal{V},\ \text{-}).$ SPEC $(\lambda(mem,\ k).\ \neg$ alloc-failed mem $\longrightarrow k \notin\#$ dom-m $\mathcal{V}))$›

definition *import-variableS*
  :: ‹'string ⇒ ('nat, 'string) shared-vars ⇒
  (memory-allocation × ('nat, 'string) shared-vars × 'nat) nres›
where
  ‹import-variableS v = $(\lambda(\mathcal{D},\ \mathcal{V},\ \mathcal{V}').$ do {
    (mem, k) ← find-new-idx $(\mathcal{D},\ \mathcal{V},\ \mathcal{V}')$;
    if alloc-failed mem then do {k ← RES (UNIV :: 'nat set); RETURN (mem, $(\mathcal{D},\ \mathcal{V},\ \mathcal{V}')$, k)}
    else RETURN (Allocated, (add-mset v $\mathcal{D}$, fmupd k v $\mathcal{V}$, fmupd v k $\mathcal{V}'$), k)
  })›

definition *import-variable*
  :: ‹'string ⇒ ('nat, 'string) vars ⇒ (memory-allocation × ('nat, 'string) vars × 'string) nres›
  where
  ‹import-variable v = $(\lambda\mathcal{V}.$ do {
    ASSERT($v \notin\# \mathcal{V}$);
    SPEC(λ(mem, $\mathcal{V}'$, k::'string). ¬alloc-failed mem $\longrightarrow \mathcal{V}' =$ add-mset k $\mathcal{V} \wedge k = v$)
  })›

definition *is-new-variableS* :: ‹'string ⇒ ('nat, 'string) shared-vars ⇒ bool nres› **where**
  ‹is-new-variableS v = $(\lambda(\mathcal{D},\ \mathcal{V},\ \mathcal{V}').$
  RETURN ($v \notin\#$ dom-m $\mathcal{V}'$)
)›

definition *is-new-variable* :: ‹′*string* ⇒ (′*nat*, ′*string*) *vars* ⇒ *bool nres*› **where**
‹*is-new-variable v* = (λ𝒱′.
RETURN (*v* ∉# 𝒱′)
)›

lemma *import-variableS-import-variable*:
fixes 𝒱 :: ‹(′*nat*, ′*string*) *vars*›
assumes ‹(𝒜, 𝒱) ∈ *perfectly-shared-vars-rel*› **and** ‹(*v*, *v*′) ∈ *Id*›
shows ‹*import-variableS v* 𝒜 ≤ ⇓({((*mem*, 𝒜′, *i*), (*mem*′, 𝒱′, *j*)). *mem* = *mem*′ ∧
(𝒜′, 𝒱′) ∈ *perfectly-shared-vars-rel* ∧
(¬*alloc-failed mem*′ ⟶ (*i*, *j*) ∈ *perfectly-shared-var-rel* 𝒜′) ∧
(∀ *xs*. *xs* ∈ *perfectly-shared-var-rel* 𝒜 ⟶ *xs* ∈ *perfectly-shared-var-rel* 𝒜′)})
(*import-variable v*′ 𝒱)›
⟨*proof*⟩

lemma *is-new-variable-spec*:
assumes ‹(𝒜, 𝒟𝒱) ∈ *perfectly-shared-vars-rel*› ‹(*v*,*v*′) ∈ *Id*›
shows ‹*is-new-variableS v* 𝒜 ≤ ⇓*bool-rel* (*is-new-variable v*′ 𝒟𝒱)›
⟨*proof*⟩

definition *import-variables*
:: ‹′*string list* ⇒ (′*nat*, ′*string*) *vars* ⇒ (*memory-allocation* × (′*nat*, ′*string*) *vars*) *nres*›
where
‹*import-variables vs* 𝒱 = *do* {
(*mem*, 𝒱, -, -) ← *WHILE*$_T$(λ(*mem*, 𝒱, *vs*, -). ¬*alloc-failed mem* ∧ *vs* ≠ [])
(λ(-, 𝒱, *vs*, *vs*′). *do* {
ASSERT(*vs* ≠ []);
let v = *hd vs*;
a ← *is-new-variable v* 𝒱;
if ¬*a then RETURN* (*Allocated* ,𝒱, *tl vs*, *vs*′ @ [*v*])
else do {
(*mem*, 𝒱, -) ← *import-variable v* 𝒱;
RETURN(*mem*, 𝒱, *tl vs*, *vs*′ @ [*v*])
}
})
(*Allocated*, 𝒱, *vs*, []);
RETURN (*mem*, 𝒱)
}›

definition *import-variablesS*
:: ‹′*string list* ⇒ (′*nat*, ′*string*) *shared-vars* ⇒ (*memory-allocation* × (′*nat*, ′*string*) *shared-vars*) *nres*›
where
‹*import-variablesS vs* 𝒱 = *do* {
(*mem*, 𝒱, -) ← *WHILE*$_T$(λ(*mem*, 𝒱, *vs*). ¬*alloc-failed mem* ∧ *vs* ≠ [])
(λ(-, 𝒱, *vs*). *do* {
ASSERT(*vs* ≠ []);
let v = *hd vs*;
a ← *is-new-variableS v* 𝒱;
if ¬*a then RETURN* (*Allocated* ,𝒱, *tl vs*)
else do {
(*mem*, 𝒱, -) ← *import-variableS v* 𝒱;
RETURN(*mem*, 𝒱, *tl vs*)
}
})

(*Allocated*, $\mathcal{V}$, *vs*);
RETURN (*mem*, $\mathcal{V}$)
}›

lemma *import-variables-spec*:
‹*import-variables vs* $\mathcal{V} \leq \Downarrow$*Id* (*SPEC*($\lambda$(*mem*, $\mathcal{V}'$). $\neg$*alloc-failed mem* $\longrightarrow$ *set-mset* $\mathcal{V}'$ = *set-mset* $\mathcal{V} \cup$ *set vs*))›
⟨*proof*⟩

lemma *import-variablesS-import-variables*:
assumes ‹($\mathcal{V}$, $\mathcal{V}'$) $\in$ *perfectly-shared-vars-rel*› **and**
‹(*vs*, *vs'*) $\in$ *Id*›
shows ‹*import-variablesS vs* $\mathcal{V} \leq \Downarrow\{(a,b).\ (a,b) \in Id \times_r$ *perfectly-shared-vars-rel* $\wedge$
($\neg$*alloc-failed* (*fst a*) $\longrightarrow$ *perfectly-shared-var-rel* $\mathcal{V} \subseteq$ *perfectly-shared-var-rel* (*snd a*))} (*import-variables vs'* $\mathcal{V}'$)›
⟨*proof*⟩

definition *get-var-name* :: ‹('*nat*, '*string*) *vars* ⇒ '*string* ⇒ '*string nres*› **where**
‹*get-var-name* $\mathcal{V}$ *x* = *do* {
ASSERT(*x* ∈# $\mathcal{V}$);
RETURN x
}›

definition *get-var-posS* :: ‹('*nat*, '*string*) *shared-vars* ⇒ '*string* ⇒ '*nat nres*› **where**
‹*get-var-posS* $\mathcal{V}$ *x* = *do* {
ASSERT(*x* ∈# *dom-m* (*snd* (*snd* $\mathcal{V}$)));
RETURN (*snd* (*snd* $\mathcal{V}$) ∝ *x*)
}›

definition *get-var-nameS* :: ‹('*nat*, '*string*) *shared-vars* ⇒ '*nat* ⇒ '*string nres*› **where**
‹*get-var-nameS* $\mathcal{V}$ *x* = *do* {
ASSERT(*x* ∈# *dom-m* (*fst* (*snd* $\mathcal{V}$)));
RETURN (*fst* (*snd* $\mathcal{V}$) ∝ *x*)
}›

lemma *get-var-posS-spec*:
fixes $\mathcal{DV}$ :: ‹('*nat*, '*string*) *vars*› **and**
$\mathcal{A}$ :: ‹('*nat*, '*string*) *shared-vars*› **and**
x :: '*string*
assumes ‹($\mathcal{A}$, $\mathcal{DV}$) $\in$ *perfectly-shared-vars-rel*› **and**
‹(*x*,*x'*) $\in$ *Id*›
shows ‹*get-var-posS* $\mathcal{A}$ *x* $\leq \Downarrow$(*perfectly-shared-var-rel* $\mathcal{A}$) (*get-var-name* $\mathcal{DV}$ *x'*)›
⟨*proof*⟩

abbreviation *perfectly-shared-monom*
:: ‹('*nat*,'*string*) *shared-vars* ⇒ ('*nat list* × '*string list*) *set*›
where
‹*perfectly-shared-monom* $\mathcal{V}$ ≡ ⟨*perfectly-shared-var-rel* $\mathcal{V}$⟩*list-rel* ›

definition *import-monom-no-newS*
:: ‹('*nat*, '*string*) *shared-vars* ⇒ '*string list* ⇒ (*bool* × '*nat list*) *nres*›
where
‹*import-monom-no-newS* $\mathcal{A}$ *xs* = *do* {
(*new*, -, *xs*) ← *WHILE*$_T$ (λ(*new*, *xs*, -). $\neg$*new* $\wedge$ *xs* ≠ [])

```
    (λ(-, xs, ys). do {
      ASSERT(xs ≠ []);
      let x = hd xs;
      b ← is-new-variableS x 𝒜;
      if b
      then RETURN (True, tl xs, ys)
      else do {
        x ← get-var-posS 𝒜 x;
        RETURN (False, tl xs, x # ys)
       }
    })
    (False, xs, []);
  RETURN (new, rev xs)
 }›
```

definition *import-monom-no-new*
  :: ‹('nat, 'string) vars ⇒ 'string list ⇒ (bool × 'string list) nres›
where

```
  ‹import-monom-no-new 𝒜 xs = do {
  (new, -, xs) ← WHILE_T (λ(new, xs, -). ¬new ∧ xs ≠ [])
  (λ(-, xs, ys). do {
    ASSERT(xs ≠ []);
    let x = hd xs;
    b ← is-new-variable x 𝒜;
    if b
    then RETURN (True, tl xs, ys)
      else do {
      x ← get-var-name 𝒜 x;
      RETURN (False, tl xs, ys @ [x])
    }
    })
    (False, xs, []);
  RETURN (new, xs)
}›
```

lemma *import-monom-no-new-spec*:
  shows ‹*import-monom-no-new* $\mathcal{A}$ *xs* ≤ ⇓ *Id*
    (*SPEC*(λ(*new*, *ys*). (*new* ⟷ ¬*set xs* ⊆ *set-mset* $\mathcal{A}$) ∧
      (¬*new* ⟶ *ys* = *xs*)))›
  ⟨*proof*⟩

lemma *import-monom-no-newS-import-monom-no-new*:
  assumes ‹($\mathcal{A}$, $\mathcal{VD}$) ∈ *perfectly-shared-vars-rel*› ‹(*xs*, *xs'*) ∈ *Id*›
  shows ‹*import-monom-no-newS* $\mathcal{A}$ *xs* ≤ ⇓(*bool-rel* $\times_r$ *perfectly-shared-monom* $\mathcal{A}$)
    (*import-monom-no-new* $\mathcal{VD}$ *xs'*)›
  ⟨*proof*⟩

definition *import-poly-no-newS*
  :: ‹('nat, 'string) shared-vars ⇒ ('string list × 'a) list ⇒ (bool × ('nat list × 'a)list) nres›
where

```
  ‹import-poly-no-newS 𝒜 xs = do {
  (new, -, xs) ← WHILE_T (λ(new, xs, -). ¬new ∧ xs ≠ [])
    (λ(-, xs, ys). do {
      ASSERT(xs ≠ []);
      let (x, n) = hd xs;
```

    (b, x) ← *import-monom-no-newS* $\mathcal{A}$ x;
    if b
    then RETURN (*True*, *tl xs*, *ys*)
    else do {
      RETURN (*False*, *tl xs*, (x, n) # *ys*)
     }
  })
  (*False*, *xs*, []);
 RETURN (*new*, *rev xs*)
}›

definition *import-poly-no-new*
:: ‹(′nat, ′string) vars ⇒ (′string list × ′a) list ⇒ (bool × (′string list × ′a) list) nres›
where
 ‹*import-poly-no-new* $\mathcal{A}$ *xs* = *do* {
 (*new*, -, *xs*) ← $WHILE_T$ (λ(*new*, *xs*, -). ¬*new* ∧ *xs* ≠ [])
 (λ(-, *xs*, *ys*). *do* {
   ASSERT(*xs* ≠ []);
   let (x, n) = *hd xs*;
   (b, x) ← *import-monom-no-new* $\mathcal{A}$ x;
   if b
   then RETURN (*True*, *tl xs*, *ys*)
    else do {
    RETURN (*False*, *tl xs*, *ys* @ [(x, n)])
  }
  })
  (*False*, *xs*, []);
 RETURN (*new*, *xs*)
}›

lemma *import-poly-no-newS-import-poly-no-new*:
 assumes ‹($\mathcal{A}$, $\mathcal{VD}$) ∈ *perfectly-shared-vars-rel*› ‹(*xs*, *xs′*) ∈ *Id*›
 shows ‹*import-poly-no-newS* $\mathcal{A}$ *xs* ≤ ⇓(*bool-rel* $\times_r$ ⟨*perfectly-shared-monom* $\mathcal{A}$ $\times_r$ *Id*⟩*list-rel*)
  (*import-poly-no-new* $\mathcal{VD}$ *xs′*)›
 ⟨*proof*⟩

lemma *import-poly-no-new-spec*:
 shows ‹*import-poly-no-new* $\mathcal{A}$ *xs* ≤ ⇓ *Id*
  (*SPEC*(λ(*new*, *ys*). ¬*new* ⟶ *ys* = *xs* ∧ *vars-llist xs* ⊆ *set-mset* $\mathcal{A}$))›
⟨*proof*⟩

definition *import-monomS*
:: ‹(′nat, ′string) shared-vars ⇒ ′string list ⇒ (- × ′nat list × (′nat, ′string) shared-vars) nres›
where
 ‹*import-monomS* $\mathcal{A}$ *xs* = *do* {
 (*new*, -, *xs*, $\mathcal{A}$) ← $WHILE_T$ (λ(*mem*, *xs*, -, -). ¬*alloc-failed mem* ∧ *xs* ≠ [])
  (λ(-, *xs*, *ys*, $\mathcal{A}$). *do* {
    ASSERT(*xs* ≠ []);
    let x = *hd xs*;
    b ← *is-new-variableS* x $\mathcal{A}$;
    if b
    then do {
      (*mem*, $\mathcal{A}$, x) ← *import-variableS* x $\mathcal{A}$;
      if alloc-failed mem

then RETURN (*mem*, *xs*, *ys*, $\mathcal{A}$)
else RETURN (*mem*, *tl xs*, *x* # *ys*, $\mathcal{A}$)
}
else do {
x ← *get-var-posS* $\mathcal{A}$ *x*;
RETURN (*Allocated*, *tl xs*, *x* # *ys*, $\mathcal{A}$)
}
})
(*Allocated*, *xs*, [], $\mathcal{A}$);
RETURN (*new*, *rev xs*, $\mathcal{A}$)
}›

definition *import-monom*
:: ‹('*nat*, '*string*) *vars* ⇒ '*string list* ⇒ (*memory-allocation* × '*string list* × ('*nat*, '*string*) *vars*) *nres*›
where
‹*import-monom* $\mathcal{A}$ *xs* = *do* {
(*new*, -, *xs*, $\mathcal{A}$) ← $WHILE_T$ (λ(*new*, *xs*, -, -). ¬*alloc-failed new* ∧ *xs* ≠ [])
(λ(*mem*, *xs*, *ys*, $\mathcal{A}$). *do* {
ASSERT(*xs* ≠ []);
let x = *hd xs*;
b ← *is-new-variable x* $\mathcal{A}$;
if b
then do {
(*mem*, $\mathcal{A}$, *x*) ← *import-variable x* $\mathcal{A}$;
if alloc-failed mem
then RETURN (*mem*, *xs*, *ys*, $\mathcal{A}$)
else RETURN (*mem*, *tl xs*, *ys* @ [*x*], $\mathcal{A}$)
}
else do {
x ← *get-var-name* $\mathcal{A}$ *x*;
RETURN (*mem*, *tl xs*, *ys* @ [*x*], $\mathcal{A}$)
}
})
(*Allocated*, *xs*, [], $\mathcal{A}$);
RETURN (*new*, *xs*, $\mathcal{A}$)
}›

lemma *import-monom-spec*:
shows ‹*import-monom* $\mathcal{A}$ *xs* ≤ ⇓ *Id*
(*SPEC*(λ(*new*, *ys*, $\mathcal{A}'$). ¬*alloc-failed new* ⟶ *ys* = *xs* ∧ *set-mset* $\mathcal{A}'$ = *set-mset* $\mathcal{A}$ ∪ *set xs*))›
⟨*proof*⟩

definition *import-polyS*
:: ‹('*nat*, '*string*) *shared-vars* ⇒ ('*string list* × '*a*) *list* ⇒
(*memory-allocation* × ('*nat list* × '*a*)*list* × ('*nat*, '*string*) *shared-vars*) *nres*›
where
‹*import-polyS* $\mathcal{A}$ *xs* = *do* {
(*mem*,-, *xs*, $\mathcal{A}$) ← $WHILE_T$ (λ(*mem*, *xs*, -, -). ¬*alloc-failed mem* ∧ *xs* ≠ [])
(λ(*mem*, *xs*, *ys*, $\mathcal{A}$). *do* {
ASSERT(*xs* ≠ []);
let (*x*, *n*) = *hd xs*;
(*mem*, *x*, $\mathcal{A}$) ← *import-monomS* $\mathcal{A}$ *x*;
if alloc-failed mem
then RETURN (*mem*, *xs*, *ys*, $\mathcal{A}$)
else do {

```
      RETURN (mem, tl xs, (x, n) # ys, 𝒜)
     }
   })
   (Allocated, xs, [], 𝒜);
  RETURN (mem, rev xs, 𝒜)
 }›
```

definition *import-poly*

```
  :: ‹('nat, 'string) vars ⇒ ('string list × 'a) list ⇒
       (memory-allocation × ('string list × 'a) list × ('nat, 'string)vars) nres›
  where
  ‹import-poly 𝒜 xs0 = do {
  (new, -, xs, 𝒜) ← WHILE_T (λ(new, xs, -). ¬alloc-failed new ∧ xs ≠ [])
  (λ(-, xs, ys, 𝒜). do {
    ASSERT(xs ≠ []);
    let (x, n) = hd xs;
    (b, x, 𝒜) ← import-monom 𝒜 x;
    if alloc-failed b
    then RETURN (b, xs, ys, 𝒜)
    else do {
      RETURN (Allocated, tl xs, ys @ [(x, n)], 𝒜)
    }
  })
     (Allocated, xs0, [], 𝒜);
  ASSERT(¬alloc-failed new ⟶ xs0 = xs);
  RETURN (new, xs, 𝒜)
}›
```

lemma *import-poly-spec*:

```
  fixes 𝒜 :: ‹('nat, 'string) vars›
  shows ‹import-poly 𝒜 xs ≤ ⇓ Id
    (SPEC(λ(new, ys, 𝒜'). ¬alloc-failed new ⟶ ys = xs ∧ set-mset 𝒜' = set-mset 𝒜 ∪ ⋃(set ‘fst ‘ set xs)))›
⟨proof⟩
```

lemma *list-rel-append-single*: ‹(xs, ys) ∈ ⟨R⟩list-rel ⟹ (x, y) ∈ R ⟹ (xs @ [x], ys @ [y]) ∈ ⟨R⟩list-rel›
  ⟨proof⟩

lemma *list-rel-mono*: ‹A ∈ ⟨R⟩list-rel ⟹ (⋀xs. xs ∈ R ⟹ xs ∈ R') ⟹ A ∈ ⟨R'⟩list-rel›
  ⟨proof⟩

lemma *import-monomS-import-monom*:

```
  fixes 𝒱𝒟 :: ‹('nat, 'string) vars› and 𝒜₀ :: ‹('nat, 'string)shared-vars› and xs xs' :: ‹'string list›
  assumes ‹(𝒜₀, 𝒱𝒟) ∈ perfectly-shared-vars-rel› ‹(xs, xs') ∈ ⟨Id⟩list-rel›
  shows ‹import-monomS 𝒜₀ xs ≤ ⇓ {((mem, xs₀, 𝒜), (mem', ys₀, 𝒜')). mem = mem' ∧
     (𝒜, 𝒜') ∈ perfectly-shared-vars-rel ∧ (¬alloc-failed mem ⟶ (xs₀, ys₀) ∈ perfectly-shared-monom 𝒜)∧
    (¬alloc-failed mem ⟶ (∀xs. xs ∈ perfectly-shared-monom 𝒜₀ ⟶ xs ∈ perfectly-shared-monom 𝒜))}
    (import-monom 𝒱𝒟 xs')›
  ⟨proof⟩
```

abbreviation *perfectly-shared-polynom*

```
  :: ‹('nat,'string) shared-vars ⇒ (('nat list × int) list × ('string list × int) list) set›
where
  ‹perfectly-shared-polynom 𝒱 ≡ ⟨perfectly-shared-monom 𝒱 ×_r int-rel⟩list-rel›
```

abbreviation *import-poly-rel* :: ‹-› **where**
‹*import-poly-rel* $\mathcal{A}_0$ *xs′* ≡
{((*mem*, *xs*$_0$, $\mathcal{A}$), (*mem′*, *ys*$_0$, $\mathcal{A}'$)). *mem* = *mem′* ∧
(¬*alloc-failed mem* ⟶ ($\mathcal{A}$, $\mathcal{A}'$) ∈ *perfectly-shared-vars-rel* ∧ *ys*$_0$ = *xs′* ∧ (*xs*$_0$, *ys*$_0$) ∈ *perfectly-shared-polynom* $\mathcal{A}$) ∧
(¬*alloc-failed mem* ⟶ *perfectly-shared-polynom* $\mathcal{A}_0$ ⊆ *perfectly-shared-polynom* $\mathcal{A}$)}›

lemma *import-polyS-import-poly*:
assumes ‹($\mathcal{A}_0$, $\mathcal{VD}$) ∈ *perfectly-shared-vars-rel*› ‹(*xs*, *xs′*) ∈ ⟨⟨*Id*⟩*list-rel*$\times_r$*Id*⟩*list-rel*›
shows ‹*import-polyS* $\mathcal{A}_0$ *xs* ≤ ⇓(*import-poly-rel* $\mathcal{A}_0$ *xs*)
(*import-poly* $\mathcal{VD}$ *xs′*)›
⟨*proof*⟩

definition *drop-content* :: ‹′*string* ⇒ (′*nat*, ′*string*) *vars* ⇒ (′*nat*, ′*string*) *vars nres*›
where
‹*drop-content* = (λ*v* $\mathcal{V}'$. *do* {
ASSERT(*v* ∈# $\mathcal{V}'$);
RETURN (*remove1-mset v* $\mathcal{V}'$)
})›

definition *drop-contentS* :: ‹′*string* ⇒ (′*nat*, ′*string*) *shared-vars* ⇒ (′*nat*, ′*string*) *shared-vars nres*›
where
‹*drop-contentS* = (λ*v* ($\mathcal{D}$, $\mathcal{V}$, $\mathcal{V}'$). *do* {
ASSERT(*v* ∈# *dom-m* $\mathcal{V}'$);
if count $\mathcal{D}$ *v* = *1*
then do {
let i = $\mathcal{V}'$ ∝ *v*;
RETURN (*remove1-mset v* $\mathcal{D}$, *fmdrop i* $\mathcal{V}$, *fmdrop v* $\mathcal{V}'$)
}
else
RETURN (*remove1-mset v* $\mathcal{D}$, $\mathcal{V}$, $\mathcal{V}'$)
})›

lemma *drop-contentS-drop-content*:
assumes ‹($\mathcal{A}$, $\mathcal{VD}$) ∈ *perfectly-shared-vars-rel*› ‹(*v*, *v′*) ∈ *Id*›
shows ‹*drop-contentS v* $\mathcal{A}$ ≤ ⇓*perfectly-shared-vars-rel* (*drop-content v′* $\mathcal{VD}$)›
⟨*proof*⟩

definition *perfectly-shared-strings-equal*
:: ‹(′*nat*, ′*string*) *vars* ⇒ ′*string* ⇒ ′*string* ⇒ *bool nres*›
where
‹*perfectly-shared-strings-equal* $\mathcal{V}$ *x y* = *do* {
ASSERT(*x* ∈# $\mathcal{V}$ ∧ *y* ∈# $\mathcal{V}$);
RETURN (*x* = *y*)
}›

definition *perfectly-shared-strings-equal-l*
:: ‹(′*nat*,′*string*)*shared-vars* ⇒ ′*nat* ⇒ ′*nat* ⇒ *bool nres*›
where
‹*perfectly-shared-strings-equal-l* $\mathcal{V}$ *x y* = *do* {
RETURN (*x* = *y*)

