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.

Vice versa if we are given the values of the polynomial f on the n-roots of unity, $y_0 = f(\omega^0), \ldots, y_{n-1} = f(\omega^{n-1})$, we can compute the vector of coefficients of f by applying DFT to the vector $\langle y_0, \ldots, y_{n-1}\rangle$ with ω^{-1} in place of ω and then divide by n. The proof of this fact is well-known. We will give a type-theoretic version of it in section 5.

One useful application of FFT is the computation the product of two polynomials. If $f(x) = \sum_{i=0}^{n-1} a_i x^i$ and $g(x) = \sum_{i=0}^{n-1} b_i x^i$, we want to compute their product $f(x)g(x) = \sum_{i=0}^{n-1} c_i x^i$. A direct computation of the coefficients would require the evaluation the formula $c_i = \sum_{j+k=i} a_j b_k$ for $i = 0, \ldots, 2n-2$, for a total of n^2 scalar products. A faster way to compute it is, first, to find the representations of the polynomials in the value domain with FFT: $f(\omega^i)$ and $g(\omega^i)$ for $i = 0, \ldots, 2n-1$; second, to multiply the corresponding values to obtain the representation of the product in the value domain: $f(\omega^i)g(\omega^i)$; and third, to return to the coefficient domain iFT. In this way we need to perform only $2n$ multiplications plus the $O(n \log n)$ multiplications needed for the conversions. The overall complexity of the multiplication algorithm is then also $O(n \log n)$, instead of quadratic.

The rest of the article describes the formalization of these ideas in type theory, using the proof tool Coq. In Section 2 we discuss the different representations for the data types involved. We choose a tree representation that is specifically designed for FFT. We prove that it is equivalent to a more natural one. In Section 3 we apply the two-level approach (see [3] and [6]) to the tree representation to prove some basic facts about it. Section 4 contains the definition and proof of correctness of FFT. Section 5 introduces iFT and proves its correctness. Finally, Section 6 discusses the tools used in the formalization and some implementation issues.

2 Data Representation

Let us fix the domain of coefficients and values. We need to work in a commutative ring with unity, and in the case of the inverse transform we will need a field. Usually it is required that the domain is an algebraically closed field, because the transform is applied to a polynomial and a root of unity. We will not do that, but just require that, when the algorithm is applied, a proof that its argument is a root of unity is supplied. This covers the useful case in which we want to apply the algorithm to finite fields (that can never be algebraically closed) or finite rings. In type theory an algebraic structure like that of ring is represented by an underlying *setoid*, which is a type with an equivalence relation, plus some operations on the setoid and proofs of the defining properties of the structure. In our case we will have the following data:

K : Set
K_eq : $K \to K \to$ Prop
K_eq_refl : (reflexive K_eq)
K_eq_symm : (symmetric K_eq)
K_eq_trans : (transitive K_eq).

K_eq is thus an equivalence relation that expresses the equality of the objects represented by terms of the type K. We will write $x \equiv y$ for (K_eq a b). The basic ring constants and operations are

$$0, 1 \colon K$$
$$\mathsf{sm}, \mathsf{ml} \colon K \to K \to K$$
$$\mathsf{sm\_inv} \colon K \to K.$$

We will use the notations $x+y$, $x \cdot y$ and $-x$ for (sm x y), (ml x y) and (sm_inv x), respectively. We need to require that they are well behaved with respect to $\equiv$, or, equivalently, that $\equiv$ is a congruence relation with respect to these operations:

$$\mathsf{sm\_congr} \colon \forall x_1, x_2, y_1, y_2 \colon K. x_1 \equiv x_2 \to y_1 \equiv y_2 \to x_1 + y_1 \equiv x_2 + y_2$$
$$\mathsf{ml\_congr} \colon \forall x_1, x_2, y_1, y_2 \colon K. x_1 \equiv x_2 \to y_1 \equiv y_2 \to x_1 \cdot y_1 \equiv x_2 \cdot y_2$$
$$\mathsf{sm\_congr} \colon \forall x_1, x_2 \colon K. x_1 \equiv x_2 \to (-x_1) \equiv (-x_2)$$

The axioms of commutative rings can now be formulated for these operations and for the equality $\equiv$.

We will often require that a certain element $\omega \colon K$ is a primitive n-root of unity. This means that $\omega^n \equiv 1$ (ω is a root) and n is the smallest non-zero element for which this happens (primitivity). At some point we will also need that, if n is even, $\omega^{n/2} \equiv -1$. Note that this fact does not generally follow from ω being a primitive n-root of unity. Indeed, if K is an integral domain, we can prove this from the fact that $\omega^{n/2}$ is a solution of the equation $x^2 \equiv 1$, but cannot be 1 by primitivity of ω. In an integral domain 1 and -1 are the only solutions of the equation. But in a commutative ring this is not always true. For example, take the finite commutative ring $\mathbb{Z}_{15}$ and the value $\omega = 4$, which is a primitive 2-root of unity ($4^2 \equiv 16 \equiv 1$). Nevertheless $4^1 \equiv 4 \not\equiv -1$. So the requirement $\omega^{n/2} \equiv -1$ will have to be stated explicitly when needed.

Polynomials in one variable are usually represented as the vectors of their coefficients. So the polynomial $a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$ is represented by the vector $\langle a_0, a_1, \ldots, a_{n-1} \rangle$. We can choose to implement such a vector in three standard ways: as a list (in which case there is no restriction on the length), as an element of a type with fixed length, or as a function from the finite type with n elements, $\mathbb{N}_n$, to K. This last representation is the most elastic for a certain number of purposes, specifically for reasoning about summations. It will be useful when prove the correctness of iFT. So we adopt it as our basic implementation of polynomials. The type of polynomials of degree $n-1$ will then be $\mathbb{N}_n \to K$.

