A Formalisation of a Fast Fourier Transform

Laurent Théry

A Formalisation of a Fast Fourier Transform

Laurent Théry
Laurent.Thery@sophia.inria.fr

Abstract

This notes explains how a standard algorithm that constructs the discrete Fourier transform has been formalised and proved correct in the Coq proof assistant using the SSReflect extension.

1 Introduction

Fast Fourier Transforms are key tools in many areas. In this note, we are going to explain how they have been formalised in a theorem prover like Coq.

There are many ways to motivate this work. As univariate polynomials is at the heart of our formalisation, we take, here, one application : the polynomial multiplication.

If we take two polynomials p q of degree n, a naive way of multiplying them requires a quadratic number n^2 of multiplications, i.e. we perform the multiplication of each term of p (there are at most n of them) by a term of q (again n of those). It is possible to get a faster algorithm using a fast Fourier transform. The idea is to take an alternative representation for polynomials : a polynomial p of degree n is uniquely defined by its value $p(x_1), ..., p(x_n)$ on n distinct points $x_1, \ldots x_n$. This representation is what is called polynomial interpolation and the x_is are called the interpolation points.

A direct way to effectively construct a polynomial p from its evaluation $p(x_1), ..., p(x_n)$ is by using Lagrange polynomial $L_{n,k} = \Pi_{i \neq k}(x - x_i)/(x_k - x_i)$. It is easy to check that $L_{n,k}(x_j) = 1$ if $j = k$ or 0 otherwise. Then, we have $p = \sum_{1 \leq i \leq k} p(x_i) L_{n,i}$. Computing the multiplication in the interpolation world is easy. The polynomial pq is of degree at most $2n$ so if we have the values of p and q at $2n$ points $x_1, \ldots, x_{2n}$, we simply need to perform $2n$ multiplication to get $p(x_1)q(x_1), ..., p(x_{2n})q(x_{2n})$. Of course, we need algorithms to move from

the usual polynomial representation to the interpolation one and back. This is what the fast Fourier transform gives us. The trick is that we are going to evaluate the polynomials at very specific points of interpolation (the x_is are roots of unit). What makes it work is that, for those points, the potential n^2 values of the different $(x_i)^j$ $(1 \leq i, j \leq n)$ consist of only n distinct values, so computing the $p(x_1), ..., p(x_n)$ can be done very efficiently in $nlog(n)$.

Note that this is not the first time fast Fourier transform has been formalised in Coq. To our knowledge, the first one was done in 2001 by Venanzio Capretta (see [2]). It is about time to revisit this work and see how concise it can get using existing libraries. Also, this initial effort was mostly interested in the recursive presentation of the algorithm. Here, we also give an iterative version.

2 Formalisation of the recursive algorithm

The algorithm manipulates three kinds of data:

- natural numbers, i.e. elements of type `nat`;
- elements of an arbitrary integral domain R;
- univariate polynomials over R, i.e. elements of type `{poly R}`.

The operations we use on natural numbers are the successor and predecessor functions (i`.+1` and i`.-1`), the doubling and halving functions (i`.*2`, i`./2` and `uphalf` i = (i`.+1./2`)), the exponentiation (i `^` j), the division (i `%/` j), and the modulo (i `%%` j). For the integral domain, we use the usual ring operations : the addition x `+` y, the multiplication by a scalar k `*:` x, the multiplication x `*` y, and the exponentiation by a natural number x `^+` n. Also, a predicate of the library that is useful in our application is the one that indicates that an element w is a n^{th} primitive root of unity. It is written as n`.-primitive_root` w. It means that w `^+` $n = 1$ and that n is actually the smallest non-zero natural number that has this property. In our formalisation, we derive two easy lemmas about primitive roots

```
Lemma prim_exp2nS n (w : R) :
  (2 ^ n.+1).-primitive_root w → w ^+ (2 ^ n) = -1.
Lemma prim_sqr n (w : R) :
  (2 ^ n.+1).-primitive_root w → (2 ^ n).-primitive_root (w ^+ 2).
```

The first lemma is used to simplify expression involving w^j. Its proof is the only place where the integrality is used (in order to get from $w^2 = 1$ that either $w = 1$ or $w = -1$). The recursive algorithm of the fast Fourier transform takes as argument a primitive root w and performs some recursive calls with w `^+ 2`. The lemma *prim_sqr* is then used to prove that the primitive root property of the argument is an invariant of the recursion.

