

Mechanically Verifying the Correctness of the Fast Fourier Transform in ACL2

Ruben A. Gamboa*

LIM International, Inc.
9390 Research Blvd., Suite II-200
Austin, TX 78759, USA
e-mail: ruben@lim.com

Abstract. In [10], Misra introduced the powerlist data structure, which is well suited to express recursive, data-parallel algorithms. In particular, Misra showed how powerlists could be used to give simple descriptions to very complex algorithms, such as the Fast Fourier Transform (FFT). Such simplicity in presentation facilitates reasoning about the resulting algorithms, and in fact Misra was able to give a stunningly simple proof of the correctness of the FFT. In this paper, we show how this proof can be verified using ACL2. This strengthens Misra's case that powerlists provide a suitable framework in which to define and reason about parallel algorithms, particularly using mechanical tools to aid in reasoning.

1 Introduction

In [10], Misra gives a concise and simple proof of the correctness of the Fast Fourier Transform (FFT). The key to this simplicity is the data structure of powerlists, which allows the FFT to be defined using a recursive function, without resorting to index arithmetic, e.g., index reversal.

In this paper, we will show how ACL2 can be used to mechanically verify this result. We begin by presenting a brief summary of powerlists, which should give the reader unfamiliar with powerlist theory enough intuition to read the remainder of the paper. Following this review, we will present Misra's proof of the correctness of the FFT. Besides paying homage to this wonderful result, our goal here will be to try to understand the proof, so that we can formulate a plan for its mechanical verification. We will then present the mechanical proof itself. Our focus will be on the *process* of translating the hand proof into an ACL2 proof. We hope this will benefit those readers using ACL2 in their own research, as well as readers using other mechanical theorem provers.

2 A Review of Powerlists

A powerlist is a linear data structure, that is to say a list of elements. What differentiates powerlists from ordinary lists is that powerlists are constructed using parallel operators.

* Work was performed while author was a student at the University of Texas at Austin.

Recall that a list is constructed by inserting an element e to the list L , as in $e.L$. The drawback is that recursive functions over lists become inherently serial, since the function must process the first element e , and then the remainder of the list L .

In contrast, powerlists are constructed from other *powerlists*. Given the powerlists L_1 and L_2 , we can construct a new powerlist L in one of two ways. We can take first the elements from L_1 and then the elements from L_2 . This is called the “tie” of L_1 and L_2 , and is written $L_1 | L_2$. Alternatively, we can take an element from L_1 , followed by an element from L_2 , and so on. This is called the “zip” of L_1 and L_2 , and we write it as $L_1 \bowtie L_2$. In order for these operations to have unique inverses, we insist that L_1 and L_2 be of the same length. Thus, the length of a powerlist is always a power of two, if we take the expedient of disallowing empty powerlists.

The powerlist algebra provides the necessary axioms about powerlists. In particular, it shows how the operators $|$ and \bowtie interact. It is summarized below:

$$\begin{aligned} \forall p \{ \text{length}(p) > 1 \Rightarrow \exists u, v, r, s \{ p = u | v \wedge p = r \bowtie s \} \} \\ \langle a \rangle | \langle b \rangle &= \langle a \rangle \bowtie \langle b \rangle \\ \langle a \rangle &= \langle b \rangle \equiv a = b \\ p | q = u | v &\equiv p = u \wedge q = v \\ p \bowtie q = u \bowtie v &\equiv p = u \wedge q = v \\ (p | q) \bowtie (u | v) &= (p \bowtie u) | (q \bowtie v) \end{aligned}$$

The powerlist algebra can be used to define recursive functions on powerlists. For example, the *length* function can be defined as follows:

$$\begin{aligned} \text{length}(\langle x \rangle) &= 1 \\ \text{length}(p | q) &= \text{length}(p) + \text{length}(q) \end{aligned}$$

Notice how this definition, similar to the corresponding definition on regular lists, is more amenable to parallel computation. Since the powerlist constructors operate on lists of the same length, the corresponding destructors split a list into equal halves – a key component in writing divide and conquer algorithms. For more information on powerlists, the reader is referred to [10].

