Fast Chinese Remaindering via Product Trees

Manuel Eberl

September 18, 2026

Abstract

Product trees are a simple data structure that, broadly speaking, allows dealing with the product of a large number of (typically coprime) integers, or more generally elements of a Euclidean ring.

This is particularly useful for efficient simultaneous modular reduction, i.e. computing $x \bmod m_i$ for a large number of moduli m_i at the same time for a fixed x, and the inverse operation thereof, i.e. modular reconstruction (also known as "Chinese Remaindering").

The algorithms formalised are adapted from the book "Modern Computer Algebra" by von zur Gathen and Gerhard [1, Chapter 10].

Contents

1 Product and remainder trees

theory *CRT_Product_Tree*
  imports *"HOL-Number_Theory.Number_Theory"*
begin

1.1 Prerequisite: A "pairwise" operator for multisets

The following is a multiset analogue of the *pairwise* operator for sets.

definition *pairwise_mset :: "('a ⇒ 'a ⇒ bool) ⇒ 'a multiset ⇒ bool"*
where
  "pairwise_mset P X ⟷ (∀x∈#X. ∀y∈#X-{#x#}. P x y)"

lemma *pairwise_msetI: "(⋀x y. x ∈# X ⟹ y ∈# X - {#x#} ⟹ P x y)*
⟹ pairwise_mset P X"
  by *(auto simp: pairwise_mset_def)*

lemma *pairwise_msetI': "(⋀x y. {#x, y#} ⊆# X ⟹ P x y) ⟹ pairwise_mset*
P X"
  by *(rule pairwise_msetI) (auto simp: insert_subset_eq_iff)*

lemma *pairwise_msetD: "pairwise_mset P X ⟹ x ∈# X ⟹ y ∈# X - {#x#}*
⟹ P x y"
  by *(auto simp: pairwise_mset_def)*

lemma *pairwise_msetD': "pairwise_mset P X ⟹ {#x, y#} ⊆# X ⟹ P x*
y"
  by *(erule pairwise_msetD) (auto simp: insert_subset_eq_iff)*

lemma *pairwise_mset_empty [simp, intro]: "pairwise_mset P {#}"*
  by *(auto simp: pairwise_mset_def)*

lemma *pairwise_mset_singleton [simp, intro]: "pairwise_mset P {#x#}"*
  by *(auto simp: pairwise_mset_def)*

lemma *pairwise_mset_doubleton_iff [simp]: "pairwise_mset P {#x, y#} ⟷*
P x y ∧ P y x"
  by *(auto simp: pairwise_mset_def)*

lemma *pairwise_mset_add_mset:*
  "pairwise_mset P (add_mset x X) ⟷ pairwise_mset P X ∧ (∀y∈#X. P
x y ∧ P y x)"
  by *(auto simp: pairwise_mset_def)*

lemma *pairwise_mset_plus:*
  "pairwise_mset P (X + Y) ⟷ pairwise_mset P X ∧ pairwise_mset P Y
∧ (∀x∈#X. ∀y∈#Y. P x y ∧ P y x)"
  by *(induction Y) (auto simp: pairwise_mset_add_mset)*

```
lemma pairwise_mset_mono:
  "pairwise_mset P X ⟹ (⋀x y. x ∈# Y ⟹ y ∈# Y - {#x#} ⟹ P x y
⟹ Q x y) ⟹ Y ⊆# X ⟹ pairwise_mset Q Y"
  unfolding pairwise_mset_def by (metis insert_DiffM insert_subset_eq_iff)

lemma sorted_wrt_sym_conv_pairwise_mset:
  assumes "⋀x y. x ∈ set xs ⟹ y ∈ set xs ⟹ P x y ⟷ P y x"
  shows   "sorted_wrt P xs ⟷ pairwise_mset P (mset xs)"
  using assms by (induction xs) (auto simp: pairwise_mset_add_mset)
```

1.2 Definition of product trees

A product tree in the classical sense is a binary tree that has a non-zero number attached to each leaf and each internal node has the product of its two immediate child nodes attached to it.

We use a slightly more general structure which can carry additional information at the leaves as well. We also introduce a relation between a product tree and the list of its leaves traversed left-to-right.

```
datatype ('a, 'b) prodtree = Leaf 'a 'b | Node "('a, 'b) prodtree" 'b
"('a, 'b) prodtree"

fun rootval_prodtree :: "('a, 'b) prodtree ⇒ 'b" where
  "rootval_prodtree (Leaf _ m) = m"
| "rootval_prodtree (Node _ m _) = m"

fun flat_prodtree :: "('a, 'b) prodtree ⇒ ('a × 'b) list"  where
  "flat_prodtree (Leaf x m) = [(x, m)]"
| "flat_prodtree (Node l _ r) = flat_prodtree l @ flat_prodtree r"


context
  assumes "SORT_CONSTRAINT('b :: semiring_gcd)"
begin

fun wf_prodtree :: "('a, 'b) prodtree ⇒ bool"  where
  "wf_prodtree (Leaf x m) ⟷ m ≠ 0"
| "wf_prodtree (Node l m r) ⟷
     m = rootval_prodtree l * rootval_prodtree r ∧
     wf_prodtree l ∧ wf_prodtree r"
```