}›

lemma *perfectly-shared-strings-equal-l-perfectly-shared-strings-equal*:
assumes ‹$(\mathcal{A}, \mathcal{V}) \in$ *perfectly-shared-vars-rel*› **and**
‹$(x, x') \in$ *perfectly-shared-var-rel* $\mathcal{A}$› **and**
‹$(y, y') \in$ *perfectly-shared-var-rel* $\mathcal{A}$›
shows ‹*perfectly-shared-strings-equal-l* $\mathcal{A}$ x $y \leq \Downarrow$*bool-rel* (*perfectly-shared-strings-equal* $\mathcal{V}$ x' y')›
⟨*proof*⟩

datatype(**in** −) *ordered* = *LESS* | *EQUAL* | *GREATER* | *UNKNOWN*

definition (**in** −)*perfect-shared-var-order* :: ‹(*nat*, *string*)*vars* ⇒ *string* ⇒ *string* ⇒ *ordered nres*› **where**
‹*perfect-shared-var-order* $\mathcal{D}$ x y = *do* {
ASSERT($x \in\# \mathcal{D} \wedge y \in\# \mathcal{D}$);
eq ← *perfectly-shared-strings-equal* $\mathcal{D}$ x y;
if eq then RETURN EQUAL
else do {
x ← *get-var-name* $\mathcal{D}$ x;
y ← *get-var-name* $\mathcal{D}$ y;
if $(x, y) \in$ *var-order-rel then RETURN* (*LESS*)
else RETURN (*GREATER*)
}
}›

lemma *var-roder-rel-total*:
‹$y \neq ya \Longrightarrow (y, ya) \notin$ *var-order-rel* $\Longrightarrow (ya, y) \in$ *var-order-rel*›
⟨*proof*⟩

lemma *perfect-shared-var-order-spec*:
assumes ‹$xs \in\# \mathcal{V}$› ‹$ys \in\# \mathcal{V}$›
shows
‹*perfect-shared-var-order* $\mathcal{V}$ xs $ys \leq \Downarrow$ *Id* (*SPEC*(λb. ((b=*LESS* ⟶ $(xs, ys) \in$ *var-order-rel*) ∧
(b=*GREATER* ⟶ $(ys, xs) \in$ *var-order-rel* ∧ ¬$(xs, ys) \in$ *var-order-rel*) ∧
(b=*EQUAL* ⟶ $xs = ys$)) ∧ $b \neq$ *UNKNOWN*))›
⟨*proof*⟩

definition (**in** −) *perfect-shared-term-order-rel-pre*
:: ‹(*nat*, *string*) *vars* ⇒ *string list* ⇒ *string list* ⇒ *bool*›
where
‹*perfect-shared-term-order-rel-pre* $\mathcal{V}$ xs ys ⟷
set $xs \subseteq$ *set-mset* $\mathcal{V}$ ∧ *set* $ys \subseteq$ *set-mset* $\mathcal{V}$›

definition (**in** −) *perfect-shared-term-order-rel*
:: ‹(*nat*, *string*) *vars* ⇒ *string list* ⇒ *string list* ⇒ *ordered nres*›
where
‹*perfect-shared-term-order-rel* $\mathcal{V}$ xs ys = *do* {
ASSERT (*perfect-shared-term-order-rel-pre* $\mathcal{V}$ xs ys);
(b, -, -) ← $WHILE_T$ ($\lambda(b, xs, ys)$. b = *UNKNOWN*)
($\lambda(b, xs, ys)$. *do* {
if $xs = [] \wedge ys = []$ *then RETURN* (*EQUAL*, xs, ys)
else if $xs = []$ *then RETURN* (*LESS*, xs, ys)
else if $ys = []$ *then RETURN* (*GREATER*, xs, ys)
else do {
ASSERT($xs \neq [] \wedge ys \neq []$);

eq ← *perfect-shared-var-order* $\mathcal{V}$ (*hd xs*) (*hd ys*);
if eq = *EQUAL then RETURN* (*b*, *tl xs*, *tl ys*)
else RETURN (*eq*, *xs*, *ys*)
}
}) (*UNKNOWN*, *xs*, *ys*);
RETURN b
}›

lemma (**in** −)*perfect-shared-term-order-rel-spec*:
assumes ‹*set xs* ⊆ *set-mset* $\mathcal{V}$› ‹*set ys* ⊆ *set-mset* $\mathcal{V}$›
shows
‹*perfect-shared-term-order-rel* $\mathcal{V}$ *xs ys* ≤ ⇓ *Id* (*SPEC*(λ*b*. ((*b*=*LESS* ⟶ (*xs*, *ys*) ∈ *term-order-rel*) ∧
(*b*=*GREATER* ⟶ (*ys*, *xs*) ∈ *term-order-rel*) ∧
(*b*=*EQUAL* ⟶ *xs* = *ys*)) ∧ *b* ≠ *UNKNOWN*))› (**is** ‹- ≤ ⇓ - (*SPEC* (λ*b*. *?f b* ∧ *b* ≠ *UNKNOWN*))›)
⟨*proof*⟩

lemma (**in**−) *trans-var-order-rel*[*simp*]: ‹*trans var-order-rel*›
⟨*proof*⟩

lemma (**in**−) *term-order-rel-irreflexive*:
‹(*x1f*, *x1d*) ∈ *term-order-rel* ⟹ (*x1d*, *x1f*) ∈ *term-order-rel* ⟹ *x1f* = *x1d*›
⟨*proof*⟩

lemma *get-var-nameS-spec*:
fixes $\mathcal{DV}$ :: ‹('*nat*, '*string*) *vars*› **and**
$\mathcal{A}$ :: ‹('*nat*, '*string*) *shared-vars*› **and**
x' :: '*string*
assumes ‹($\mathcal{A}$, $\mathcal{DV}$) ∈ *perfectly-shared-vars-rel*› **and**
‹(*x*,*x'*) ∈ *perfectly-shared-var-rel* $\mathcal{A}$›
shows ‹*get-var-nameS* $\mathcal{A}$ *x* ≤ ⇓(*Id*) (*get-var-name* $\mathcal{DV}$ *x'*)›
⟨*proof*⟩

lemma *get-var-nameS-spec2*:
fixes $\mathcal{DV}$ :: ‹('*nat*, '*string*) *vars*› **and**
$\mathcal{A}$ :: ‹('*nat*, '*string*) *shared-vars*› **and**
x' :: '*string*
assumes ‹($\mathcal{A}$, $\mathcal{DV}$) ∈ *perfectly-shared-vars-rel*› **and**
‹(*x*,*x'*) ∈ *perfectly-shared-var-rel* $\mathcal{A}$›
‹*x'* ∈# $\mathcal{DV}$›
shows ‹*get-var-nameS* $\mathcal{A}$ *x* ≤ ⇓(*Id*) (*RETURN x'*)›
⟨*proof*⟩

end
theory *EPAC-Efficient-Checker*
imports *EPAC-Checker EPAC-Perfectly-Shared*
begin
hide-const (**open**) *PAC-Checker.full-checker-l*
hide-fact (**open**) *PAC-Checker.full-checker-l-def*

type-synonym *shared-poly* = ‹(*nat list* × *int*) *list*›

definition (**in** −) *add-poly-l'* **where**

‹*add-poly-l′ - = add-poly-l*›

definition (**in** −)*add-poly-l-prep* :: ‹(*nat*,*string*)*vars* ⇒ *llist-polynomial* × *llist-polynomial* ⇒ *llist-polynomial nres*› **where**
‹*add-poly-l-prep* $\mathcal{D}$ = REC_T
(λ*add-poly-l* (*p*, *q*).
case (*p*,*q*) *of*
(*p*, []) ⇒ *RETURN p*
| ([], *q*) ⇒ *RETURN q*
| ((*xs*, *n*) # *p*, (*ys*, *m*) # *q*) ⇒ *do* {
comp ← *perfect-shared-term-order-rel* $\mathcal{D}$ *xs ys*;
if comp = *EQUAL 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 comp = *LESS*
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*)
}
})›

lemma *add-poly-alt-def*[*unfolded conc-Id id-apply*]:
fixes *xs ys* :: *llist-polynomial*
assumes ‹⋃(*set* ‘ (*fst*‘*set xs*)) ⊆ *set-mset* $\mathcal{D}$› ‹⋃(*set* ‘ *fst* ‘ *set ys*) ⊆ *set-mset* $\mathcal{D}$›
shows ‹*add-poly-l-prep* $\mathcal{D}$ (*xs*, *ys*) ≤ ⇓ *Id* (*add-poly-l′* $\mathcal{D}$ (*xs*, *ys*))›
⟨*proof*⟩

definition (**in** −)*normalize-poly-shared*
:: ‹(*nat*,*string*) *vars* ⇒ *llist-polynomial* ⇒
(*bool* × *llist-polynomial*) *nres*›
where
‹*normalize-poly-shared* $\mathcal{A}$ *xs* = *do* {
xs ← *full-normalize-poly xs*;
import-poly-no-new $\mathcal{A}$ *xs*
}›

definition *normalize-poly-sharedS*
:: ‹(*nat*,*string*) *shared-vars* ⇒ *llist-polynomial* ⇒
(*bool* × *shared-poly*) *nres*›
where
‹*normalize-poly-sharedS* $\mathcal{A}$ *xs* = *do* {
xs ← *full-normalize-poly xs*;
import-poly-no-newS $\mathcal{A}$ *xs*
}›

definition (**in** −) *mult-monoms-prep* :: ‹(*nat*,*string*)*vars* ⇒ *term-poly-list* ⇒ *term-poly-list* ⇒ *term-poly-list nres*› **where**
‹*mult-monoms-prep* $\mathcal{D}$ *xs ys* = REC_T (λ*f* (*xs*, *ys*).
do {

if xs = [] *then RETURN ys*
else if ys = [] *then RETURN xs*
else do {
ASSERT(*xs* ≠ [] ∧ *ys* ≠ []);
comp ← *perfect-shared-var-order* $\mathcal{D}$ (*hd xs*) (*hd ys*);
if comp = *EQUAL then do* {
pq ← *f* (*tl xs*, *tl ys*);
RETURN (*hd xs* # *pq*)
}
else if comp = *LESS then do* {
pq ← *f* (*tl xs*, *ys*);
RETURN (*hd xs* # *pq*)
}
else do {
pq ← *f* (*xs*, *tl ys*);
RETURN (*hd ys* # *pq*)
}
}
}) (*xs*, *ys*)›

lemma (**in** −) *mult-monoms-prep-mult-monoms*:
assumes ‹*set xs* ⊆ *set-mset* $\mathcal{V}$› ‹*set ys* ⊆ *set-mset* $\mathcal{V}$›
shows ‹*mult-monoms-prep* $\mathcal{V}$ *xs ys* ≤ ⇓*Id* (*SPEC* ((=) (*mult-monoms xs ys*)))›
⟨*proof*⟩
definition *mult-monoms-prop* :: ‹(*nat*,*string*)*vars*⇒ *llist-polynomial* ⇒ - ⇒ *llist-polynomial* ⇒ *llist-polynomial nres*› **where**
‹*mult-monoms-prop* = (λ$\mathcal{V}$ *qs* (*p*, *m*) *b*. *nfoldli qs* (λ-. *True*) (λ(*q*, *n*) *b*. *do* {*pq* ← *mult-monoms-prep* $\mathcal{V}$ *p q*; *RETURN* ((*pq*, *m* ∗ *n*) # *b*)}) *b*)›

definition *mult-poly-raw-prop* :: ‹(*nat*,*string*) *vars*⇒ *llist-polynomial* ⇒ *llist-polynomial* ⇒ *llist-polynomial nres*› **where**
‹*mult-poly-raw-prop* $\mathcal{V}$ *p q* = *nfoldli p* (λ-. *True*) (*mult-monoms-prop* $\mathcal{V}$ *q*) []›

lemma *mult-monoms-prop-mult-monomials*:
assumes ‹*vars-llist qs* ⊆ *set-mset* $\mathcal{V}$› ‹*set* (*fst m*) ⊆ *set-mset* $\mathcal{V}$›
shows ‹*mult-monoms-prop* $\mathcal{V}$ *qs m b* ≤ ⇓{(*xs*, *ys*). *mset xs* = *mset ys*} (*RES*{*map* (*mult-monomials m*) *qs* @ *b*})›
⟨*proof*⟩

lemma *mult-poly-raw-prop-mult-poly-raw*:
assumes ‹*vars-llist qs* ⊆ *set-mset* $\mathcal{V}$› ‹*vars-llist ps* ⊆ *set-mset* $\mathcal{V}$›
shows ‹*mult-poly-raw-prop* $\mathcal{V}$ *ps qs* ≤
(*SPEC* (λ*c*. (*c*, *PAC-Polynomials-Operations.mult-poly-raw ps qs*) ∈ {(*xs*, *ys*). *mset xs* = *mset ys*}))›
⟨*proof*⟩

definition (**in** −) *mult-poly-full-prop* :: ‹-› **where**
‹*mult-poly-full-prop* $\mathcal{V}$ *p q* = *do* {
pq ← *mult-poly-raw-prop* $\mathcal{V}$ *p q*;
ASSERT(*vars-llist pq* ⊆ *vars-llist p* ∪ *vars-llist q*);
normalize-poly pq
}›

lemma *vars-llist-mset-eq*: ‹*mset p* = *mset q* ⟹ *vars-llist p* = *vars-llist q*›

⟨*proof*⟩
lemma *mult-poly-full-prop-mult-poly-full*:
assumes ‹*vars-llist qs* ⊆ *set-mset* 𝒱› ‹*vars-llist ps* ⊆ *set-mset* 𝒱›
‹(*ps*, *ps'*) ∈ *Id*› ‹(*qs*, *qs'*) ∈ *Id*›
shows ‹*mult-poly-full-prop* 𝒱 *ps qs* ≤ ⇓*Id* (*mult-poly-full ps' qs'*)›
⟨*proof*⟩

definition (**in** −) *linear-combi-l-prep2* **where**
‹*linear-combi-l-prep2 i A* 𝒱 *xs* = *do* {
  ASSERT(*linear-combi-l-pre i A* (*set-mset* 𝒱) *xs*);
  WHILE$_T$
    (λ(*p*, *xs*, *err*). *xs* ≠ [] ∧ ¬*is-cfailed err*)
    (λ(*p*, *xs*, -). *do* {
      ASSERT(*xs* ≠ []);
      let (q_0 :: *llist-polynomial*, *i*) = *hd xs*;
      if (*i* ∉# *dom-m A* ∨ ¬(*vars-llist* q_0 ⊆ *set-mset* 𝒱))
      then do {
        err ← *check-linear-combi-l-dom-err* q_0 *i*;
        RETURN (*p*, *xs*, *error-msg i err*)
      } *else do* {
        ASSERT(*fmlookup A i* ≠ *None*);
        let r = *the* (*fmlookup A i*);
        ASSERT(*vars-llist r* ⊆ *set-mset* 𝒱);
        if q_0 = [([],*1*)] *then do* {
          pq ← *add-poly-l-prep* 𝒱 (*p*, *r*);
          RETURN (*pq*, *tl xs*, *CSUCCESS*)
        } *else do* {
          (-, *q*) ← *normalize-poly-shared* 𝒱 (q_0);
          ASSERT(*vars-llist q* ⊆ *set-mset* 𝒱);
          pq ← *mult-poly-full-prop* 𝒱 *q r*;
          ASSERT(*vars-llist pq* ⊆ *set-mset* 𝒱);
          pq ← *add-poly-l-prep* 𝒱 (*p*, *pq*);
          RETURN (*pq*, *tl xs*, *CSUCCESS*)
        }
      }
    })
    ([], *xs*, *CSUCCESS*)
  }›
lemma (**in** −) *import-poly-no-new-spec*:
‹*import-poly-no-new* 𝒱 *xs* ≤ ⇓*Id* (*SPEC*(λ(*b*, *xs'*). (¬*b* ⟶ *xs* = *xs'*) ∧ (¬*b* ⟷ *vars-llist xs* ⊆ *set-mset* 𝒱)))›
⟨*proof*⟩

lemma *linear-combi-l-prep2-linear-combi-l*:
assumes 𝒱: ‹(𝒱,𝒱') ∈ {(*x*, *y*). *y* = *set-mset x*}›‹(*i*,*i'*)∈*nat-rel*›‹(*A*,*A'*)∈*Id*›‹(*xs*,*xs'*)∈*Id*›
shows ‹*linear-combi-l-prep2 i A* 𝒱 *xs* ≤ ⇓*Id* (*linear-combi-l i' A'* 𝒱' *xs'*)›
⟨*proof*⟩

definition *check-linear-combi-l-prop* **where**
‹*check-linear-combi-l-prop spec A* 𝒱 *i xs r* = *do* {
(mem-err, *r*) ← *import-poly-no-new* 𝒱 *r*;
if mem-err ∨ *i* ∈# *dom-m A* ∨ *xs* = []
then do {
  err ← *check-linear-combi-l-pre-err i* (*i* ∈# *dom-m A*) (*xs* = []) (*mem-err*);
  RETURN (*error-msg i err*, *r*)

```
  }
  else do {
    (p, -, err) ← linear-combi-l-prep2 i A 𝒱 xs;
    if (is-cfailed err)
    then do {
      RETURN (err, r)
    }
    else do {
      b ← weak-equality-l p r;
      b′ ← weak-equality-l r spec;
      if b then (if b′ then RETURN (CFOUND, r) else RETURN (CSUCCESS, r)) else do {
        c ← check-linear-combi-l-mult-err p r;
        RETURN (error-msg i c, r)
      }
    }
  }}›
```

lemma *check-linear-combi-l-prop-check-linear-combi-l*:
assumes ‹(𝒱,𝒱′) ∈ {(x, y). y = set-mset x}› ‹(A, A′) ∈ Id› ‹(i,i′)∈nat-rel› ‹(xs,xs′)∈Id›‹(r,r′)∈Id› ‹(spec,spec′)∈Id›
shows ‹check-linear-combi-l-prop spec A 𝒱 i xs r ≤ ⇓{((b,r′), b′). b=b′ ∧ (¬is-cfailed b ⟶ r=r′)} (check-linear-combi-l spec′ A′ 𝒱′ i′ xs′ r′)›
⟨*proof*⟩

definition (**in** −)*check-extension-l2-prop*
:: ‹- ⇒ - ⇒ string multiset ⇒ nat ⇒ string ⇒ llist-polynomial ⇒ (string code-status × llist-polynomial × string multiset × string) nres›
where

```
‹check-extension-l2-prop spec A 𝒱 i v p = do {
  (pre, nonew, mem, mem′, p, 𝒱, v) ← do {
      let pre = i ∉# dom-m A ∧ v ∉ set-mset 𝒱;
      let b = vars-llist p ⊆ set-mset 𝒱;
      (mem, p, 𝒱) ← import-poly 𝒱 p;
      (mem′, 𝒱, v) ← if b ∧ pre ∧ ¬ alloc-failed mem then import-variable v 𝒱 else RETURN (mem, 𝒱, v);
      RETURN (pre ∧ ¬alloc-failed mem ∧ ¬ alloc-failed mem′, b, mem, mem′, p, 𝒱, v)
   };
  if ¬pre
  then do {
    c ← check-extension-l-dom-err i;
    RETURN (error-msg i c, [], 𝒱, v)
  } else do {
      if ¬nonew
      then do {
        c ← check-extension-l-new-var-multiple-err v p;
        RETURN (error-msg i c, [], 𝒱, v)
      }
      else do {
         ASSERT(vars-llist p ⊆ set-mset 𝒱);
         p2 ←  mult-poly-full-prop 𝒱 p p;
         ASSERT(vars-llist p2 ⊆ set-mset 𝒱);
         let p′′ = map (λ(a,b). (a, −b)) p;
         ASSERT(vars-llist p′′ ⊆ set-mset 𝒱);
         q ← add-poly-l-prep 𝒱 (p2, p′′);
         ASSERT(vars-llist q ⊆ set-mset 𝒱);
```

```
        eq ← weak-equality-l q [];
        if eq then do {
          RETURN (CSUCCESS, p, 𝒱, v)
        } else do {
         c ← check-extension-l-side-cond-err v p q;
         RETURN (error-msg i c, [], 𝒱, v)
        }
      }
    }
  }›
```