3 The Fast Fourier Transform

The Fourier transform of a real or complex vector $P = (p_1, p_2, p_3, \dots, p_n)$ is defined as $FT(P) = (\overline{P}(w_n), \overline{P}(w_n^2), \overline{P}(w_n^3), \dots, \overline{P}(w_n^n))$, where w_n is the n^{th} principal root of 1, and \overline{P} is the polynomial constructed from P by $\overline{P}(x) = \sum_{i=1}^n p_i \cdot x^{i-1}$.

Naively, the Fourier Transform of P can be computed in n^2 sequential steps, by evaluating $\overline{P}(x)$ at each of the n powers of w_n . This naive implementation will serve as our formal specification.

Following [10], we begin with the function ep which evaluates a polynomial P at a vector V . We will write ep in infix notation:

$$\langle x \rangle ep v = \langle x \rangle \tag{1}$$

$$(p \bowtie q) ep v = p ep v^2 + v \cdot (q ep v^2) \tag{2}$$

$$p ep (u | v) = (p ep u) | (p ep v) \tag{3}$$

Note that in the case $\langle x \rangle ep (u | v)$ we can proceed using either rule 3 or rule 1. Sadly, this will result in different answers. Thus, we tacitly restrict using rule 1 while rule 3 is applicable. We claim, without proof for now, that this is the only inconsistency, as long as the arithmetic operators used in rule 2 are assumed to apply pointwise to vectors. The Fourier transform can now be defined simply as

$$FT(p) = p ep W_n \quad (4)$$

where n is the length of p and $W_n = (w_n, w_n^2, \dots, w_n^n)$.

The FFT is an algorithm which evaluates the Fourier transform in $O(n \log n)$ sequential steps, by using the special properties of the vector of powers of w_n . In particular, let $W_n = (w_n, w_n^2, \dots, w_n^n)$, for $n > 1$ a power of two. Then, we find that W_n can be written as

$$\begin{aligned} W_n &= u | -u \\ W_{n/2} &= u^2 \end{aligned}$$

The first property is true because w_n is the n^{th} principal root of 1, and so $w_n^n = 1$ and therefore $w_n^{n/2} = -1$ (since w_n is the n^{th} principal root and $n/2$ is an integer less than n – recall that $n > 1$ is a power of two – $w_n^{n/2} \neq 1$). So, for any $n/2 < k \leq n$, we have that $w_n^k = w_n^{n/2} \cdot w_n^{k-n/2} = -w_n^{k-n/2}$. The second property is true because the first $n/2$ values of W_n are the first $n/2$ powers of the n^{th} principal root of 1. These are precisely the (principal) square roots of the $n/2$ powers of the $(n/2)^{\text{th}}$ principal root of 1, that is, $W_{n/2}$.

Since we are only interested in the powers of 2, it is convenient to use the notation $\overline{W}_N = W_{2^N}$. This way, we have the properties

$$\begin{aligned} \overline{W}_n &= u | -u \\ \overline{W}_{n-1} &= u^2 \end{aligned}$$

which are more amenable to induction.

We can derive the Fast Fourier Transform as follows. For singleton powerlists,

$$FT(\langle x \rangle) = \langle x \rangle ep \overline{W}_0 \quad (5)$$

$$= \langle x \rangle \quad (6)$$

since \overline{W}_0 is a singleton (equal to 1), and so we can use rule 1 of the definition of ep . For a powerlist of length $2^N > 1$, we have that

$$FT(p \bowtie q) = (p \bowtie q) ep \overline{W}_N \quad (7)$$

$$= (p \bowtie q) ep (u | -u) \quad (8)$$

$$= ((p \bowtie q) ep u) | ((p \bowtie q) ep -u) \quad (9)$$

$$= (p ep u^2 + u \cdot (q ep u^2)) | (p ep u^2 - u \cdot (q ep u^2)) \quad (10)$$