The algorithm is mainly manipulating univariate polynomials. A polynomial is represented by a list whose last element (the leading term), if it exists, is non-zero. The empty list represents the null polynomial. A polynomial p can be automatically converted to a list, so `size` p is understood as the length of the list representing p. So, if p is not null, it is the usual degree of the polynomial incremented by one. We can access the n^{th} term of polynomial p by p`'_`n. The leading term is then p`'_ (size` p`).-1`. The variable of the univariate polynomials is written `'X` and `'X^`n its power. Evaluating a polynomial p at point x is written p`.[`x`]`. Composition two polynomials p and q which consists in lifting the evaluation from points to polynomials p`.[`q`]` is written p `\Po` q. We can turn a function F into a polynomial using `\poly_`$(i$ `<` n$)$ F i that builds a polynomial of size n whose i^{th} term is F i. A special notation is available for constant polynomials where one can write c`%:P` in order to build a polynomial that only contains the constant term c.

The recursive algorithm is taking two arguments : a polynomial p and a primitive root w of degree 2^n and returns the evaluation of p at points 1, w, w^2, ..., w^{2^n-1} as the polynomial p`.[1] +` p`.[`w`]` X `+` ... `+` p`.[`w^{2^n-1}`]` X^{2^n-1}. The recursive calls are made on the even and odd parts of the polynomial p. If $p = 1 + 2X + 3X^2 + 4X^3 + 5X^4$, its even part is $1 + 3X + 5X^2$ and its even part $2 + 4X$. As these operations on polynomials are not in the library we had to define them.

```
Definition even_poly p : {poly R} := \poly_(i < uphalf (size p)) p'_i.*2.
Definition odd_poly  p : {poly R} := \poly_(i < (size p)./2) p'_i.*2.+1.
```

It is then easy to derive the key lemma that justifies the decomposition of the polynomial in the recursive calls

```
Lemma poly_even_odd p : (even_poly p \Po 'X^2) + (odd_poly p \Po 'X^2)) * 'X = p.
```

We are now ready to present the algorithm we have proved correct:

```
Fixpoint fft (n : nat) (w : R) (p : {poly R}) : {poly R} :=
  if n is n1.+1 then
    let ev := fft n1 (w ^+ 2) (even_poly p) in
    let ov := fft n1 (w ^+ 2) (odd_poly  p) in
    \poly_(i < 2 ^ (n1.+1)) let j := i %% (2 ^ n1) in ev'_j + ov'_ j * w ^+ i
  else (p'_0)%:P.
```

It takes a natural number n, a root of unity w and a polynomial p and returns a polynomial whose coefficients are the value at the interpolation point w^i. More formally, the correctness lemma is the following :

```
Lemma fftE n (w : R) p :
  size p ≤ 2 ^ n → (2 ^ n).-primitive_root w →
  fft n w p = \poly_(i < 2 ^ n) p.[w ^+ i].
```