lemma *check-extension-l2-prop-check-extension-l2*:
  assumes ‹(𝒱,𝒱′) ∈ {(x, y). y = set-mset x}› ‹(spec, spec′) ∈ Id› ‹(A, A′) ∈ Id› ‹(i,i′) ∈ nat-rel› ‹(v, v′) ∈ Id› ‹(p, p′) ∈ Id›
  shows ‹check-extension-l2-prop spec A 𝒱 i v p ≤⇓{((err, q, 𝒜, va), b). (b = err) ∧ (¬is-cfailed err ⟶ q=p ∧ v=va ∧ set-mset 𝒜 = insert v 𝒱′)}
    (check-extension-l2 spec′ A′ 𝒱′ i′ v′ p′)›
⟨proof⟩

definition *PAC-checker-l-step-prep* :: ‹- ⇒ string code-status × string multiset × - ⇒ (llist-polynomial, string, nat) pac-step ⇒ -› **where**

```
  ‹PAC-checker-l-step-prep = (λspec (st′, 𝒱, A) st. do {
    ASSERT (PAC-checker-l-step-inv spec st′ (set-mset 𝒱) A);
    ASSERT (¬is-cfailed st′);
    case st of
     CL - - - ⇒
       do {
        r ← full-normalize-poly (pac-res st);
        (eq, r) ← check-linear-combi-l-prop spec A 𝒱 (new-id st) (pac-srcs st) r;
        let - = eq;
        if ¬is-cfailed eq
        then RETURN (merge-cstatus st′ eq, 𝒱, fmupd (new-id st) r A)
       else RETURN (eq, 𝒱, A)
     }
    | Del - ⇒
       do {
        eq ← check-del-l spec A (pac-src1 st);
        let - = eq;
        if ¬is-cfailed eq
        then RETURN (merge-cstatus st′ eq, 𝒱, fmdrop (pac-src1 st) A)
        else RETURN (eq, 𝒱, A)
     }
   | Extension - - - ⇒
       do {
         r ← full-normalize-poly (pac-res st);
        (eq, r, 𝒱, v) ← check-extension-l2-prop spec A (𝒱) (new-id st) (new-var st) r;
        if ¬is-cfailed eq
      then do {
        r ← add-poly-l-prep 𝒱 ([([v], −1)], r);
        RETURN (st′, 𝒱, fmupd (new-id st) r A)
        }
        else RETURN (eq, 𝒱, A)
     }}
          )›
```

lemma *PAC-checker-l-step-prep-PAC-checker-l-step*:
  assumes ‹(*state*, *state'*) ∈ {((*st*, $\mathcal{V}$, *A*), (*st'*, $\mathcal{V}'$, *A'*)). (*st*,*st'*)∈*Id* ∧ (*A*,*A'*)∈*Id* ∧ (¬*is-cfailed st* ⟶ ($\mathcal{V}$,$\mathcal{V}'$)∈ {(*x*, *y*). *y* = *set-mset x*})}›
    ‹(*spec*,*spec'*)∈*Id*›
    ‹(*step*,*step'*)∈*Id*›
  shows ‹*PAC-checker-l-step-prep spec state step* ≤
    ⇓{((*st*, $\mathcal{V}$, *A*), (*st'*, $\mathcal{V}'$, *A'*)). (*st*,*st'*)∈*Id* ∧ (*A*,*A'*)∈*Id* ∧ (¬*is-cfailed st* ⟶ ($\mathcal{V}$,$\mathcal{V}'$)∈ {(*x*, *y*). *y* = *set-mset x*})}
    (*PAC-checker-l-step spec' state' step'*)›
⟨*proof*⟩

definition (**in** −) *remap-polys-l2-with-err*
  :: ‹*llist-polynomial* ⇒ *llist-polynomial* ⇒ (*nat*, *string*) *vars* ⇒ (*nat*, *llist-polynomial*) *fmap* ⇒
  (*string code-status* × (*nat*, *string*) *vars* × (*nat*, *llist-polynomial*) *fmap*) *nres*› **where**
  ‹*remap-polys-l2-with-err spec' spec0* = (λ($\mathcal{V}$:: (*nat*, *string*) *vars*) *A*. *do*{
   ASSERT(*vars-llist spec'* ⊆ *vars-llist spec0*);
   dom ← *SPEC*(λ*dom*. *set-mset* (*dom-m A*) ⊆ *dom* ∧ *finite dom*);
   (*mem*, $\mathcal{V}$) ← *SPEC*(λ(*mem*, $\mathcal{V}'$). ¬*alloc-failed mem* ⟶ *set-mset* $\mathcal{V}'$ = *set-mset* $\mathcal{V}$ ∪ *vars-llist spec0*);
   (*mem'*, *spec*, $\mathcal{V}$) ← *if* ¬*alloc-failed mem then import-poly* $\mathcal{V}$ *spec' else SPEC*(λ-. *True*);
   failed ← *SPEC*(λ*b*::*bool*. *alloc-failed mem* ∨ *alloc-failed mem'* ⟶ *b*);
   ASSERT(¬*failed* ⟶ *spec* = *spec'*);
   if failed
   then do {
     c ← *remap-polys-l-dom-err*;
     SPEC (λ(*mem*, -, -). *mem* = *error-msg* (*0*::*nat*) *c*)
   }
   else do {
     (*err*, $\mathcal{V}$, *A*) ← $FOREACH_C$ *dom* (λ(*err*, $\mathcal{V}$, *A'*). ¬*is-cfailed err*)
       (λ*i* (*err*, $\mathcal{V}$, *A'*).
         if i ∈# *dom-m A*
         then do {
          (*err'*, *p*, $\mathcal{V}$) ← *import-poly* $\mathcal{V}$ (*the* (*fmlookup A i*));
           if alloc-failed err' then RETURN((*CFAILED* ''*memory out*'', $\mathcal{V}$, *A'*))
           else do {
             ASSERT(*vars-llist p* ⊆ *set-mset* $\mathcal{V}$);
             p ← *full-normalize-poly p*;
             eq ← *weak-equality-l p spec*;
             let $\mathcal{V}$ = $\mathcal{V}$;
             RETURN((*if eq then CFOUND else CSUCCESS*), $\mathcal{V}$, *fmupd i p A'*)
           }
         } *else RETURN* (*err*, $\mathcal{V}$, *A'*))
       (*CSUCCESS*, $\mathcal{V}$, *fmempty*);
     RETURN (*err*, $\mathcal{V}$, *A*)
  }})›

lemma *remap-polys-l-with-err-alt-def*:
  ‹*remap-polys-l-with-err spec spec0* = (λ$\mathcal{V}$ *A*. *do*{
  ASSERT (*remap-polys-l-with-err-pre spec spec0* $\mathcal{V}$ *A*);
  dom ← *SPEC*(λ*dom*. *set-mset* (*dom-m A*) ⊆ *dom* ∧ *finite dom*);
   $\mathcal{V}$ ← *RETURN* ($\mathcal{V}$ ∪ *vars-llist spec0*);
   spec ← *RETURN spec*;
   failed ← *SPEC*(λ-::*bool*. *True*);
   if failed
   then do {

c ← remap-polys-l-dom-err;
SPEC (λ(mem, -, -). mem = error-msg (0::nat) c)
}
else do {
(err, 𝒱, A) ← $FOREACH_C$ dom (λ(err, 𝒱, A′). ¬is-cfailed err)
(λi (err, 𝒱, A′).
if i ∈# dom-m A
then do {
err′ ← SPEC(λerr. err ≠ CFOUND);
if is-cfailed err′ then RETURN((err′, 𝒱, A′))
else do {
p ← full-normalize-poly (the (fmlookup A i));
eq ← weak-equality-l p spec;
𝒱 ← RETURN(𝒱 ∪ vars-llist (the (fmlookup A i)));
RETURN((if eq then CFOUND else CSUCCESS), 𝒱, fmupd i p A′)
}
} else RETURN (err, 𝒱, A′))
(CSUCCESS, 𝒱, fmempty);
RETURN (err, 𝒱, A)
}})›
⟨proof⟩

lemma *remap-polys-l2-with-err-polys-l2-with-err*:
assumes ‹(𝒱, 𝒱′) ∈ {(x, y). y = set-mset x}› ‹(A,A′) ∈ Id› ‹(spec, spec′)∈Id› ‹(spec0, spec0′)∈Id›
shows ‹remap-polys-l2-with-err spec spec0 𝒱 A ≤ ⇓{((st, 𝒱, A), st′, 𝒱′, A′).
(st, st′) ∈ Id ∧
(A, A′) ∈ Id ∧
(¬ is-cfailed st ⟶ (𝒱, 𝒱′) ∈ {(x, y). y = set-mset x})}
(remap-polys-l-with-err spec′ spec0′ 𝒱′ A′)›
⟨proof⟩

definition *PAC-checker-l2* **where**
‹PAC-checker-l2 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-prep spec bA (hd n);
RETURN (S, tl n)
})
((b, A), st);
RETURN S
}›

lemma *PAC-checker-l2-PAC-checker-l*:
assumes ‹(A, A′) ∈ {(x, y). y = set-mset x} $\times_r$ Id› ‹(spec, spec′)∈Id› ‹(st, st′)∈Id› ‹(b,b′)∈Id›
shows ‹PAC-checker-l2 spec A b st ≤ ⇓{((b, A, st), (b′, A′, st′)).
(¬is-cfailed b ⟶ (A, A′) ∈ {(x, y). y = set-mset x} ∧ (st, st′)∈Id) ∧ (b,b′)∈Id} (PAC-checker-l spec′ A′ b′ st′)›
⟨proof⟩

definition (**in** −) *remap-polys-l2-with-err-prep* :: ‹llist-polynomial ⇒ llist-polynomial ⇒ (nat, string) vars ⇒ (nat, llist-polynomial) fmap ⇒
(string code-status × (nat, string) vars × (nat, llist-polynomial) fmap × llist-polynomial) nres› **where**
‹remap-polys-l2-with-err-prep spec spec0 = (λ(𝒱:: (nat, string) vars) A. do{

ASSERT(vars-llist spec ⊆ vars-llist spec0);
dom ← SPEC(λdom. set-mset (dom-m A) ⊆ dom ∧ finite dom);
(mem, 𝒱) ← SPEC(λ(mem, 𝒱′). ¬alloc-failed mem ⟶ set-mset 𝒱′ = set-mset 𝒱 ∪ vars-llist spec0);
(mem′, spec, 𝒱) ← if ¬alloc-failed mem then import-poly 𝒱 spec else SPEC(λ-. True);
failed ← SPEC(λb::bool. alloc-failed mem ∨ alloc-failed mem′ ⟶ b);
if failed
then do {
c ← remap-polys-l-dom-err;
SPEC (λ(mem, -, -, -). mem = error-msg (0::nat) c)
}
else do {
(err, 𝒱, A) ← $FOREACH_C$ dom (λ(err, 𝒱, A′). ¬is-cfailed err)
(λi (err, 𝒱, A′).
if i ∈# dom-m A
then do {
(err′, p, 𝒱) ← import-poly 𝒱 (the (fmlookup A i));
if alloc-failed err′ then RETURN((CFAILED ′′memory out′′, 𝒱, A′))
else do {
ASSERT(vars-llist p ⊆ set-mset 𝒱);
p ← full-normalize-poly p;
eq ← weak-equality-l p spec;
let 𝒱 = 𝒱;
RETURN((if eq then CFOUND else CSUCCESS), 𝒱, fmupd i p A′)
}
} else RETURN (err, 𝒱, A′))
(CSUCCESS, 𝒱, fmempty);
RETURN (err, 𝒱, A, spec)
}})›

lemma *remap-polys-l2-with-err-prep-remap-polys-l2-with-err*:
assumes ‹(p, p′) ∈ Id› ‹(q, q′) ∈ Id› ‹(A,A′) ∈ ⟨Id, Id⟩fmap-rel› **and** ‹(V,V′) ∈ Id›
shows ‹remap-polys-l2-with-err-prep p q V A ≤ ⇓{((b, A, st, spec′), (b′, A′, st′)).
((b, A, st), (b′, A′, st′)) ∈ Id ∧
(¬is-cfailed b ⟶ spec′ = p′)} (remap-polys-l2-with-err p′ q′ V′ A′)›
⟨proof⟩

definition *full-checker-l-prep*
:: ‹llist-polynomial ⇒ (nat, llist-polynomial) fmap ⇒ (-, string, nat) pac-step list ⇒
(string code-status × -) nres›
where
‹full-checker-l-prep spec A st = do {
spec′ ← full-normalize-poly spec;
(b, 𝒱, A, spec) ← remap-polys-l2-with-err-prep spec′ spec {#} A;
if is-cfailed b
then RETURN (b, 𝒱, A)
else do {
let 𝒱 = 𝒱;
PAC-checker-l2 spec (𝒱, A) b st
}
}›

lemma *remap-polys-l2-with-err-polys-l-with-err*:
assumes ‹(𝒱, 𝒱′) ∈ {(x, y). y = set-mset x}› ‹(A,A′) ∈ Id› ‹(spec, spec′)∈Id› ‹(spec0, spec0′)∈Id›
shows ‹remap-polys-l2-with-err-prep spec spec0 𝒱 A ≤ ⇓{((st, 𝒱, A, spec′′), st′, 𝒱′, A′).
(st, st′) ∈ Id ∧

(A, A′) ∈ Id ∧
(¬ is-cfailed st ⟶ (𝒱, 𝒱′) ∈ {(x, y). y = set-mset x} ∧ spec′′ = spec)}
(remap-polys-l-with-err spec′ spec0′ 𝒱′ A′)›
⟨proof⟩

lemma *full-checker-l-prep-full-checker-l*:
assumes ‹(spec, spec′)∈Id› ‹(st, st′)∈Id› ‹(A,A′)∈Id›
shows ‹full-checker-l-prep spec A st ≤⇓{((b, A, st), (b′, A′, st′)).
(¬is-cfailed b ⟶ (A, A′) ∈ {(x, y). y = set-mset x} ∧ (st, st′)∈Id) ∧ (b,b′)∈Id}
(full-checker-l spec′ A′ st′)›
⟨proof⟩

lemma *full-checker-l-prep-full-checker-l2′*:
shows ‹(uncurry2 full-checker-l-prep, uncurry2 full-checker-l) ∈ (Id $\times_r$ Id) $\times_r$ Id $\rightarrow_f$
⟨{((b, A, st), (b′, A′, st′)). (¬is-cfailed b ⟶ (A, A′) ∈ {(x, y). y = set-mset x} ∧ (st, st′)∈Id) ∧ (b,b′)∈Id}⟩nres-rel›
⟨proof⟩

end
theory *EPAC-Perfectly-Shared-Vars*
imports *EPAC-Perfectly-Shared*
PAC-Checker.PAC-Checker-Relation
PAC-Checker.PAC-Map-Rel
begin
thm *import-variableS-def*
term *hm.assn*
term *iam.assn*
term *is-iam*
term *iam-rel*

type-synonym (′string2, ′nat) shared-vars-c = ‹′string2 list × (′string2, ′nat) fmap›

definition *perfect-shared-vars-rel-c* :: ‹(′string2 × ′string) set ⇒ ((′string2, nat) shared-vars-c × (nat, ′string)shared-vars) set› **where**
‹perfect-shared-vars-rel-c R =
{((𝒱, 𝒜), (𝒟′, 𝒱′, 𝒜′)). (∀ i∈#dom-m 𝒱′. i < length 𝒱) ∧
(∀ i∈#dom-m 𝒱′. i < length 𝒱 ∧ (𝒱 ! i, the (fmlookup 𝒱′ i))∈ R) ∧
(𝒜, 𝒜′) ∈ ⟨R,nat-rel⟩fmap-rel}›

Random conditions with the idea to use machine words eventually

definition *find-new-idx-c* :: ‹(′string, nat) shared-vars-c ⇒ (memory-allocation × nat) nres› **where**
‹find-new-idx-c = (λ(𝒱, 𝒜). let k = length 𝒱 in if k < 2^63−1 then RETURN (Allocated, k) else RETURN (Mem-Out, 0))›

definition *insert-variable-c* :: ‹′string ⇒ nat ⇒ (′string, nat) shared-vars-c ⇒ (′string, nat) shared-vars-c›
where
‹insert-variable-c v k′ = (λ(𝒱, 𝒜). (𝒱 @ [v], fmupd v k′ 𝒜))›

definition *import-variable-c* :: ‹′string ⇒ (′string, nat) shared-vars-c ⇒ (memory-allocation × (′string, nat) shared-vars-c × nat) nres› **where**
‹import-variable-c v = (λ(𝒱𝒜). do {
(err, k′) ← find-new-idx-c (𝒱𝒜);
if alloc-failed err then do {let k′=k′; RETURN (err, (𝒱𝒜), k′)}

else do{
ASSERT(*k*′ < *2^63−1*);
RETURN (*Allocated*, *insert-variable-c v k*′ 𝒱𝒜, *k*′)
}
})›

lemma *import-variable-c-alt-def*:
‹*import-variable-c v* = (λ(𝒱, 𝒜). *do* {
(*err*, *k*′) ← *find-new-idx-c* (𝒱, 𝒜);
if alloc-failed err then do {*let k*′=*k*′; *RETURN* (*err*, (𝒱, 𝒜), *k*′)}
else do{
ASSERT(*k*′ < *2^63−1*);
RETURN (*Allocated*, (𝒱 @ [*v*], *fmupd v k*′ 𝒜), *k*′)
}
})›
⟨*proof*⟩

lemma *import-variable-c-import-variableS*:
fixes *A*′ :: ‹(*nat*,′*string*) *shared-vars*›
assumes
A: ‹(*A*,*A*′)∈*perfect-shared-vars-rel-c R*› **and**
v: ‹(*v*,*v*′)∈*R*› ‹*single-valued R*› ‹*single-valued* (R^{-1})›
shows ‹*import-variable-c v A* ≤⇓(*Id* $\times_r$ (*perfect-shared-vars-rel-c R* $\times_r$ *nat-rel*)) (*import-variableS v*′ *A*′)›
⟨*proof*⟩

definition *is-new-variable-c* :: ‹′*string* ⇒ (′*string*, ′*nat*) *shared-vars-c* ⇒ *bool nres*› **where**
‹*is-new-variable-c v* = (λ(𝒱, 𝒱′).
RETURN (*v* ∉# *dom-m* 𝒱′)
)›

lemma *fset-fmdom-dom-m*: ‹*fset* (*fmdom A*) = *set-mset* (*dom-m A*)›
⟨*proof*⟩

lemma *fmap-rel-nat-rel-dom-m-iff*:
‹(*A*, *B*) ∈ ⟨*R*, *S*⟩*fmap-rel* ⟹ (*v*,*v*′)∈*R* ⟹ *v*∈#*dom-m A* ⟷*v*′∈# *dom-m B*›
⟨*proof*⟩

lemma *is-new-variable-c-is-new-variableS*:
shows ‹(*uncurry is-new-variable-c*, *uncurry is-new-variableS*) ∈ *R* $\times_r$ *perfect-shared-vars-rel-c R* $\to_f$ ⟨*bool-rel*⟩*nres-rel*›
⟨*proof*⟩

definition *get-var-pos-c* :: ‹ (′*string*, *nat*) *shared-vars-c* ⇒ - ⇒ *nat nres*› **where**
‹*get-var-pos-c* = (λ(*xs*, 𝒱) *x*. *do* {
ASSERT(*x* ∈# *dom-m* 𝒱);
RETURN (*the* (*fmlookup* 𝒱 *x*))
})›

lemma *get-var-pos-c-get-var-posS*:

fixes A' :: ‹(*nat*,′*string*) *shared-vars*›
assumes
V: ‹*single-valued R*› ‹*single-valued* (R^{-1})›
shows ‹(*uncurry get-var-pos-c*, *uncurry get-var-posS*) ∈ *perfect-shared-vars-rel-c* $R \times_r R \rightarrow_f$ ⟨*nat-rel*⟩*nres-rel*›
⟨*proof*⟩

definition *get-var-name-c* :: ‹ (′*string*, *nat*) *shared-vars-c* ⇒ *nat* ⇒ ′*string nres*› **where**
‹*get-var-name-c* = (λ(*xs*, $\mathcal{V}$) *x*. *do* {
ASSERT(*x* < *length xs*);
RETURN (*xs* ! *x*)
})›

lemma *get-var-name-c-get-var-nameS*:
fixes A' :: ‹(*nat*,′*string*) *shared-vars*›
assumes
V: ‹*single-valued R*› ‹*single-valued* (R^{-1})›
shows ‹(*uncurry get-var-name-c*, *uncurry get-var-nameS*) ∈ *perfect-shared-vars-rel-c* $R \times_r$ *Id* $\rightarrow_f$ ⟨*R*⟩*nres-rel*›
⟨*proof*⟩

abbreviation *perfect-shared-vars-assn* :: ‹(*string*, *nat*) *shared-vars-c* ⇒ - ⇒ *assn*› **where**
‹*perfect-shared-vars-assn* ≡ *arl-assn string-assn* $\times_a$ *hm-fmap-assn string-assn uint64-nat-assn*›
abbreviation *shared-vars-assn* **where**
‹*shared-vars-assn* ≡ *hr-comp perfect-shared-vars-assn* (*perfect-shared-vars-rel-c Id*)›