$$\begin{aligned} &= (p ep \overline{W}_{N-1} + u \cdot (q ep \overline{W}_{N-1})) | \\ &\quad (p ep \overline{W}_{N-1} - u \cdot (q ep \overline{W}_{N-1})) \end{aligned} \quad (11)$$

$$= (FT(p) + u \cdot FT(q)) | (FT(p) - u \cdot FT(q)) \quad (12)$$

Using these results, we can derive the Fast Fourier Transform as follows:

$$FFT(\langle x \rangle) = \langle x \rangle \quad (13)$$

$$FFT(p \bowtie q) = (FFT(p) + u \cdot FFT(q)) \mid (FFT(p) - u \cdot FFT(q)) \quad (14)$$

where the vector u contains the first $2^N/2$ elements of \overline{W}_N , and 2^N is the length of $p \bowtie q$. It is clear that $FFT(p \bowtie q)$ can be computed in $O(2^N)$ time from $FFT(p)$ and $FFT(q)$. Thus, it can be computed in $O(N2^N)$ (sequential) time, which is $O(n \log n)$ time, where $n = 2^N$ is the length of $p \bowtie q$. By unraveling the recursive calls, it is possible to synthesize a parallel circuit to implement the FFT. This requires $O(n \log n)$ computation nodes and $O(\log n)$ depth[4].

4 Reasoning About Powerlists in ACL2

ACL2 is a theorem prover over a total, first-order, quantifier-free logic. Historically, ACL2 derives from the Boyer-Moore theorem prover, or Nqthm, which established itself as the premier theorem over the natural numbers. Among the pioneering efforts in Nqthm were the automatic generation of induction plans from recursive functions and the “waterfall” design, which allowed conjectures to be successively modified in such a way as to permit a “clean” inductive proof. Moreover, Nqthm was based on an *executable* logic, specifically a simplified dialect of LISP, hence functions in Nqthm could be directly executed. These characteristics have been carried over into ACL2. It is fair to say that ACL2 grew out of a desire to “do Nqthm, only better” [8].

The syntax of ACL2 is essentially that of Common LISP, with `defun` as the primary way to introduce new function symbols into the logic, and `defthm` as the primary way to prove theorems about these functions. With its LISP heritage, carried over from Nqthm, it is no surprise that ACL2 is best suited to prove theorems by induction. That is, ACL2 proves theorems about recursive functions by finding suitable induction hypotheses. It is reasonable, therefore, to assume that ACL2 presents a natural vehicle for reasoning about powerlists, since powerlists are recursive data structures, and powerlist algorithms are mostly written as recursive functions, e.g., the definition of FFT in the previous section. An implementation of the powerlist logic in ACL2 is presented in [5]. We will use that framework in this paper. In the remainder of this section, we will present the necessary background.

First of all, we create powerlists using the operators `zip` (\bowtie) and `tie` (\mid). Since ACL2 uses the syntax of LISP, we cannot define functions in the pattern-matching style traditional in powerlists. This forces us to define the explicit powerlist destructors `p-untie-l` and `p-untie-r` which are defined so that `p-untie-l(p|q)` is equal to p and `p-untie-r(p|q)` is equal to q . Similarly, we define `p-unzip-l` and `p-unzip-r` to let us define functions in terms of `zip`. The only thing left is the type predicate `powerlist-p` which lets us differentiate powerlists from scalars; this corresponds to the scalar case in the regular powerlist notation, e.g., the definition of $FT(\langle x \rangle)$. For notational convenience, we will use a more traditional logical notation, instead of ACL2’s LISP syntax.