However, in the proof of correctness of FFT, a different representation results more useful. A fundamental step of the computation consists in breaking the polynomial f in two parts f_e and f_o, consisting of the even and odd terms, respectively. We apply the algorithm recursively on these two polynomials. The recursion is wellfounded because f_e and f_o have smaller degree than f. This requires the use of course-of-value recursion on the degree of the polynomial, which can be realized in type theory using a general method for wellfounded recursion (see [1]). The formalization of the algorithm that we obtain is more complex than necessary, because it contains the proofs of the fact that the degrees

decrease. Simpler algorithms are obtained by using structural recursion in place of wellfounded recursion. To use structural recursion we need that the algorithm calls itself recursively only on structurally smaller arguments.

So we are led to look for a different implementation of polynomials whose structure reflects the steps of the algorithm. A general method to obtain a data type whose structure is derived from the recursive definition of a function is presented in [7]. In our case we obtain the result by representing polynomials as tree structures in which the left subtree contains the even coefficients and the right subtree contains the odd coefficients. This results in the recursive definition

$$\begin{aligned}&\mathsf{Tree}\colon \mathbb{N} \to \mathsf{Set} :=\\ &\mathsf{Tree}(0) := K\\ &\mathsf{Tree}(k+1) := \mathsf{Tree}(k) \times \mathsf{Tree}(k).\end{aligned}$$

An element of $\mathsf{Tree}(k)$ is a binary tree of depth k whose leaves are elements of the coefficient domain K. We will use the notation $\mathsf{leaf}(a)$ to denote an element $a\colon K$ when it is intended as a tree of depth 0, that is, an element of $\mathsf{Tree}(0) = K$. We will use the notation $\mathsf{node}(t_1, t_2)$ to denote the element $\langle t_1, t_2\rangle\colon \mathsf{Tree}(k+1) = \mathsf{Tree}(k) \times \mathsf{Tree}(k)$, if t_1 and t_2 are elements of $\mathsf{Tree}(k)$. The number of leaves of such a tree is 2^k. This is not a problem since, for the application of FFT, we always assume that the degree of the input polynomial is one less than a power of 2. Otherwise we adjust it to the closest power of 2 by adding terms with zero coefficients.

The equality $\equiv$ on K is extended to the equality $\cong$ on trees. We say that two elements $t_1, t_2\colon \mathsf{Tree}(k)$ are equal, and write $t_1 \cong t_2$, when the relation $\equiv$ holds between corresponding leaves. The relation $\cong$ can be formally defined by recursion on k.

A polynomial is represented by putting the coefficients of the even powers of x in the left subtree and the coefficients of the odd powers of x in the right one, and this procedure is repeated recursively on the two subtrees. If we have, for example, a polynomial of degree 7 $(= 2^3 - 1)$,

$$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7,$$

we break it into two parts

$$f_e(y) := a_0 + a_2y + a_4y^2 + a_6y^3, \quad f_o(y) := a_1 + a_3y + a_5y^2 + a_7y^3$$

so that $f(x) = f_e(x^2) + xf_o(x^2)$. Recursively $f_e(y)$ and $f_o(y)$ can be broken into even and odd terms

$$\begin{aligned} f_{ee}(z) &:= a_0 + a_4z, \quad f_{eo}(z) := a_2 + a_6z,\\ f_{oe}(z) &:= a_1 + a_5z, \quad f_{oo}(z) := a_3 + a_7z\end{aligned}$$

so that $f_e(y) = f_{ee}(y^2) + yf_{eo}(y^2)$ and $f_o(y) = f_{oe}(y^2) + yf_{oo}(y^2)$. With another step we reach single coefficients:

$$\begin{aligned} f_{eee}(u) &:= a_0, \quad f_{eeo}(u) := a_4, \quad f_{eoe}(u) := a_2, \quad f_{eoo}(u) := a_6,\\ f_{oee}(u) &:= a_1, \quad f_{oeo}(u) := a_5, \quad f_{ooe}(u) := a_3, \quad f_{ooo}(u) := a_7\end{aligned}$$

with $f_{ee}(z) = f_{eee}(z^2) + zf_{eeo}(z^2)$, $f_{eo}(z) = f_{eoe}(z^2) + zf_{eoo}(z^2)$, $f_{oe}(z) = f_{oee}(z^2) + zf_{oeo}(z^2)$, $f_{oo}(z) = f_{ooe}(z^2) + zf_{ooo}(z^2)$. Now we transform each of these polynomials in trees, starting with the single-coefficient ones. We simply obtain

$$\begin{array}{ll} t_{eee} := \mathsf{leaf}(a_0) = a_0, & t_{eeo} := \mathsf{leaf}(a_4) = a_4, \\ t_{eoe} := \mathsf{leaf}(a_2) = a_2, & t_{eoo} := \mathsf{leaf}(a_6) = a_6, \\ t_{oee} := \mathsf{leaf}(a_1) = a_1, & t_{oeo} := \mathsf{leaf}(a_5) = a_5, \\ t_{ooe} := \mathsf{leaf}(a_3) = a_3, & t_{ooo} := \mathsf{leaf}(a_7) = a_7. \end{array}$$

The polynomials of degree one are then represented by the trees

$$\begin{array}{ll} t_{ee} := \mathsf{node}(t_{eee}, t_{eeo}) = \langle a_0, a_4 \rangle, & t_{eo} := \mathsf{node}(t_{eoe}, t_{eoo}) = \langle a_2, a_6 \rangle, \\ t_{oe} := \mathsf{node}(t_{oee}, t_{oeo}) = \langle a_1, a_5 \rangle, & t_{oo} := \mathsf{node}(t_{ooe}, t_{ooo}) = \langle a_3, a_7 \rangle. \end{array}$$