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

sepref-definition *get-var-pos-c-impl*
is ‹*uncurry get-var-pos-c*›
:: ‹$\textit{perfect-shared-vars-assn}^k *_a \textit{string-assn}^k \rightarrow_a$ *uint64-nat-assn*›
⟨*proof*⟩

sepref-definition *is-new-variable-c-impl*
is ‹*uncurry is-new-variable-c*›
:: ‹$\textit{string-assn}^k *_a \textit{perfect-shared-vars-assn}^k \rightarrow_a$ *bool-assn*›
⟨*proof*⟩

definition *nth-uint64* **where**
‹*nth-uint64* = (!)›

definition *arl-get′* :: ‹′*a*::*heap array-list* ⇒ *integer* ⇒ ′*a Heap*› **where**
[*code del*]: ‹*arl-get′ a i* = *arl-get a* (*nat-of-integer i*)›

definition *arl-get-u* :: ‹′*a*::*heap array-list* ⇒ *uint64* ⇒ ′*a Heap*› **where**
‹*arl-get-u* ≡ λ*a i*. *arl-get′ a* (*integer-of-uint64 i*)›

lemma *arl-get-hnr-u*[*sepref-fr-rules*]:
assumes ‹*CONSTRAINT is-pure A*›
shows ‹(*uncurry arl-get-u*, *uncurry* (*RETURN* ∘∘ *op-list-get*))
∈ $[\textit{pre-list-get}]_a\ (\textit{arl-assn}\ A)^k *_a \textit{uint64-nat-assn}^k \rightarrow A$›
⟨*proof*⟩

definition *arl-get-u′* **where**
[*symmetric*, *code*]: ‹*arl-get-u′* = *arl-get-u*›

lemma *arl-get′-nth′*[*code*]: ‹*arl-get′* = (λ(*a*, *n*). *Array.nth′ a*)›
⟨*proof*⟩

definition *nat-of-uint64-s* :: ‹*nat* ⇒ *nat*› **where**
[*simp*]: ‹*nat-of-uint64-s x* = *x*›

lemma [*refine*]:
‹(*return o nat-of-uint64*, *RETURN o nat-of-uint64-s*) ∈ $uint64\text{-}nat\text{-}assn^k \rightarrow_a$ *nat-assn*›
⟨*proof*⟩

sepref-definition *get-var-name-c-impl*
is ‹*uncurry get-var-name-c*›
:: ‹$perfect\text{-}shared\text{-}vars\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow_a$ *string-assn*›
⟨*proof*⟩

lemma [*sepref-fr-rules*]:
‹(*uncurry is-new-variable-c-impl*, *uncurry is-new-variableS*) ∈ $string\text{-}assn^k *_a shared\text{-}vars\text{-}assn^k \rightarrow_a$ *bool-assn*›
⟨*proof*⟩

lemma [*sepref-fr-rules*]:
‹(*uncurry get-var-pos-c-impl*, *uncurry get-var-posS*) ∈ $shared\text{-}vars\text{-}assn^k *_a string\text{-}assn^k \rightarrow_a$ *uint64-nat-assn*›
⟨*proof*⟩

lemma [*sepref-fr-rules*]:
‹(*uncurry get-var-name-c-impl*, *uncurry get-var-nameS*) ∈ $shared\text{-}vars\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow_a$ *string-assn*›
⟨*proof*⟩

sepref-register *get-var-nameS get-var-posS is-new-variableS*

abbreviation *memory-allocation-rel* :: ‹(*memory-allocation* × *memory-allocation*) *set*› **where**
‹*memory-allocation-rel* ≡ *Id*›

abbreviation *memory-allocation-assn* :: ‹*memory-allocation* ⇒ *memory-allocation* ⇒ *assn*› **where**
‹*memory-allocation-assn* ≡ *id-assn*›

instantiation *memory-allocation* :: *default*
begin
definition *default-memory-allocation* :: ‹*memory-allocation*› **where**
‹*default-memory-allocation* = *Allocated*›
instance
⟨*proof*⟩
end

term *import-polyS*
lemma [*sepref-import-param*]:
‹(*Allocated*, *Allocated*) ∈ *memory-allocation-rel*›
‹(*Mem-Out*, *Mem-Out*) ∈ *memory-allocation-rel*›

‹(alloc-failed, alloc-failed) ∈ memory-allocation-rel → bool-rel›
⟨proof⟩

lemma *pow-2-63-1*: ‹2 ^ 63 − 1 = (9223372036854775807 :: nat)›
⟨proof⟩
definition *zero-uint64-nat* **where**
‹zero-uint64-nat = 0›
sepref-register *zero-uint64-nat*
lemma [*sepref-fr-rules*]:
‹(uncurry0 (return 0), uncurry0 (RETURN zero-uint64-nat))∈unit-assn^k $\to_a$ uint64-nat-assn›
⟨proof⟩

definition *length-uint64-nat* **where**
[*simp*]: ‹length-uint64-nat = length›

definition *length-arl-u-code* :: ‹('a::heap) array-list ⇒ uint64 Heap› **where**
‹length-arl-u-code xs = do {
n ← arl-length xs;
return (uint64-of-nat n)}›

definition *uint64-max* :: *nat* **where**
‹uint64-max = 2 ^64 − 1›

lemma *nat-of-uint64-uint64-of-nat*: ‹b ≤ uint64-max ⟹ nat-of-uint64 (uint64-of-nat b) = b›
⟨proof⟩

lemma *length-arl-u-hnr*[*sepref-fr-rules*]:
‹(length-arl-u-code, RETURN o length-uint64-nat) ∈
[λxs. length xs ≤ uint64-max]$_a$ (arl-assn R)^k → uint64-nat-assn›
⟨proof⟩

lemma *find-new-idx-c-alt-def*:
‹find-new-idx-c = (λ(𝒱, 𝒜). let k = length 𝒱 in if k < 2^63−1 then RETURN (Allocated, length-uint64-nat 𝒱) else RETURN (Mem-Out, 0))›
⟨proof⟩

sepref-definition *find-new-idx-c-impl*
is ‹find-new-idx-c›
:: ‹perfect-shared-vars-assn^k $\to_a$ id-assn $\times_a$ uint64-nat-assn›
⟨proof⟩

instantiation *String.literal* :: *default*
begin
definition *default-literal* :: ‹String.literal› **where**
‹default-literal = String.implode ''''›
instance
⟨proof⟩
end

sepref-definition *insert-variable-c-impl*
is ‹uncurry2 (RETURN ooo insert-variable-c)›
:: ‹string-assn^k $*_a$ uint64-nat-assn^k $*_a$ perfect-shared-vars-assn^d $\to_a$ perfect-shared-vars-assn›
⟨proof⟩

lemmas [*sepref-fr-rules*] =
find-new-idx-c-impl.refine insert-variable-c-impl.refine

sepref-definition *import-variable-c-impl*
is ‹*uncurry import-variable-c*›
:: ‹$string\text{-}assn^k *_a perfect\text{-}shared\text{-}vars\text{-}assn^d \rightarrow_a id\text{-}assn \times_a perfect\text{-}shared\text{-}vars\text{-}assn \times_a uint64\text{-}nat\text{-}assn$›
⟨*proof*⟩

lemma *import-variable-c-import-variableS'*:
assumes ‹*single-valued R*› ‹*single-valued* (R^{-1})›
shows ‹(*uncurry import-variable-c*, *uncurry import-variableS*) ∈ $R \times_r$ *perfect-shared-vars-rel-c* $R \rightarrow_f$
⟨*memory-allocation-rel* $\times_r$ *perfect-shared-vars-rel-c* $R \times_r$ *nat-rel*⟩*nres-rel*›
⟨*proof*⟩

lemma [*sepref-fr-rules*]:
‹(*uncurry import-variable-c-impl*, *uncurry import-variableS*)
∈ $string\text{-}assn^k *_a shared\text{-}vars\text{-}assn^d \rightarrow_a memory\text{-}allocation\text{-}assn \times_a shared\text{-}vars\text{-}assn \times_a uint64\text{-}nat\text{-}assn$›
⟨*proof*⟩

definition *empty-shared-vars* :: ‹(*nat*, *string*) *shared-vars*› **where**
‹*empty-shared-vars* = ({#}, *fmempty*, *fmempty*)›

definition *empty-shared-vars-int* :: ‹(*string*, *nat*) *shared-vars-c*› **where**
‹*empty-shared-vars-int* = ([], *fmempty*)›

sepref-definition *empty-shared-vars-int-impl*
is ‹*uncurry0* (*RETURN empty-shared-vars-int*)›
:: ‹$unit\text{-}assn^k \rightarrow_a perfect\text{-}shared\text{-}vars\text{-}assn$›
⟨*proof*⟩

lemma *empty-shared-vars-int-empty-shared-vars*:
‹(*uncurry0* (*RETURN empty-shared-vars-int*), *uncurry0* (*RETURN empty-shared-vars*)) ∈ *unit-rel* $\rightarrow_f$
⟨*perfect-shared-vars-rel-c R*⟩*nres-rel*›
⟨*proof*⟩

lemma [*sepref-fr-rules*]:
‹(*uncurry0 empty-shared-vars-int-impl*, *uncurry0* (*RETURN empty-shared-vars*))
∈ $unit\text{-}assn^k \rightarrow_a shared\text{-}vars\text{-}assn$›
⟨*proof*⟩
sepref-register *empty-shared-vars*
end
theory *EPAC-Efficient-Checker-Synthesis*
imports *EPAC-Efficient-Checker*
EPAC-Perfectly-Shared-Vars
PAC-Checker.PAC-Checker-Synthesis
EPAC-Steps-Refine
PAC-Checker.PAC-Checker-Synthesis
begin

lemma *in-set-rel-inD*: ‹(*x*,*y*) ∈⟨*R*⟩*list-rel* ⟹ *a* ∈ *set x* ⟹ ∃ *b* ∈ *set y*. (*a*,*b*)∈ *R*›
⟨*proof*⟩

lemma *perfectly-shared-monom-eqD*: ‹(*a*, *ab*) ∈ *perfectly-shared-monom* $\mathcal{V}$ ⟹ *ab* = *map* ((*the* ∘∘ *fm-*

lookup) (*fst* (*snd* $\mathcal{V}$))) *a*›
⟨*proof*⟩

lemma *perfectly-shared-monom-unique-left*:
‹$(x, y) \in$ *perfectly-shared-monom* $\mathcal{V} \Longrightarrow (x, y') \in$ *perfectly-shared-monom* $\mathcal{V} \Longrightarrow y = y'$›
⟨*proof*⟩

lemma *perfectly-shared-monom-unique-right*:
‹$(\mathcal{V}, \mathcal{DV}) \in$ *perfectly-shared-vars-rel* $\Longrightarrow$
$(x, y) \in$ *perfectly-shared-monom* $\mathcal{V} \Longrightarrow (x', y) \in$ *perfectly-shared-monom* $\mathcal{V} \Longrightarrow x = x'$›
⟨*proof*⟩

lemma *perfectly-shared-polynom-unique-left*:
‹$(x, y) \in$ *perfectly-shared-polynom* $\mathcal{V} \Longrightarrow (x, y') \in$ *perfectly-shared-polynom* $\mathcal{V} \Longrightarrow y = y'$›
⟨*proof*⟩
lemma *perfectly-shared-polynom-unique-right*:
‹$(\mathcal{V}, \mathcal{DV}) \in$ *perfectly-shared-vars-rel* $\Longrightarrow$
$(x, y) \in$ *perfectly-shared-polynom* $\mathcal{V} \Longrightarrow (x', y) \in$ *perfectly-shared-polynom* $\mathcal{V} \Longrightarrow x = x'$›
⟨*proof*⟩

definition (**in** −)*perfect-shared-var-order-s* :: ‹(*nat*, *string*)*shared-vars* ⇒ *nat* ⇒ *nat* ⇒ *ordered nres*›
where
‹*perfect-shared-var-order-s* $\mathcal{D}$ *x y* = *do* {
  eq ← *perfectly-shared-strings-equal-l* $\mathcal{D}$ *x y*;
  if eq then RETURN EQUAL
  else do {
    x ← *get-var-nameS* $\mathcal{D}$ *x*;
    y ← *get-var-nameS* $\mathcal{D}$ *y*;
    if (*x*, *y*) ∈ *var-order-rel then RETURN* (*LESS*)
    else RETURN (*GREATER*)
      }}›

lemma *perfect-shared-var-order-s-perfect-shared-var-order*:
  assumes ‹$(\mathcal{V}, \mathcal{VD}) \in$ *perfectly-shared-vars-rel*› **and**
  ‹$(i, i') \in$ *perfectly-shared-var-rel* $\mathcal{V}$›**and**
  ‹$(j, j') \in$ *perfectly-shared-var-rel* $\mathcal{V}$›
  shows ‹*perfect-shared-var-order-s* $\mathcal{V}$ *i j* ≤⇓*Id* (*perfect-shared-var-order* $\mathcal{VD}$ i' j')›
⟨*proof*⟩

definition (**in** −) *perfect-shared-term-order-rel-s*
  :: ‹(*nat*, *string*) *shared-vars* ⇒ *nat list*⇒ *nat list* ⇒ *ordered nres*›
where
‹*perfect-shared-term-order-rel-s* $\mathcal{V}$ *xs ys* = *do* {
  (*b*, -, -) ← $WHILE_T$ (λ(*b*, *xs*, *ys*). *b* = *UNKNOWN*)
  (λ(*b*, *xs*, *ys*). *do* {
    if xs = [] ∧ *ys* = [] *then RETURN* (*EQUAL*, *xs*, *ys*)
    else if xs = [] *then RETURN* (*LESS*, *xs*, *ys*)
    else if ys = [] *then RETURN* (*GREATER*, *xs*, *ys*)
    else do {
      ASSERT(*xs* ≠ [] ∧ *ys* ≠ []);
      eq ← *perfect-shared-var-order-s* $\mathcal{V}$ (*hd xs*) (*hd ys*);
      if eq = *EQUAL then RETURN* (*b*, *tl xs*, *tl ys*)
      else RETURN (*eq*, *xs*, *ys*)
    }
  }) (*UNKNOWN*, *xs*, *ys*);

RETURN b
}›

lemma *perfect-shared-term-order-rel-s-perfect-shared-term-order-rel*:
assumes ‹(𝒱, 𝒱𝒟) ∈ *perfectly-shared-vars-rel*› **and**
‹(*xs*, *xs'*) ∈ *perfectly-shared-monom* 𝒱› **and**
‹(*ys*, *ys'*) ∈ *perfectly-shared-monom* 𝒱›
shows ‹*perfect-shared-term-order-rel-s* 𝒱 *xs ys* ≤ ⇓*Id* (*perfect-shared-term-order-rel* 𝒱𝒟 *xs' ys'*)›
⟨*proof*⟩

fun *mergeR* :: - ⇒ - ⇒ *'a list* ⇒ *'a list* ⇒ *'a list nres*
where
mergeR Φ *f* (*x*#*xs*) (*y*#*ys*) = *do* {
    ASSERT(Φ *x y*);
    b ← *f x y*;
    if b then do {*zs* ← *mergeR* Φ *f xs* (*y*#*ys*); *RETURN* (*x* # *zs*)}
    else do {*zs* ← *mergeR* Φ *f* (*x*#*xs*) *ys*; *RETURN* (*y* # *zs*)}
  }
| *mergeR* Φ *f xs* [] = *RETURN xs*
| *mergeR* Φ *f* [] *ys* = *RETURN ys*

lemma *mergeR-merge*:
assumes ‹⋀*x y*. *x*∈*set xs* ∪ *set ys* ⟹ *y*∈*set xs* ∪ *set ys* ⟹Φ *x y*› **and**
‹⋀*x y*. *x*∈*set xs* ∪ *set ys* ⟹ *y*∈*set xs* ∪ *set ys* ⟹ *f x y* ≤ ⇓*Id* (*RETURN* (*f' x y*))› **and**
‹(*xs*,*xs'*)∈*Id*›**and**
‹(*ys*,*ys'*)∈*Id*›
shows
‹*mergeR* Φ *f xs ys* ≤ ⇓*Id* (*RETURN* (*merge f' xs' ys'*))›
⟨*proof*⟩

lemma *merge-alt*:
RETURN (*merge f xs ys*) = *SPEC*(λ*zs*. *zs* = *merge f xs ys* ∧ *set zs* = *set xs* ∪ *set ys*)
⟨*proof*⟩

fun *msortR* :: - ⇒ - ⇒ *'a list* ⇒ *'a list nres*
where
msortR Φ *f* [] = *RETURN* []
| *msortR* Φ *f* [*x*] = *RETURN* [*x*]
| *msortR* Φ *f xs* = *do* {
  as ← *msortR* Φ *f* (*take* (*size xs div 2*) *xs*);
  bs ← *msortR* Φ *f* (*drop* (*size xs div 2*) *xs*);
  mergeR Φ *f as bs*
}

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

lemma *msortR-msort*:
assumes ‹⋀*x y*. *x*∈*set xs* ⟹ *y*∈*set xs* ⟹Φ *x y*› **and**
‹⋀*x y*. *x*∈*set xs* ⟹ *y*∈*set xs* ⟹ *f x y* ≤ ⇓*Id* (*RETURN* (*f' x y*))›
shows
‹*msortR* Φ *f xs* ≤ ⇓*Id* (*RETURN* (*msort f' xs*))›
⟨*proof*⟩

lemma *merge-list-rel*:

assumes ‹⋀*x y x′ y′*. *x*∈*set xs* ⟹ *y*∈*set ys* ⟹ *x′*∈*set xs′* ⟹ *y′*∈*set ys′* ⟹ (*x*,*x′*)∈*R* ⟹ (*y*,*y′*)∈*R* ⟹ *f x y* = *f′ x′ y′*› **and**
‹(*xs*,*xs′*) ∈ ⟨*R*⟩*list-rel*› **and**
‹(*ys*,*ys′*) ∈ ⟨*R*⟩*list-rel*›
shows ‹(*merge f xs ys*, *merge f′ xs′ ys′*) ∈ ⟨*R*⟩*list-rel*›
⟨*proof*⟩

lemma *msort-list-rel*:
assumes ‹⋀*x y x′ y′*. *x*∈*set xs* ⟹ *y*∈*set xs* ⟹ *x′*∈*set xs′* ⟹ *y′*∈*set xs′* ⟹ (*x*,*x′*)∈*R* ⟹ (*y*,*y′*)∈*R* ⟹ *f x y* = *f′ x′ y′*› **and**
‹(*xs*,*xs′*) ∈ ⟨*R*⟩*list-rel*›
shows ‹(*msort f xs*, *msort f′ xs′*) ∈ ⟨*R*⟩*list-rel*›
⟨*proof*⟩

lemma *msortR-alt-def*:
‹(*msortR* Φ *f xs*) = REC_T(λ*msortR′ xs*.
if length xs ≤ *1 then RETURN xs else do* {
let xs1 = (*take* ((*size xs*) *div 2*) *xs*);
let xs2 = (*drop* ((*size xs*) *div 2*) *xs*);
as ← *msortR′ xs1*;
bs ← *msortR′ xs2*;
(*mergeR* Φ *f as bs*)
}) *xs*
›
⟨*proof*⟩

definition *sort-poly-spec-s* **where**
‹*sort-poly-spec-s* 𝒱 *xs* = *msortR* (λ*xs ys*. (∀ *a*∈*set* (*fst xs*). *a* ∈# *dom-m* (*fst* (*snd* 𝒱))) ∧ (∀ *a*∈*set*(*fst ys*). *a* ∈# *dom-m* (*fst* (*snd* 𝒱))))
(λ*xs ys*. *do* {*a* ← *perfect-shared-term-order-rel-s* 𝒱 (*fst xs*) (*fst ys*); *RETURN* (*a* ≠ *GREATER*)}) *xs*›

lemma *sort-poly-spec-s-sort-poly-spec*:
assumes ‹(𝒱, 𝒱𝒟) ∈ *perfectly-shared-vars-rel*› **and**
‹(*xs*, *xs′*) ∈ *perfectly-shared-polynom* 𝒱› **and**
‹*vars-llist xs′* ⊆ *set-mset* 𝒱𝒟›
shows
‹*sort-poly-spec-s* 𝒱 *xs*
≤⇓(*perfectly-shared-polynom* 𝒱)
(*sort-poly-spec xs′*)
›
⟨*proof*⟩

definition *msort-coeff-s* :: ‹(*nat*,*string*)*shared-vars* ⇒ *nat list* ⇒ *nat list nres*› **where**
‹*msort-coeff-s* 𝒱 *xs* = *msortR* (λ*a b*. *a* ∈ *set xs* ∧ *b* ∈ *set xs*)
(λ*a b*. *do* {
x ← *get-var-nameS* 𝒱 *a*;
y ← *get-var-nameS* 𝒱 *b*;
RETURN(*a* = *b* ∨ *var-order x y*)
}) *xs*›

lemma *perfectly-shared-var-rel-unique-left*:
‹(*x*, *y*) ∈ *perfectly-shared-var-rel* 𝒱 ⟹ (*x*, *y′*) ∈ *perfectly-shared-var-rel* 𝒱 ⟹ *y* = *y′*›

⟨*proof*⟩

lemma *perfectly-shared-var-rel-unique-right*:
‹($\mathcal{V}$, $\mathcal{DV}$) ∈ *perfectly-shared-vars-rel* ⟹ (*x*, *y*) ∈ *perfectly-shared-var-rel* $\mathcal{V}$ ⟹ (*x'*, *y*) ∈ *perfectly-shared-var-rel* $\mathcal{V}$ ⟹ *x* = *x'*›
⟨*proof*⟩

lemma *msort-coeff-s-sort-coeff*:
fixes *xs'* :: ‹*string list*› **and**
$\mathcal{V}$ :: ‹(*nat*,*string*)*shared-vars*›
assumes
‹(*xs*, *xs'*) ∈ *perfectly-shared-monom* $\mathcal{V}$› **and**
‹($\mathcal{V}$, $\mathcal{DV}$) ∈ *perfectly-shared-vars-rel*› **and**
‹*set xs'* ⊆ *set-mset* $\mathcal{DV}$›
shows ‹*msort-coeff-s* $\mathcal{V}$ *xs* ≤ ⇓(*perfectly-shared-monom* $\mathcal{V}$) (*sort-coeff xs'*)›
⟨*proof*⟩

type-synonym *sllist-polynomial* = ‹(*nat list* × *int*) *list*›

definition *sort-all-coeffs-s* :: ‹(*nat*,*string*)*shared-vars* ⇒ *sllist-polynomial* ⇒ *sllist-polynomial nres*› **where**
‹*sort-all-coeffs-s* $\mathcal{V}$ *xs* = *monadic-nfoldli xs* (λ-. *RETURN True*) (λ(*a*, *n*) *b*. *do* {*ASSERT*((*a*,*n*)∈*set xs*);*a* ← *msort-coeff-s* $\mathcal{V}$ *a*; *RETURN* ((*a*, *n*) # *b*)}) []›

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

lemma *merge-coeffs0-s-merge-coeffs0*:
fixes *xs* :: ‹*sllist-polynomial*› **and**
$\mathcal{V}$ :: ‹(*nat*,*string*)*shared-vars*›
assumes
‹(*xs*, *xs'*) ∈ *perfectly-shared-polynom* $\mathcal{V}$› **and**
$\mathcal{V}$: ‹($\mathcal{V}$, $\mathcal{DV}$) ∈ *perfectly-shared-vars-rel*›
shows ‹(*merge-coeffs0-s xs*, *merge-coeffs0 xs'*) ∈ *perfectly-shared-polynom* $\mathcal{V}$›
⟨*proof*⟩

lemma *list-rel-mono-strong*: ‹*A* ∈ ⟨*R*⟩*list-rel* ⟹ (⋀*xs*. *fst xs* ∈ *set* (*fst A*) ⟹ *snd xs* ∈ *set* (*snd A*) ⟹ *xs* ∈ *R* ⟹ *xs* ∈ *R'*) ⟹ *A* ∈ ⟨*R'*⟩*list-rel*›
⟨*proof*⟩