Departing from the tradition of powerlists, we think of powerlists not as linear lists, but as *trees* constructed with `tie`. Since ACL2 is a total logic, we do not restrict the

length of powerlists to the powers of two. Instead, a powerlist can be created with an arbitrary tie tree structure. Most theorems about powerlists continue to be true in this more general setting, but obviously some do not. The predicate `p-regular-p` recognizes the powerlists with a power of two length. More precisely, a powerlist is `p-regular-p` if its tie tree representation is a complete binary tree. Often, we will be interested in `p-similar-p` powerlists, which are those with isomorphic tie trees. Similar powerlists are the only ones for which we can reasonable define point-wise operators, e.g., a function to add two powerlists.

5 Verifying the Fast Fourier Transform in ACL2

In this section, we will translate the hand proof found in Sect. 3 into ACL2. We begin by translating the function `ep` into ACL2. Recall, the definition of `P ep V` was non-deterministic, in that it was possible to recurse based on the polynomial `P` or the vector `V`. We disambiguate in favor of the vector `V`, so we split `ep` into two functions, `eval-poly` and `eval-poly-at-point`. Their definitions are straightforward:

Definition.

```
eval-poly-at-point (p, x)
=
if powerlist-p (p)
  then  eval-poly-at-point (p-unzip-l (p), x · x)
        + x · eval-poly-at-point (p-unzip-r (p), x · x)
  else fix (p)
```

Definition.

```
eval-poly (p, x)
=
if powerlist-p (x)
  then eval-poly (p, p-untie-l (x)) | eval-poly (p, p-untie-r (x))
  else eval-poly-at-point (p, x)
```

We use `fix (p)` instead of simply `p` in the definition of `eval-poly-at-point` because we want the value returned to be numeric, even when `p` is not. This preserves the traditions of ACL2 that treat all non-numeric arguments to a numeric function as zero and force numeric functions to *always* return a numeric value.

The correctness proof uses not only the definition of `ep` over points, but also over vectors. In particular, the step

$$\begin{aligned} & ((p \bowtie q) \text{ ep } u) \mid ((p \bowtie q) \text{ ep } - u) & (9) \\ & = (p \text{ ep } u^2 + u \cdot (q \text{ ep } u^2)) \mid (p \text{ ep } u^2 - u \cdot (q \text{ ep } u^2)) & (10) \end{aligned}$$

uses polynomial versions of the arithmetic operators. ACL2 reserves the arithmetic operators for numbers only; in fact, `x · 1` is equal to zero for all non-numeric arguments `x`, including vectors represented as powerlists. So, we must define our own “arithmetic”

operators over powerlists: \oplus , \ominus , and \otimes for pairwise addition, subtraction and multiplication, respectively. With these operators, we can rewrite polynomial evaluation over vectors, using the following theorem:

Theorem (eval-poly-lemma).

$$\begin{aligned} & \text{powerlist-p}(p) \\ \rightarrow & \text{eval-poly}(p, x) \\ = & \text{eval-poly}(\text{p-unzip-l}(p), x \otimes x) \\ & \oplus x \otimes \text{eval-poly}(\text{p-unzip-r}(p), x \otimes x) \end{aligned}$$

The theorem eval-poly-lemma is almost sufficient to prove (10). However, (10) also uses properties of $-u$, such as $(-u)^2 = u^2$. To prove these facts in ACL2, we introduce unary minus on powerlists and prove some basic lemmas about its interaction with the other arithmetic operators:

Theorem.

$$(\ominus x) \otimes y = \ominus(x \otimes y)$$

Theorem (minus-times-minus).

$$\text{p-similar-p}(x, y) \rightarrow (\ominus x) \otimes (\ominus y) = x \otimes y$$

The first theorem is simple enough, stating how a unary minus in the first argument of a product can be factored out of the product. The second theorem seems odd, because of the p-similar-p requirement. It is needed because the function \otimes is defined in terms of the structure of the first argument, so it is possible that y will “run out” of terms before x does, in which case \otimes will recurse using the p-untie-l and p-untie-r of a non-powerlist object. Neither p-untie-l nor p-untie-r guarantee a particular value when applied to a non-powerlist; in fact, it is possible to choose values of x and y that invalidate the theorem without the p-similar-p hypothesis.