The polynomials of degree three are represented by

$$\begin{array}{l} t_e := \mathsf{node}(t_{ee}, t_{eo}) = \langle \langle a_0, a_4 \rangle, \langle a_2, a_6 \rangle \rangle, \\ t_o := \mathsf{node}(t_{oe}, t_{oo}) = \langle \langle a_1, a_5 \rangle, \langle a_3, a_7 \rangle \rangle. \end{array}$$

Finally, the original polynomial is represented by

$$t := \mathsf{node}(t_e, t_o) = \langle \langle \langle a_0, a_4 \rangle, \langle a_2, a_6 \rangle \rangle, \langle \langle a_1, a_5 \rangle, \langle a_3, a_7 \rangle \rangle \rangle.$$

It is clear that the two representations are equivalent, in the sense that the types $\mathsf{Tree}(k)$ and $\mathbb{N}_{2^k} \to K$ are isomorphic, with the isomorphism outlined above.

The type $\mathsf{Tree}(k)$ is similar to the structure of powerlists by Misra [11], used by Gamboa in the verification of FFT in ACL2 [10]. The difference consists in the fact that powerlists are presented as an abstract data type that can be constructed and read in two different ways: by concatenation or by interleaving. It is not specified how powerlists are actually represented. One could implement them as simple lists or as a structure like $\mathsf{Tree}(k)$. The important fact is that there are functions doing and undoing the two different construction methods, and that we have corresponding recursion and induction principles. Here we made the choice of committing to the particular representation $\mathsf{Tree}(k)$ and keep it both for polynomials and for argument lists, avoiding costly representation transformations. Instead of using the normal list operations we have then to define equivalent ones for the tree types.

We go on to the definition of DFT. It is defined simply as the vector of evaluations of a polynomial on the powers of an argument, that is, if f is a polynomial of degree $2^k - 1$ and w is an element of K, we want that $\mathsf{DFT}(f, w) = \langle f(w^0), f(w^1), f(w^2), \ldots, f(\omega^{2^k-1}) \rangle$. For consistency we also want that this result vector is represented in the same form as the polynomials, that is, as an element of $\mathbb{N}_{2^k} \to K$ in the first representation and as an element of $\mathsf{Tree}(k)$ in the second representation with the values interleaved in the same way as the coefficient. For example if $k = 3$ we want that

$$\mathsf{DFT}(f, w) = \langle \langle \langle f(w^0), f(w^4) \rangle, \langle f(w^2), f(w^6) \rangle \rangle, \langle \langle f(w^1), f(w^5) \rangle, \langle f(w^3), f(w^7) \rangle \rangle \rangle.$$

The proof that the two definitions of DFT for the two representations are equivalent via the isomorphism of the types $\mathsf{Tree}(k)$ and $\mathbb{N}_{2^k} \to K$ is straightforward.

3 The Two-Level Approach for Trees

We need some operations on the type of trees and tools that facilitate reasoning about them. First of all, we define the mapping of the operations of the domain K on the trees: When we write $t_1 \circ t_2$ with $t_1, t_2 \colon \mathsf{Tree}(k)$ and $\circ$ one of the binary operations of K $(\cdot, +, -)$, we mean that the operation must be applied to pairs of corresponding leaves on the two trees, to obtain a new tree of the same type. Similarly, the unary operation of additive inversion $(-)$ will be applied to each leaf of the argument tree. Multiplication by a scalar, also indicated by $\cdot$, is the operation that takes an element a of K and a tree t and multiplies a for each of the leaves of t.

We denote the evaluation of a polynomial with $(t \xrightarrow{e} w)$, meaning "the value given by the evaluation of the polynomial represented by the tree t in the point w":

$$\begin{aligned}
&(_ \xrightarrow{e} _) \colon \mathsf{Tree}(k) \times K \to K \\
&(\mathsf{leaf}(a) \xrightarrow{e} w) := a \\
&(\mathsf{node}(t_1, t_2) \xrightarrow{e} w) := (t_1 \xrightarrow{e} w^2) + w \cdot (t_2 \xrightarrow{e} w^2).
\end{aligned}$$

The evaluation operation can be extended to evaluate a polynomial in all the leaves of a tree. This extension is achieved by mapping $\xrightarrow{e}$ to the trees in a way similar to the basic ring operations,

$$(\_ \xrightarrow{e\!\!\!\!\rightarrow} \_) \colon \mathsf{Tree}(k) \to \mathsf{Tree}(h) \to \mathsf{Tree}(h).$$

Two other operations that we need are the duplication of the leaves of a tree, in which every leaf is replaced by a tree containing two copies of it, and a duplication with change of the sign of the second copy:

$$\begin{aligned}
&\Downarrow \colon \mathsf{Tree}(k) \to \mathsf{Tree}(k+1) && \pm \colon \mathsf{Tree}(k) \to \mathsf{Tree}(k+1) \\
&\Downarrow \mathsf{leaf}(a) := \mathsf{node}(a, a) && \pm \mathsf{leaf}(a) := \mathsf{node}(a, -a) \\
&\Downarrow \mathsf{node}(t_1, t_2) := \mathsf{node}(\Downarrow t_1, \Downarrow t_2), && \pm \mathsf{node}(t_1, t_2) := \mathsf{node}(\pm t_1, \pm t_2).
\end{aligned}$$

We use $\pm$ also as a binary operator: We write $t_1 \pm t_2$ for $\Downarrow t_1 + (\pm t_2)$.