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

lemma *sort-all-coeffs-s-sort-all-coeffs*:
fixes *xs* :: ‹*sllist-polynomial*› **and**
$\mathcal{V}$ :: ‹(*nat*,*string*)*shared-vars*›

assumes
‹(*xs*, *xs*′) ∈ *perfectly-shared-polynom* 𝒱› **and**
𝒱: ‹(𝒱, 𝒟𝒱) ∈ *perfectly-shared-vars-rel*› **and**
‹*vars-llist xs*′ ⊆ *set-mset* 𝒟𝒱›
shows ‹*sort-all-coeffs-s* 𝒱 *xs* ≤ ⇓(*perfectly-shared-polynom* 𝒱) (*sort-all-coeffs xs*′)›
⟨*proof*⟩

definition *vars-llist-in-s* :: ‹(*nat*, *string*) *shared-vars* ⇒ *llist-polynomial* ⇒ *bool*› **where**
‹*vars-llist-in-s* = (λ(𝒱,𝒟,𝒟′) *p*. *vars-llist p* ⊆ *set-mset* (*dom-m* 𝒟′))›

lemma *vars-llist-in-s-vars-llist*[*simp*]:
assumes ‹(𝒱, 𝒟𝒱) ∈ *perfectly-shared-vars-rel*›
shows ‹*vars-llist-in-s* 𝒱 *p* ⟷ *vars-llist p* ⊆ *set-mset* 𝒟𝒱›
⟨*proof*⟩

definition (**in** −)*add-poly-l-s* :: ‹(*nat*,*string*)*shared-vars* ⇒ *sllist-polynomial* × *sllist-polynomial* ⇒ *sllist-polynomial nres*› **where**
‹*add-poly-l-s* 𝒟 = REC_T
(λ*add-poly-l* (*p*, *q*).
case (*p*,*q*) *of*
(*p*, []) ⇒ *RETURN p*
| ([], *q*) ⇒ *RETURN q*
| ((*xs*, *n*) # *p*, (*ys*, *m*) # *q*) ⇒ *do* {
comp ← *perfect-shared-term-order-rel-s* 𝒟 *xs ys*;
if comp = *EQUAL 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 comp = *LESS*
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*)
}
})›

lemma *add-poly-l-s-add-poly-l*:
fixes *xs* :: ‹*sllist-polynomial* × *sllist-polynomial*›
assumes ‹(𝒱, 𝒱𝒟) ∈ *perfectly-shared-vars-rel*› **and**
‹(*xs*, *xs*′) ∈ *perfectly-shared-polynom* 𝒱 $\times_r$ *perfectly-shared-polynom* 𝒱›
shows ‹*add-poly-l-s* 𝒱 *xs* ≤ ⇓(*perfectly-shared-polynom* 𝒱) (*add-poly-l-prep* 𝒱𝒟 *xs*′)›
⟨*proof*⟩

definition (**in** −) *mult-monoms-s* :: ‹(*nat*,*string*)*shared-vars* ⇒ *nat list* ⇒ *nat list* ⇒ *nat list nres*›
where
‹*mult-monoms-s* 𝒟 *xs ys* = REC_T (λ*f* (*xs*, *ys*).
do {
if xs = [] *then RETURN ys*
else if ys = [] *then RETURN xs*

```
    else do {
      ASSERT(xs ≠ [] ∧ ys ≠ []);
      comp ← perfect-shared-var-order-s 𝒟 (hd xs) (hd ys);
      if comp = EQUAL then do {
        pq ← f (tl xs, tl ys);
        RETURN (hd xs # pq)
      }
      else if comp = LESS then do {
        pq ← f (tl xs, ys);
        RETURN (hd xs # pq)
      }
      else do {
        pq ← f (xs, tl ys);
        RETURN (hd ys # pq)
      }
    }
  }) (xs, ys)›
```

lemma *mult-monoms-s-simps*:

```
  ‹mult-monoms-s 𝒱 xs ys =
  do {
    if xs = [] then RETURN ys
    else if ys = [] then RETURN xs
    else do {
      ASSERT(xs ≠ [] ∧ ys ≠ []);
      comp ← perfect-shared-var-order-s 𝒱 (hd xs) (hd ys);
      if comp = EQUAL then do {
        pq ← mult-monoms-s 𝒱 (tl xs) (tl ys);
        RETURN (hd xs # pq)
      }
      else if comp = LESS then do {
        pq ← mult-monoms-s 𝒱 (tl xs) ys;
        RETURN (hd xs # pq)
      }
      else do {
        pq ← mult-monoms-s 𝒱 xs (tl ys);
        RETURN (hd ys # pq)
      }
    }
  }›
  ⟨proof⟩
```

lemma *mult-monoms-s-mult-monoms-prep*:
  fixes *xs*
  assumes ‹(𝒱, 𝒱𝒟) ∈ *perfectly-shared-vars-rel*› **and**
    ‹(*xs*, *xs′*) ∈ *perfectly-shared-monom* 𝒱›
    ‹(*ys*, *ys′*) ∈ *perfectly-shared-monom* 𝒱›
  shows ‹*mult-monoms-s* 𝒱 *xs ys* ≤ ⇓(*perfectly-shared-monom* 𝒱) ((*mult-monoms-prep* 𝒱𝒟 *xs′ ys′*))›
⟨*proof*⟩

definition (**in** −) *mult-term-s*
  :: ‹(*nat*,*string*)*shared-vars*⇒ *sllist-polynomial* ⇒ - ⇒ *sllist-polynomial* ⇒ *sllist-polynomial nres*›
where
  ‹*mult-term-s* = (λ𝒱 *qs* (*p*, *m*) *b*. *nfoldli qs* (λ-. *True*) (λ(*q*, *n*) *b*. *do* {*pq* ← *mult-monoms-s* 𝒱 *p q*;

RETURN ((*pq*, *m* ∗ *n*) # *b*)}) *b*)›

definition *mult-poly-s* :: ‹(*nat*,*string*) *shared-vars*⇒ *sllist-polynomial* ⇒ *sllist-polynomial* ⇒ *sllist-polynomial nres*› **where**
‹*mult-poly-s* 𝒱 *p* *q* = *nfoldli* *p* (λ-. *True*) (*mult-term-s* 𝒱 *q*) []›

lemma *mult-term-s-mult-monoms-prop*:
fixes *xs*
assumes ‹(𝒱, 𝒱𝒟) ∈ *perfectly-shared-vars-rel*› **and**
‹(*xs*, *xs'*) ∈ *perfectly-shared-polynom* 𝒱›
‹(*ys*, *ys'*) ∈ *perfectly-shared-monom* 𝒱 $\times_r$ *int-rel*›
‹(*zs*, *zs'*) ∈ *perfectly-shared-polynom* 𝒱›
shows ‹*mult-term-s* 𝒱 *xs* *ys* *zs* ≤ ⇓(*perfectly-shared-polynom* 𝒱) (*mult-monoms-prop* 𝒱𝒟 *xs'* *ys'* *zs'*)›
⟨*proof*⟩

lemma *mult-poly-s-mult-poly-raw-prop*:
fixes *xs*
assumes ‹(𝒱, 𝒱𝒟) ∈ *perfectly-shared-vars-rel*› **and**
‹(*xs*, *xs'*) ∈ *perfectly-shared-polynom* 𝒱›
‹(*ys*, *ys'*) ∈ *perfectly-shared-polynom* 𝒱›
shows ‹*mult-poly-s* 𝒱 *xs* *ys* ≤ ⇓(*perfectly-shared-polynom* 𝒱) (*mult-poly-raw-prop* 𝒱𝒟 *xs'* *ys'*)›
⟨*proof*⟩

lemma *op-eq-uint64-nat*[*sepref-fr-rules*]:
‹(*uncurry* (*return* *oo* ((=) :: *uint64* ⇒ -)), *uncurry* (*RETURN* *oo* (=))) ∈
uint64-nat-assn^k $*_a$ *uint64-nat-assn*^{k} $\rightarrow_a$ *bool-assn*›
⟨*proof*⟩

abbreviation *ordered-assn* :: ‹*ordered* ⇒ - ⇒ -› **where**
‹*ordered-assn* ≡ *id-assn*›

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

abbreviation *monom-s-rel* **where**
‹*monom-s-rel* ≡ ⟨*uint64-nat-rel*⟩*list-rel*›

abbreviation *monom-s-assn* **where**
‹*monom-s-assn* ≡ *list-assn* *uint64-nat-assn*›

abbreviation *poly-s-assn* **where**
‹*poly-s-assn* ≡ *list-assn* (*monom-s-assn* $\times_a$ *int-assn*)›

sepref-decl-intf *wordered* **is** *ordered*

sepref-register *EQUAL LESS GREATER UNKNOWN get-var-nameS perfect-shared-var-order-s perfect-shared-term-ord*
lemma [*sepref-fr-rules*]:
‹(*uncurry0* (*return* *EQUAL*), *uncurry0* (*RETURN* *EQUAL*)) ∈ *unit-assn*^{k} $\rightarrow_a$ *id-assn*›
‹(*uncurry0* (*return* *LESS*), *uncurry0* (*RETURN* *LESS*)) ∈ *unit-assn*^{k} $\rightarrow_a$ *id-assn*›
‹(*uncurry0* (*return* *GREATER*), *uncurry0* (*RETURN* *GREATER*)) ∈ *unit-assn*^{k} $\rightarrow_a$ *id-assn*›
‹(*uncurry0* (*return* *UNKNOWN*), *uncurry0* (*RETURN* *UNKNOWN*)) ∈ *unit-assn*^{k} $\rightarrow_a$ *id-assn*›

⟨*proof*⟩

sepref-definition *perfect-shared-var-order-s-impl*
is ‹*uncurry2 perfect-shared-var-order-s*›
:: ‹*shared-vars-assn*^{k} $*_a$ *uint64-nat-assn*^{k} $*_a$ *uint64-nat-assn*^{k} $\rightarrow_a$ *id-assn*›
⟨*proof*⟩

lemmas [*sepref-fr-rules*] = *perfect-shared-var-order-s-impl.refine*

sepref-definition *perfect-shared-term-order-rel-s-impl*
is ‹*uncurry2 perfect-shared-term-order-rel-s*›
:: ‹*shared-vars-assn*^{k} $*_a$ *monom-s-assn*^{k} $*_a$ *monom-s-assn*^{k} $\rightarrow_a$ *id-assn*›
⟨*proof*⟩

lemmas [*sepref-fr-rules*] = *perfect-shared-term-order-rel-s-impl.refine*

sepref-definition *add-poly-l-prep-impl*
is ‹*uncurry add-poly-l-s*›
:: ‹*shared-vars-assn*^{k} $*_a$ (*poly-s-assn* $\times_a$ *poly-s-assn*)^k $\rightarrow_a$ *poly-s-assn*›
⟨*proof*⟩

lemma [*sepref-fr-rules*]:
‹(*return o is-Nil*, *RETURN o is-Nil*)∈ (*list-assn R*)^k $\rightarrow_a$ *bool-assn*›
⟨*proof*⟩

sepref-definition *mult-monoms-s-impl*
is ‹*uncurry2 mult-monoms-s*›
:: ‹*shared-vars-assn*^{k} $*_a$ *monom-s-assn*^{k} $*_a$ *monom-s-assn*^{k} $\rightarrow_a$ *monom-s-assn*›
⟨*proof*⟩

lemmas [*sepref-fr-rules*] =
mult-monoms-s-impl.refine

sepref-definition *mult-term-s-impl*
is ‹*uncurry3 mult-term-s*›
:: ‹*shared-vars-assn*^{k} $*_a$ *poly-s-assn*^{k} $*_a$ (*monom-s-assn* $\times_a$ *int-assn*)^k $*_a$ *poly-s-assn*^{k}$\rightarrow_a$ *poly-s-assn*›
⟨*proof*⟩

lemmas [*sepref-fr-rules*] =
mult-term-s-impl.refine

sepref-definition *mult-poly-s-impl*
is ‹*uncurry2 mult-poly-s*›
:: ‹*shared-vars-assn*^{k} $*_a$ *poly-s-assn*^{k} $*_a$ *poly-s-assn*^{k}$\rightarrow_a$ *poly-s-assn*›
⟨*proof*⟩

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

sepref-register *take drop*
lemma [*sepref-fr-rules*]:
assumes ‹*CONSTRAINT is-pure R*›
shows ‹(*uncurry* (*return oo take*), *uncurry* (*RETURN oo take*)) ∈ *nat-assn*^{k} $*_a$ (*list-assn R*)^k $\rightarrow_a$ *list-assn R*›

⟨proof⟩

lemma [*sepref-fr-rules*]:
assumes ‹*CONSTRAINT is-pure R*›
shows ‹(*uncurry* (*return oo drop*), *uncurry* (*RETURN oo drop*)) ∈ *nat-assn*^{k} $*_a$ (*list-assn R*)^k $\rightarrow_a$ *list-assn R*›
⟨proof⟩

definition *mergeR-vars* :: ‹(*nat*, *string*) *shared-vars* ⇒ *sllist-polynomial* ⇒ *sllist-polynomial* ⇒ *sllist-polynomial nres*› **where**
‹*mergeR-vars* $\mathcal{V}$ = *mergeR*
(λ*xs ys*. (∀ *a*∈*set* (*fst xs*). *a* ∈# *dom-m* (*fst* (*snd* $\mathcal{V}$))) ∧ (∀ *a*∈*set*(*fst ys*). *a* ∈# *dom-m* (*fst* (*snd* $\mathcal{V}$))))
(λ*xs ys*. *do* {*a* ← *perfect-shared-term-order-rel-s* $\mathcal{V}$ (*fst xs*) (*fst ys*); *RETURN* (*a* ≠ *GREATER*)})›
lemma *mergeR-alt-def*:
‹(*mergeR* Φ *f xs ys*) = REC_T(λ*mergeR xs*.
case xs of
([], *ys*) ⇒ *RETURN ys*
| (*xs*, []) ⇒ *RETURN xs*
| (*x* # *xs*, *y* # *ys*) ⇒ *do* {
ASSERT(Φ *x y*);
b ← *f x y*;
if b then do {
zs ← *mergeR* (*xs*, *y* # *ys*);
RETURN (*x* # *zs*)
}
else do {
zs ← *mergeR* (*x* # *xs*, *ys*);
RETURN (*y* # *zs*)
}
})
(*xs*, *ys*)›
⟨proof⟩

sepref-definition *mergeR-vars-impl*
is ‹*uncurry2 mergeR-vars*›
:: ‹*shared-vars-assn*^{k} $*_a$ *poly-s-assn*^{k} $*_a$ *poly-s-assn*^{k} $\rightarrow_a$ *poly-s-assn*›
⟨proof⟩

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

abbreviation *msortR-vars* **where**
‹*msortR-vars* ≡ *sort-poly-spec-s*›
lemmas *msortR-vars-def* = *sort-poly-spec-s-def*

sepref-register *mergeR-vars msortR-vars*

sepref-definition *msortR-vars-impl*
is ‹*uncurry msortR-vars*›
:: ‹*shared-vars-assn*^{k} $*_a$ *poly-s-assn*^{k} $\rightarrow_a$ *poly-s-assn*›
⟨proof⟩

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

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

lemma *perfectly-shared-merge-coeffs-merge-coeffs*:
assumes
‹($\mathcal{V}$, $\mathcal{DV}$) ∈ *perfectly-shared-vars-rel*›
‹(*xs*, *xs'*) ∈ *perfectly-shared-polynom* $\mathcal{V}$›
shows ‹(*merge-coeffs-s xs*, *merge-coeffs xs'*) ∈ (*perfectly-shared-polynom* $\mathcal{V}$)›
⟨*proof*⟩

definition *normalize-poly-s* :: ‹-› **where**
‹*normalize-poly-s* $\mathcal{V}$ *p* = *do* {
p ← *msortR-vars* $\mathcal{V}$ *p*;
RETURN (*merge-coeffs-s p*)
}›

lemma *normalize-poly-s-normalize-poly-s*:
assumes
‹($\mathcal{V}$, $\mathcal{DV}$) ∈ *perfectly-shared-vars-rel*›
‹(*xs*, *xs'*) ∈ *perfectly-shared-polynom* $\mathcal{V}$› **and**
‹*vars-llist xs'* ⊆ *set-mset* $\mathcal{DV}$›
shows ‹*normalize-poly-s* $\mathcal{V}$ *xs* ≤ ⇓ (*perfectly-shared-polynom* $\mathcal{V}$) (*normalize-poly xs'*)›
⟨*proof*⟩

definition *check-linear-combi-l-s-dom-err* :: ‹*sllist-polynomial* ⇒ *nat* ⇒ *string nres*› **where**
‹*check-linear-combi-l-s-dom-err p r* = *SPEC* (λ-. *True*)›

definition *mult-poly-full-s* :: ‹-› **where**
‹*mult-poly-full-s* $\mathcal{V}$ *p q* = *do* {
pq ← *mult-poly-s* $\mathcal{V}$ *p q*;
normalize-poly-s $\mathcal{V}$ *pq*
}›

lemma *mult-poly-full-s-mult-poly-full-prop*:
assumes
‹($\mathcal{V}$, $\mathcal{DV}$) ∈ *perfectly-shared-vars-rel*›
‹(*xs*, *xs'*) ∈ *perfectly-shared-polynom* $\mathcal{V}$› **and**
‹(*ys*, *ys'*) ∈ *perfectly-shared-polynom* $\mathcal{V}$› **and**
‹*vars-llist xs'* ⊆ *set-mset* $\mathcal{DV}$› **and**
‹*vars-llist ys'* ⊆ *set-mset* $\mathcal{DV}$›
shows ‹*mult-poly-full-s* $\mathcal{V}$ *xs ys* ≤ ⇓ (*perfectly-shared-polynom* $\mathcal{V}$) (*mult-poly-full-prop* $\mathcal{DV}$ *xs' ys'*)›
⟨*proof*⟩
definition (**in** −)*linear-combi-l-prep-s*
:: ‹*nat* ⇒ - ⇒ (*nat*, *string*) *shared-vars* ⇒ - ⇒ (*sllist-polynomial* × (*llist-polynomial* × *nat*) *list* × *string code-status*) *nres*›
where
‹*linear-combi-l-prep-s i A* $\mathcal{V}$ *xs* = *do* {
$WHILE_T$
(λ(*p*, *xs*, *err*). *xs* ≠ [] ∧ ¬*is-cfailed err*)

```
    (λ(p, xs, -). do {
      ASSERT(xs ≠ []);
      let (q :: llist-polynomial, i) = hd xs;
      if (i ∉# dom-m A ∨ ¬(vars-llist-in-s 𝒱 q))
      then do {
        err ← check-linear-combi-l-s-dom-err p i;
        RETURN (p, xs, error-msg i err)
      } else do {
        ASSERT(fmlookup A i ≠ None);
        let r = the (fmlookup A i);
        if q = [([], 1)]
        then do {
          pq ← add-poly-l-s 𝒱 (p, r);
          RETURN (pq, tl xs, CSUCCESS)}
        else do {
          (no-new, q) ← normalize-poly-sharedS 𝒱 (q);
          q ← mult-poly-full-s 𝒱 q r;
          pq ← add-poly-l-s 𝒱 (p, q);
          RETURN (pq, tl xs, CSUCCESS)
        }
        }
        })
        ([], xs, CSUCCESS)
          }›
```

lemma *normalize-poly-sharedS-normalize-poly-shared*:
  assumes
    ‹(𝒱, 𝒟𝒱) ∈ perfectly-shared-vars-rel›
    ‹(xs, xs′) ∈ Id›
  shows ‹normalize-poly-sharedS 𝒱 xs
    ≤ ⇓(bool-rel $\times_r$ perfectly-shared-polynom 𝒱)
    (normalize-poly-shared 𝒟𝒱 xs′)›
⟨proof⟩

lemma *linear-combi-l-prep-s-linear-combi-l-prep*:
  assumes
    ‹(𝒱, 𝒟𝒱) ∈ perfectly-shared-vars-rel›
    ‹(A,B) ∈ ⟨nat-rel, perfectly-shared-polynom 𝒱⟩fmap-rel›
    ‹(xs,xs′) ∈ Id›
  shows ‹linear-combi-l-prep-s i A 𝒱 xs
    ≤ ⇓(perfectly-shared-polynom 𝒱 $\times_r$ Id $\times_r$ Id)
    (linear-combi-l-prep2 j B 𝒟𝒱 xs′)›
⟨proof⟩

definition *check-linear-combi-l-s-mult-err* :: ‹sllist-polynomial ⇒ sllist-polynomial ⇒ string nres› **where**
  ‹check-linear-combi-l-s-mult-err pq r = SPEC (λ-. True)›

definition *weak-equality-l-s* :: ‹sllist-polynomial ⇒ sllist-polynomial ⇒ bool nres› **where**
  ‹weak-equality-l-s p q = RETURN (p = q)›

definition *check-linear-combi-l-s* **where**
  ‹check-linear-combi-l-s spec A 𝒱 i xs r = do {
  (mem-err, r) ← import-poly-no-newS 𝒱 r;

if mem-err ∨ *i* ∈# *dom-m A* ∨ *xs* = []
then do {
err ← *check-linear-combi-l-pre-err i* (*i* ∈# *dom-m A*) (*xs* = []) (*mem-err*);
RETURN (*error-msg i err*, *r*)
}
else do {
(*p*, -, *err*) ← *linear-combi-l-prep-s i A* 𝒱 *xs*;
if (*is-cfailed err*)
then do {
RETURN (*err*, *r*)
}
else do {
b ← *weak-equality-l-s p r*;
b' ← *weak-equality-l-s r spec*;
if b then (*if b' then RETURN* (*CFOUND*, *r*) *else RETURN* (*CSUCCESS*, *r*)) *else do* {
c ← *check-linear-combi-l-s-mult-err p r*;
RETURN (*error-msg i c*, *r*)
}
}
}}›

definition *weak-equality-l-s'* :: ‹-› **where**
‹*weak-equality-l-s'* - = *weak-equality-l-s*›