ACL2 prefers to see rewrite rules without any hypotheses. Since we are only interested in the square of powerlists, we can write a more specific rule:

Theorem.

$$(\ominus x) \otimes (\ominus x) = x \otimes x$$

This more special rule uses the fact that any powerlist is p-similar-p to itself.

Note, a given term may rewrite using any one of the above rules. Certainly, if the last rewrite rule applies, so will the two earlier ones. It is important, therefore, that the rules be given to ACL2 *in this order*. That is, the most specific rules should be given last, since the more recent rules are tried first.

The only remaining rule deals with unary minus and addition. In particular, we can prove that

Theorem (plus-minus).

$$\text{p-similar-p}(x, y) \rightarrow x \oplus (\ominus y) = x \ominus y$$

As before, the similarity requirement can not be relaxed.

We are ready to attempt the following theorem, justifying step 10 of the proof:

Theorem (eval-poly-u).

$$\begin{aligned}
& \text{powerlist-p}(x) \\
\rightarrow & \text{eval-poly}(x, u \mid \ominus u) \\
= & \text{eval-poly}(\text{p-unzip-l}(x), u \otimes u) \\
& \oplus u \otimes \text{eval-poly}(\text{p-unzip-r}(x), u \otimes u) \\
& \mid \text{eval-poly}(\text{p-unzip-l}(x), u \otimes u) \\
& \ominus u \otimes \text{eval-poly}(\text{p-unzip-r}(x), u \otimes u)
\end{aligned}$$

Unfortunately, this proof attempt fails. The reason is that the ACL2 rewriter will not use the rewrite rules about unary minus, because of the similarity requirement. For example, part of the proof requires $\text{eval-poly}(x, u \otimes u)$ to be similar to $u \otimes \text{eval-poly}(y, u \otimes u)$ which, while true, is not obvious to the ACL2 rewriter, and hence the rewrite rule taking $(x \text{ ep } u^2) + (-(u \cdot y \text{ ep } u^2))$ to the simpler $(x \text{ ep } u^2) - (u \cdot y \text{ ep } u^2)$ is not applied.

There are two solutions to this problem. The first is to add a number of rules to help ACL2 determine when two objects are similar. This approach is successful, but it results in a large number of tedious lemmas. ACL2 provides a more immediate approach: “forcing.” Essentially, ACL2 allows a hypothesis to be marked as “forceable,” which means that it is assumed true by the rewriter, allowing the proof to proceed. At the end of the proof, any forced hypotheses are tackled, using the full power of the theorem prover, not just the rewriter. To take advantage of this, the similarity conditions are marked as forceable in the theorems minus-times-minus and plus-minus. At this point, ACL2 proves eval-poly-u without a problem. It may be tempting to consider forcing as a panacea. Why not, one may ask, simply force all the hypotheses, allowing the theorem prover to proceed at blinding speed, only to discard those pesky hypotheses at a later time? There are two answers to that. First, if we use a rewrite rule with a false forced hypothesis, the proof attempt will subsequently fail – even if some *other* rewrite rule could have been applied at that time. This means that one should never force a hypothesis that is not expected to be “always” true, where by “always” we mean in the terms that the theorem prover will encounter. In our case, since we are dealing with similar powerlists, the p-similar-p hypothesis seems like a good candidate for forcing. There is a second caveat, however. In the forcing round, ACL2 does not restore *all* the facts that were available when the forced rewrite rule was used. In particular, it is possible for ACL2 to “drop” a hypothesis that will be needed when ACL2 attempts to prove the forced hypothesis.

