

Fast Chinese Remaindering via Product Trees

Manuel Eberl

October 6, 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	2
1.1	Prerequisite: A “pairwise” operator for multisets	2
1.2	Definition of product trees	3
1.3	Product trees with coprime branches	5
1.4	Mapping	5
1.5	Product trees in the classical sense	6
1.6	Remainder trees	7
1.7	Linear combinations	7
1.8	Chinese remaindering	8
1.9	Examples and Benchmarks	9

1 Product and remainder trees

```
theory CRT_Product_Tree
  imports "HOL-Number_Theory.Number_Theory"
begin
  <proof><proof><proof><proof><proof><proof><proof>
```

1.1 Prerequisite: A “pairwise” operator for multisets

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

definition *pairwise_mset* :: "('a $\Rightarrow$ 'a $\Rightarrow$ bool) $\Rightarrow$ 'a multiset $\Rightarrow$ bool"

where

"pairwise_mset P X $\longleftrightarrow$ ($\forall x \in \#X. \forall y \in \#X - \{x\}. P\ x\ y$)"

lemma *pairwise_msetI*: " $(\bigwedge x\ y. x \in \#X \implies y \in \#X - \{x\} \implies P\ x\ y) \implies$ pairwise_mset P X"
 <proof>

lemma *pairwise_msetI'*: " $(\bigwedge x\ y. \{x, y\} \subseteq \#X \implies P\ x\ y) \implies$ pairwise_mset P X"
 <proof>

lemma *pairwise_msetD*: "pairwise_mset P X $\implies x \in \#X \implies y \in \#X - \{x\} \implies P\ x\ y$ "
 <proof>

lemma *pairwise_msetD'*: "pairwise_mset P X $\implies \{x, y\} \subseteq \#X \implies P\ x\ y$ "
 <proof>

lemma *pairwise_mset_empty* [simp, intro]: "pairwise_mset P {#}"
 <proof>

lemma *pairwise_mset_singleton* [simp, intro]: "pairwise_mset P {#x#}"
 <proof>

lemma *pairwise_mset_doubleton_iff* [simp]: "pairwise_mset P {#x, y#} $\longleftrightarrow P\ x\ y \wedge P\ y\ x$ "
 <proof>

lemma *pairwise_mset_add_mset*:
 "pairwise_mset P (add_mset x X) $\longleftrightarrow$ pairwise_mset P X $\wedge (\forall y \in \#X. P\ x\ y \wedge P\ y\ x)$ "
 <proof>

lemma *pairwise_mset_plus*:
 "pairwise_mset P (X + Y) $\longleftrightarrow$ pairwise_mset P X $\wedge$ pairwise_mset P Y
 $\wedge (\forall x \in \#X. \forall y \in \#Y. P\ x\ y \wedge P\ y\ x)$ "
 <proof>

```

lemma pairwise_mset_mono:
  "pairwise_mset P X  $\implies$  ( $\bigwedge x y. x \in \# Y \implies y \in \# Y - \{\#x\} \implies P x y$ 
 $\implies Q x y$ )  $\implies Y \subseteq \# X \implies$  pairwise_mset Q Y"
  <proof>

lemma sorted_wrt_sym_conv_pairwise_mset:
  assumes " $\bigwedge x y. x \in \text{set } xs \implies y \in \text{set } xs \implies P x y \longleftrightarrow P y x$ "
  shows "sorted_wrt P xs  $\longleftrightarrow$  pairwise_mset P (mset xs)"
  <proof>

```

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  $\Rightarrow$  'b" where
  "rootval_prodtree (Leaf _ m) = m"
| "rootval_prodtree (Node _ m _) = m"

fun flat_prodtree :: "('a, 'b) prodtree  $\Rightarrow$  ('a  $\times$  '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  $\Rightarrow$  bool" where
  "wf_prodtree (Leaf x m)  $\longleftrightarrow$  m  $\neq$  0"
| "wf_prodtree (Node l m r)  $\longleftrightarrow$ 
  m = rootval_prodtree l * rootval_prodtree r  $\wedge$ 
  wf_prodtree l  $\wedge$  wf_prodtree r"

```

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