definition *weak-equality-l'* :: ‹-› **where**
‹*weak-equality-l'* - = *weak-equality-l*›

lemma *weak-equality-l-s-weak-equality-l*:
fixes *a* :: *sllist-polynomial* **and** *b* :: *llist-polynomial* **and** 𝒱 :: ‹(*nat*,*string*)*shared-vars*›
assumes
‹(𝒱, 𝒟𝒱) ∈ *perfectly-shared-vars-rel*›
‹(*a*,*b*) ∈ *perfectly-shared-polynom* 𝒱›
‹(*c*,*d*) ∈ *perfectly-shared-polynom* 𝒱›
shows
‹*weak-equality-l-s'* 𝒱 *a c* ≤⇓*bool-rel* (*weak-equality-l'* 𝒟𝒱 *b d*)›
⟨*proof*⟩

lemma *check-linear-combi-l-s-check-linear-combi-l*:
assumes
‹(𝒱, 𝒟𝒱) ∈ *perfectly-shared-vars-rel*›
‹(*A*,*B*) ∈ ⟨*nat-rel*, *perfectly-shared-polynom* 𝒱⟩*fmap-rel*› **and**
‹(*xs*,*xs'*)∈ *Id*›
‹(*r*,*r'*)∈*Id*›
‹(*i*,*j*)∈*nat-rel*›
‹(*spec*, *spec'*) ∈ *perfectly-shared-polynom* 𝒱›
shows ‹*check-linear-combi-l-s spec A* 𝒱 *i r xs*
≤ ⇓(*Id* $\times_r$ *perfectly-shared-polynom* 𝒱)
(*check-linear-combi-l-prop spec' B* 𝒟𝒱 *j r' xs'*)›
⟨*proof*⟩

definition *check-extension-l-s-new-var-multiple-err* :: ‹*string* ⇒ *sllist-polynomial* ⇒ *string nres*› **where**
‹*check-extension-l-s-new-var-multiple-err v p* = *SPEC* (λ-. *True*)›

definition *check-extension-l-s-side-cond-err*
:: ‹*string* ⇒ *sllist-polynomial* ⇒ *sllist-polynomial* ⇒ *sllist-polynomial* ⇒ *string nres*›
where

```
  ‹check-extension-l-s-side-cond-err v p p' q = SPEC (λ-. True)›
```
term *is-new-variable*
definition (**in** −)*check-extension-l2-s*
```
  :: ‹- ⇒ - ⇒ (nat,string)shared-vars ⇒ nat ⇒ string ⇒ llist-polynomial ⇒
     (string code-status × sllist-polynomial × (nat,string)shared-vars × nat) nres›
```
where
```
  ‹check-extension-l2-s spec A 𝒱 i v p = do {
  n ← is-new-variableS v 𝒱;
  let pre = i ∉# dom-m A ∧ n;
  let nonew = vars-llist-in-s 𝒱 p;
  (mem, p, 𝒱) ← import-polyS 𝒱 p;
  let pre = (pre ∧ ¬alloc-failed mem);
  if ¬pre
  then do {
    c ← check-extension-l-dom-err i;
    RETURN (error-msg i c, [], 𝒱, 0)
  } else do {
      if ¬nonew
      then do {
        c ← check-extension-l-s-new-var-multiple-err v p;
        RETURN (error-msg i c, [], 𝒱, 0)
      }
      else do {
        (mem', 𝒱, v') ← import-variableS v 𝒱;
        if alloc-failed mem'
        then do {
          c ← check-extension-l-dom-err i;
          RETURN (error-msg i c, [], 𝒱, 0)
        } else
        do {
         p2 ← mult-poly-full-s 𝒱 p p;
         let p'' = map (λ(a,b). (a, −b)) p;
         q ← add-poly-l-s 𝒱 (p2, p'');
         eq ← weak-equality-l-s q [];
         if eq then do {
           RETURN (CSUCCESS, p, 𝒱, v')
         } else do {
          c ← check-extension-l-s-side-cond-err v p p'' q;
          RETURN (error-msg i c, [], 𝒱, v')
        }
       }
      }
     }
  }
 }›
```
lemma *list-rel-tlD*: ‹(a, b) ∈ ⟨R⟩list-rel ⟹ (tl a, tl b) ∈ ⟨R⟩list-rel›
  ⟨*proof*⟩

lemma *check-extension-l2-prop-alt-def*:
```
  ‹check-extension-l2-prop spec A 𝒱 i v p = do {
  n ← is-new-variable v 𝒱;
  let pre = i ∉# dom-m A ∧ n;
  let nonew = vars-llist p ⊆ set-mset 𝒱;
  (mem, p, 𝒱) ← import-poly 𝒱 p;
  (mem', 𝒱, va) ← if pre ∧ nonew ∧ ¬ alloc-failed mem then import-variable v 𝒱 else RETURN (mem,
𝒱, v);
```

let pre = ((*pre* ∧ ¬*alloc-failed mem*) ∧ ¬*alloc-failed mem′*);

if ¬*pre*
then do {
  c ← *check-extension-l-dom-err i*;
  RETURN (*error-msg i c*, [], 𝒱, *va*)
} *else do* {
    if ¬*nonew*
    then do {
      c ← *check-extension-l-new-var-multiple-err v p*;
      RETURN (*error-msg i c*, [], 𝒱, *va*)
    }
    else do {
       ASSERT(*vars-llist p* ⊆ *set-mset* 𝒱);
       p2 ← *mult-poly-full-prop* 𝒱 *p p*;
       ASSERT(*vars-llist p2* ⊆ *set-mset* 𝒱);
       let p″ = *map* (λ(*a*,*b*). (*a*, −*b*)) *p*;
       ASSERT(*vars-llist p″* ⊆ *set-mset* 𝒱);
       q ← *add-poly-l-prep* 𝒱 (*p2*, *p″*);
       ASSERT(*vars-llist q* ⊆ *set-mset* 𝒱);
       eq ← *weak-equality-l q* [];
       if eq then do {
         RETURN (*CSUCCESS*, *p*, 𝒱, *va*)
       } *else do* {
        c ← *check-extension-l-side-cond-err v p q*;
        RETURN (*error-msg i c*, [], 𝒱, *va*)
      }
    }
  }
}›
⟨*proof*⟩

lemma *check-extension-l2-prop-alt-def2*:
‹*check-extension-l2-prop spec A* 𝒱 *i v p* = *do* {
n ← *is-new-variable v* 𝒱;
let pre = *i* ∉# *dom-m A* ∧ *n*;
let nonew = *vars-llist p* ⊆ *set-mset* 𝒱;
(*mem*, *p*, 𝒱) ← *import-poly* 𝒱 *p*;
let pre = (*pre* ∧ ¬*alloc-failed mem*);
if ¬*pre*
then do {
  c ← *check-extension-l-dom-err i*;
  RETURN (*error-msg i c*, [], 𝒱, *v*)
 } *else do* {
    if ¬*nonew*
    then do {
      c ← *check-extension-l-new-var-multiple-err v p*;
      RETURN (*error-msg i c*, [], 𝒱, *v*)
    }
    else do {
    (*mem′*, 𝒱, *va*) ← *import-variable v* 𝒱;
    if (*alloc-failed mem′*)
    then do {
     c ← *check-extension-l-dom-err i*;
     RETURN (*error-msg i c*, [], 𝒱, *va*)

      }
      else do {
        ASSERT(*vars-llist p* ⊆ *set-mset* 𝒱);
        p2 ← *mult-poly-full-prop* 𝒱 *p p*;
        ASSERT(*vars-llist p2* ⊆ *set-mset* 𝒱);
        let p'' = *map* (λ(*a*,*b*). (*a*, −*b*)) *p*;
        ASSERT(*vars-llist* p'' ⊆ *set-mset* 𝒱);
        q ← *add-poly-l-prep* 𝒱 (*p2*, p'');
        ASSERT(*vars-llist q* ⊆ *set-mset* 𝒱);
        eq ← *weak-equality-l q* [];
        if eq then do {
          RETURN (*CSUCCESS*, *p*, 𝒱, *va*)
        } *else do* {
          c ← *check-extension-l-side-cond-err v p q*;
          RETURN (*error-msg i c*, [], 𝒱, *va*)
        }
      }
    }
  }
}›
⟨*proof*⟩

lemma *list-rel-mapI*: ‹(*xs*,*ys*) ∈ ⟨*R*⟩*list-rel* ⟹ (⋀*x y*. *x* ∈ *set xs* ⟹ *y* ∈ *set ys* ⟹ (*x*,*y*)∈*R* ⟹ (*f x*, *g y*) ∈ *S*) ⟹ (*map f xs*, *map g ys*) ∈ ⟨*S*⟩*list-rel*›
⟨*proof*⟩

lemma *perfectly-shared-var-rel-perfectly-shared-monom-mono*:
‹(∀ *xs*. *xs* ∈ *perfectly-shared-var-rel* 𝒜 ⟶ *xs* ∈ *perfectly-shared-var-rel* 𝒜′) ⟷
(∀ *xs*. *xs* ∈ *perfectly-shared-monom* 𝒜 ⟶ *xs* ∈ *perfectly-shared-monom* 𝒜′)›
⟨*proof*⟩

lemma *perfectly-shared-var-rel-perfectly-shared-polynom-mono*:
‹(∀ *xs*. *xs* ∈ *perfectly-shared-var-rel* 𝒜 ⟶ *xs* ∈ *perfectly-shared-var-rel* 𝒜′) ⟷
(∀ *xs*. *xs* ∈ *perfectly-shared-polynom* 𝒜 ⟶ *xs* ∈ *perfectly-shared-polynom* 𝒜′)›
⟨*proof*⟩

lemma *check-extension-l2-s-check-extension-l2*:
  assumes
    ‹(𝒱, 𝒟𝒱) ∈ *perfectly-shared-vars-rel*›
    ‹(*A*,*B*) ∈ ⟨*nat-rel*, *perfectly-shared-polynom* 𝒱⟩*fmap-rel*› **and**
    ‹(*r*,*r*′)∈*Id*›
    ‹(*i*,*j*)∈*nat-rel*›
    ‹(*spec*, *spec*′) ∈ *perfectly-shared-polynom* 𝒱›
    ‹(*v*,*v*′)∈*Id*›
  shows ‹*check-extension-l2-s spec A* 𝒱 *i v r*
    ≤ ⇓{((*err*, *p*, *A*, *v*), (*err*′, *p*′, *A*′, *v*′)).
    (*err*, *err*′) ∈ *Id* ∧
    (¬*is-cfailed err* ⟶
    (*p*, *p*′) ∈ *perfectly-shared-polynom A* ∧ (*v*, *v*′) ∈ *perfectly-shared-var-rel A* ∧
    (*A*, *A*′) ∈ {(*a*,*b*). (*a*,*b*) ∈ *perfectly-shared-vars-rel* ∧ *perfectly-shared-polynom* 𝒱 ⊆ *perfectly-shared-polynom a*})}
    (*check-extension-l2-prop spec*′ *B* 𝒟𝒱 *j v*′ *r*′)›
⟨*proof*⟩

definition *PAC-checker-l-step-s*
:: ‹*sllist-polynomial* ⇒ *string code-status* × (*nat*,*string*)*shared-vars* × - ⇒ (*llist-polynomial*, *string*, *nat*) *pac-step* ⇒ -›
where
‹*PAC-checker-l-step-s* = (λ*spec* (*st'*, 𝒱, *A*) *st*. *do* {
ASSERT (¬*is-cfailed st'*);
case st of
CL - - - ⇒
do {
r ← *full-normalize-poly* (*pac-res st*);
(*eq*, *r*) ← *check-linear-combi-l-s spec A* 𝒱 (*new-id st*) (*pac-srcs st*) *r*;
let - = *eq*;
if ¬*is-cfailed eq*
then RETURN (*merge-cstatus st' eq*, 𝒱, *fmupd* (*new-id st*) *r A*)
else RETURN (*eq*, 𝒱, *A*)
}
| *Del* - ⇒
do {
eq ← *check-del-l spec A* (*pac-src1 st*);
let - = *eq*;
if ¬*is-cfailed eq*
then RETURN (*merge-cstatus st' eq*, 𝒱, *fmdrop* (*pac-src1 st*) *A*)
else RETURN (*eq*, 𝒱, *A*)
}
| *Extension* - - - ⇒
do {
r ← *full-normalize-poly* (*pac-res st*);
(*eq*, *r*, 𝒱, *v*) ← *check-extension-l2-s spec A* (𝒱) (*new-id st*) (*new-var st*) *r*;
if ¬*is-cfailed eq*
then do {
r ← *add-poly-l-s* 𝒱 ([([*v*], −*1*)], *r*);
RETURN (*st'*, 𝒱, *fmupd* (*new-id st*) *r A*)
}
else RETURN (*eq*, 𝒱, *A*)
}}
)›

lemma *is-cfailed-merge-cstatus*:
is-cfailed (*merge-cstatus c d*) ⟷ *is-cfailed c* ∨ *is-cfailed d*
⟨*proof*⟩

lemma (**in** −) *fmap-rel-mono2*:
‹*x* ∈ ⟨*A*,*B*⟩*fmap-rel* ⟹ *B* ⊆*B'* ⟹ *x* ∈ ⟨*A*,*B'*⟩*fmap-rel*›
⟨*proof*⟩

lemma *PAC-checker-l-step-s-PAC-checker-l-step-s*:
assumes
‹(𝒱, 𝒟𝒱) ∈ *perfectly-shared-vars-rel*›
‹(*A*,*B*) ∈ ⟨*nat-rel*, *perfectly-shared-polynom* 𝒱⟩*fmap-rel*› **and**
‹(*spec*, *spec'*) ∈ *perfectly-shared-polynom* 𝒱› **and**
‹(*err*, *err'*) ∈ *Id*› **and**
‹(*st*,*st'*)∈*Id*›
shows ‹*PAC-checker-l-step-s spec* (*err*, 𝒱, *A*) *st*
≤ ⇓{((*err*, 𝒱', *A'*), (*err'*, 𝒟𝒱', *B'*)).
(*err*, *err'*) ∈ *Id* ∧
(¬*is-cfailed err* ⟶ ((𝒱', 𝒟𝒱') ∈ *perfectly-shared-vars-rel* ∧(*A'*,*B'*) ∈ ⟨*nat-rel*, *perfectly-shared-polynom* 𝒱'⟩*fmap-rel* ∧

perfectly-shared-polynom $\mathcal{V}$ ⊆ *perfectly-shared-polynom* $\mathcal{V}'$))}
(*PAC-checker-l-step-prep spec'* (*err'*, $\mathcal{DV}$, *B*) *st'*)›
⟨*proof*⟩

lemma *PAC-checker-l-step-s-PAC-checker-l-step-s2*:
assumes
‹(*st*,*st'*)∈*Id*›
‹(*spec*, *spec'*) ∈ *perfectly-shared-polynom* (*fst* (*snd errVA*))› **and**
‹((*errVA*), (*err'DVB*)) ∈ *Id* $\times_r$ *perfectly-shared-vars-rel* $\times_r$ ⟨*nat-rel*, *perfectly-shared-polynom* (*fst* (*snd errVA*))⟩*fmap-rel*›
shows ‹*PAC-checker-l-step-s spec* (*errVA*) *st*
≤ ⇓{((*err*, $\mathcal{V}'$, *A'*), (*err'*, $\mathcal{DV}'$, *B'*)).
(*err*, *err'*) ∈ *Id* ∧
(¬*is-cfailed err* ⟶ (($\mathcal{V}'$, $\mathcal{DV}'$) ∈ *perfectly-shared-vars-rel* ∧(*A'*,*B'*) ∈ ⟨*nat-rel*, *perfectly-shared-polynom* $\mathcal{V}'$⟩*fmap-rel* ∧
perfectly-shared-polynom (*fst* (*snd errVA*)) ⊆ *perfectly-shared-polynom* $\mathcal{V}'$))}
(*PAC-checker-l-step-prep spec'* (*err'DVB*) *st'*)›
⟨*proof*⟩

definition *fully-normalize-and-import* **where**
‹*fully-normalize-and-import* $\mathcal{V}$ *p* = *do* {
p ← *sort-all-coeffs p*;
(*err*, *p*, $\mathcal{V}$) ← *import-polyS* $\mathcal{V}$ *p*;
if alloc-failed err
then RETURN (*err*, *p*, $\mathcal{V}$)
else do {
p ← *normalize-poly-s* $\mathcal{V}$ *p*;
RETURN (*err*, *p*, $\mathcal{V}$)
}}›

fun *vars-llist-l* **where**
‹*vars-llist-l* [] = []› |
‹*vars-llist-l* (*x*#*xs*) = *fst x* @ *vars-llist-l xs*›

lemma *set-vars-llist-l*[*simp*]: ‹*set*(*vars-llist-l xs*) = *vars-llist xs*›
⟨*proof*⟩

lemma *vars-llist-l-append*[*simp*]: ‹*vars-llist-l* (*a* @ *b*) = *vars-llist-l a* @ *vars-llist-l b*›
⟨*proof*⟩

definition (**in** −) *remap-polys-s-with-err* :: ‹*llist-polynomial* ⇒ *llist-polynomial* ⇒ (*nat*, *string*) *shared-vars* ⇒ (*nat*, *llist-polynomial*) *fmap* ⇒
(*string code-status* × (*nat*, *string*) *shared-vars* × (*nat*, *sllist-polynomial*) *fmap* × *sllist-polynomial*) *nres*› **where**
‹*remap-polys-s-with-err spec spec0* = (λ($\mathcal{V}$:: (*nat*, *string*) *shared-vars*) *A*. *do*{
ASSERT(*vars-llist spec* ⊆ *vars-llist spec0*);
dom ← *SPEC*(λ*dom*. *set-mset* (*dom-m A*) ⊆ *dom* ∧ *finite dom*);
(*mem*, $\mathcal{V}$) ← *import-variablesS* (*vars-llist-l spec0*) $\mathcal{V}$;
(*mem'*, *spec*, $\mathcal{V}$) ← *if* ¬*alloc-failed mem then import-polyS* $\mathcal{V}$ *spec else RETURN* (*mem*, [], $\mathcal{V}$);
failed ← *SPEC*(λ*b*::*bool*. *alloc-failed mem* ∨ *alloc-failed mem'* ⟶ *b*);
if failed
then do {
c ← *remap-polys-l-dom-err*;
RETURN (*error-msg* (*0* :: *nat*) *c*, $\mathcal{V}$, *fmempty*, [])
}

else do {
(err, 𝒱, A) ← FOREACH$_C$ dom (λ(err, 𝒱, A′). ¬is-cfailed err)
(λi (err, 𝒱, A′).
if i ∈# dom-m A
then do {
(err′, p, 𝒱) ← import-polyS 𝒱 (the (fmlookup A i));
if alloc-failed err′ then RETURN((CFAILED ″memory out″, 𝒱, A′))
else do {
p ← full-normalize-poly-s 𝒱 p;
eq ← weak-equality-l-s′ 𝒱 p spec;
let 𝒱 = 𝒱;
RETURN((if eq then CFOUND else CSUCCESS), 𝒱, fmupd i p A′)
}
} else RETURN (err, 𝒱, A′))
(CSUCCESS, 𝒱, fmempty);
RETURN (err, 𝒱, A, spec)
}})›

lemma full-normalize-poly-alt-def:
‹full-normalize-poly p0 = do {
p ← sort-all-coeffs p0;
ASSERT(vars-llist p ⊆ vars-llist p0);
p ← sort-poly-spec p;
ASSERT(vars-llist p ⊆ vars-llist p0);
RETURN (merge-coeffs0 p)
}› (**is** ‹?A = ?B›)
⟨proof⟩

definition full-normalize-poly′ :: ‹-› **where**
‹full-normalize-poly′ - = full-normalize-poly›

lemma full-normalize-poly-s-full-normalize-poly:
fixes xs :: ‹sllist-polynomial› **and**
𝒱 :: ‹(nat,string)shared-vars›
assumes
‹(xs, xs′) ∈ perfectly-shared-polynom 𝒱› **and**
𝒱: ‹(𝒱, 𝒟𝒱) ∈ perfectly-shared-vars-rel› **and**
‹vars-llist xs′ ⊆ set-mset 𝒟𝒱›
shows ‹full-normalize-poly-s 𝒱 xs ≤ ⇓(perfectly-shared-polynom 𝒱) (full-normalize-poly′ 𝒟𝒱 xs′)›
⟨proof⟩

lemma remap-polys-l2-with-err-prep-alt-def:
‹remap-polys-l2-with-err-prep spec spec0 = (λ(𝒱:: (nat, string) vars) A. do{
ASSERT(vars-llist spec ⊆ vars-llist spec0);
dom ← SPEC(λdom. set-mset (dom-m A) ⊆ dom ∧ finite dom);
(mem, 𝒱) ← SPEC(λ(mem, 𝒱′). ¬alloc-failed mem ⟶ set-mset 𝒱′ = set-mset 𝒱 ∪ vars-llist spec0);
(mem′, spec, 𝒱) ← if ¬alloc-failed mem then import-poly 𝒱 spec else SPEC(λ-. True);
failed ← SPEC(λb::bool. alloc-failed mem ∨ alloc-failed mem′ ⟶ b);
if failed
then do {
c ← remap-polys-l-dom-err;
SPEC (λ(mem, -, -, -). mem = error-msg (0::nat) c)
}
else do {
(err, 𝒱, A) ← FOREACH$_C$ dom (λ(err, 𝒱, A′). ¬is-cfailed err)