We are now ready to consider the lists \overline{W}_n . The only properties of this function that we actually need are the following:

$$\begin{aligned}
\overline{W}_n &= u \mid -u \\
\overline{W}_{n-1} &= u^2
\end{aligned}$$

Since the function \overline{W}_n is quite complicated, involving powers of the principal $(2^n)^{\text{th}}$ power of 1, it is advantageous to pursue the proof at an abstract level, where the only known properties are the ones stated above. ACL2 provides a mechanism for this called “encapsulate.” Using encapsulate, we can soundly introduce new function symbols, without making their definition visible. Instead, these functions are identified only by

some of their properties, called constraints. Once a theorem is proved about a constrained function, it can be automatically proved about an arbitrary function, given that it satisfies all the constraints of the constrained function. The constraints are as follows:

Constraint.

$\text{acl2-numberp}(\text{p-omega}(0))$

Constraint.

$\neg \text{zp}(n) \rightarrow \text{p-omega}(n) = \text{p-omega-sqrt}(n-1) \mid \ominus \text{p-omega-sqrt}(n-1)$

Constraint.

$\text{p-omega-sqrt}(n) \otimes \text{p-omega-sqrt}(n) = \text{p-omega}(n)$

Note, the ACL2 idiom $\neg \text{zp}(n)$ is used to recognize the positive integers; it is traditional to use $\text{zp}(n)$ in recursive definitions.

The preceding event introduces the two constrained function symbols p-omega and p-omega-sqrt . It is important that the constrained functions are actually defined inside the encapsulate, because this allows ACL2 to verify that the constraints assumed about them are not contradictory, thus preventing a user from unknowingly (or deliberately) introducing unsoundness into his theory. It is a tradition of convenience to choose simple “witness” functions in place of the constrained functions. This simplifies the theorem proving requirement inside the encapsulate, while not affecting the remainder of the proof – for our purposes, the zero-tree function for p-omega suffices. Outside of the encapsulate, the only facts known about the constrained functions are the actual constraints. Note in particular that we had to define a specific function for u , since it’s a *different* u for each value of n . We call this function p-omega-sqrt , as suggested by the last constraint.

We can prove the following theorem, justifying step 9 in Misra’s proof:

Theorem (eval-poly-omega-n).

$$\begin{aligned}
 & \text{powerlist-p}(x) \wedge \neg \text{zp}(n) \\
 \rightarrow & \text{eval-poly}(x, \text{p-omega}(n)) \\
 = & \text{eval-poly}(\text{p-unzip-l}(x), \text{p-omega}(n-1)) \\
 & \oplus \text{p-omega-sqrt}(n-1) \otimes \text{eval-poly}(\text{p-unzip-r}(x), \text{p-omega}(n-1)) \\
 & \mid \text{eval-poly}(\text{p-unzip-l}(x), \text{p-omega}(n-1)) \\
 & \ominus \text{p-omega-sqrt}(n-1) \\
 & \otimes \text{eval-poly}(\text{p-unzip-r}(x), \text{p-omega}(n-1))
 \end{aligned}$$

Proving this theorem requires a hint to encourage ACL2 to use the rule converting $\text{p-omega-sqrt}(n-1)$ into its $u \mid -u$ equivalent, so that the theorem eval-poly-u can apply. We also need hints to keep ACL2 from considering lemmas relating to several functions. This is because the intermediate terms are so large, they contain many function instances which ACL2 would like to consider further – unfortunately, once ACL2 starts going down that path, it loses the special structure of the theorem that allows a simple proof. It is rare that one needs to override the ACL2 heuristics quite so much.

At this point, we are almost ready to prove the main result. However, at this stage our reasoning is very general, since it deals with *any* sequence of powers of roots of

1. It is not restricted to the specific sequence with as many elements as required by the Fourier Transform. To do so, we need to reason about the length of a list, or better yet, about the logarithm of its length, i.e., its depth as a binary tree. This yields the following lemma:

Definition.