The proof is straightforward. It is done by induction. We have to prove the equality of two polynomials, so we show that their i^{th} coefficients are equal

```
  (fft n (w ^+ 2) (even_poly p))'_(i %% 2 ^ n) +
  (fft n (w ^+ 2) (odd_poly  p))'_(i %% 2 ^ n) * w ^+ i = p.[w ^+ i]
```

with the assumption that w is a primitive root of order 2^{n+1}. Using the induction hypothesis on the left of the equality and the decomposition lemma *odd_even_polyE* on the right, we get

```
  (even_poly p).[(w ^+ 2) ^+ (i %% 2 ^ n)] +
  (odd_poly  p).[(w ^+ 2) ^+ (i %% 2 ^ n)] * w ^+ i =
  (even_poly p).[(w ^+ i) ^+ 2] + (odd_poly p).[(w ^+ i) ^+ 2] * w ^+ i
```

This means we are left with proving the following equality

```
  (w ^+ 2) ^+ (i %% 2 ^ n) = (w ^+ i) ^+ 2
```

that directly follows from the fact that $w^{2^{n+1}} = 1$.

Finally, we also prove an alternative version of the algorithm that more explicitly exhibits the data path, the so-called butterfly.

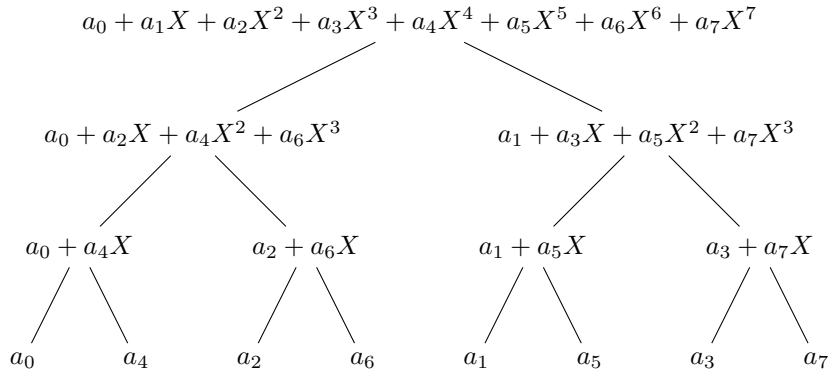
```
Fixpoint fft1 n w p : {poly R} :=
  if n is n1.+1 then
  let ev := fft1 n1 (w ^+ 2) (even_poly p) in
  let ov := fft1 n1 (w ^+ 2) (odd_poly p)    in
  \sum_(j < 2 ^ n1)
    ((ev'_j + ov'_j * w ^+ j) *: 'X^j +
     (ev'_j - ov'_j * w ^+ j) *: 'X^(j + 2 ^ n1))
  else (p'_0)%:P.
```

It is straightforward to prove that both recursive versions compute the same thing.

```
Lemma fft1E n (w : R) p : (2 ^ n).-primitive_root w → fft1 n w p = fft n w p.
```

3 Formalisation of the iterative algorithm

In our formalisation, we are going to derive an iterative version from the recursive in a very straightforward way. Let us explain it on an example with a polynomial of degree 7 ($2^3 - 1$). The depth of the recursion is 3 and the binary tree of the recursive calls looks like:



The idea is to put all the results at depth i in a single polynomial. Here, at depth 3 it is a polynomial containing 8 sub polynomials of degree 0, at depth 2, 4 polynomials of degree 1, at depth 1, 2 polynomials of degree 3 and finally one polynomial of degree 7. The final result is build bottom up. Initially we start with the polynomials that contains all the leaves. Then, one step of the iteration simply take the results at depth i and returns the results at depth $i-1$.

Let us first concentrate on the initial value, the values of all the leaves. If we remember that we use an even/odd partition, putting the odd part on the left and the right part on the right, this means that if we look at the binary representation, the bits from the right to the left let us sort. If we take our example, we have to sort

$$[0; 1; 2; 3; 4; 5; 6; 7]$$

With their binary representation, it gives

$$[0 \rightsquigarrow 000; 1 \rightsquigarrow 001; 2 \rightsquigarrow 010 \rightsquigarrow; 3 \rightsquigarrow 011; 4 \rightsquigarrow 100; 5 \rightsquigarrow 101; 6 \rightsquigarrow 110; 7 \rightsquigarrow 111]$$