Given any scalar $x \colon K$, we want to generate the tree containing the powers of x in the interleaved order. So, for example for $k = 3$, we want to obtain

$$\langle\langle\langle x^0, x^4\rangle, \langle x^2, x^6\rangle\rangle, \langle\langle x^1, x^5\rangle, \langle x^3, x^7\rangle\rangle\rangle.$$

This is achieved by the function

$$\begin{aligned}
&(_ \Uparrow \text{-}) \colon K \times \mathbb{N} \to \mathsf{Tree}(k) \\
&(x \Uparrow^0) := \mathsf{leaf}(1) \\
&(x \Uparrow^{k+1}) := \mathsf{node}(t, x \cdot t) \text{ with } t := (x^2 \Uparrow^k).
\end{aligned}$$

To facilitate reasoning about trees, we use the method known as *the two-level approach* (see, for example, [3] and [6]). It is a general technique to automate the proof of a class of goals by internalizing it as an inductive type. Suppose we want to implement a decision procedure for a class of goals $\mathcal{G} \subseteq \mathsf{Prop}$. First

of all we define a type of codes for the goals, $\mathsf{goal}\colon \mathsf{Set}$, and an interpretation function $[\![\_]\!]\colon \mathsf{goal} \to \mathsf{Prop}$, such that the image of the whole type goal is $\mathcal{G}$. Then we define a decision procedure as a function $\mathsf{dec}\colon \mathsf{goal} \to \mathsf{bool}$ and we prove that it is correct, that is, $\mathsf{correctness}\colon \forall g\colon \mathsf{goal}.\mathsf{dec}(g) = \mathsf{true} \to [\![g]\!]$. As a consequence, whenever we want to prove a goal $P \in \mathcal{G}$, we can do it with just one command, $\mathsf{correctness}(g)(\mathsf{eq\_refl}(\mathsf{true}))$, where g is the code corresponding to P. In the case of equational reasoning the method can be further refined. We have a certain type of objects $T\colon \mathsf{Set}$, and a certain number of operations over it. We want to be able to prove the equality of two expressions of type T build from constants and variables using the operations. We define a type $\mathsf{code}^T\colon \mathsf{Set}$ of codes for such expressions. It is defined as an inductive type having as basis names for the constants and variables, and as constructors names for the operations. Then we define an interpretation function that associate an object of T to a code under a certain assignment of values to the variables, that is, if α is an assignment that associates an element of T to every variable, and $c\colon \mathsf{code}^T$ is a code, we obtain an element $[\![c]\!]_\alpha\colon T$. We can now use the syntactic level code^T to implement different tactics to decide equality of terms. For example, we may have a simplifying function $\mathsf{simplify}\colon \mathsf{code}^T \to \mathsf{code}^T$. If we prove that the simplification does not change the interpretation of the term, $\forall c\colon \mathsf{code}^T.\forall \alpha.[\![c]\!]_\alpha = [\![\mathsf{simplify}(c)]\!]_\alpha$, then we can easily prove the equality of two terms by simply checking if the simplifications of their codes are equal.

A very helpful use of the two-level approach consists in proving equalities obtained by substitution of equal elements in a context. Suppose that we need to prove $C[\cdots a \cdots] = C[\cdots b \cdots]$ for a certain context $C[\cdots]$ and two objects a and b of which we know that they are equal. If the equality considered is Leibniz equality, then the goal can be proved by rewriting. But if we are using a defined equality (a generic equivalence relation), this method will not work and we will have to decompose the context $C[\cdots]$ and apply various times the proofs that the operations preserve equality. If the context is complex, this may result in very long and tedious proofs. We want to be able to solve such goals in one step. This can be done by simply encoding the context as an element of code^T containing a variable corresponding to the position of the objects a and b. If we have a proof that the interpretation of a code does not change when we assign equal values to a variable, we are done.

We apply this general method to trees. The objects that we are considering, trees, do not form a single type but a family of types indexed on the natural numbers according to their depth. This means that also the type of tree codes needs to be parameterized on the depth. Also the variables appearing inside a tree expression can have different depths, not necessarily equal to the depth of the whole expression. Having expressions in which several variables of different depth may appear would create complications that we do not need or desire. Instead we implement expressions in which only one variable appears, in other words, we formalize the notion of a context with a hole that can be filled with different trees. Be careful not to confuse this kind of variable, a hole in the context, from a regular variable of type $\mathsf{Tree}(k)$, that at the syntactic level is treated as a

constant. We could call the hole variables *metavariables*, but contrary to usual metavariables, there can be occurrences of at most one hole variable in a term. Here is the definition of tree expressions, it has two natural-number parameters, the first for the depth of the metavariable, the second for the depth of the whole tree:

$$
\begin{array}{ll}
\mathsf{Inductive\ Tree\_exp}\colon \mathbb{N} \to \mathbb{N} \to \mathsf{Set} := & \\
\mathsf{trex\_var}(k)\colon (\mathsf{Tree\_exp}\ k\ k) & \text{for } k\colon \mathbb{N} \\
\mathsf{trex\_const}(h,k,t)\colon (\mathsf{Tree\_exp}\ h\ k) & \text{for } h,k\colon \mathbb{N}; t\colon \mathsf{Tree}(k) \\
\mathsf{trex\_leaf}(h,x)\colon (\mathsf{Tree\_exp}\ h\ 0) & \text{for } h\colon \mathbb{N}; x\colon K \\
\mathsf{trex\_node}(h,k,e_1,e_2)\colon (\mathsf{Tree\_exp}\ h\ k+1) & \text{for } h,k\colon \mathbb{N}; e_1,e_2\colon (\mathsf{Tree\_exp}\ h\ k) \\
\mathsf{trex\_sm}(h,k,e_1,e_2)\colon (\mathsf{Tree\_exp}\ h\ k) & \text{for } h,k\colon \mathbb{N}; e_1,e_2\colon (\mathsf{Tree\_exp}\ h\ k) \\
\mathsf{trex\_ml}(h,k,e_1,e_2)\colon (\mathsf{Tree\_exp}\ h\ k) & \text{for } h,k\colon \mathbb{N}; e_1,e_2\colon (\mathsf{Tree\_exp}\ h\ k) \\
\mathsf{trex\_mn}(h,k,e_1,e_2)\colon (\mathsf{Tree\_exp}\ h\ k) & \text{for } h,k\colon \mathbb{N}; e_1,e_2\colon (\mathsf{Tree\_exp}\ h\ k) \\
\mathsf{trex\_sc\_ml}(h,k,x,e)\colon (\mathsf{Tree\_exp}\ h\ k) & \text{for } h,k\colon \mathbb{N}; x\colon K; e\colon (\mathsf{Tree\_exp}\ k\ h) \\
\mathsf{trex\_pow}(h,k,x,e)\colon (\mathsf{Tree\_exp}\ h\ k) & \text{for } h\colon \mathbb{N}; x\colon K; k\colon \mathbb{N} \\
\mathsf{trex\_dupl}(h,k,e)\colon (\mathsf{Tree\_exp}\ h\ k+1) & \text{for } h,k\colon \mathbb{N}; e\colon (\mathsf{Tree\_exp}\ h\ k) \\
\mathsf{trex\_id\_inv}(h,k,e)\colon (\mathsf{Tree\_exp}\ h\ k+1) & \text{for } h,k\colon \mathbb{N}; e\colon (\mathsf{Tree\_exp}\ h\ k) \\
\mathsf{trex\_sm\_mn}(h,k,e_1,e_2)\colon (\mathsf{Tree\_exp}\ h\ k+1) & \text{for } h,k\colon \mathbb{N}; e_1,e_2\colon (\mathsf{Tree\_exp}\ h\ k) \\
\mathsf{trex\_eval}(h,k_1,k_2,e_1,e_2)\colon (\mathsf{Tree\_exp}\ h\ k_2) & \text{for } h,k_1,k_2\colon \mathbb{N}; e_1\colon (\mathsf{Tree\_exp}\ h\ k_1); \\
 & \qquad e_2\colon (\mathsf{Tree\_exp}\ h\ k_2).
\end{array}
$$

Each of the constructors of Tree_exp corresponds to an operation on trees, except trex_var, that introduces a metavariable, and trex_const that *quotes* an actual tree inside an expression. We now define the interpretation function that takes a tree expression, a tree in which the metavariable must be interpreted, of depth equal to the first argument of the tree expression type, and gives a tree as a result. We omit the first two arguments of the function, the natural-number parameters h and k, because they can be inferred from the types of the other arguments.

$$
\begin{array}{l}
[\![\_]\!]\_\colon (h,k\colon \mathbb{N})(\mathsf{Tree\_exp}\ h\ k) \to \mathsf{Tree}(h) \to \mathsf{Tree}(k) \\
[\![\mathsf{trex\_var}(k)]\!]_s := s \\
[\![\mathsf{trex\_const}(h,k,t)]\!]_s := t \\
[\![\mathsf{trex\_leaf}(h,x)]\!]_s := \mathsf{leaf}(x) = x \\
[\![\mathsf{trex\_node}(h,k,e_1,e_2)]\!]_s := \mathsf{node}([\![e_1]\!]_s, [\![e_2]\!]_s) = \langle [\![e_1]\!]_s, [\![e_2]\!]_s \rangle \\
[\![\mathsf{trex\_sm}(h,k,e_1,e_2)]\!]_s := [\![e_1]\!]_s + [\![e_2]\!]_s \\
[\![\mathsf{trex\_ml}(h,k,e_1,e_2)]\!]_s := [\![e_1]\!]_s \cdot [\![e_2]\!]_s \\
[\![\mathsf{trex\_mn}(h,k,e_1,e_2)]\!]_s := [\![e_1]\!]_s - [\![e_2]\!]_s \\
[\![\mathsf{trex\_sc\_ml}(h,k,x,e_2)]\!]_s := x \cdot [\![e_2]\!]_s \\
[\![\mathsf{trex\_pow}(h,x,k)]\!]_s := x \Uparrow^k \\
[\![\mathsf{trex\_dupl}(h,k,e)]\!]_s := \downdownarrows [\![e]\!]_s \\
[\![\mathsf{trex\_id\_inv}(h,k,e)]\!]_s := \pm [\![e]\!]_s \\
[\![\mathsf{trex\_sm\_mn}(h,k,e_1,e_2)]\!]_s := [\![e_1]\!]_s \pm [\![e_2]\!]_s \\
[\![\mathsf{trex\_eval}(h,k_1,k_2,e_1,e_2)]\!]_s := ([\![e_1]\!]_s \overset{e}{\twoheadrightarrow} [\![e_2]\!]_s)
\end{array}
$$

The most important use of this setup is in proving equality of substitutions inside a context.

Theorem 1 (Tree Reflection). *Let $h, k\colon \mathbb{N}$; for every context, given as a tree expression $e\colon$ (Tree_exp h k), and for every pair of trees $t_1, t_2\colon$ Tree(h); if $t_1 \cong t_2$, then the interpretations of the context under t_1 and t_2 are equal, $[\![e]\!]_{t_1} \cong [\![e]\!]_{t_2}$.*

Proof: A routine induction on the structure of e, using the proofs that the various operations considered preserve tree equality.

This theorem has been formalized in Coq with the name tree_reflection and is the most powerful tool in the development and proof of correctness of FFT. Whenever we need to prove a goal in the form $C[\cdots a \cdots] \cong C[\cdots a \cdots]$ with trees a and b for which we have a proof $p\colon a \cong b$, we can do it with the single command tree_reflection(h, k, e, a, b, p), where e is an encoding of C as an element of (Tree_exp h k).