$p\text{-depth}(x)$
 $=$
if $\text{powerlist-}p(x)$ **then** $1 + p\text{-depth}(p\text{-untie-}l(x))$
else 0

Theorem.

$\text{powerlist-}p(x)$
 $\rightarrow \text{eval-poly}(x, p\text{-omega}(p\text{-depth}(x)))$
 $=$
 $\text{eval-poly}(p\text{-unzip-}l(x), p\text{-omega}(p\text{-depth}(x)-1))$
 $\oplus p\text{-omega-sqrt}(p\text{-depth}(x)-1)$
 $\otimes \text{eval-poly}(p\text{-unzip-}r(x), p\text{-omega}(p\text{-depth}(x)-1))$
 $|$
 $\text{eval-poly}(p\text{-unzip-}l(x), p\text{-omega}(p\text{-depth}(x)-1))$
 $\ominus p\text{-omega-sqrt}(p\text{-depth}(x)-1)$
 $\otimes \text{eval-poly}(p\text{-unzip-}r(x), p\text{-omega}(p\text{-depth}(x)-1))$

This proof almost works, but ACL2 gets confused because of the hypothesis in $\text{eval-poly-omega-}n$ that N is positive. This hypothesis is clearly satisfied, since for $\text{powerlist } x$, $p\text{-depth}(x)$ is at least 1. Rather than forcing the hypothesis, as before, we will prove this simple lemma first:

Theorem.

$\text{powerlist-}p(x) \rightarrow \neg zp(p\text{-depth}(x))$

Once this fact is known, ACL2 has no more problems with the theorem.

This is a good time to actually define the Fourier Transform in ACL2:

Definition.

$p\text{-ft-omega}(x) = \text{eval-poly}(x, p\text{-omega}(p\text{-depth}(x)))$

We would like to prove the main result, which extends $\text{eval-poly-omega-depth}$ into $p\text{-ft-omega}$, but this will force us to reason about the $p\text{-depth}$ of p , given the $p\text{-depth}$ of $p \bowtie q$. We proceed, therefore, with the following technical lemma:

Theorem.

$\text{powerlist-}p(x) \wedge p\text{-regular-}p(x)$
 $\rightarrow p\text{-depth}(p\text{-unzip-}l(x)) = p\text{-depth}(x)-1$
 $\wedge p\text{-depth}(p\text{-unzip-}r(x)) = p\text{-depth}(x)-1$

It may be surprising that this is the only place where we require the $\text{powerlist } x$ to be regular.

Finally, we can prove the main result given in the hand-proof of Sect. 3:

Theorem.

$$\begin{aligned}
& \text{powerlist-p}(x) \wedge \text{p-regular-p}(x) \\
\rightarrow & \text{p-ft-omega}(x) \\
= & \text{p-ft-omega}(\text{p-unzip-l}(x)) \\
& \oplus \text{p-omega-sqrt}(\text{p-depth}(x)-1) \otimes \text{p-ft-omega}(\text{p-unzip-r}(x)) \\
& | \\
& \text{p-ft-omega}(\text{p-unzip-l}(x)) \\
& \ominus \text{p-omega-sqrt}(\text{p-depth}(x)-1) \otimes \text{p-ft-omega}(\text{p-unzip-r}(x))
\end{aligned}$$

To complete the proof, we need only introduce the ACL2 version of the Fast Fourier Transform:

Definition.

$$\begin{aligned}
& \text{p-fft-omega}(x) \\
= & \\
\text{if } & \text{powerlist-p}(x) \\
\text{then } & \text{p-fft-omega}(\text{p-unzip-l}(x)) \\
& \oplus \text{p-omega-sqrt}(\text{p-depth}(x)-1) \otimes \text{p-fft-omega}(\text{p-unzip-r}(x)) \\
& | \\
& \text{p-fft-omega}(\text{p-unzip-l}(x)) \\
& \ominus \text{p-omega-sqrt}(\text{p-depth}(x)-1) \otimes \text{p-fft-omega}(\text{p-unzip-r}(x)) \\
\text{else } & \text{fix}(x)
\end{aligned}$$