Reversing them, we get

$$[0 \rightsquigarrow 000; 1 \rightsquigarrow 100; 2 \rightsquigarrow 010; 3 \rightsquigarrow 110; 4 \rightsquigarrow 001; 5 \rightsquigarrow 101; 6 \rightsquigarrow 011; 7 \rightsquigarrow 111]$$

Translating them back to decimal numbers, we have

$$[0 \rightsquigarrow 0 \quad ; 1 \rightsquigarrow 4 \quad ; 2 \rightsquigarrow 2 \quad ; 3 \rightsquigarrow 6 \quad ; 4 \rightsquigarrow 1 \quad ; 5 \rightsquigarrow 5 \quad ; 6 \rightsquigarrow 3 \quad ; 7 \rightsquigarrow 7 \quad]$$

To build this initial polynomial, we first define *digitn b n m* that computes the m^{th} digit of n in base b. We then use it to reverse a number : *rdigitn b n m* reverses the n bits in base b of m. Finally, the initial polynomial with 2^n terms is created by *reverse_poly n p* using an appropriate permutation of the coefficient of p.

```
Definition digitn b n m := (n %/ b ^ m) %% b.
Definition rdigitn b n m := \sum_(i < n) digitn b m (n.-1 - i) * b ^ i.
Definition reverse_poly n (p : {poly R}) := \poly_(i < 2 ^ n) p'_(rdigitn 2 n i).
```

On our example, *reverse_poly* 3 p returns

$$p_0 + p_4X + p_2X^2 + p_6X^3 + p_1X^4 + p_5X^5 + p_3X^6 + p_7X^7$$

.

Now, we want to express that after each step of the iteration we get all the results at depth i. In the recursion, the polynomial is split in two using the even and odd part, but the results are glued together using the left to right concatenation. So, we need some operations to get the low terms or the high term of a polynomial.

```
Definition take_poly m (p : {poly R}) := \poly_(i < m) p'_i.
Definition drop_poly m (p : {poly R}) := \poly_(i < size p - m) p'_(i + m).
Lemma poly_take_drop m p : take_poly m p + drop_poly m p * 'X^m = p.
```

take_poly m p returns the polynomial that has the m low terms of p while *drop_poly* m p returns the polynomial with the high terms of p skipping the m low terms.

We want to express that we have in a polynomial q all the results of calling the recursive algorithm p cutting at depth n, so every leaf is a call of fft_1 with the appropriate part of the polynomial p. This is done using a recursive predicate. If n is not zero, we split p in two with even and odd part and q with left part and right part. If n is zero, we are at a leaf so the result must be a call to fft_1.

```
Fixpoint all_results_fft1 n m w p q :=
  if n is n1.+1 then
  all_results_fft1 n1 m w (even_poly p) (take_poly (2 ^ (m + n1)) q) ∧
  all_results_fft1 n1 m w (odd_poly p)  (drop_poly (2 ^ (m + n1)) q)
  else q = fft1 m w p.
```

For the initial polynomial, we prove that, given a polynomial p of size less than 2^n the reverse polynomial has all the results of fft_1 0 at depth n

```
Lemma all_results_fft1_reverse_poly p n w :
  size p ≤ 2 ^ n → all_results_fft1 n 0 w p (reverse_poly n p).
```

This lemma is proved by induction on n. The key lemma for proving it is the fact *reverse_poly* decomposes nicely using odd and even parts.

```
Lemma reverse_polyS n p :
  reverse_poly n.+1 p =
  reverse_poly n (even_poly p) + reverse_poly n (odd_poly p) * 'X^(2 ^ n).
```