```

definition rel_prodtree' :: "('a, 'b) prodtree  $\Rightarrow$  ('a  $\times$  'b) list  $\Rightarrow$  bool"
where
  "rel_prodtree' t xs  $\longleftrightarrow$  xs = flat_prodtree t  $\wedge$  wf_prodtree t"

```

```

lemma rel_prodtree'I [intro?]: "xs = flat_prodtree t  $\implies$  wf_prodtree
t  $\implies$  rel_prodtree' t xs"

```

```

    <proof>

lemma rel_prodtree_Leaf [simp]: "rel_prodtree' (Leaf x m) xs  $\longleftrightarrow$  xs
= [(x, m)]  $\wedge$  m  $\neq$  0"
    <proof>

lemma flat_prodtree_neq_Nil [simp]: "flat_prodtree t  $\neq$  Nil"
    <proof>

lemma Nil_neq_flat_prodtree [simp]: "Nil  $\neq$  flat_prodtree t"
    <proof>

lemma not_rel_prodtree'_Nil [simp]: " $\neg$ rel_prodtree' t []"
    <proof>

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)"
    <proof>

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"
    <proof>

lemma rel_prodtree'_flat:
  assumes "rel_prodtree' t xs"
  shows "flat_prodtree t = xs"
    <proof>

lemma set_flat_prodtree_dvd: "wf_prodtree t  $\implies$  (x, y)  $\in$  set (flat_prodtree
t)  $\implies$  y dvd rootval_prodtree t"
    <proof>

lemma wf_prodtree_imp_nonzero: "wf_prodtree t  $\implies$  (x, y)  $\in$  set (flat_prodtree
t)  $\implies$  y  $\neq$  0"
    <proof>

lemma prod_list_flat_prodtree:
  assumes "wf_prodtree t"
  shows "prod_list (map snd (flat_prodtree t)) = rootval_prodtree t"
    <proof>

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

```

$\langle proof \rangle$

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  $\Rightarrow$  bool" where
  "coprime_prodtree (Leaf x m)  $\longleftrightarrow$  True"
| "coprime_prodtree (Node l m r)  $\longleftrightarrow$ 
  coprime (rootval_prodtree l) (rootval_prodtree r)  $\wedge$  coprime_prodtree l  $\wedge$  coprime_prodtree r"
```

```
lemma coprime_prodtree_flat:
  assumes "coprime_prodtree t" "wf_prodtree t"
  shows "sorted_wrt coprime (map snd (flat_prodtree t))"
 $\langle proof \rangle$ 
```

```
lemma coprime_prodtree_altdef:
  assumes "wf_prodtree t"
  shows "coprime_prodtree t  $\longleftrightarrow$  sorted_wrt coprime (map snd (flat_prodtree t))"
 $\langle proof \rangle$ 
```

```
lemma coprime_prodtree_rel_prodtree':
  assumes "rel_prodtree' t xs"
  shows "coprime_prodtree t  $\longleftrightarrow$  sorted_wrt coprime (map snd (flat_prodtree t))"
 $\langle proof \rangle$ 
```

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  $\Rightarrow$  'c)  $\Rightarrow$  ('d  $\Rightarrow$  'a  $\Rightarrow$  'e)  $\Rightarrow$  ('d, 'a) prodtree  $\Rightarrow$  ('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 " $\bigwedge m. m \text{ dvd } \text{rootval\_prodtree } t \implies f m = f' m$ "
  assumes " $\bigwedge m x. m \text{ dvd } \text{rootval\_prodtree } t \implies x \in \text{set1\_prodtree } t \implies g x m = g' x m$ " " $t = t'$ "
  shows "map_prodtree' f g t = map_prodtree' f' g' t'"
 $\langle proof \rangle$ 
```