This method has been repeatedly used in the formal proofs of the following lemmas, important steps towards the proof of correctness of FFT.

Lemma 1 (scalar_ml_tree_ml). *For every $x_1, x_2\colon K$ and for every pair of trees $t_1, t_2\colon$ Tree(k),*

$$(x_1 \cdot x_2) \cdot (t_1 \cdot t_2) \cong (x_1 \cdot t_1) \cdot (x_2 \cdot t_2).$$

(The operator $\cdot$ is overloaded and has three different meanings.)

Lemma 2 (tree_ml_tree_ml). *For every quadruple of trees $t_1, t_2, s_1, s_2\colon$ Tree(k),*

$$(t_1 \cdot t_2) \cdot (s_1 \cdot s_2) \cong (t_1 \cdot s_1) \cdot (t_2 \cdot s_2).$$

Lemma 3 (pow_tree_square). *For every $k\colon \mathbb{N}$ and $x\colon K$,*

$$(x \cdot x)\Uparrow^k \cong (x\Uparrow^k \cdot x\Uparrow^k).$$

(Here also $\cdot$ has a different meaning on the left-hand and right-hand side.)

Lemma 4 (eval_duplicate). *For every polynomial represented as a tree $t\colon$ Tree(k) and for every vector of arguments represented as a tree $u\colon$ Tree(h), the evaluation of t on the tree obtained by duplicating the leaves of u is equal to the tree obtained by duplicating the leaves of the evaluation of t on u,*

$$(t \overset{e}{\twoheadrightarrow} \downdownarrows u) \cong \downdownarrows (t \overset{e}{\twoheadrightarrow} u).$$

Lemma 5 (tree_eval_step). *A polynomial represented by the tree composed of its even and odd halves, $t = \mathsf{node}(t_e, t_o)\colon$ Tree$(k+1)$ is evaluated on a tree of arguments $u\colon$ Tree(h) by the equality*

$$(\mathsf{node}(t_e, t_o) \overset{e}{\twoheadrightarrow} u) \cong (t_e \overset{e}{\twoheadrightarrow} u \cdot u) + u \cdot (t_o \overset{e}{\twoheadrightarrow} u \cdot u).$$

(Note that this rule is simply the recursive definition of evaluation when we have a single element of K as argument, but it needs to be proved when the argument is a tree.)

Lemma 6. *Let* $k\colon \mathbb{N}$, $t, t_1, t_2, s_1, s_2\colon \mathsf{Tree}(k)$, *and* $x\colon K$. *The following equalities hold:*

$$\begin{aligned}
&\mathsf{sm_mn_duplicate_id_inv}\colon && t_1 \pm t_2 \cong (\Downarrow t_1) + (\pm t_2);\\
&\mathsf{ml_id_inv_duplicate}\colon && \pm t_1 \cdot \Downarrow t_2 \cong \pm(t_1 \cdot t_2);\\
&\mathsf{scalar_ml_in_inv}\colon && x \cdot (\pm t) \cong \pm(x \cdot t);\\
&\mathsf{scalar_ml_duplicate}\colon && x \cdot (\Downarrow t) \cong \Downarrow(x \cdot t);\\
&\mathsf{node_duplicate}\colon && \mathsf{node}(\Downarrow t_1, \Downarrow t_2) \cong \Downarrow \mathsf{node}(t_1, t_2).
\end{aligned}$$

The method of tree reflection and the above lemmas are extensively used in the following sections.

4 Definition and Correctness of FFT

We have build enough theory to obtain a short formulation of FFT and to prove its correctness.

Definition 1 (FFT). *The algorithm computing the Fast Fourier Transform of a polynomial represented by a tree* $t\colon \mathsf{Tree}(k)$ *(polynomial of degree* $2^k - 1$*) on the roots of unity generated by a primitive* 2^k*-root* $w\colon K$ *is given by the type-theoretic function*

$$\begin{aligned}
&\mathsf{FFT}\colon (k\colon \mathbb{N})\mathsf{Tree}(k) \to K \to \mathsf{Tree}(k)\\
&\mathsf{FFT}(0, \mathsf{leaf}(a_0), w) := \mathsf{leaf}(a_0)\\
&\mathsf{FFT}(k+1, \mathsf{node}(t_1, t_2), w) := \mathsf{FFT}(k, t_1, w^2) \pm (w\Uparrow^k \cdot \mathsf{FFT}(k, t_2, w^2))
\end{aligned}$$

We actually do not require that w is a root of unity, but we allow it to be any element of K to keep the definition of the algorithm simple. The correctness statement will hold only when w is a primitive root of unity and states that FFT computes the same function as DFT.

Definition 2 (DFT). *The Discrete Fourier Transform of a polynomial represented by a tree* $t\colon \mathsf{Tree}(k)$ *(polynomial of degree* $2^k - 1$*) on the roots of unity generated by a primitive* 2^k*-root* $w\colon K$ *is given by the evaluation of the polynomial on every root*

$$\begin{aligned}
&\mathsf{DFT}\colon (k\colon \mathbb{N})\mathsf{Tree}(k) \to K \to \mathsf{Tree}(k)\\
&\mathsf{DFT}(k, t, w) := t \overset{e}{\twoheadrightarrow} w\Uparrow^k
\end{aligned}$$

The fundamental step in the proof of equivalence of the two functions consists in proving that DFT satisfies the equality expressed in the recursion step of FFT. The equivalence follows by induction on the steps of FFT, that is, by induction on the tree structure of the argument.