Note, again, the use of `fix` to ensure `p-fft-omega` always returns a numeric result – this is *required* here, because of our choice to do so in `eval-poly`. Otherwise, we would be unable to prove our main theorem, which equates the Fast Fourier Transform with the Fourier Transform:

Theorem (fft-omega-correctness).

$$\text{p-regular-p}(x) \rightarrow \text{p-fft-omega}(x) = \text{p-ft-omega}(x)$$

ACL2 needs a subtle hint to use the main lemma in the inductive part of the proof. The proof can be refined by defining instances of `p-omega` and `p-omega-sqrt` in terms of complex exponentiation. These instances correspond to the traditional definition of the Fourier Transform, and the correctness result can be established directly by functional instantiation.

6 Conclusions

In this paper, we showed how ACL2 can be used to prove the correctness of the Fast Fourier Transform algorithm, using the notational convenience of powerlists. The proof itself was taken from Misra’s seminal paper on powerlists [10]. We consider it a victory for ACL2 that it is able to follow a proof as elegant as the one given by Misra.

We believe this establishes the fact that ACL2 is a wonderful vehicle for automated reasoning about recursive data structures and algorithms, e.g., the powerlist data structures. This reinforces the feeling developed in [5], where powerlists were introduced into ACL2.

Moreover, the FFT proof suggests that ACL2 can be used to reason about *numeric* algorithms, especially those based on recursive definitions. That is, while ACL2 may

not be the theorem prover of choice to prove topological facts about the reals – a higher-order theorem prover would be better suited for this task – it is a perfectly wonderful system for proving recursive algorithms about the reals, such as the FFT, in much the same way that its predecessor, Nqthm, was the theorem prover of choice for algorithms over the naturals, e.g., the Euclidean algorithm. This is only possible, however, in a version of ACL2 which has been enhanced with support for the real numbers.

The source code for all the ACL2 examples listed here, as well as the refined proof using complex exponentiation, can be found in our web page at the following URL: <http://www.lim.com/~ruben/research/acl2/powerlists>. This code was processed with ACL2 version 2.1(r), the enhanced version of ACL2 2.1 with support for the real numbers.

References

1. Brock, B., Kaufmann, M., Moore, J: Acl2 theorems about commercial microprocessors. *Formal Methods in Computer-Aided Design (FMCAD'96)*. (1996) 275–293.
2. Boyer, R., Moore, J: *A Computational Logic*. Academic Press. Orlando. (1979)
3. Boyer, R., Moore, J: *A Computational Logic Handbook*. Academic Press. San Diego. (1988)
4. Corman, T., Leiserson, C., Rivest, R.: *Introduction to Algorithms*, chapter 32. McGraw-Hill. New York. (1990)
5. Gamboa, R.: Defthms about zip and tie: Reasoning about powerlists in ACL2. Univ. of Texas Comp. Sci. Tech. Rep. TR97-02. (1997)
6. Kaufmann, M., Moore J: ACL2 Version 1.9 Documentation. Computational Logic, Inc.
7. Kaufmann, M., Moore J: Design goals for ACL2. Computational Logic, Inc. Tech. Rep. 101. (1994)
8. Kaufmann, M., Moore J: An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering*. **23(4)** (1997) 203–213
9. Kaufmann, M., Pecchiari, P.: Interaction with the Boyer-Moore theorem prover: A tutorial study using the arithmetic-geometric mean theorem. Computational Logic, Inc. Tech. Rep. 100. (1994)
10. Misra, J.: Powerlists: A structure for parallel recursion. *ACM Trans. on Prog. Lang. and Systems*. **16(6)** (1994) 1737–1767