We introduce a relator between a product tree and its flattening as a list.

```
definition rel_prodtree' :: "('a, 'b) prodtree ⇒ ('a × 'b) list ⇒ bool"
where
  "rel_prodtree' t xs ⟷ xs = flat_prodtree t ∧ wf_prodtree t"

lemma rel_prodtree'I [intro?]: "xs = flat_prodtree t ⟹ wf_prodtree
t ⟹ rel_prodtree' t xs"
```

```
  by (auto simp: rel_prodtree'_def)

lemma rel_prodtree_Leaf [simp]: "rel_prodtree' (Leaf x m) xs ⟷ xs
= [(x, m)] ∧ m ≠ 0"
  by (auto simp: rel_prodtree'_def)

lemma flat_prodtree_neq_Nil [simp]: "flat_prodtree t ≠ Nil"
  by (induction t) auto

lemma Nil_neq_flat_prodtree [simp]: "Nil ≠ flat_prodtree t"
  by (induction t) auto

lemma not_rel_prodtree'_Nil [simp]: "¬rel_prodtree' t []"
  by (auto simp: rel_prodtree'_def)

lemma rel_prodtree'_NodeI:
  assumes "rel_prodtree' l xs" "rel_prodtree' r ys" "m = rootval_prodtree
l * rootval_prodtree r"
  shows   "rel_prodtree' (Node l m r) (xs @ ys)"
  using assms by (auto simp: rel_prodtree'_def)

lemma rel_prodtree'_NodeE:
  assumes "rel_prodtree' (Node l m r) zs"
  obtains xs ys where
    "zs = xs @ ys" "rel_prodtree' l xs" "rel_prodtree' r ys"
    "m = rootval_prodtree l * rootval_prodtree r"
    using assms by (auto simp: rel_prodtree'_def)

lemma rel_prodtree'_flat:
  assumes "rel_prodtree' t xs"
  shows   "flat_prodtree t = xs"
  using assms by (simp add: rel_prodtree'_def)

lemma set_flat_prodtree_dvd: "wf_prodtree t ⟹ (x, y) ∈ set (flat_prodtree
t) ⟹ y dvd rootval_prodtree t"
  by (induction t; fastforce simp: image_def)

lemma wf_prodtree_imp_nonzero: "wf_prodtree t ⟹ (x, y) ∈ set (flat_prodtree
t) ⟹ y ≠ 0"
  by (induction t) auto

lemma prod_list_flat_prodtree:
  assumes "wf_prodtree t"
  shows   "prod_list (map snd (flat_prodtree t)) = rootval_prodtree t"
  using assms by (induction t) auto

lemma rel_prodtree'_rootval_prodtree:
  assumes "rel_prodtree' t xs"
  shows   "rootval_prodtree t = prod_list (map snd xs)"
```

```
  using assms unfolding rel_prodtree'_def by (simp add: prod_list_flat_prodtree)
```

1.3 Product trees with coprime branches

In some contexts one requires an additional invariant in the product tree, namely that the numbers attached to two disjoint branches are all coprime.

```
primrec coprime_prodtree :: "('b, 'c :: algebraic_semidom) prodtree ⇒
bool" where
  "coprime_prodtree (Leaf x m) ⟷ True"
| "coprime_prodtree (Node l m r) ⟷
    coprime (rootval_prodtree l) (rootval_prodtree r) ∧ coprime_prodtree
l ∧ coprime_prodtree r"

lemma coprime_prodtree_flat:
  assumes "coprime_prodtree t" "wf_prodtree t"
  shows   "sorted_wrt coprime (map snd (flat_prodtree t))"
  using assms
proof (induction t)
  case (Node l m r)
  thus ?case
    by (auto simp: sorted_wrt_append coprime_divisors dest!: set_flat_prodtree_dvd[rotated]
qed auto

lemma coprime_prodtree_altdef:
  assumes "wf_prodtree t"
  shows   "coprime_prodtree t ⟷ sorted_wrt coprime (map snd (flat_prodtree
t))"
proof
  assume "sorted_wrt coprime (map snd (flat_prodtree t))"
  thus "coprime_prodtree t"
    using assms
  proof (induction t)
    case (Node l m r)
    have "coprime (prod_list (map snd (flat_prodtree l))) (prod_list (map
snd (flat_prodtree r)))"
      by (intro prod_list_coprime_left prod_list_coprime_right) (use Node
in <auto simp: sorted_wrt_append>)
    hence "coprime (rootval_prodtree l) (rootval_prodtree r)"
      by (subst (asm) (1 2) prod_list_flat_prodtree) (use Node.prems in
auto)
    thus ?case
      using Node by (auto simp: sorted_wrt_append)
  qed auto
qed (use coprime_prodtree_flat[OF _ assms] in auto)

lemma coprime_prodtree_rel_prodtree':
  assumes "rel_prodtree' t xs"
  shows   "coprime_prodtree t ⟷ sorted_wrt coprime (map snd (flat_prodtree
t))"
```

using *assms* **by** *(simp add: rel_prodtree'_def coprime_prodtree_altdef)*

1.4 Mapping

We introduce a map operator that can modify both the number attached to the nodes and the extra information at the leaves.