Now, we can define a step of the iteration. If we are at depth m, we have 2^{m+1} results of size 2^n

```
Definition step m n w (p : {poly R}) :=
\sum_(l < 2 ^ m)
  let ev := \poly_(i < 2 ^ n) p'_(i + l * 2 ^ n.+1) in
  let ov := \poly_(i < 2 ^ n) p'_(i + l * 2 ^ n.+1 + 2 ^ n) in
    \sum_(j < 2 ^ n)
      ((ev'_ j + ov'_ j * w ^+ j) *: 'X^(j + l * 2 ^ n.+1) +
       (ev'_ j - ov'_ j * w ^+ j) *: 'X^(j + l * 2 ^ n.+1 + 2 ^ n)).
```

and the correctness is that applying a step decreases the depth m while increasing the size n.

```
Lemma all_results_fft1_step m n w (p q : {poly R}) :
  size p ≤ 2 ^ (m + n).+1 →
  size q ≤ 2 ^ (m + n).+1 →
  all_results_fft1 m.+1 n (w ^+ 2) p q →
  all_results_fft1 m n.+1 w p (step m n w q).
```

Again, the key lemmas are the ones that show that *step* behaves well with *take_poly* and *drop_poly*.

```
Lemma take_step m n w (p : {poly R}) :
  size p ≤ 2 ^ (m + n).+2 →
  take_poly (2 ^ (m + n).+1) (step m.+1 n w p) =
  step m n w (take_poly (2 ^ (m + n).+1) p).
Lemma drop_step m n w (p : {poly R}) :
  size p ≤ 2 ^ (m + n).+2 →
  drop_poly (2 ^ (m + n).+1) (step m.+1 n w p) =
  step m n w (drop_poly (2 ^ (m + n).+1) p).
```

Now, we code the iteration of *step*, it is straightforward to prove that we get the same result as the recursive algorithm.

```
Fixpoint istep_aux m n w p :=
  if m is m1.+1 then istep_aux m1 n.+1 w (step m1 n (w ^+ (2 ^ m1)) p) else p.
Definition istep n w p := istep_aux n 0 w (reverse_poly n p).
Lemma istep_fft1 n p w : size p ≤ 2 ^ n → istep n w p = fft1 n w p.
```

Similarly we prove the correctness of an alternative and more direct iterative version.

```
Definition step1 m n w (p : {poly R}) :=
  \poly_(i < 2 ^ (m + n).+1)
  let j := i %% 2 ^ n.+1 in
  if j < 2 ^ n then
    p'_i + p'_(i + 2 ^ n) * w ^+ j
  else
    p'_(i - 2 ^ n) - p'_i * w ^+ (j - 2 ^ n).
Lemma step1E m n w p : step1 m n w p = step m n w p.
```

4 Inverse algorithm

We have seen how to go from polynomial to interpolation. What about the other way around. In fact, we can use the same algorithm. From the direct direction R was an integral ring. Here, we need a field F. The notation for inverse is `x^-1`. The idea of the inverse algorithm is to use $1/w$ instead of w.

```
Definition ifft n w p : {poly F} := (2 ^ n)%:R ^-1%:P * (fft n w^-1 p).
```

where `n%:R` is the coercion for natural number n into F. Its correctness follows if the characteristic of F is not 2-.

```
Lemma fftK n (w : F) p :
  2%:R != 0 → size p ≤ 2 ^ n → (2 ^ n).-primitive_root w →
  ifft n w (fft n w p) = p.
```

5 Conclusion

We have presented our formalisation of the fast Fourier transform. It makes used intensively of the polynomial library and the big operators [1]. The complete source code is about 600 lines. It is available at

https://github.com/thery/mathcomp-extra/blob/master/fft.v

References

[1] Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pașca. Canonical Big Operators. In *TPHOLs*, volume 5170 of *LNCS*, Montreal, Canada, 2008.

[2] Venanzio Capretta. Certifying the Fast Fourier Transform with Coq. In *TPHOLs'01*, number 2152 in LNCS, pages 169–184, Edinburgh, Scotland, 2001.