§1. General Problem Area

§1.1. The Chinese Remainder Theorem

In Number Theory, a linear equation of the form $ax \equiv b(\mod m)$ will have a solution for x if and only if the greatest common denominator (d) of a and m divides b. In the cases where $d \mid b$, x can be represented as a set which is given by the residue class modulo m/d. In the case, however, that there are multiple linear congruences within a system who have different modulo m and simultaneously solve for x, a theorem known as the Chinese Remainder Theorem can be used to determine the solution to the system. This theorem states that if the values $m_1, m_2, \ldots m_k$ are pairwise co-prime positive integers and $a_1, a_2, \ldots a_k$ are arbitrary integers within a linear system that is written in the form

$$\begin{aligned} x &\equiv a_1(\mod m_1) \\ x &\equiv a_2(\mod m_2) \\ \cdots & \quad \cdots \quad \cdots \\ x &\equiv a_k(\mod m_k) \end{aligned}$$

then the linear system has an integral solution x, which is unique in the range $[0, \prod_{i=1}^{k} m_i)$.

In its simplest definition, the Chinese Remainder Theorem is a theorem that given a system of linear congruence with co-prime moduli, returns a unique solution. It is a clean and efficient way to solve several different problems in the subject of Number Theory.

§1.2. Theorem implementation

The process of implementing the Chinese Remainder Theorem to solve a collection of linear congruence with co-prime moduli is detailed as such:

1) First, the congruences are ordered by ascending value of the moduli. The last congruence is then rewritten from the form $x \equiv a_k(\mod m_k)$ to the equation $x = m_k b_k + a_k$ for some positive integer b_k
2) Following this, the rewritten equation must be substituted for x in the congruence containing the next largest moduli, $x \equiv a_{k-1}(\mod m_{k-1})$. Doing so, this second congruence becomes $m_k b_k + a_k \equiv a_{k-1}(\mod m_{k-1})$. This expression must then be solved to find an expression for b_k.
3) The expression for b_k must now be once again substituted into the congruence for x
4) Continue substituting the equations in the same way stated above until the equation for the variable x is the final solution to the system of linear equations.

§1.3. Implementation Example

To clarify the steps above, we will provide an example of how a unique solution can be found to a system of linear congruences with relatively prime moduli. The example system that we will use contains the following x values and a values:

a	m
2	3
19	5
24	7

This simplifies to the following system:

$$x \equiv 2 \pmod 3$$

$$x \equiv 4 \pmod 5$$

$$x \equiv 3 \pmod 7$$

The first congruence that is chosen will be $x \equiv 3 \pmod 7$, since it has the largest modulo. This can also be written as $x = 7j + 3$, for any integer value of j. We will then substitute this expression for x into the congruence with the next largest modulo, $x \equiv 4 \pmod 5$, to form the congruence $7j + 3 \equiv 4 \pmod 5$, which solves to $j \equiv 3 \pmod 5$. This congruence can then be written as $j = 5k + 3$, for any integer value k, similarly to what we had done previously when we first determined the implementation of j. Again through substitution, we can then substitute $5k + 3$ for j in the equation $x = 7j + 3$. Thus, $x = 7(5k + 3) + 3$ which simplifies to $x = 35k + 24$. We will then do the same steps for this equation and the third (and final) congruence, $x \equiv 2 \pmod 3$, substituting into the equation as such: $35k + 24 \equiv 2 \pmod 3$,

from which we may derive the solution $k \equiv 1 \pmod 3$. This can be reformulated as $k = 3l + 1$ and substituted back in the equation $x = 35k + 24$, revealing $x = 35(3l + 1) + 24$.

Finally, this equation can be written as $x = 105l + 59$ where l is any given integer, which gives us the unique solution to the system of linear congruences: $x \equiv 59(\mod 105)$.

§1.4. Mathematical Proof

The mathematical proof for the Chinese Remainder Theorem is written as such:

Given a system of linear congruences written as above, where all the modulo values m_i are pairwise co-prime to one another, let there be a value M that is the product of all the moduli: $\prod_{i=i}^{k} m_i$. Notice that for each value i less than k, M/m_i will always be an integer that is relatively prime to m_i (thus it is a unit modulo m_i). Now, assume that there is a value y_i that is the inverse of M/m_i modulo M. When there is a value j that is not equivalent to the integer value i, $(M/m_i)y_j$ is congruent to the value of 0 modulo m_j, because m_j divides M/m_i. Furthermore, assume that a value that is equivalent to the summation of all the values of $\frac{M}{m_j}y_ja_j$ (where j is equal to all the integers between 1 and k) is a simultaneous solution to all the congruences given in the system. This summation can be written as $\sum_{j=1}^{k} \frac{M}{m_j}y_ja_j$, and modulo m_i, this can be simplified to a_i (mod m_i) since as stated previously, y_i and M/m_i are inverses. This determines the validity of the Chinese Remainder Theorem. Next, we prove the uniqueness of the solution:

Assume that both p and q are both solutions that satisfy a given system of congruences. Then, $m_1 | (p - q)$ since $(p - q)$ is 0 mod m_1. Thus by a similar reasoning, $m_i | (p - q)$ for all values $1 \leq i \leq k$. But since all values of m_i are relatively prime, their product (M) divides

$(p - q)$. Therefore, p and q are congruent modulo M, proving the uniqueness of the solution within the bounds [0, M).

§1.5. Our Project

Within this project, we attempted to create a solver in ACL2s which, given a system of linear congruences satisfying the preconditions of the Chinese Remainder Theorem, derives a single integer solution to the system. Using the solver we had derived, we also wrote formalized proofs verifying the validity and uniqueness of the solutions our solver produced, where uniqueness is defined as there being no other non-negative solution to the system which is less than the product of the moduli.

§2. Approach & Methodology

§2.1. Initial Approach

Before creating our main solver, we wrote a brute force solver. This solver iterates through the integer values from 0 to the product of the moduli, checking whether each individual value is a solution to the Chinese Remainder Theorem. This solver seemed to work, but was never proven because we moved on to a solver that is more closely reflective of the process described above. This old method was considered to be worse than the new method because it was more computationally expensive. The upper bound on the running time is proportional to the product of the moduli. The new method, as described below, has a running time proportional to the sum of the moduli.

§2.2. The Solver Program

In order to implement a solver adherent to the Chinese Remainder Theorem in ACL2s, we broke down the steps described in §1.2 and demonstrated in §1.3 into extremely modular components of a fine granularity. To begin with, we defined two key data types. The first is a congruence pair, denoted in our code as an `anp`. An `anp` is a pair consisting of one integer, *a*, and one positive integer, *n*. This pair represents the right-hand side of the expression $x \equiv a(\mod n)$. The second primary data type we defined was a `loanp`, which is a non-empty list of `anp`'s, describing the system of linear congruences for which we are attempting to find a solution. While not core to our implementation, we also defined the data type `lop`, representing a list of positive integers.

In order to satisfy the input constraints of the Chinese Remainder Theorem, we had to further define a function `good-modsp: loanp -> bool`, which determines if the *n* values in a `loanp` are all pairwise-coprime to one another. This method uses the helper `ex-mods: loanp -> lop`, which extracts the right values from each `anp` pair in the input into a list, then uses the function `all-rpp: lop -> bool` to determine if all the values are relatively coprime. This `good-modsp` function was then used throughout the remainder of our implementation as an input contract on many methods. Our final preprocessing function was a helper `mod-as: loanp -> loanp`, which simplifies an inputted system of linear congruences by modding all of the *a* values over their corresponding *n* values.

Our solver proceeds in steps as follows, from the top level down:

1. The `solve-crt` function receives a system of linear congruences in the form of a `loanp`, and it's input contracts verify that the moduli are pairwise co-prime (assumed from here on out)
2. `solve-crt` passes the system to `fold-loanp: loanp -> anp`, whose purpose is to condense the system into a single pair consisting of the solution and the product of all the moduli
3. `fold-loanp` checks to determine if the provided list contains only one pair, for if it does, that pair is the solution. This is because a one element true list is the base case of a non-empty list, and thus if a single element list is reached, either we've recurred to the end of

the list or the input only ever had one congruence to begin with (and the solution to a single congruence system is trivial).

4. If the system has more than one element, then the result of `fold-loanp` is a call to `next-term`, taking as its first argument the first pair in the input and as its second argument the recursive call to `fold-loanp` on the remainder of the input.

5. `next-term: anp X anp -> anp` is a function which, given two congruence pairs, derives a congruence pair where the *a* value is congruent to both input *a*'s modulo their respective *n*'s, and the *n* value of the result is the product of the input *n*'s (i.e., the function derives a solution to a system of linear congruences containing only two expressions). In order to do so, `next-term` makes a call to `find-d` in order to generate a pair (d_a, d_n), where $d_a(\mod d_n)$ is a solution to the form $n_1 d_a + a_1 \equiv a_2(\mod n_2)$. In this format, assume that the arguments to `next-term` represent (a_1, n_1) and (a_2, n_2) respectively.