primrec *map_prodtree' :: "('a ⇒ 'c) ⇒ ('d ⇒ 'a ⇒ 'e) ⇒ ('d, 'a) prodtree ⇒ ('e, 'c) prodtree"* **where**
"map_prodtree' f g (Leaf x m) = Leaf (g x m) (f m)"
| "map_prodtree' f g (Node l m r) = Node (map_prodtree' f g l) (f m) (map_prodtree' f g r)"

lemma *map_prodtree'_cong:*
assumes *"wf_prodtree t"*
assumes *"⋀m. m dvd rootval_prodtree t ⟹ f m = f' m"*
assumes *"⋀m x. m dvd rootval_prodtree t ⟹ x ∈ set1_prodtree t ⟹ g x m = g' x m" "t = t'"*
shows *"map_prodtree' f g t = map_prodtree' f' g' t'"*
unfolding *assms(4)[symmetric]* **using** *assms(1-3)* **by** *(induction t) auto*

lemma *flat_map_prodtree' [simp]:*
"flat_prodtree (map_prodtree' f g t) = map (λ(x,m). (g x m, f m)) (flat_prodtree t)"
by *(induction t) auto*

lemma *rootval_map_prodtree' [simp]:*
"rootval_prodtree (map_prodtree' f g t) = f (rootval_prodtree t)"
by *(cases t) auto*

lemma *wf_map_prodtree':*
assumes *"wf_prodtree t" "⋀x. f x = 0 ⟹ x = 0" "⋀x y. f (x * y) = f x * f y"*
shows *"wf_prodtree (map_prodtree' f g t)"*
using *assms(1)* **by** *(induction t) (auto simp: assms(2-))*

lemma *rel_prodtree'_map:*
assumes *"rel_prodtree' t xs"*
assumes *"⋀x. f x = 0 ⟹ x = 0" "⋀x y. f (x * y) = f x * f y"*
shows *"rel_prodtree' (map_prodtree' f g t) (map (λ(x,m). (g x m, f m)) xs)"*
using *assms* **unfolding** *rel_prodtree'_def* **by** *(auto intro!: wf_map_prodtree')*

1.5 Product trees in the classical sense

The operations below turn a list of numbers (possibly tagged with some additional information) into the corresponding product tree.

