Certifying the Fast Fourier Transform with Coq

Venanzio Capretta*

Computing Science Institute, University of Nijmegen
Postbus 9010, 6500 GL Nijmegen, The Netherlands
venanzio@cs.kun.nl
telephone: +31+24+3652647, fax: +31+24+3553450

Abstract. We program the Fast Fourier Transform in type theory, using the tool Coq. We prove its correctness and the correctness of the Inverse Fourier Transform. A type of trees representing vectors with interleaved elements is defined to facilitate the definition of the transform by structural recursion. We define several operations and proof tools for this data structure, leading to a simple proof of correctness of the algorithm. The inverse transform, on the other hand, is implemented on a different representation of the data, that makes reasoning about summations easier. The link between the two data types is given by an isomorphism. This work is an illustration of the *two-level approach* to proof development and of the principle of adapting the data representation to the specific algorithm under study. CtCoq, a graphical user interface of Coq, helped in the development. We discuss the characteristics and usefulness of this tool.

1 Introduction

An important field of research in formalized mathematics tackles the verification of classical algorithms widely used in computer science. It is important for a theorem prover to have a good library of proof-checked functions, that can be used both to extract a formally certified program and to quickly verify software that uses the algorithm. The Fast Fourier Transform [8] is one of the most widely used algorithms, so I chose it as a case study in formalization using the type-theory proof tool Coq [2]. Here I present the formalization and discuss in detail the parts of it that are more interesting in the general topic of formal verification in type theory.

Previous work on the computer formalization of FFT was done by Ruben Gamboa [10] in ACL2 using the data structure of powerlists introduced by Jayadev Misra [11], which is similar to the structure of polynomial trees that we use here.

* I worked on the formalization of FFT during a two-month stay at the INRIA research center in Sophia Antipolis, made possible by a grant from the Dutch Organization for Scientific Research (NWO, Dossiernummer F 62-556). I am indebted to the people of the Lemme group for their support and collaboration. In particular, I want to thank Yves Bertot for his general support and for teaching me how to use CtCoq, and Loïc Pottier for his help in formulating the Fast Fourier Transform in type theory.

R.J. Boulton and P.B. Jackson (Eds.): TPHOLs 2001, LNCS 2152, pp. 154–168, 2001.

The Discrete Fourier Transform is a function commuting between two representations of polynomials over a commutative ring, usually the algebraic field $\mathbb{C}$. One representation is in the *coefficient domain*, where a polynomial is given by the list of its coefficients. The second representation is in the *value domain*, where a polynomial of degree $n-1$ is given by its values on n distinct points. The function from the coefficient domain to the value domain is called *evaluation*, the inverse function is called *interpolation*. The Discrete Fourier Transform (DFT) and the Inverse Discrete Fourier Transform (iDFT) are the evaluation and interpolation functions in the case in which the points of evaluation are distinct n-roots of the unit element of the ring. The reason to consider such particular evaluation points is that, in this case, an efficient recursive algorithm exists to perform evaluation, the Fast Fourier Transform (FFT), and interpolation, the Inverse Fourier Transform (iFT). Let

$$f(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1} = \sum_{i=0}^{n-1} a_i x^i$$

be a polynomial of degree $n-1$ and ω be a primitive n-root of unity, that is, $\omega^n = 1$ but $\omega^j \neq 1$ for $0 < j < n$. We must compute $f(\omega^j)$ for $j = 0, 1, \ldots, n-1$. First of all we write $f(x)$ as the sum of two components, the first comprising the monomials of even power and the second the monomials of odd power, for which we can collect a factor x:

$$f(x) = f_e(x^2) + x f_o(x^2). \tag{1}$$

The polynomials f_e and f_o have degree $n/2-1$ (assuming that n is even). We could apply our algorithm recursively to them and to ω^2, which is an $n/2$-root of unity. We obtain the values

$$f_e((\omega^2)^0),\ f_e((\omega^2)^1),\ \ldots,\ f_e((\omega^2)^{n/2-1});$$
$$f_o((\omega^2)^0),\ f_o((\omega^2)^1),\ \ldots,\ f_o((\omega^2)^{n/2-1}).$$

We have, therefore, $f_e((\omega^2)^i) = f_e((\omega^i)^2)$ for $i = 0, \ldots, n/2-1$ which we can feed into Formula 1. The only problem is that Formula 1 must be evaluated for $x = \omega^i$ when $i = 0, \ldots, n-1$. We are still missing the values for $i = n/2, \ldots, n-1$. Here is where the fact that ω is a primitive n-root of unity comes useful: $\omega^{n/2} = -1$, so for $j = 0, \ldots, n/2-1$ we have that

$$\omega^{n/2+j} = \omega^{n/2}\omega^j = -\omega^j$$

and therefore $f_e((\omega^{n/2+j})^2) = f_e((\omega^j)^2)$. So the values of the first term of Formula 1 for $i = n/2, \ldots, n-1$ are equal to the values for $i = 0, \ldots, n/2-1$ and we don't need to compute them. A similar argument holds for f_o. If we measure the algorithmic complexity by the number of multiplications of scalars that need to be performed, we see that the algorithm calls itself twice on inputs of half size and then must still perform n multiplications (multiply x by $f_o(x)$ in Formula 1). This gives an algorithm of complexity $O(n \log n)$, much better than the naive quadratic algorithm.