Lemma 7 (DFT_step). *Let* $t_1, t_2\colon \mathsf{Tree}(k)$ *and* ω *be a* 2^{k+1} *principal root of unity such that* $\omega^{2^k} = -1$*; then*

$$\mathsf{DFT}(k+1, \mathsf{node}(t_1, t_2), \omega) \cong \mathsf{DFT}(k, t_1, \omega^2) \pm (\omega\Uparrow^k \cdot \mathsf{DFT}(k, t_2, \omega^2)).$$

Proof: We prove the equality through the intermediate steps

$$\begin{aligned}\mathsf{DFT}(k+1,\mathsf{node}(t_1,t_2),\omega) &\cong (t_1 \overset{e}{\twoheadrightarrow} \omega^2 \Uparrow^{k+1}) + \omega \Uparrow^{k+1} \cdot (t_2 \overset{e}{\twoheadrightarrow} \omega^2 \Uparrow^{k+1}) \\ &\cong \Downarrow\!\Downarrow (t_1 \overset{e}{\twoheadrightarrow} \omega^2 \Uparrow^{k}) + \omega \Uparrow^{k+1} \cdot \Downarrow\!\Downarrow (t_2 \overset{e}{\twoheadrightarrow} \omega^2 \Uparrow^{k}) \\ &\cong \Downarrow\!\Downarrow \mathsf{DFT}(k,t_1,\omega^2) + \omega \Uparrow^{k+1} \cdot \Downarrow\!\Downarrow \mathsf{DFT}(k,t_2,\omega^2) \\ &\cong \mathsf{DFT}(k,t_1,\omega^2) \pm (\omega \Uparrow^{k} \cdot \mathsf{DFT}(k,t_2,\omega^2)).\end{aligned}$$

The first step follows from the definition of DFT and Lemma tree_eval_step. The other steps are proved using the lemmas from the previous section, the method of tree reflection, and induction on the structure of trees. In the last step the hypothesis $\omega^{2^k} = -1$ must be used.

Theorem 2 (FFT_correct)**.** *Let $k\colon \mathbb{N}$, $t\colon \mathsf{Tree}(k)$ representing a polynomial, and ω be a principal 2^k-root of unity with the property that $\omega^{2^{k-1}} = -1$; then*

$$\mathsf{FFT}(k,t,\omega) \cong \mathsf{DFT}(k,t,\omega).$$

Proof: By straightforward induction on the structure of t using the previous lemma in the recursive case.

5 The Inverse Fourier Transform

We formulate and prove the correctness of iFT. We need that K is a field, not just a commutative ring. This means that we assume that there is an operation

$$\mathsf{ml\_inv}\colon (x : K)x \not\equiv 0 \to K$$

satisfying the usual properties of multiplicative inverse. We can therefore define a division operation

$$\mathsf{dv}\colon (x, y\colon K)y \not\equiv 0 \to K.$$

Division is a function of three arguments: two elements of K, x and y, and a proof p that y is not 0. We use the notation x/y for $(\mathsf{dv}\ x\ y\ p)$, hiding the proof p.

The tree representation for polynomials is very useful for the verification of FFT, but the proof of correctness of iFT is easier with the function representation: A polynomial $f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$ is represented as a function $a\colon \mathbb{N}_n \to K$, and we write a_i for the value $a(i)$. We already proved in Section 2 that the two representations are equivalent, so we can freely switch between the two to develop our proofs. Let then DFT^f be the version of the Discrete Fourier Transform for the functional representation of polynomials. Here is the definition of the inverse transform.

Definition 3. *The* Inverse Discrete Fourier Transform *is the function that applies* DFT *to the multiplicative inverse of a root of unity and then divides by the degree:*

$$\begin{aligned}&\mathsf{iDFT}^f\colon (n : \mathbb{N})(\mathbb{N}_n \to K) \to (\omega\colon K)\omega \not\equiv 0 \to (\mathbb{N}_n \to K) \\ &\mathsf{iDFT}^f(n,a,\omega,p) := \tfrac{1}{n}\mathsf{DFT}(n,a,\omega^{-1}).\end{aligned}$$

Our goal is to prove that this function is indeed the inverse of DFT^f, that is, $\mathsf{iDFT}^f(n, \mathsf{DFT}(n, a, \omega), \omega) = a$. The proof of this fact follows closely the standard proof given in the computer algebra literature, so we do not linger over its details. We only point out the passages where extra work must be done to obtain a formalization in type theory. First of all, we need to define the summation of a vector of values of K. The vector is represented by a function $v\colon \mathbb{N}_n \to K$. The summation $\sum v$ is defined by recursion on n. The main reason for choosing the representation $\mathbb{N}_n \to K$, in place of $\mathsf{Tree}(k)$, is that we often need to exchange the order of double summations, that is, we need the equality

$$\sum_{i=0}^{n-1}\sum_{j=0}^{m-1} a_{ij} = \sum_{j=0}^{m-1}\sum_{i=0}^{n-1} a_{ij}.$$

With the function notation, this becomes very easy: $a\colon \mathbb{N}_n \to \mathbb{N}_m \to K$ and the equality is written

$$\sum \lambda i\colon \mathbb{N}_n.\sum a(i) \equiv \sum \lambda j\colon \mathbb{N}_m.\sum \lambda i\colon \mathbb{N}_n a(i)(j).$$

We can easily define a summation function for $\mathsf{Tree}(k)$, but in that case it becomes much more complicated to formulate a double summation. A matrix of values like $\{a_{ij}\}$ would have to be represented as a tree having trees for leaves. Then the swap of the indexes in the summation would correspond to lifting the tree structure of the leaves to the main tree and lowering the main tree structure to the leaves. This would require much more work than simply changing representation.