fun *prodtree_aux1 :: "('a, 'b) prodtree list ⇒ ('a, 'b) prodtree list"* **where**

```
  "prodtree_aux1 [] = []"
| "prodtree_aux1 [x] = [x]"
| "prodtree_aux1 (x # y # xs) = Node x (rootval_prodtree x * rootval_prodtree
y) y # prodtree_aux1 xs"

lemma length_prodtree_aux [simp]: "length (prodtree_aux1 xs) = (length
xs + 1) div 2"
  by (induction xs rule: prodtree_aux1.induct) auto

fun prodtree_aux2 :: "('a, 'b) prodtree list ⇒ ('a, 'b) prodtree" where
  "prodtree_aux2 [] = undefined"
| "prodtree_aux2 [t] = t"
| "prodtree_aux2 ts = prodtree_aux2 (prodtree_aux1 ts)"

definition prodtree' :: "('a × 'b) list ⇒ ('a, 'b) prodtree" where
  "prodtree' xs = prodtree_aux2 (map (λ(x,m). Leaf x m) xs)"

definition prodtree :: "'b list ⇒ (unit, 'b) prodtree" where
  "prodtree xs = prodtree_aux2 (map (λm. Leaf () m) xs)"

lemma rootval_prodtree' [simp]:
  assumes [simp]: "xs ≠ []"
  shows   "rootval_prodtree (prodtree' xs) = prod_list (map snd xs)"
proof -
  have [simp]: "prod_list (map rootval_prodtree (prodtree_aux1 ts)) =
prod_list (map rootval_prodtree ts)"
    for ts :: "('a, 'b) prodtree list"
    by (induction ts rule: prodtree_aux1.induct) (auto simp: mult_ac)
  have *: "rootval_prodtree (prodtree_aux2 ts) = prod_list (map rootval_prodtree
ts)" if "ts ≠ []"
    for ts :: "('a, 'b) prodtree list"
    using that by (induction ts rule: prodtree_aux2.induct) (auto simp:
mult_ac)
  show ?thesis
    unfolding prodtree'_def by (subst *) (auto simp: o_def case_prod_unfold)
qed

lemma rel_prodtree'_prodtree':
  assumes "xs ≠ []" "0 ∉ snd ` set xs"
  shows   "rel_prodtree' (prodtree' xs) xs"
proof
  have 1: "concat (map flat_prodtree (prodtree_aux1 ts)) = concat (map
flat_prodtree ts)"
    for ts :: "('a, 'b) prodtree list"
    by (induction ts rule: prodtree_aux1.induct) auto
  have 2: "flat_prodtree (prodtree_aux2 ts) = concat (map flat_prodtree
ts)" if "ts ≠ []"
    for ts :: "('a, 'b) prodtree list"
    using that by (induction ts rule: prodtree_aux2.induct) (auto simp:
```

```
1)
  show "xs = flat_prodtree (prodtree' xs)"
    unfolding prodtree'_def using assms(1) by (subst 2) (auto simp: o_def
case_prod_unfold)
next
  define P :: "('a, 'b) prodtree list ⇒ bool"
    where "P = (λts. list_all wf_prodtree ts ∧ 0 ∉ rootval_prodtree ` set ts)"

  have *: "rootval_prodtree ` set (prodtree_aux1 ts) ⊆ rootval_prodtree
` set ts ∪ (λ(m1,m2). rootval_prodtree m1 * rootval_prodtree m2) ` (set
ts × set ts)"
    for ts :: "('a, 'b) prodtree list"
    by (induction ts rule: prodtree_aux1.induct) auto

  have 1: "P (prodtree_aux1 ts)" if "P ts" for ts
    using that
  proof (induction ts rule: prodtree_aux1.induct)
    case (3 t1 t2 ts)
    show ?case
      unfolding P_def
    proof (intro conjI)
      show "list_all wf_prodtree (prodtree_aux1 (t1 # t2 # ts))"
        using 3 by (simp_all add: P_def pairwise_def)
    next
      have "rootval_prodtree ` set (prodtree_aux1 (t1 # t2 # ts)) ⊆
              insert (rootval_prodtree t1 * rootval_prodtree t2)
                (rootval_prodtree ` set ts ∪ (λ(m1,m2). rootval_prodtree
m1 * rootval_prodtree m2) ` (set ts × set ts))"
        using *[of ts] by auto
      also have "... ⊆ -{0}"
        using 3 by (auto simp: P_def)
      finally show "0 ∉ rootval_prodtree ` set (prodtree_aux1 (t1 # t2
# ts))"
        by auto
    qed
  qed (auto simp: P_def)

  have 2: "wf_prodtree (prodtree_aux2 ts)" if "P ts" "ts ≠ []" for ts
    using that
  proof (induction ts rule: prodtree_aux2.induct)
    case (3 t1 t2 ts)
    show ?case
      unfolding prodtree_aux2.simps by (intro "3.IH" 1) (use "3.prems"
in auto)
  qed (auto simp: P_def)

  show "wf_prodtree (prodtree' xs)"
    unfolding prodtree'_def
```

```
    by (intro 2) (use assms in ‹auto simp: P_def o_def case_prod_unfold
list.pred_set image_def›)
qed

end
```

1.6 Remainder trees

Using a product tree of non-zero numbers $(m_1, \ldots, m_k)$, one can efficiently compute the remainders $(n \bmod m_1, \ldots, n \bmod m_k)$ simultaneously.

```
context
  assumes "SORT_CONSTRAINT('a :: {euclidean_semiring_cancel, euclidean_semiring_gcd})"
begin

fun mod_prodtree :: "'a ⇒ ('b, 'a) prodtree ⇒ ('b × 'a, 'a) prodtree"
where
  "mod_prodtree n (Leaf x m) = Leaf (x, n mod m) m"
| "mod_prodtree n (Node l m r) = (
     let n' = n mod m
     in  Node (mod_prodtree n' l) m (mod_prodtree n' r))"

lemma mod_prodtree_altdef:
  assumes "wf_prodtree t"
  shows   "mod_prodtree n t = map_prodtree' id (λx m. (x, n mod m)) t"
  using assms
  by (induction t rule: mod_prodtree.induct)
     (auto simp: Let_def mod_mod_cancel intro!: map_prodtree'_cong)

lemma rel_prodtree'_mod:
  assumes "rel_prodtree' t xs"
  shows   "rel_prodtree' (mod_prodtree n t) (map (λ(x,m). ((x, n mod
m), m)) xs)"
proof -
  have wf: "wf_prodtree t"
    using assms by (auto simp: rel_prodtree'_def)
  have "rel_prodtree' (mod_prodtree n t) (map (λ(x, m). ((x, n mod m),
id m)) xs)"
    unfolding mod_prodtree_altdef[OF wf] by (rule rel_prodtree'_map) (use
assms in auto)
  thus ?thesis
    by simp
qed

end
```