6. In order to generate this pair, the function `find-d: anp X anp -> anp` uses the `cdr` of it's second input as the `cdr` of it's resulting pair. In order to find the `car` (i.e. the d_a value), however, it makes a call to `find-ad.`

7. To compute d_a, `find-ad: anp X anp -> nat` uses the accumulating helper `find-ad-from: anp X anp X nat -> nat` to check every natural number between zero and n_2, as by modular arithmetic one of those values must solve the congruence.

8. To generate a solution for it’s two inputs, `next-term`, now having a value for the pair (d_a, d_n), returns a pair equivalent to $(d_a n_1 + a_1, n_1 n_2)$, which is guaranteed to be a solution for the system $\{x \equiv a_1(\mod n_1), x \equiv a_2(\mod n_2)\}$
9. Now in possession of results for a system of two congruences, `fold-loanp` can apply this function all the way up the list, using `next-term` to join each input pair with the result of the rest of the system.
10. Finally, `solve-crt` is provided with a pair whose first element is the solution to the system and whose second element is the product of all the moduli (denoted *M* as above). In order to produce a final result, `solve-crt` takes the mod of the solution over the product to ensure it returns the smallest solution.

This approach has a running time bounded by the sum of the moduli for the following reason:

To find a given d_i value, we iterate through the values from 0 possibly up to $n_i + 1$. We perform this process for each of the moduli except for one. This means that the total number of d values checked is bounded by the sum of the number of checks that are done for each n, which is the sum of the moduli.

§2.3. The Formalized Proof

The solver was shown to compute a valid, unique solution for any set of congruences that are valid under the constraints of the CRT. We first verified that the solution produced by the solver was a correct solution to the given set of congruences. We wrote the following function:

```
is-solp: loanp X int -> bool
```

which returns true iff the provided `int` satisfies all of the congruences provided in the `loanp`. We then wrote a conjecture that, when given an input that does not violate its input contracts, `solve-crt` will produce a result that satisfies `is-solp`. This is written as:

```
(is-solp l (solve-crt l))
```

This conjecture was shown to be true through the proof methods that were taught in CS2800 lectures.

We then conjectured that the solution was unique within the bounds of the problem. That is, we ensured that there was no integer between 0 and M, other than the solution produced by solve-crt, that was a solution to the problem. This proof was also performed using methods taught in CS2800. Please see the `proof.lisp` file in our repository for the full technical proof of this conjecture.

§3. Metrics & Summary

§3.1. Results

Our ACL2 implementation was largely successful, with a few caveats. We were able to admit all of the functions from the bottom up to `next-term`. In other words, the implementation can compute the solution for exactly two terms. However, in order to proceed to the general solution, this function must be called recursively on the `loanp` representing the problem, which is done in the `fold-loanp` function. The `fold-loanp` function is not admissible in ACL2. When we attempted to admit it, it rejected the function dude to a perceived guard violation. This problem persisted even when `verify-guards` was turned off in the `xargs` of the function. We believe that this failure to admit the function was erroneous, as we were able to prove through traditional pen-and-paper means that the function should successfully solve the theorem (see §2.3).

§3.2. Next Steps

Now having developed a solver in ACL2S that implements the Chinese Remainder Theorem, we are able to look further into other related problems that are now within easier reach. The first being that we can develop a solver that uses the Chinese Remainder Theorem in a way that the Chinese Remainder Theorem can be generalized to an arbitrary ring *R*, and not solely bound by the integers. Given this information, we can determine a ring isomorphism that states that $\mathbb{Z}/M\mathbb{Z} \cong (\mathbb{Z}/p_1^{a_1}\mathbb{Z} \times \ldots \times \mathbb{Z}/p_k^{a_k}\mathbb{Z})$ where $M = \prod_{i=0}^{k} p_i^{a_i}$ is the prime factorization of M.

Another way that we can expand upon our Chinese Remainder Theorem solver is that we can use the general principles of the Chinese Remainder Theorem in order to consider a case where the moduli have common divisors and are not co-prime. The way to go about this would be to compute the common divisors and separate the congruences to a point where they do not have any common divisors or to factor the moduli into prime powers. Thus this solves the problem of the moduli not being co-prime and the Chinese Remainder Theorem can then be implemented to system of congruences to determine a general solution.

The final example of related problem that we will discuss is using the Chinese Remainder Theorem as a way to solve polynomial congruences of a higher degree. This can be done by reducing the congruence to where the modulus is a prime power and the Chinese Remainder Theorem can be applied to find the solutions of the polynomial congruence.