$(\lambda i\ (err,\ \mathcal{V},\ \ A').$
if $i \in\#$ *dom-m* A
then *do* {
$(err',\ p,\ \mathcal{V}) \leftarrow$ *import-poly* $\mathcal{V}$ (*the* (*fmlookup* A i));
if alloc-failed err′ then RETURN((*CFAILED ″memory out″*, $\mathcal{V}$, A'))
else do {
ASSERT(*vars-llist* $p \subseteq$ *set-mset* $\mathcal{V}$);
$p \leftarrow$ *full-normalize-poly′* $\mathcal{V}$ p;
eq $\leftarrow$ *weak-equality-l′* $\mathcal{V}$ p *spec*;
let $\mathcal{V} = \mathcal{V}$;
RETURN((*if eq then CFOUND else CSUCCESS*), $\mathcal{V}$, *fmupd* i p A')
}
} *else RETURN* (*err*, $\mathcal{V}$, A'))
(*CSUCCESS*, $\mathcal{V}$, *fmempty*);
RETURN (*err*, $\mathcal{V}$, A, *spec*)
}})›
⟨*proof*⟩

lemma *remap-polys-s-with-err-remap-polys-l2-with-err-prep*:
fixes $\mathcal{V}$ :: ‹(*nat*, *string*) *shared-vars*›
assumes
$\mathcal{V}$: ‹($\mathcal{V}$, $\mathcal{DV}$) ∈ *perfectly-shared-vars-rel*› **and**
AB: ‹(*A*,*B*) ∈ ⟨*nat-rel*, *Id*⟩*fmap-rel*› **and**
‹(*spec*, *spec′*) ∈ ⟨⟨*Id*⟩*list-rel* $\times_r$ *int-rel*⟩*list-rel*› **and**
spec0: ‹(*spec0*, *spec0′*) ∈ ⟨⟨*Id*⟩*list-rel* $\times_r$ *int-rel*⟩*list-rel*›
shows
‹*remap-polys-s-with-err spec spec0* $\mathcal{V}$ A ≤
⇓{((*err*, $\mathcal{V}$, A, *fspec*), (*err′*, $\mathcal{V}'$, A', *fspec′*)).
(*err*, *err′*) ∈ *Id* ∧
(¬*is-cfailed err* ⟶ (*fspec*, *fspec′*) ∈ *perfectly-shared-polynom* $\mathcal{V}$ ∧
((*err*, $\mathcal{V}$, A), (*err′*, $\mathcal{V}'$, A')) ∈ *Id* $\times_r$ *perfectly-shared-vars-rel* $\times_r$⟨*nat-rel*, *perfectly-shared-polynom* $\mathcal{V}$⟩*fmap-rel*)}
(*remap-polys-l2-with-err-prep spec′ spec0′* $\mathcal{DV}$ B)›
⟨*proof*⟩

definition *PAC-checker-l-s* **where**
‹*PAC-checker-l-s 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-s spec bA* (*hd n*);
RETURN (*S*, *tl n*)
})
((*b*, *A*), *st*);
RETURN S
}›

lemma *PAC-checker-l-s-PAC-checker-l-prep-s*:
assumes
‹($\mathcal{V}$, $\mathcal{DV}$) ∈ *perfectly-shared-vars-rel*›
‹(*A*,*B*) ∈ ⟨*nat-rel*, *perfectly-shared-polynom* $\mathcal{V}$⟩*fmap-rel*› **and**
‹(*spec*, *spec′*) ∈ *perfectly-shared-polynom* $\mathcal{V}$› **and**
‹(*err*, *err′*) ∈ *Id*› **and**

‹$(st,st')\in Id$›
shows ‹*PAC-checker-l-s spec* ($\mathcal{V}$, *A*) *err st*
$\leq \Downarrow\{((err,\ \mathcal{V}',\ A'),\ (err',\ \mathcal{DV}',\ B')).$
$(err,\ err') \in Id \wedge$
$(\neg is\text{-}cfailed\ err \longrightarrow ((\mathcal{V}',\ \mathcal{DV}') \in perfectly\text{-}shared\text{-}vars\text{-}rel \wedge (A',B') \in \langle nat\text{-}rel,\ perfectly\text{-}shared\text{-}polynom$
$\mathcal{V}'\rangle fmap\text{-}rel))\}$
(*PAC-checker-l2 spec'* ($\mathcal{DV}$, *B*) *err' st'*)›
⟨*proof*⟩

definition *full-checker-l-s*
:: ‹*llist-polynomial* ⇒ (*nat*, *llist-polynomial*) *fmap* ⇒ (-, *string*, *nat*) *pac-step list* ⇒
(*string code-status* × -) *nres*›
where
‹*full-checker-l-s spec A st* = *do* {
spec' ← *full-normalize-poly spec*;
(*b*, $\mathcal{V}$, *A*, *spec'*) ← *remap-polys-s-with-err spec' spec* ({#}, *fmempty*, *fmempty*) *A*;
if is-cfailed b
then RETURN (*b*, $\mathcal{V}$, *A*)
else do {
PAC-checker-l-s spec' ($\mathcal{V}$, *A*) *b st*
}
}›
lemma *full-checker-l-s-full-checker-l-prep*:
assumes
‹(*A*,*B*) ∈ ⟨*nat-rel*, *Id*⟩*fmap-rel*› **and**
‹(*spec*, *spec'*) ∈ ⟨⟨*Id*⟩*list-rel* $\times_r$ *int-rel*⟩*list-rel*› **and**
‹$(st,st')\in Id$›
shows ‹*full-checker-l-s spec A st*
$\leq \Downarrow\{((err,\ \text{-}),\ (err',\ \text{-})).\ (err,\ err') \in Id\}$
(*full-checker-l-prep spec' B st'*)›
⟨*proof*⟩

lemma *full-checker-l-s-full-checker-l-prep'*:
‹(*uncurry2 full-checker-l-s*, *uncurry2 full-checker-l-prep*)∈
((⟨⟨*Id*⟩*list-rel* $\times_r$ *int-rel*⟩*list-rel* $\times_r$ ⟨*nat-rel*, *Id*⟩*fmap-rel*) $\times_r$ *Id* $\rightarrow_f$
⟨$\{((err,\ \text{-}),\ (err',\ \text{-})).\ (err,\ err') \in Id\}$⟩*nres-rel*›
⟨*proof*⟩

definition *merge-coeff-s* :: ‹(*nat*,*string*)*shared-vars* ⇒ *nat list* ⇒ *nat list* ⇒ *nat list* ⇒ *nat list nres*›
where
‹*merge-coeff-s* $\mathcal{V}$ *xs* = *mergeR* ($\lambda a\ b.\ a \in set\ xs \wedge b \in set\ xs$)
($\lambda a\ b.$ *do* {
x ← *get-var-nameS* $\mathcal{V}$ *a*;
y ← *get-var-nameS* $\mathcal{V}$ *b*;
RETURN($a = b \vee var\text{-}order\ x\ y$)
})›

term *get-var-nameS*
sepref-definition *merge-coeff-s-impl*
is ‹*uncurry3 merge-coeff-s*›
:: ‹$shared\text{-}vars\text{-}assn^k *_a (monom\text{-}s\text{-}assn)^k *_a (monom\text{-}s\text{-}assn)^k *_a (monom\text{-}s\text{-}assn)^k \rightarrow_a monom\text{-}s\text{-}assn$›
⟨*proof*⟩

sepref-register *merge-coeff-s msort-coeff-s sort-all-coeffs-s*
lemmas [*sepref-fr-rules*] = *merge-coeff-s-impl.refine*

lemma *msort-coeff-s-alt-def*:
‹*msort-coeff-s* $\mathcal{V}$ *xs* = *do* {
let zs = *COPY xs*;
REC_T
(λ*msortR′ xsa*. *if length xsa* ≤ *1 then RETURN* (*ASSN-ANNOT monom-s-assn xsa*) *else do* {
let xs1 = *ASSN-ANNOT monom-s-assn* (*take* (*length xsa div 2*) *xsa*);
let xs2 = *ASSN-ANNOT monom-s-assn* (*drop* (*length xsa div 2*) *xsa*);
as ← *msortR′ xs1*;
let as = *ASSN-ANNOT monom-s-assn as*;
bs ← *msortR′ xs2*;
let bs = *ASSN-ANNOT monom-s-assn bs*;
merge-coeff-s $\mathcal{V}$ *zs as bs*
})
xs}›
⟨*proof*⟩

sepref-definition *msort-coeff-s-impl*
is ‹*uncurry msort-coeff-s*›
:: ‹$shared\text{-}vars\text{-}assn^k *_a (monom\text{-}s\text{-}assn)^k \rightarrow_a monom\text{-}s\text{-}assn$›
⟨*proof*⟩

lemmas [*sepref-fr-rules*] = *msort-coeff-s-impl.refine*

sepref-definition *sort-all-coeffs-s′-impl*
is ‹*uncurry sort-all-coeffs-s*›
:: ‹$shared\text{-}vars\text{-}assn^k *_a poly\text{-}s\text{-}assn^d \rightarrow_a poly\text{-}s\text{-}assn$›
⟨*proof*⟩

lemmas [*sepref-fr-rules*] = *sort-all-coeffs-s′-impl.refine*

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

lemma [*sepref-import-param*]: ‹(((=)), ((=))) ∈ ⟨*uint64-nat-rel*⟩ *list-rel* → ⟨*uint64-nat-rel*⟩ *list-rel* → *bool-rel*›
⟨*proof*⟩

lemma *is-pure-monom-s-assn*: ‹*is-pure monom-s-assn*›

‹*is-pure* (*monom-s-assn* $\times_a$ *int-assn*)›
⟨*proof*⟩

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

lemmas [*sepref-fr-rules*] = *merge-coeffs0-s-impl.refine*

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

lemma *weak-equality-l-s-alt-def*:
‹*weak-equality-l-s* = *RETURN oo* ($\lambda p\ q.\ p = q$)›
⟨*proof*⟩

lemma [*sepref-import-param*]
: ‹(((=)), ((=))) ∈ ⟨⟨*uint64-nat-rel*⟩ *list-rel* $\times_r$ *int-rel*⟩ *list-rel* → ⟨⟨*uint64-nat-rel*⟩ *list-rel* $\times_r$ *int-rel*⟩ *list-rel* → *bool-rel*›
⟨*proof*⟩

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

code-printing constant *arl-get-u'* ⇀ (*SML*) (*fn*/ ()/ =>/ *Array.sub*/ ((*fn*/ (*a*,*b*)/ =>/ *a*) ((-)),/ *Word32.toInt* ((-))))

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

sepref-definition *import-monom-no-newS-impl*
is ‹*uncurry* (*import-monom-no-newS* :: (*nat*,*string*)*shared-vars* ⇒ - ⇒(*bool* × -) *nres*)›
:: ‹*shared-vars-assn*^{k} $*_a$ (*list-assn string-assn*)^k $\rightarrow_a$ *bool-assn* $\times_a$ *list-assn uint64-nat-assn*›
⟨*proof*⟩
sepref-register *import-monom-no-newS import-poly-no-newS check-linear-combi-l-pre-err*
lemmas [*sepref-fr-rules*] =
import-monom-no-newS-impl.refine weak-equality-l-s-impl.refine

sepref-definition *import-poly-no-newS-impl*
is ‹*uncurry* (*import-poly-no-newS* :: (*nat*,*string*)*shared-vars* ⇒ *llist-polynomial* ⇒(*bool* × *sllist-polynomial*) *nres*)›
:: ‹*shared-vars-assn*^{k} $*_a$ *poly-assn*^{k} $\rightarrow_a$ *bool-assn* $\times_a$ *poly-s-assn*›
⟨*proof*⟩

lemmas [*sepref-fr-rules*] =
import-poly-no-newS-impl.refine

definition *check-linear-combi-l-pre-err-impl* **where**
⟨*check-linear-combi-l-pre-err-impl i pd p mem* =
(*if pd then ''The polynomial with id '' @ show* (*nat-of-uint64 i*) @ *'' was not found'' else ''''*) @
(*if p then ''The co−factor from '' @ show* (*nat-of-uint64 i*) @ *'' was empty'' else ''''*)@
(*if mem then ''Memory out'' else ''''*)⟩

definition *check-mult-l-mult-err-impl* **where**
⟨*check-mult-l-mult-err-impl p q pq r* =
''Multiplying '' @ show p @ '' by '' @ show q @ '' gives '' @ show pq @ '' and not '' @ show r⟩

lemma [*sepref-fr-rules*]:
⟨(*uncurry3* ((*λx y. return oo* (*check-linear-combi-l-pre-err-impl x y*))),
uncurry3 (*check-linear-combi-l-pre-err*)) ∈ *uint64-nat-assn*^{k} $*_a$ *bool-assn*^{k} $*_a$ *bool-assn*^{k} $*_a$ *bool-assn*^{k}
$\rightarrow_a$ *raw-string-assn*⟩
⟨*proof*⟩

lemma *vars-llist-in-s-single*: ⟨*RETURN* (*vars-llist-in-s* 𝒱 [(*xs*, *a*)]) =
REC$_T$ (*λf xs. case xs of*
[] ⇒ *RETURN True*
| *x* # *xs* ⇒ *do* {
b ← *is-new-variableS x* 𝒱;
if b then RETURN False
else f xs
}) (*xs*)⟩
⟨*proof*⟩

lemma *vars-llist-in-s-alt-def*: ⟨(*RETURN oo vars-llist-in-s*) 𝒱 *xs* =
REC$_T$ (*λf xs. case xs of*
[] ⇒ *RETURN True*
| (*x*, *a*) # *xs* ⇒ *do* {
b ← *RETURN* (*vars-llist-in-s* 𝒱 [(*x*, *a*)]);
if ¬*b then RETURN False*
else f xs
}) *xs*⟩
⟨*proof*⟩

sepref-definition *vars-llist-in-s-impl*
is ⟨*uncurry* (*RETURN oo vars-llist-in-s*)⟩
:: ⟨*shared-vars-assn*^{k} $*_a$ *poly-assn*^{k} $\rightarrow_a$ *bool-assn*⟩
⟨*proof*⟩
lemmas [*sepref-fr-rules*] = *vars-llist-in-s-impl.refine*

definition *check-linear-combi-l-s-dom-err-impl* :: ⟨- ⇒ *uint64* ⇒ -⟩ **where**
⟨*check-linear-combi-l-s-dom-err-impl x p* =
''Poly not found in CL from x '' @ show (*nat-of-uint64 p*)⟩

lemma [*sepref-fr-rules*]:
⟨(*uncurry* (*return oo* (*check-linear-combi-l-s-dom-err-impl*)),
uncurry (*check-linear-combi-l-s-dom-err*)) ∈ *poly-s-assn*^{k} $*_a$ *uint64-nat-assn*^{k} $\rightarrow_a$ *raw-string-assn*⟩
⟨*proof*⟩
sepref-register *check-linear-combi-l-s-dom-err-impl mult-poly-s normalize-poly-s*

sepref-definition *normalize-poly-sharedS-impl*
is ⟨*uncurry normalize-poly-sharedS*⟩
:: ⟨ *shared-vars-assn*^{k} $*_a$ *poly-assn*^{k} $\rightarrow_a$ *bool-assn* $\times_a$ *poly-s-assn*⟩

⟨proof⟩

lemmas [*sepref-fr-rules*] = *normalize-poly-sharedS-impl.refine*
mult-poly-s-impl.refine
lemma *merge-coeffs-s-alt-def*:
‹(*RETURN o merge-coeffs-s*) *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*) # *COPY p*) *else f p*
else do {*p* ← *f* ((*ys*, *m*) # *p*); *RETURN* ((*xs*, *n*) # *p*)})))
p›
⟨proof⟩

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

lemmas [*sepref-fr-rules*] = *merge-coeffs-s-impl.refine*

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

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

sepref-definition *mult-poly-full-s-impl*
is ‹*uncurry2 mult-poly-full-s*›
:: ‹$shared\text{-}vars\text{-}assn^k *_a poly\text{-}s\text{-}assn^k *_a poly\text{-}s\text{-}assn^k \rightarrow_a poly\text{-}s\text{-}assn$›
⟨proof⟩

lemmas [*sepref-fr-rules*] = *mult-poly-full-s-impl.refine*
add-poly-l-prep-impl.refine

sepref-register *add-poly-l-s*

sepref-definition *linear-combi-l-prep-s-impl*
is ‹*uncurry3 linear-combi-l-prep-s*›
:: ‹$uint64\text{-}nat\text{-}assn^k *_a polys\text{-}s\text{-}assn^k *_a shared\text{-}vars\text{-}assn^k *_a$
$(list\text{-}assn\ (poly\text{-}assn \times_a uint64\text{-}nat\text{-}assn))^d \rightarrow_a poly\text{-}s\text{-}assn \times_a (list\text{-}assn\ (poly\text{-}assn \times_a uint64\text{-}nat\text{-}assn))$
$\times_a$ *status-assn raw-string-assn*
›
⟨proof⟩

lemmas [*sepref-fr-rules*] = *linear-combi-l-prep-s-impl.refine*

definition *check-linear-combi-l-s-mult-err-impl* :: ‹- ⇒ - ⇒ -› **where**
‹*check-linear-combi-l-s-mult-err-impl x p* =
''*Unequal polynom found in CL* '' @ *show* (*map* (λ(*a*,*b*). (*map nat-of-uint64 a*, *b*)) *p*) @
'' *but* '' @ *show* (*map* (λ(*a*,*b*). (*map nat-of-uint64 a*, *b*)) *x*)›

lemma [*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (*check-linear-combi-l-s-mult-err-impl*)),
uncurry (*check-linear-combi-l-s-mult-err*)) ∈ $poly\text{-}s\text{-}assn^k *_a poly\text{-}s\text{-}assn^k \rightarrow_a raw\text{-}string\text{-}assn$›
⟨*proof*⟩

sepref-definition *check-linear-combi-l-s-impl*
is ‹*uncurry5 check-linear-combi-l-s*›
:: ‹$poly\text{-}s\text{-}assn^k *_a polys\text{-}s\text{-}assn^k *_a shared\text{-}vars\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k *_a$
$(list\text{-}assn\ (poly\text{-}assn \times_a uint64\text{-}nat\text{-}assn))^d *_a poly\text{-}assn^k \rightarrow_a status\text{-}assn\ raw\text{-}string\text{-}assn \times_a poly\text{-}s\text{-}assn$
›
⟨*proof*⟩

sepref-register *fmlookup′*
lemma *check-extension-l2-s-alt-def*:
‹*check-extension-l2-s spec A 𝒱 i v p = do* {
n ← is-new-variableS v 𝒱;
let t = fmlookup′ i A;
pre ← RETURN (t = None);
let pre = pre ∧ n;
let nonew = vars-llist-in-s 𝒱 p;
(*mem*, *p*, *𝒱*) ← *import-polyS 𝒱 p*;
let pre = (*pre* ∧ ¬*alloc-failed mem*);
if ¬*pre*
then do {
  c ← check-extension-l-dom-err i;
  RETURN (*error-msg i c*, [], *𝒱*, *0*)
} *else do* {
    if ¬*nonew*
    then do {
      c ← check-extension-l-s-new-var-multiple-err v p;
      RETURN (*error-msg i c*, [], *𝒱*, *0*)
    }
    else do {
      (*mem′*, *𝒱*, *v′*) ← *import-variableS v 𝒱*;
      if alloc-failed mem′
      then do {
        c ← check-extension-l-dom-err i;
        RETURN (*error-msg i c*, [], *𝒱*, *0*)
      } *else*
      do {
       p2 ← mult-poly-full-s 𝒱 p p;
       let p′′ = map (λ(*a*,*b*). (*a*, −*b*)) *p*;
       q ← add-poly-l-s 𝒱 (*p2*, *p′′*);
       eq ← weak-equality-l-s q [];
       if eq then do {
         RETURN (*CSUCCESS*, *p*, *𝒱*, *v′*)
       } *else do* {
         c ← check-extension-l-s-side-cond-err v p p′′ q;
         RETURN (*error-msg i c*, [], *𝒱*, *v′*)
       }
     }
    }
  }
}
}›

⟨proof⟩

definition *uminus-poly* :: ‹- ⇒ -› **where**
‹*uminus-poly* p' = *map* (λ(*a*, *b*). (*a*, − *b*)) p'›

lemma [*sepref-import-param*]: ‹(*uminus-poly*, *uminus-poly*) ∈ ⟨*monom-s-rel* $\times_r$ *int-rel*⟩*list-rel* → ⟨*monom-s-rel* $\times_r$ *int-rel*⟩*list-rel*›
⟨proof⟩

sepref-register *import-monomS import-polyS*

sepref-definition *import-monomS-impl*
is ‹*uncurry import-monomS*›
:: ‹*shared-vars-assn*^{d} $*_a$ *monom-assn*^{k} $\rightarrow_a$ *memory-allocation-assn* $\times_a$ *monom-s-assn* $\times_a$ *shared-vars-assn*›
⟨proof⟩

lemmas [*sepref-fr-rules*] =
import-monomS-impl.refine

sepref-definition *import-polyS-impl*
is ‹*uncurry import-polyS*›
:: ‹*shared-vars-assn*^{d} $*_a$ *poly-assn*^{k} $\rightarrow_a$ *memory-allocation-assn* $\times_a$ *poly-s-assn* $\times_a$ *shared-vars-assn*›
⟨proof⟩

lemmas [*sepref-fr-rules*] =
import-polyS-impl.refine

definition *check-extension-l-s-new-var-multiple-err-impl* :: ‹*String.literal* ⇒ - ⇒ -› **where**
‹*check-extension-l-s-new-var-multiple-err-impl x p* =
''*Variable already defined* '' @ *show x* @
'' *but* '' @ *show* (*map* (λ(*a*,*b*). (*map nat-of-uint64 a*, *b*)) *p*)›

lemma [*sepref-fr-rules*]:
‹(*uncurry* (*return oo* (*check-extension-l-s-new-var-multiple-err-impl*)),
uncurry (*check-extension-l-s-new-var-multiple-err*)) ∈ *string-assn*^{k} $*_a$ *poly-s-assn*^{k} $\rightarrow_a$ *raw-string-assn*›
⟨proof⟩

definition *check-extension-l-s-side-cond-err-impl* :: ‹*String.literal* ⇒ - ⇒ -› **where**
‹*check-extension-l-s-side-cond-err-impl x p* p' q' =
''*p*^*2*− *p* != *0* '' @ *show x* @
'' *but* '' @ *show* (*map* (λ(*a*,*b*). (*map nat-of-uint64 a*, *b*)) *p*) @
'' *and* '' @ *show* (*map* (λ(*a*,*b*). (*map nat-of-uint64 a*, *b*)) p') @
'' *and* '' @ *show* (*map* (λ(*a*,*b*). (*map nat-of-uint64 a*, *b*)) q')›

abbreviation *comp4* (**infixl** *oooo 55*) **where** *f oooo g* ≡ λ*x*. *f ooo* (*g x*)
abbreviation *comp5* (**infixl** *ooooo 55*) **where** *f ooooo g* ≡ λ*x*. *f oooo* (*g x*)

lemma [*sepref-fr-rules*]:
‹(*uncurry3* (*return oooo* (*check-extension-l-s-side-cond-err-impl*)),
uncurry3 (*check-extension-l-s-side-cond-err*)) ∈ *string-assn*^{k} $*_a$ *poly-s-assn*^{k}$*_a$ *poly-s-assn*^{k}$*_a$ *poly-s-assn*^{k} $\rightarrow_a$ *raw-string-assn*›
⟨proof⟩