1.7 Linear combinations

Given a product tree of the pairwise non-zero numbers $(m_1, \ldots, m_k)$ and a sequence of numbers $(a_1, \ldots, a_k)$ attached to them, one can efficiently compute the linear combination $\sum_{i=1}^{k} a_i M/m_i$, where $M = m_1 \cdot \ldots \cdot m_k$.

```
context
  assumes "SORT_CONSTRAINT('b :: semiring_gcd)"
begin

fun comb_prodtree :: "('b, 'b) prodtree ⇒ 'b" where
  "comb_prodtree (Leaf x m) = x"
| "comb_prodtree (Node l m r) =
     comb_prodtree l * rootval_prodtree r + comb_prodtree r * rootval_prodtree
l"

lemma rel_prodtree'_comb:
  assumes "rel_prodtree' t xs"
  defines "M ≡ prod_list (map snd xs)"
  shows   "comb_prodtree t = (∑ (x, m)←xs. x * (M div m))"
proof -
  have "wf_prodtree t"
    using assms by (simp add: rel_prodtree'_def)
  hence "comb_prodtree t = (∑ (x, m)←flat_prodtree t. x * (rootval_prodtree
t div m))"
    unfolding M_def
  proof (induction t)
    case (Node l m r)
    define ls where "ls = flat_prodtree l"
    define rs where "rs = flat_prodtree r"
    define m1 where "m1 = rootval_prodtree l"
    define m2 where "m2 = rootval_prodtree r"
    have "(∑ (x, a)←flat_prodtree (Node l m r). x * (rootval_prodtree
(Node l m r) div a)) =
          (∑ (x, a)←ls. x * ((m1 * m2) div a)) + (∑ (x, a)←rs. x * ((m1
* m2) div a))"
      using Node.prems by (simp add: ls_def rs_def m1_def m2_def)

    also have "map (λ(x, a). x * ((m1 * m2) div a)) ls = map (λ(x,a).
x * m2 * (m1 div a)) ls"
      using Node.prems
      by (intro map_cong refl)
         (auto simp: ls_def m1_def div_mult_swap[OF set_flat_prodtree_dvd]
mult_ac)
    also have "sum_list ... = (∑ (x, a)←ls. x * (m1 div a)) * m2"
      by (induction ls) (auto simp: algebra_simps)
    also have "... = comb_prodtree l * m2"
      using Node by (simp add: ls_def m1_def m2_def)

    also have "map (λ(x, a). x * ((m1 * m2) div a)) rs = map (λ(x,a).
```

```
x * (m1 * (m2 div a))) rs"
      using Node.prems
      by (intro map_cong refl)
         (auto simp: rs_def m2_def div_mult_swap[OF set_flat_prodtree_dvd]
mult_ac)
    also have "sum_list ... = (∑(x, a)←rs. x * (m2 div a)) * m1"
      by (induction rs) (auto simp: algebra_simps)
    also have "... = comb_prodtree r * m1"
      using Node by (simp add: rs_def m1_def m2_def)

    finally show ?case
      by (simp add: m1_def m2_def mult_ac)
  qed auto

  also have "flat_prodtree t = xs"
    using assms by (simp add: rel_prodtree'_def)
  also have "rootval_prodtree t = M"
    using rel_prodtree'_rootval_prodtree[OF assms(1)] by (simp add: M_def)
  finally show ?thesis .
qed

end
```

1.8 Chinese remaindering

Putting all these ingredients together, we can implement an efficient Chinese remaindering algorithm using product trees: given k pairs of numbers (a_i, m_i) where the m_i are pairwise coprime and non-zero, we compute the unique number $x \in [0, M)$ such that

$$x \bmod m_1 = a_1 \quad \ldots \quad x \bmod m_k = a_k$$

where $M = m_1 \cdot \ldots \cdot m_k$.