In the proof we make essential use of the formula for the summation of a geometric series, expressed in type theory by

Lemma 8 (geometric_series). *For every $n\colon \mathbb{N}$, $x\colon K$ such that $x \not\equiv 1$,*

$$\sum \lambda i\colon \mathbb{N}_n.x^i \equiv \frac{x^n - 1}{x - 1}$$

where we simplified notation treating i as an element of $\mathbb{N}$ (we should really write $x^{(\mathsf{fin\_to\_nat}(i))}$ in place of x^i, where $\mathsf{fin\_to\_nat}\colon \mathbb{N}_n \to \mathbb{N}$ is an embedding function).

Proof: Standard.

Lemma 9 (summation_root_delta). *For $n\colon \mathbb{N}$, $\omega\colon K$ a primitive n-root of unity, and $k, j\colon \mathbb{N}_n$,*

$$\sum \lambda i\colon \mathbb{N}_n.(\omega^i)^j \cdot ((\omega^{-1})^k)^i \equiv n\delta_{jk}$$

where δ_{ij}, the Kronecker symbol, is 1 if $i = j$, 0 otherwise. Once again we abused notation leaving out the application of the conversion function fin_to_nat. *On the right-hand side we used juxtaposition of the natural number n to an element of K to indicate the function giving the n-fold sum in K.*

Proof: Standard.

Finally we can prove the correctness of the inverse transform.

Theorem 3 (inverse_DFT). *Let* $n\colon \mathbb{N}$, $a\colon \mathbb{N}_n \to K$, $\omega\colon K$ *a primitive* n*-root of unity; then for every* $k\colon \mathbb{N}_n$,

$$\mathsf{DFT}^f(n, \mathsf{DFT}^f(n, a, \omega), \omega^{-1})_k \equiv n \cdot a_k.$$

Proof: Standard using the previous lemmas.

Once iDFT has been defined and proved correct for the functional representation of polynomials, we can use the equivalence of the two representations to obtain a proof of correctness of the version of the inverse transform for trees using the fast algorithm:

$$\mathsf{iFT}(n, t, \omega) := \frac{1}{n}\mathsf{FFT}(n, t, \omega^{-1}).$$

6 Conclusion

The definition and the proof of correctness for FFT and iFT have been completely formalized using the proof tool Coq. I have used the graphical interface CtCoq [4] to develop the formalization. CtCoq was extremely useful for several reasons. It affords extendable notation which allows the printing of terms in nice mathematical formalism, hiding parts that have no mathematical content (for example the applications of the commutation function fin_to_nat or the presence of a proof that the denominator is not zero in a division). The technique of *proof-by-pointing* [5] allows the user to construct complicated tactics with a few clicks of the mouse. It is easy to search for theorems and lemmas previously proved and apply them by just pressing a key. I found that the use of CtCoq increased the efficiency and speed of work, and I could complete the whole development in just a couple of months.

The proof uses two main techniques. First, instead of using the most natural data type to represent the objects on which the algorithm operates, we chose an alternative data type that makes it easier to reason about the computations by structural recursion. Second, we used the two-level approach to automate part of the generation of proofs of equality for tree expressions.

Future work will concentrate in controlling more carefully how the algorithm exploits time and space resources. It is known that FFT runs in $O(n \log n)$ time, but we didn't prove it formally in Coq. The problem here is that there is no formal study of algorithmic complexity in type theory. In general it is difficult to reason about the running time of functional programs, since the reduction strategy is not fixed. A solution could be achieved by translating FFT into a development of imperative programming and then reasoning about its complexity in that framework. Another point is the use of memory. One important feature of FFT is that it can be computed *in place*, that means that the memory occupied by the input data can be reused during the computation, without the need of extra memory. Also memory management is a sore point in functional programming. I think it is possible to use the *uniqueness* types of the programming language Clean [9] to force the algorithm to reuse the space occupied by the data.

References

1. Antonia Balaa and Yves Bertot. Fix-point equations for well-founded recursion in type theory. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2000.
2. Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, Henri Laulhère, César Muñoz, Chetan Murthy, Catherine Parent-Vigouroux, Patrick Loiseleur, Christine Paulin-Mohring, Amokrane Saïbi, and Benjanin Werner. *The Coq Proof Assistant Reference Manual. Version 6.3*. INRIA, 1999.
3. G. Barthe, M. Ruys, and H. P. Barendregt. A two-level approach towards lean proof-checking. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs (TYPES'95)*, volume 1158 of *LNCS*, pages 16–35. Springer, 1995.
4. Yves Bertot. The CtCoq system: Design and architecture. *Formal aspects of Computing*, 11:225–243, 1999.
5. Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In *Symposium on Theoretical Aspects Computer Software (STACS), Sendai (Japan)*, volume 789 of *LNCS*. Springer, April 1994.
6. Samuel Boutin. Using reflection to build efficient and certified decision procedures. In Martín Abadi and Takayasu Ito, editors, *Theoretical Aspects of Computer Software. Third International Symposium, TACS'97*, volume 1281 of *LNCS*, pages 515–529. Springer, 1997.
7. Ana Bove and Venanzio Capretta. Nested general recursion and partiality in type theory. `http://www.cs.kun.nl/ venanzio/publications/nested.ps.gz`, 2001.
8. James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, April 1965.
9. Paul de Mast, Jan-Marten Jansen, Dick Bruin, Jeroen Fokker, Pieter Koopman, Sjaak Smetsers, Marko van Eekelen, and Rinus Plasmeijer. *Functional Programming in Clean*. Computing Science Institute, University of Nijmegen.
10. Ruben A. Gamboa. The correctness of the Fast Fourier Transform: a structured proof in ACL2. *Formal Methods in System Design, Special Issue on UNITY*, 2001. in print.
11. Jayadev Misra. Powerlist: a structure for parallel recursion. *TOPLAS*, 16(6):1737–1767, November 1994.