sepref-register *mult-poly-full-s weak-equality-l-s check-extension-l-s-side-cond-err check-extension-l2-s check-linear-combi-l-s is-cfailed check-del-l*

sepref-definition *check-extension-l-impl*
is ‹*uncurry5 check-extension-l2-s*›
:: ‹$poly\text{-}s\text{-}assn^k *_a polys\text{-}s\text{-}assn^k *_a shared\text{-}vars\text{-}assn^d *_a uint64\text{-}nat\text{-}assn^k *_a$
$string\text{-}assn^k *_a poly\text{-}assn^k \rightarrow_a status\text{-}assn\ raw\text{-}string\text{-}assn \times_a poly\text{-}s\text{-}assn \times_a shared\text{-}vars\text{-}assn \times_a$
uint64-nat-assn
›
⟨*proof*⟩

lemma [*sepref-fr-rules*]:
‹(*return o is-cfailed*, *RETURN o is-cfailed*) ∈ $(status\text{-}assn\ raw\text{-}string\text{-}assn)^k \rightarrow_a bool\text{-}assn$›
⟨*proof*⟩

sepref-definition *check-del-l-impl*
is ‹*uncurry2 check-del-l*›
:: ‹$poly\text{-}s\text{-}assn^k *_a polys\text{-}s\text{-}assn^k *_a uint64\text{-}nat\text{-}assn^k \rightarrow_a status\text{-}assn\ raw\text{-}string\text{-}assn$›
⟨*proof*⟩

lemmas [*sepref-fr-rules*] =
check-extension-l-impl.refine
check-linear-combi-l-s-impl.refine
check-del-l-impl.refine

sepref-definition *PAC-checker-l-step-s-impl*
is ‹*uncurry2 PAC-checker-l-step-s*›
:: ‹$poly\text{-}s\text{-}assn^k *_a (status\text{-}assn\ raw\text{-}string\text{-}assn \times_a shared\text{-}vars\text{-}assn \times_a polys\text{-}s\text{-}assn)^d *_a$
$(pac\text{-}step\text{-}rel\text{-}assn\ (uint64\text{-}nat\text{-}assn)\ poly\text{-}assn\ string\text{-}assn)^k \rightarrow_a status\text{-}assn\ raw\text{-}string\text{-}assn \times_a$
$shared\text{-}vars\text{-}assn \times_a polys\text{-}s\text{-}assn$
›
⟨*proof*⟩

lemmas [*sepref-fr-rules*] = *PAC-checker-l-step-s-impl.refine*

fun *vars-llist-s2* :: ‹- ⇒ - *list*› **where**
‹*vars-llist-s2* [] = []› |
‹*vars-llist-s2* ((*a*,-) # *xs*) = *a* @ *vars-llist-s2 xs*›

lemma [*sepref-import-param*]:
‹(*vars-llist-s2*, *vars-llist-s2*) ∈ ⟨⟨*string-rel*⟩*list-rel* $\times_r$ *int-rel*⟩*list-rel* → ⟨*string-rel*⟩*list-rel*›
⟨*proof*⟩
sepref-register *PAC-checker-l-step-s*
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*)›
⟨*proof*⟩

lemma *safe-epac-step-rel-assn*[*safe-constraint-rules*]:
‹*CONSTRAINT is-pure K* ⟹ *CONSTRAINT is-pure V* ⟹ *CONSTRAINT is-pure R* ⟹
CONSTRAINT is-pure (*EPAC-Checker.pac-step-rel-assn K V R*)›
⟨*proof*⟩

sepref-definition *PAC-checker-l-s-impl*
is ‹*uncurry3 PAC-checker-l-s*›

:: ‹*poly-s-assn*^{k} $*_a$ (*shared-vars-assn* $\times_a$ *polys-s-assn*)^d $*_a$(*status-assn raw-string-assn*)^d $*_a$
(*list-assn* (*pac-step-rel-assn* (*uint64-nat-assn*) *poly-assn string-assn*))^d $\rightarrow_a$
status-assn raw-string-assn $\times_a$ *shared-vars-assn* $\times_a$ *polys-s-assn*
›
⟨*proof*⟩

lemmas [*sepref-fr-rules*] = *PAC-checker-l-s-impl.refine*

definition *memory-out-msg* :: ‹*string*› **where**
‹*memory-out-msg* = ''*memory out*''›

lemma [*sepref-fr-rules*]: ‹(*uncurry0* (*return memory-out-msg*), *uncurry0* (*RETURN memory-out-msg*))
∈ *unit-assn*^{k} $\rightarrow_a$ *raw-string-assn*›
⟨*proof*⟩

definition (**in** −) *remap-polys-l2-with-err-s* :: ‹*llist-polynomial* ⇒ *llist-polynomial* ⇒ (*nat*, *llist-polynomial*)
fmap ⇒ (*nat*, *string*) *shared-vars* ⇒
(*string code-status* × (*nat*, *string*) *shared-vars* × (*nat*, *sllist-polynomial*) *fmap* × *sllist-polynomial*)
nres› **where**
‹*remap-polys-l2-with-err-s spec spec0 A* (𝒱 :: (*nat*, *string*) *shared-vars*) = *do*{
ASSERT(*vars-llist spec* ⊆ *vars-llist spec0*);
n ← *upper-bound-on-dom A*;
(*mem*, 𝒱) ← *import-variablesS* (*vars-llist-s2 spec0*) 𝒱;
(*mem'*, *spec*, 𝒱) ← *if* ¬*alloc-failed mem then import-polyS* 𝒱 *spec else RETURN* (*mem*, [], 𝒱);
failed ← *RETURN* (*alloc-failed mem* ∨ *alloc-failed mem'* ∨ *n* ≥ *2^64*);
if failed
then do {
c ← *remap-polys-l-dom-err*;
RETURN (*error-msg* (*0*::*nat*) *c*, 𝒱, *fmempty*, [])
}
else do {
(*err*, *A*, 𝒱) ← *nfoldli* ([*0*..<*n*]) (λ(*err*, *A'*, 𝒱). ¬*is-cfailed err*)
(λ*i* (*err*, *A'* :: (*nat*, *sllist-polynomial*) *fmap*, 𝒱 :: (*nat*,*string*) *shared-vars*).
if i ∈# *dom-m A*
then do {
(*err'*, *p*, 𝒱 :: (*nat*,*string*) *shared-vars*) ← *import-polyS* (𝒱 :: (*nat*,*string*) *shared-vars*) (*the*
(*fmlookup A i*));
if alloc-failed err' then RETURN((*CFAILED* ''*memory out*'', *A'*, 𝒱 :: (*nat*,*string*) *shared-vars*))
else do {
p ← *full-normalize-poly-s* 𝒱 *p*;
eq ← *weak-equality-l-s p spec*;
RETURN((*if eq then CFOUND else CSUCCESS*), *fmupd i p A'*, 𝒱 :: (*nat*,*string*) *shared-vars*)
}
} *else RETURN* (*err*, *A'*, 𝒱 :: (*nat*,*string*) *shared-vars*))
(*CSUCCESS*, *fmempty* :: (*nat*, *sllist-polynomial*) *fmap*, 𝒱 :: (*nat*,*string*) *shared-vars*);
RETURN (*err*, 𝒱, *A*, *spec*)
}}›

lemma *set-vars-llist-s2* [*simp*]: ‹*set* (*vars-llist-s2 b*) = *vars-llist b*›
⟨*proof*⟩

sepref-register *upper-bound-on-dom import-variablesS vars-llist-s2 memory-out-msg*

sepref-definition *import-variablesS-impl*
is ‹*uncurry import-variablesS*›

:: ‹(*list-assn string-assn*)^k $*_a$ *shared-vars-assn*^{d} $\rightarrow_a$ *memory-allocation-assn* $\times_a$ *shared-vars-assn*›
⟨*proof*⟩

lemmas [*sepref-fr-rules*] =
import-variablesS-impl.refine full-normalize-poly′-impl.refine
lemma [*sepref-fr-rules*]:
‹*CONSTRAINT is-pure R* $\Longrightarrow$ ((*return o CFAILED*), *RETURN o CFAILED*) $\in R^k \rightarrow_a$ *status-assn R*›
⟨*proof*⟩

sepref-definition *remap-polys-l2-with-err-s-impl*
is ‹*uncurry3 remap-polys-l2-with-err-s*›
:: ‹*poly-assn*^{k} $*_a$ *poly-assn*^{k} $*_a$ *polys-assn-input*^{k} $*_a$ *shared-vars-assn*^{d} $\rightarrow_a$
status-assn raw-string-assn $\times_a$ *shared-vars-assn* $\times_a$ *polys-s-assn* $\times_a$ *poly-s-assn*›
⟨*proof*⟩

lemmas [*sepref-fr-rules*] =
remap-polys-l2-with-err-s-impl.refine

definition *full-checker-l-s2*
:: ‹*llist-polynomial* ⇒ (*nat*, *llist-polynomial*) *fmap* ⇒ (-, *string*, *nat*) *pac-step list* ⇒
(*string code-status* × -) *nres*›
where
‹*full-checker-l-s2 spec A st* = *do* {
spec′ ← *full-normalize-poly spec*;
(*b*, $\mathcal{V}$, *A*, *spec′*) ← *remap-polys-l2-with-err-s spec′ spec A* ({#}, *fmempty*, *fmempty*);
if is-cfailed b
then RETURN (*b*, $\mathcal{V}$, *A*)
else do {
PAC-checker-l-s spec′ ($\mathcal{V}$, *A*) *b st*
}
}›

sepref-register *remap-polys-l2-with-err-s full-checker-l-s2 PAC-checker-l-s*

sepref-definition *full-checker-l-s2-impl*
is ‹*uncurry2 full-checker-l-s2*›
:: ‹*poly-assn*^{k} $*_a$ *polys-assn-input*^{k} $*_a$ (*list-assn* (*pac-step-rel-assn* (*uint64-nat-assn*) *poly-assn string-assn*))^k $\rightarrow_a$
status-assn raw-string-assn $\times_a$ *shared-vars-assn* $\times_a$ *polys-s-assn*›
⟨*proof*⟩

7 Correctness theorem

context *poly-embed*
begin

definition *fully-epac-assn* **where**
‹*fully-epac-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*))))›

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 *remap-polys-l2-with-err-s-remap-polys-s-with-err*:
  assumes ‹((*spec*, *a*, *b*, *c*), (*spec'*, *a'*, *c'*, *b'*)) ∈ *Id*›
  shows ‹*remap-polys-l2-with-err-s spec a b c*
    ≤ ⇓ *Id*
  (*remap-polys-s-with-err spec' a' b' c'*)›
⟨*proof*⟩

lemma *full-checker-l-s2-full-checker-l-s*:
  ‹(*uncurry2 full-checker-l-s2*, *uncurry2 full-checker-l-s*) ∈ (*Id* $\times_r$ *Id*) $\times_r$ *Id* $\to_f$ ⟨*Id*⟩*nres-rel*›
⟨*proof*⟩

lemma *full-poly-input-assn-alt-def*:
  ‹*full-poly-input-assn* = (*hr-comp*
  (*hr-comp* (*hr-comp polys-assn-input* (⟨*nat-rel*, *Id*⟩*fmap-rel*))
  (⟨*nat-rel*, *fully-unsorted-poly-rel O mset-poly-rel*⟩*fmap-rel*))
  polys-rel)›
⟨*proof*⟩

lemma *PAC-full-correctness*:
  ‹(*uncurry2 full-checker-l-s2-impl*,
 uncurry2 (λ*spec A* -. *PAC-checker-specification spec A*))
∈ *full-poly-assn*^{k} $*_a$ *full-poly-input-assn*^{k} $*_a$
  fully-epac-assn^k $\to_a$ *hr-comp* (*status-assn raw-string-assn* $\times_a$ *shared-vars-assn* $\times_a$ *polys-s-assn*)
          {((*err*, -), *err'*, -). (*err*, *err'*) ∈ *code-status-status-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
end

theory *EPAC-Checker-MLton*
  imports *EPAC-Checker-Synthesis*
begin

export-code *PAC-checker-l-impl PAC-update-impl PAC-empty-impl the-error is-cfailed is-cfound*
  int-of-integer Del CL nat-of-integer String.implode remap-polys-l-impl
  fully-normalize-poly-impl union-vars-poly-impl empty-vars-impl
  full-checker-l-impl check-step-impl CSUCCESS
  Extension hashcode-literal′ version
  in *SML-imp* **module-name** *PAC-Checker*
  file-prefix *checker*

Here is how to compile it:

```
compile-generated-files -
  external-files
    ‹code/no-sharing/parser.sml›
    ‹code/no-sharing/pasteque.sml›
    ‹code/no-sharing/pasteque.mlb›
  where ‹fn dir =>
  let
    val exec = Generated-Files.execute (Path.append dir (Path.basic code));
     val - = exec ‹Copy files›
    (cp checker.ML ^((File.bash-path path ‹$ISAFOL›) ^/PAC-Checker2/code/no-sharing/checker.ML));
     val - = exec ‹Copy files›
      (cp no-sharing/∗ .);
    val - = exec ‹Copy files›
      (ls .) |> @{print};
    val - =
        exec ‹Compilation›
          (File.bash-path path ‹$ISABELLE-MLTON› ^ ^
            −const 'MLton.safe false' −verbose 1 −default−type int64 −output pasteque ^
            −codegen native −inline 700 −cc−opt −O3 pasteque.mlb);
    in () end›
```

end

theory *EPAC-Efficient-Checker-MLton*
  imports *EPAC-Efficient-Checker-Synthesis*
begin
local-setup ‹

```
  let
    val version =
      trim-line (#1 (Isabelle-System.bash-output (cd $ISAFOL/ && git rev−parse −−short HEAD ||
echo unknown)))
  in
    Local-Theory.define
      ((binding ‹version›, NoSyn),
        ((binding ‹version-def›, []), HOLogic.mk-literal version)) #> #2
  end
›
```

declare *version-def* [*code*]

definition *uint32-of-uint64* :: ‹*uint64* ⇒ *uint32*› **where**
‹*uint32-of-uint64 n* = *uint32-of-nat* (*nat-of-uint64 n*)›

lemma [*code*]: ‹*hashcode n* = *uint32-of-uint64* (*n AND 4294967295*)› **for** *n* :: *uint64*
⟨*proof*⟩

code-printing code-module *Uint64* ⇀ (*SML*) ‹(∗ *Test that words can handle numbers between 0 and 63* ∗)

```
val - = if 6 <= Word.wordSize then () else raise (Fail (wordSize less than 6));

structure Uint64 : sig
  eqtype uint64;
  val zero : uint64;
  val one : uint64;
  val fromInt : IntInf.int −> uint64;
  val toInt : uint64 −> IntInf.int;
  val toFixedInt : uint64 −> Int.int;
  val toLarge : uint64 −> LargeWord.word;
  val fromLarge : LargeWord.word −> uint64
  val fromFixedInt : Int.int −> uint64
  val toWord32: uint64 −> Word32.word
  val plus : uint64 −> uint64 −> uint64;
  val minus : uint64 −> uint64 −> uint64;
  val times : uint64 −> uint64 −> uint64;
  val divide : uint64 −> uint64 −> uint64;
  val modulus : uint64 −> uint64 −> uint64;
  val negate : uint64 −> uint64;
  val less-eq : uint64 −> uint64 −> bool;
  val less : uint64 −> uint64 −> bool;
  val notb : uint64 −> uint64;
  val andb : uint64 −> uint64 −> uint64;
  val orb : uint64 −> uint64 −> uint64;
  val xorb : uint64 −> uint64 −> uint64;
  val shiftl : uint64 −> IntInf.int −> uint64;
  val shiftr : uint64 −> IntInf.int −> uint64;
  val shiftr-signed : uint64 −> IntInf.int −> uint64;
  val set-bit : uint64 −> IntInf.int −> bool −> uint64;
  val test-bit : uint64 −> IntInf.int −> bool;
end = struct

type uint64 = Word64.word;
```

```
val zero = (0wx0 : uint64);

val one = (0wx1 : uint64);

fun fromInt x = Word64.fromLargeInt (IntInf.toLarge x);

fun toInt x = IntInf.fromLarge (Word64.toLargeInt x);

fun toFixedInt x = Word64.toInt x;

fun fromLarge x = Word64.fromLarge x;

fun fromFixedInt x = Word64.fromInt x;

fun toLarge x = Word64.toLarge x;

fun toWord32 x = Word32.fromLarge x

fun plus x y = Word64.+(x, y);

fun minus x y = Word64.-(x, y);

fun negate x = Word64.~(x);

fun times x y = Word64.*(x, y);

fun divide x y = Word64.div(x, y);

fun modulus x y = Word64.mod(x, y);

fun less-eq x y = Word64.<=(x, y);

fun less x y = Word64.<(x, y);

fun set-bit x n b =
  let val mask = Word64.<< (0wx1, Word.fromLargeInt (IntInf.toLarge n))
  in if b then Word64.orb (x, mask)
     else Word64.andb (x, Word64.notb mask)
  end

fun shiftl x n =
  Word64.<< (x, Word.fromLargeInt (IntInf.toLarge n))

fun shiftr x n =
  Word64.>> (x, Word.fromLargeInt (IntInf.toLarge n))

fun shiftr-signed x n =
  Word64.~>> (x, Word.fromLargeInt (IntInf.toLarge n))

fun test-bit x n =
  Word64.andb (x, Word64.<< (0wx1, Word.fromLargeInt (IntInf.toLarge n))) <> Word64.fromInt 0

val notb = Word64.notb
```

fun andb x y = Word64.andb(x, y);

fun orb x y = Word64.orb(x, y);

fun xorb x y = Word64.xorb(x, y);

end (∗struct Uint64∗)
›

code-printing constant *arl-get-u′* ⇀ (*SML*) (*fn/* ()/ =>/ *Array.sub/* ((*fn/* (*a*,*b*)/ =>/ *a*) ((-)),/ *Word64.toInt* (*Uint64.toLarge* ((-)))))

definition *uint32-of-uint64′* **where**
[*symmetric*, *code*]: *uint32-of-uint64′* = *uint32-of-uint64*
code-printing constant *uint32-of-uint64′* ⇀ (*SML*) *Uint64.toWord32* ((-))
thm *hashcode-literal-def*[*unfolded hashcode-list-def*]

definition *string-nth* **where**
‹*string-nth s x* = *literal.explode s* ! *x*›

definition *string-nth′* **where**
‹*string-nth′ s x* = *literal.explode s* ! *nat x*›

lemma [*code*]: ‹*string-nth s x* = *string-nth′ s* (*int x*)›
⟨*proof*⟩

definition *string-size* :: ‹*String.literal*⇒*nat*› **where**
‹*string-size s* = *size s*›

definition *string-size′* **where**
[*symmetric*,*code*]: ‹*string-size′* = *string-size*›

lemma [*code*]: ‹*size* = *string-size*›
⟨*proof*⟩

code-printing constant *string-nth′* ⇀ (*SML*) (*String.sub/* ((-),/ *IntInf.toInt* ((*integer′-of′-int* ((-))))))
code-printing constant *string-size′* ⇀ (*SML*) *nat′-of′-integer* ((*IntInf.fromInt* ((*String.size* ((-))))))

function *hashcode-eff* **where**
[*simp del*]: ‹*hashcode-eff s h i* = (*if i* ≥ *size s then h else hashcode-eff s* (*h* ∗ *33* + *hashcode* (*s* ! *i*)) (*i*+*1*))›
⟨*proof*⟩
termination
⟨*proof*⟩

definition *hashcode-eff′* **where**
‹*hashcode-eff′ s h i* = *hashcode-eff* (*String.explode s*) *h i*›

lemma *hashcode-eff′-code*[*code*]:
‹*hashcode-eff′ s h i* = (*if i* ≥ *size s then h else hashcode-eff′ s* (*h* ∗ *33* + *hashcode* (*string-nth s i*)) (*i*+*1*))›
⟨*proof*⟩

lemma [*simp*]: ‹*length s* ≤ *i* =⇒ *hashcode-eff s h i* = *h*›
⟨*proof*⟩

lemma [*simp*]: ‹*hashcode-eff* (*a* # *s*) *h* (*Suc i*) = *hashcode-eff* (*s*) *h* (*i*)›
⟨*proof*⟩

lemma *hashcode-eff-def*[*unfolded hashcode-eff'-def*[*symmetric*], *code*]:
‹*hashcode s* = *hashcode-eff* (*String.explode s*)*5381 0*› **for** *s*::*String.literal*
⟨*proof*⟩

export-code *hashcode* :: *String.literal* ⇒ -
in *SML-imp* **module-name** *PAC-Checker*

code-printing code-module *array-blit* ⇀ (*SML*)
‹
```
fun array-blit src si dst di len = (
  src=dst andalso raise Fail (array-blit: Same arrays);
  ArraySlice.copy {
    di = IntInf.toInt di,
    src = ArraySlice.slice (src,IntInf.toInt si,SOME (IntInf.toInt len)),
    dst = dst})

fun array-nth-oo v a i () = if IntInf.toInt i >= Array.length a then v
  else Array.sub(a,IntInf.toInt i) handle Overflow => v
fun array-upd-oo f i x a () =
  if IntInf.toInt i >= Array.length a then f ()
  else
    (Array.update(a,IntInf.toInt i,x); a) handle Overflow => f ()
```
›

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*

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*⟩

export-code
full-checker-l-s2-impl int-of-integer Del CL nat-of-integer String.implode remap-polys-l2-with-err-s-impl
PAC-update-impl PAC-empty-impl the-error is-cfailed is-cfound
fully-normalize-poly-impl empty-shared-vars-int-impl
PAC-checker-l-s-impl PAC-checker-l-step-s-impl version
in *SML-imp* **module-name** *PAC-Checker*
file-prefix *checker*

compile-generated-files -
external-files
‹*code/parser.sml*›
‹*code/pasteque.sml*›
‹*code/pasteque.mlb*›
where ‹*fn dir* =>
let

val exec = *Generated-Files.execute* (*Path.append dir* (*Path.basic code*));
val - = *exec* ‹*Copy files*›
(*cp checker.ML* ^((*File.bash-path* ***path***‹$*ISAFOL*›) ^/*PAC-Checker2*/*code*/*checker.ML*));
val - =
exec ‹*Compilation*›
(*File.bash-path* ***path***‹$*ISABELLE-MLTON*› ^ ^
−*const* ′*MLton.safe false*′ −*verbose 1* −*default*−*type int64* −*output pasteque* ^
−*codegen native* −*inline 700* −*cc*−*opt* −*O3 pasteque.mlb*);
in () *end*›

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.