In contrast to the presentation in the book, we do some deforestation and fuse as many operations as possible together for a more concise implementation. The proof does not actually become any more complicated.

```
context
  assumes "SORT_CONSTRAINT('a :: {euclidean_semiring_cancel, euclidean_ring_gcd})"
begin

fun crt_prodtree_aux :: "'a ⇒ ('a, 'a) prodtree ⇒ 'a" where
  "crt_prodtree_aux M (Leaf x m) = x * modular_inverse m (M mod (m^2)
div m)"
| "crt_prodtree_aux M (Node l m r) = (let M' = M mod m^2 in
     crt_prodtree_aux M' l * rootval_prodtree r + crt_prodtree_aux M'
r * rootval_prodtree l)"

definition crt_prodtree :: "('a × 'a) list ⇒ 'a × 'a" where
```

```
  "crt_prodtree xs =
     (let t1 = prodtree' xs; M = rootval_prodtree t1 in (crt_prodtree_aux
M t1 mod M, M))"


lemma crt_prodtree_aux_eq:
  fixes M :: 'a
  assumes "wf_prodtree t"
  defines "i ≡ (λM m. modular_inverse m (M mod (m^2) div m))"
  shows   "crt_prodtree_aux M t = (∑ (x,m)←flat_prodtree t. x * i M
m * rootval_prodtree t div m)"
  using assms(1)
proof (induction t arbitrary: M)
  case (Node l m' r)
  define L where "L = rootval_prodtree l"
  define R where "R = rootval_prodtree r"

  have "crt_prodtree_aux M (Node l m' r) =
          (∑ (x,m)←flat_prodtree l. x * modular_inverse m (M mod (L *
R)^2 mod m^2 div m) * L div m * R) +
          (∑ (x,m)←flat_prodtree r. x * modular_inverse m (M mod (L *
R)^2 mod m^2 div m) * R div m * L)"
    using Node by (simp add: i_def Let_def L_def R_def case_prod_unfold
flip: sum_list_mult_const)
  also have "(∑ (x,m)←flat_prodtree l. x * modular_inverse m (M mod (L
* R)^2 mod m^2 div m) * L div m * R) =
             (∑ (x,m)←flat_prodtree l. x * modular_inverse m (M mod (m^2)
div m) * (L*R) div m)"
  proof (intro arg_cong[of _ _ sum_list] map_cong refl, safe)
    fix x m assume xm: "(x, m) ∈ set (flat_prodtree l)"
    have "m dvd L"
      using Node(3) set_flat_prodtree_dvd[OF _ xm] by (force simp: L_def)
    show "x * modular_inverse m (M mod (L * R)^2 mod m^2 div m) * L div
m * R =
          x * modular_inverse m (M mod m^2 div m) * (L*R) div m"
      using ‹m dvd L› by (auto simp: mod_mod_cancel dvd_power_same)
  qed
  also have "(∑ (x,m)←flat_prodtree r. x * modular_inverse m (M mod (L
* R)^2 mod m^2 div m) * R div m * L) =
             (∑ (x,m)←flat_prodtree r. x * modular_inverse m (M mod (m^2)
div m) * (L*R) div m)"
  proof (intro arg_cong[of _ _ sum_list] map_cong refl, safe)
    fix x m assume xm: "(x, m) ∈ set (flat_prodtree r)"
    have "m dvd R"
      using Node(3) set_flat_prodtree_dvd[OF _ xm] by (force simp: R_def)
    show "x * modular_inverse m (M mod (L * R)^2 mod m^2 div m) * R div
m * L =
          x * modular_inverse m (M mod m^2 div m) * (L*R) div m"
      using ‹m dvd R› by (auto simp: mod_mod_cancel dvd_power_same)
```

```
  qed
  finally show ?case using Node.prems
    by (simp add: i_def Let_def case_prod_unfold L_def R_def)
qed (auto simp: i_def)

end


context
  assumes "SORT_CONSTRAINT('a :: {euclidean_semiring_cancel, euclidean_ring_gcd})"
begin

lemma crt_prodtree_aux_cong:
  assumes "wf_prodtree t" and "coprime_prodtree t" and "(y, n) ∈ set
(flat_prodtree t)"
  defines "M ≡ rootval_prodtree t"
  shows   "[crt_prodtree_aux M t = y] (mod n)"
proof -
  define A' where "A' = mset (flat_prodtree t)"
  define A where "A = image_mset snd (mset (flat_prodtree t))"
  have pw_coprime: "pairwise_mset coprime A"
    unfolding A_def using coprime_prodtree_flat[OF assms(2,1)]
    by (simp add: coprime_commute sorted_wrt_sym_conv_pairwise_mset)

  have "crt_prodtree_aux M t = (∑ (x,m)←flat_prodtree t. x * modular_inverse
m (M mod (m^2) div m) * M div m)"
    unfolding M_def by (rule crt_prodtree_aux_eq) fact
  also have "[... = (∑ (x,m)←flat_prodtree t. if m = n then x else 0)]
(mod n)"
  proof (intro cong_sum_list' list.rel_refl_strong, safe)
    fix x m
    assume xm: "(x, m) ∈ set (flat_prodtree t)"
    have "m ∈# A" "n ∈# A"
      using xm assms unfolding A_def by force+

    have "m dvd M" "n dvd M"
      using assms xm by (simp_all add: set_flat_prodtree_dvd)
    show "[x * modular_inverse m (M mod m^2 div m) * M div m = (if m =
n then x else 0)] (mod n)"
    proof (cases "m = n")
      case True
      define z where "z = modular_inverse m (M mod m^2 div m)"
      have cong: "[M div m = M mod m^2 div m] (mod m)"
        by (smt (verit) <m dvd M> div_by_0 dvdE mod_mod_trivial mod_mult_mult1
nonzero_mult_div_cancel_left power2_eq_square unique_euclidean_semiring_class.cong_def)

      have "coprime m (M div m)"
      proof -
        define X where "X = A - {#m#}"
```

```