```

lemma flat_map_prodtree' [simp]:
  "flat_prodtree (map_prodtree' f g t) = map ( $\lambda(x,m). (g\ x\ m, f\ m)$ ) (flat_prodtree t)"
  <proof>

lemma rootval_map_prodtree' [simp]:
  "rootval_prodtree (map_prodtree' f g t) = f (rootval_prodtree t)"
  <proof>

lemma wf_map_prodtree':
  assumes "wf_prodtree t" " $\bigwedge x. f\ x = 0 \implies x = 0$ " " $\bigwedge x\ y. f\ (x * y) = f\ x * f\ y$ "
  shows "wf_prodtree (map_prodtree' f g t)"
  <proof>

lemma rel_prodtree'_map:
  assumes "rel_prodtree' t xs"
  assumes " $\bigwedge x. f\ x = 0 \implies x = 0$ " " $\bigwedge x\ y. f\ (x * y) = f\ x * f\ y$ "
  shows "rel_prodtree' (map_prodtree' f g t) (map ( $\lambda(x,m). (g\ x\ m, f\ m)$ ) xs)"
  <proof>

```

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  $\Rightarrow$  ('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"
  <proof>

```

```

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

```

```

definition prodtree' :: "('a  $\times$  'b) list  $\Rightarrow$  ('a, 'b) prodtree" where
  "prodtree' xs = prodtree_aux2 (map ( $\lambda(x,m). \text{Leaf } x\ m$ ) xs)"

```

```

definition prodtree :: "'b list  $\Rightarrow$  (unit, 'b) prodtree" where
  "prodtree xs = prodtree_aux2 (map ( $\lambda m. \text{Leaf } ()\ m$ ) xs)"

```

```

lemma rootval_prodtree' [simp]:

```

```

    assumes [simp]: "xs ≠ []"
    shows "rootval_prodtree (prodtree' xs) = prod_list (map snd xs)"
  <proof>

lemma rel_prodtree'_prodtree':
  assumes "xs ≠ []" "0 ∉ snd ` set xs"
  shows "rel_prodtree' (prodtree' xs) xs"
  <proof>

end

```

1.6 Remainder trees

Using a product tree of non-zero numbers $(m_1, \dots, m_k)$, one can efficiently compute the remainders $(n \bmod m_1, \dots, n \bmod m_k)$ 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"
  <proof>

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>

end

```

1.7 Linear combinations

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

```

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

```

```

fun comb_prodtree :: "('b, 'b) prodtree  $\Rightarrow$  '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  $\equiv$  prod_list (map snd xs)"
  shows "comb_prodtree t = ( $\sum (x, m) \leftarrow xs. x * (M \text{ div } m)$ )"
  <proof>

```

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

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

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

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  $\Rightarrow$  ('a, 'a) prodtree  $\Rightarrow$  '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  $\times$  'a) list  $\Rightarrow$  'a  $\times$  '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  $\equiv$  ( $\lambda M m. modular\_inverse m (M \text{ mod } (m^2) \text{ div } m)$ )"

```

```

    shows "crt_prodtree_aux M t = ( $\sum (x,m) \leftarrow \text{flat\_prodtree } t. x * i \text{ M}$ 
    m * rootval_prodtree t div m)"
    <proof>

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>

theorem snd_crt_prodtree: "xs ≠ [] ⇒ snd (crt_prodtree xs) = prod_list
  (map snd xs)"
  <proof>

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>

end

end

```

1.9 Examples and Benchmarks

```

theory CRT_Product_Tree_Benchmark
  imports CRT_Product_Tree "HOL-Library.Code_Target_Natural"
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 divide-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"
  ⟨proof⟩

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))"
  ⟨proof⟩
termination ⟨proof⟩

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 $\tilde{O}(n)$, whereas the naïve implementation takes time $\tilde{O}(n^2)$.

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.

$\langle ML \rangle$

end

References

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