

Fast Fourier Transformation

Clemens Ballarin

12th December 2009

Contents

1 Preliminaries	1
2 Complex Roots of Unity	3
2.1 Basic Lemmas	3
2.2 Derived Lemmas	4
3 Discrete Fourier Transformation	4
4 Discrete, Fast Fourier Transformation	6

```
theory FFT
imports Complex_Main
begin
```

We formalise a functional implementation of the FFT algorithm over the complex numbers, and its inverse. Both are shown equivalent to the usual definitions of these operations through Vandermonde matrices. They are also shown to be inverse to each other, more precisely, that composition of the inverse and the transformation yield the identity up to a scalar.

The presentation closely follows Section 30.2 of Cormen *et al.*, *Introduction to Algorithms*, 2nd edition, MIT Press, 2003.

1 Preliminaries

```
lemma of_nat_cplx:
  "of_nat n = Complex (of_nat n) 0"
  <proof>
```

The following two lemmas are useful for experimenting with the transformations, at a vector length of four.

lemma Iv14:

```
"{0..<4::nat} = {0, 1, 2, 3}"
⟨proof⟩
```

lemma Sum4:

```
"(∑ i=0..<4::nat. x i) = x 0 + x 1 + x 2 + x 3"
⟨proof⟩
```

A number of specialised lemmas for the summation operator, where the index set is the natural numbers

lemma setsum_add_nat_iv1_singleton:

```
assumes less: "m < (n::nat)"
shows "f m + setsum f {m<..

```

lemma setsum_add_split_nat_iv1_singleton:

```
assumes less: "m < (n::nat)"
and g: "!!i. [| m < i; i < n |] ==> g i = f i"
shows "f m + setsum g {m<..

```

lemma setsum_add_split_nat_iv1:

```
assumes le: "m <= (k::nat)" "k <= n"
and g: "!!i. [| m <= i; i < k |] ==> g i = f i"
and h: "!!i. [| k <= i; i < n |] ==> h i = f i"
shows "setsum g {m..

```

lemma iv1_splice_Un:

```
"{0..<2*n::nat} = (op * 2 ‘ {0..

```

lemma iv1_splice_Int:

```
"(op * 2 ‘ {0..

```

lemma double_inj_on:

```
"inj_on (%i. 2*i::nat) A"
⟨proof⟩
```

lemma Suc_double_inj_on:

```
"inj_on (%i. Suc (2*i)) A"
⟨proof⟩
```

```
lemma setsum_splice:
  "(∑ i::nat = 0..<2*n. f i) = (∑ i = 0..<n. f (2*i)) + (∑ i = 0..<n. f (2*i+1))"
⟨proof⟩
```

2 Complex Roots of Unity

The function `cis` from the complex library returns the point on the unity circle corresponding to the argument angle. It is the base for our definition of `root`. The main property, De Moirve's formula is already there in the library.

```
constdefs
  root :: "nat => complex"
  "root n == cis (2*pi/(real (n::nat)))"
```

```
lemma sin_periodic_pi_diff [simp]: "sin (x - pi) = - sin x"
⟨proof⟩
```

```
lemma sin_cos_between_zero_two_pi:
  assumes 0: "0 < x" and pi: "x < 2 * pi"
  shows "sin x ≠ 0 ∨ cos x ≠ 1"
⟨proof⟩
```

2.1 Basic Lemmas

```
lemma root_nonzero:
  "root n ~ 0"
⟨proof⟩
```

```
lemma root_unity:
  "root n ^ n = 1"
⟨proof⟩
```

```
lemma root_cancel:
  "0 < d ==> root (d * n) ^ (d * k) = root n ^ k"
⟨proof⟩
```

```
lemma root_summation:
  assumes k: "0 < k" "k < n"
  shows "(∑ i=0..<n. (root n ^ k) ^ i) = 0"
⟨proof⟩
```

```

lemma root_summation_inv:
  assumes k: "0 < k" "k < n"
  shows " $(\sum_{i=0..<n}. ((1 / \text{root } n) ^ k) ^ i) = 0$ "
  <proof>

```

```

lemma root0 [simp]:
  "root 0 = 1"
  <proof>

```

```

lemma root1 [simp]:
  "root 1 = 1"
  <proof>

```

```

lemma root2 [simp]:
  "root 2 = Complex -1 0"
  <proof>

```

```

lemma root4 [simp]:
  "root 4 = ii"
  <proof>

```

2.2 Derived Lemmas

```

lemma root_cancel1:
  "root (2 * m) ^ (i * (2 * j)) = root m ^ (i * j)"
  <proof>

```

```

lemma root_cancel2:
  "0 < n ==> root (2 * n) ^ n = - 1" <proof>

```

3 Discrete Fourier Transformation

We define operations DFT and IDFT for the discrete Fourier Transform and its inverse. Vectors are simply functions of type `nat => complex`.