        have [simp]: "m ≠ 0"
          using xm assms wf_prodtree_imp_nonzero by fast
        have "M = prod_mset A" using assms
          by (simp add: A_def M_def prod_mset_prod_list prod_list_flat_prodtree
flip: mset_map)
        also have "A = {#m#} + X"
          using ‹m ∈# A› unfolding X_def A_def by auto
        also have "prod_mset ... = m * prod_mset X"
          by simp
        finally have "M div m = prod_mset X"
          by simp

        have "coprime m (prod_mset X)"
        proof (rule prod_mset_coprime_right)
          fix x assume x: "x ∈# X"
          show "coprime m x"
            using pw_coprime by (rule pairwise_msetD) (use x ‹m ∈# A›
in ‹auto simp: X_def›)
        qed
        with ‹M div m = prod_mset X› show ?thesis
          by simp
      qed
      hence coprime: "coprime (M mod m^2 div m) m"
        using cong_imp_coprime coprime_commute cong by blast

      have "x * modular_inverse m (M mod m^2 div m) * M div m = x * (z
* (M div m))"
        using ‹m dvd M› by (auto simp: algebra_simps z_def elim!: dvdE)
      also have "[... = x * (z * (M mod m^2 div m))] (mod m)"
        using cong by (intro cong_mult cong_refl)
      also have "[x * (z * (M mod m^2 div m)) = x * 1] (mod m)"
        unfolding z_def using coprime by (intro cong_mult cong_refl cong_modular_inverse2)
      finally show ?thesis
        by (simp add: True)
    next
      case False
      from pw_coprime have "coprime m n"
        by (rule pairwise_msetD') (use False ‹m ∈# A› ‹n ∈# A› in ‹auto
simp: insert_subset_eq_iff›)
      with ‹m dvd M› and ‹n dvd M› have "n dvd (M div m)"
        using assms by (metis coprime_commute coprime_dvd_mult_left_iff
dvd_div_mult_self)
      thus ?thesis
        using False ‹m dvd M› by (auto simp: cong_0_iff elim!: dvdE)
    qed
  qed
  also have "(∑ (x,m)←flat_prodtree t. if m = n then x else 0) =
             (∑ (x,m)←filter (λ(x,m). m = n) (flat_prodtree t). x)"
    by (simp add: sum_list_map_filter' case_prod_unfold)
```

```
  also have "[... = y] (mod n)"
  proof (cases "is_unit n")
    case False
    have "(∑ (x,m)←filter (λ(x,m). m = n) (flat_prodtree t). x) =
          (∑ (x,m)∈#filter_mset (λ(x,m). m = n) A'. x)"
      by (simp flip: sum_mset_sum_list add: A'_def)
    also have "filter_mset (λ(x,m). m = n) A' = {#(y, n)#}"
    proof -
      define X where "X = A' - {#(y, n)#}"
      have *: "A' = {#(y, n)#} + X"
        using assms by (auto simp: X_def A'_def)
      hence "filter_mset (λ(x, m). m = n) A' = {#(y, n)#} + filter_mset
(λ(x, m). m = n) X"
        by simp
      also have "filter_mset (λ(x, m). m = n) X = filter_mset (λ_. False)
X"
      proof (intro filter_mset_cong refl; safe)
        fix x assume xn: "(x, n) ∈# X"
        have "{#(x,n), (y,n)#} ⊆# A'"
          using xn assms by (auto simp: *)
        hence "image_mset snd {#(x,n), (y,n)#} ⊆# A"
          unfolding A'_def A_def by (rule image_mset_subseteq_mono)
        hence "{#n, n#} ⊆# A"
          by simp
        from pw_coprime and this have "coprime n n"
          by (rule pairwise_msetD')
        hence "is_unit n"
          by auto
        thus False
          using ‹¬is_unit n› by contradiction
      qed
      finally show ?thesis
        by simp
    qed
    finally show ?thesis
      by simp
  qed (simp_all add: cong_def unit_imp_mod_eq_0)

  finally show ?thesis
    by simp
qed

theorem snd_crt_prodtree: "xs ≠ [] ⟹ snd (crt_prodtree xs) = prod_list
(map snd xs)"
  by (simp add: crt_prodtree_def Let_def)

theorem cong_fst_crt_prodtree:
  assumes "sorted_wrt coprime (map snd xs)" "0 ∉ snd ` set xs" "xs ≠
```

```
[]" "(x, m) ∈ set xs"
  shows   "[fst (crt_prodtree xs) = x] (mod m)"
proof -
  define t where "t = prodtree' xs"
  have t: "rel_prodtree' t xs"
    unfolding t_def by (rule rel_prodtree'_prodtree') (use assms in auto)

  have "[fst (crt_prodtree xs) = crt_prodtree_aux (rootval_prodtree t)
t] (mod (prod_list (map snd xs)))"
    using ‹xs ≠ []› by (simp add: t_def crt_prodtree_def Let_def)
  moreover have "m dvd prod_mset (mset (map snd xs))"
    using assms by (intro dvd_prod_mset) force+
  ultimately have "[fst (crt_prodtree xs) = crt_prodtree_aux (rootval_prodtree
t) t] (mod m)"
    unfolding prod_mset_prod_list using cong_dvd_mono_modulus by blast
  also have "[crt_prodtree_aux (rootval_prodtree t) t = x] (mod m)"
  proof (rule crt_prodtree_aux_cong)
    show "wf_prodtree t" "(x, m) ∈ set (flat_prodtree t)" "coprime_prodtree
t"
      using t coprime_prodtree_rel_prodtree'[OF t] assms by (auto simp:
rel_prodtree'_def)
  qed
  finally show "[fst (crt_prodtree xs) = x] (mod m)" .
qed

end

end
```

1.9 Examples and Benchmarks

```
theory CRT_Product_Tree_Benchmark
  imports CRT_Product_Tree "HOL-Library.Code_Target_Numeral"
begin
```

As a simple example, consider the following system of congruences:

$$\begin{aligned} x &\equiv 0 \pmod 3 \\ x &\equiv 3 \pmod 4 \\ x &\equiv 4 \pmod 5 \end{aligned}$$

We use our algorithm `crt_prodtree` to find that this is equivalent to the single congruence $x \equiv 39 \pmod{60}$:

```
value "crt_prodtree [(0, 3), (3, 4), (4, 5 :: int)]"
```

Next, we compare the product tree implementation to a more naïve divice-and-conquer implementation that simply computes the CRT by computing the binary CRT of adjacent pairs in a list, thereby halving the list, and doing this iteratively until there is only one pair left.

```
fun slow_crt_aux :: "(int × int) list ⇒ (int × int) list" where
  "slow_crt_aux [] = []"
| "slow_crt_aux [x] = [x]"
| "slow_crt_aux ((x1, m1) # (x2, m2) # xs) =
     (case bezout_coefficients m1 m2 of
       (a1, a2) ⇒ ((x1 * a2) mod m1 * m2 + (x2 * a1) mod m2 * m1, m1
* m2)) # slow_crt_aux xs"

lemma length_slow_crt_aux [simp]: "length (slow_crt_aux xs) = (length
xs + 1) div 2"
  by (induction xs rule: slow_crt_aux.induct) auto

function slow_crt :: "(int × int) list ⇒ int × int" where
  "slow_crt xs =
     (case xs of
        [] ⇒ (0, 0)
      | (x, m) # xs' ⇒ if xs' = [] then (x mod m, m) else slow_crt (slow_crt_aux
xs))"
  by auto
termination by (relation "measure length") auto

definition mk_test_data :: "(integer × integer) list ⇒ (int × int) list"
where
  "mk_test_data = map (map_prod int_of_integer int_of_integer)"
definition "test_crt_prodtree = map_prod integer_of_int integer_of_int
o crt_prodtree"
definition "test_slow_crt = map_prod integer_of_int integer_of_int o slow_crt"
```

We test the implementations on a list of n pairs of the form (i, p_i), where p_i is the i-th prime number. The benchmark shows that, empirically, the product tree implementation takes time $\widetilde{O}(n)$, whereas the naïve implementation takes time $\widetilde{O}(n^2)$.

This agrees with what one would expect from the theory: the naïve implementation performs $\frac{n}{2^i}$ Extended Euclidean algorithm (EEA) computations on $\Theta(2^i)$ bit numbers in the i-th iteration. The standard implementation of the EEA is quadratic, leading to quadratic effort overall. The product tree implementation only performs a constant number of multiplications and divisions (which are significantly faster than quadratic) at every node of the tree, leading to linear behaviour.

It should be noted that using an asymptotically faster EEA implementation would theoretically fix this issue and make the naïve divide-and-conquer algorithm roughly as fast as the product tree one, but these algorithms are very complicated and have large constant factor overheads, making this impractical.

```
ML ‹
(*
  a simple Eratosthenes-style sieve;
```

```
  result array contains smallest prime divisor for composites and 0 for
primes, 0, and 1
*)
fun prime_sieve n =
  let
    val a = Array.array (n+1, 0)
    fun mark i k =
      if i > n then ()
      else let val _ = Array.update (a, i, k) in mark (i + k) k end
    fun go k =
      if k * k > n then ()
      else if Array.sub (a, k) <> 0 then go (k+1)
      else let val _ = mark (k * k) k in go (k+1) end
    val _ = go 2
  in
    a
  end

fun primes_upto n =
  let
    val a = prime_sieve n
    fun go i acc =
      if i > n then rev acc
      else go (i+1) (if Array.sub (a, i) = 0 then i :: acc else acc)
  in
    go 2 []
  end
>
```

ML <

```
local
  exception NOT_ENOUGH_PRIMES
  val max_prime = 500000
  val mk_test_data_aux = @{code mk_test_data}

  val (t_ps, ps) = Timing.timing primes_upto max_prime
  val _ = writeln ( "Sieving primes up to " ^ Int.toString max_prime
     ^ ": " ^ Time.message (#elapsed t_ps) ^ "\n")
  val n_ps = length ps

  fun mk_test_data n =
    if n > n_ps then
      raise NOT_ENOUGH_PRIMES
    else
      take n ps |> map_index I |> mk_test_data_aux

  val test_crt_prodtree = @{code test_crt_prodtree}
  val test_slow_crt = @{code test_slow_crt}
  fun test n =
```

```
    let
      val dat = mk_test_data n
      val _ = writeln ("Testing with " ^ Int.toString n ^ " primes.")
      val (t1, _) = Timing.timing test_crt_prodtree dat
      val _ = writeln ("crt_prodtree: " ^ Time.message (#elapsed t1))
      val (t2, _) = Timing.timing test_slow_crt dat
      val _ = writeln ("slow_crt: " ^ Time.message (#elapsed t2))
      val speedup = Time.toReal (#elapsed t2) / Time.toReal (#elapsed
t1)
      val _ = writeln ("Speedup: " ^ Real.fmt (StringCvt.FIX (SOME 2))
speedup ^ "\n")
    in
      ()
    end
in
  val _ = map test [10000, 14142, 20000, 28284, 40000]
end
>
```

end

References

[1] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 3 edition, 2013.