DFT `n a` is the transform of vector `a` of length `n`, IDFT its inverse.

constdefs

```

DFT :: "nat => (nat => complex) => (nat => complex)"
"DFT n a == (%i.  $\sum_{j=0..<n}. (\text{root } n) ^ (i * j) * (a j)$ )"
IDFT :: "nat => (nat => complex) => (nat => complex)"
"IDFT n a == (%i.  $(\sum_{k=0..<n}. (a k) / (\text{root } n) ^ (i * k))$ )"

```

lemma "map (DFT 4 a) [0, 1, 2, 3] = ?x"

<proof>

Lemmas for the correctness proof.

lemma DFT_lower:

"DFT (2 * m) a i =
DFT m (%i. a (2 * i)) i +
(root (2 * m)) ^ i * DFT m (%i. a (2 * i + 1)) i"

<proof>

lemma DFT_upper:

assumes mbound: "0 < m" and ibound: "m <= i"
shows "DFT (2 * m) a i =
DFT m (%i. a (2 * i)) (i - m) -
root (2 * m) ^ (i - m) * DFT m (%i. a (2 * i + 1)) (i - m)"

<proof>

lemma IDFT_lower:

"IDFT (2 * m) a i =
IDFT m (%i. a (2 * i)) i +
(1 / root (2 * m)) ^ i * IDFT m (%i. a (2 * i + 1)) i"

<proof>

lemma IDFT_upper:

assumes mbound: "0 < m" and ibound: "m <= i"
shows "IDFT (2 * m) a i =
IDFT m (%i. a (2 * i)) (i - m) -
(1 / root (2 * m)) ^ (i - m) *
IDFT m (%i. a (2 * i + 1)) (i - m)"

<proof>

DFT und IDFT are inverses.

declare divide_divide_eq_right [simp del]

divide_divide_eq_left [simp del]

lemma power_diff_inverse:

assumes nz: "(a::'a::field) ~= 0"
shows "m <= n ==> (inverse a) ^ (n-m) = (a^m) / (a^n)"

<proof>

lemma power_diff_rev_if:

assumes nz: "(a::'a::field) ~= 0"

```

shows "(a^m) / (a^n) = (if n <= m then a ^ (m-n) else (1/a) ^ (n-m))"
<proof>

```

```

lemma power_divides_special:
  "(a::'a::field) ^ n = 0 ==>
  a ^ (i * j) / a ^ (k * i) = (a ^ j / a ^ k) ^ i"
<proof>

```

```

theorem DFT_inverse:
  assumes i_less: "i < n"
  shows "IDFT n (DFT n a) i = of_nat n * a i"
<proof>

```

4 Discrete, Fast Fourier Transformation

FFT k a is the transform of vector a of length 2^k , IFFT its inverse.

consts

```

FFT :: "nat => (nat => complex) => (nat => complex)"
IFFT :: "nat => (nat => complex) => (nat => complex)"

```

primrec

```

"FFT 0 a = a"
"FFT (Suc k) a =
  (let (x, y) = (FFT k (%i. a (2*i)), FFT k (%i. a (2*i+1)))
  in (%i. if i < 2^k
    then x i + (root (2 ^ (Suc k))) ^ i * y i
    else x (i- 2^k) - (root (2 ^ (Suc k))) ^ (i- 2^k) * y (i- 2^k)))"

```

primrec

```

"IFFT 0 a = a"
"IFFT (Suc k) a =
  (let (x, y) = (IFFT k (%i. a (2*i)), IFFT k (%i. a (2*i+1)))
  in (%i. if i < 2^k
    then x i + (1 / root (2 ^ (Suc k))) ^ i * y i
    else x (i - 2^k) -
      (1 / root (2 ^ (Suc k))) ^ (i - 2^k) * y (i - 2^k)))"

```

Finally, for vectors of length 2^k , DFT and FFT, and IDFT and IFFT are equivalent.

theorem DFT_FFT:

```

"!!a i. i < 2 ^ k ==> DFT (2 ^ k) a i = FFT k a i"
<proof>

```

theorem IDFT_IFFT:

```

"!!a i. i < 2 ^ k ==> IDFT (2 ^ k) a i = IFFT k a i"

```

<proof>

lemma "map (FFT (Suc (Suc 0)) a) [0, 1, 2, 3] = ?x"
<proof>

end