Journal of Algebraic Combinatorics, , (), 1–15

The Correctness of the Fast Fourier Transform: A Structured Proof in ACL2

RUBEN A. GAMBOA* ruben@lim.com

Logical Information Machines, Inc.
9390 Research Blvd., Suite II-200
Austin, TX 78759, USA

Received August 13, 1998; Revised April 28, 1999

Editor: Dominique Mery and Beverly Sanders

Abstract. The powerlists data structure, created by Misra in the early 90s, is well suited to express recursive, data-parallel algorithms. Misra has shown how powerlists can 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 has presented a stunningly simple proof of the correctness of the FFT. In this paper, we show how this proof can be mechanically verified using the ACL2 theorem prover. This supports Misra's belief that powerlists provide a suitable framework in which to define and reason about parallel algorithms, particularly using mechanical tools. It also illustrates the use of ACL2 in the formal verification of a distributed algorithm.

Keywords: Mechanical verification, formal methods, ACL2

1. Introduction

In [16], 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 the theorem prover 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. Our focus will be on the *process* of translating the hand proof into an ACL2 proof. In particular, we will show how the mechanical proof can be structured hierarchically, making it easier for the theorem prover to find the proof in the first place, as well as resulting in a simpler presentation of the proof to humans. We hope this will benefit those readers using ACL2 in their own research, as well as readers using other mechanical theorem provers.

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

2. A Review of Powerlists

A powerlist is a linear data structure, a list of elements. What differentiates powerlists from ordinary lists is that powerlists are constructed using parallel operators. 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 $L_1 \mid L_2$. 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.

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

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

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

$$
\begin{array}{rcl}
length(\langle x \rangle) & = & 1 \\
length(p \mid q) & = & length(p) + length(q)
\end{array}
$$

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 [16].

3. The Fast Fourier Transform

The Fourier transform of a real or complex vector $P = (p_1, p_2, p_3, \ldots, p_n)$ is defined as $FT(P) = (\overline{P}(w_n), \overline{P}(w_n^2), \overline{P}(w_n^3), \ldots, \overline{P}(w_n^n))$, 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 by evaluating $\overline{P}(x)$ at each of the n powers of w_n. This naive implementation will serve as our formal specification.

Following [16], we begin with the function ep which evaluates a polynomial P at a vector V. We will write ep in infix notation:

$$\begin{aligned} \langle x \rangle \ ep \ v &= \langle x \rangle && (1)\\ (p \bowtie q) \ ep \ v &= p \ ep \ v^2 + v \cdot (q \ ep \ v^2) && (2)\\ p \ ep \ (u \mid v) &= (p \ ep \ u) \mid (p \ ep \ v) && (3) \end{aligned}$$

Note that in the case $\langle x \rangle \ ep \ (u \mid v)$ we can proceed using either rule 3 or rule 1. Unfortunately, this will result in different answers. Thus, we tacitly restrict using rule 1 while rule 3 is applicable. We claim, without proof for now, 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 $W_n = (w_n, w_n^2, \ldots, w_n^n)$.

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 $W_n = (w_n, w_n^2, \ldots, w_n^n)$, for n a power of two greater than one. Then, we find that W_n can be written as

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

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

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

which are more amenable to induction.

We derive the Fast Fourier Transform as follows. For singleton powerlists, it is clear that

$$\begin{aligned} FT(\langle x \rangle) &= \langle x \rangle \ ep \ \overline{W}_0 && (5)\\ &= \langle x \rangle && (6) \end{aligned}$$

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

$$\begin{aligned} FT(p \bowtie q) &= (p \bowtie q) \ ep \ \overline{W}_N && (7)\\ &= (p \bowtie q) \ ep \ (u \mid -u) && (8)\\ &= ((p \bowtie q) \ ep \ u) \mid ((p \bowtie q) \ ep \ -u) && (9)\\ &= (p \ ep \ u^2 + u \cdot (q \ ep \ u^2)) \mid (p \ ep \ u^2 - u \cdot (q \ ep \ u^2)) && (10)\\ &= (p \ ep \ \overline{W}_{N-1} + u \cdot (q \ ep \ \overline{W}_{N-1})) \mid \\ &\quad (p \ ep \ \overline{W}_{N-1} - u \cdot (q \ ep \ \overline{W}_{N-1})) && (11)\\ &= (FT(p) + u \cdot FT(q)) \mid (FT(p) - u \cdot FT(q)) && (12) \end{aligned}$$

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

$$FFT(\langle x \rangle) = \langle x \rangle \tag{13}$$

$$FFT(p \bowtie q) = (FFT(p) + u \cdot FFT(q)) \mid (FFT(p) - u \cdot FFT(q)) \tag{14}$$

where the vector u contains the first $2^N/2$ elements of $\overline{W}_N$, and 2^N is the length of $p \bowtie q$.

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, 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" [13].

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. In fact, Kapur and Subramaniam report similar success in formalizing powerlists using the RRL theorem prover [7, 8, 9]. We will use the ACL2 formalization of powerlists presented in [6]. 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` ($|$). 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: 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 1

```
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 2

```
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 tradition of ACL2 that treats all non-numeric arguments to a numeric function as zero and forces 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 steps 9-10 use polynomial versions of the arithmetic operators. ACL2 reserves the arithmetic operators for numbers only; in fact, $x \cdot 1$ is equal to zero for all non-numeric arguments x, including vectors represented as powerlists. 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 1 (eval-poly-lemma)

```
    powerlist-p (p)
→   eval-poly (p, x)
  =   eval-poly (p-unzip-l (p), x ⊗ x)
```

$$\oplus\ x \otimes \texttt{eval-poly}\,(\texttt{p-unzip-r}\,(p),\ x \otimes x)$$

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, such as $(\ominus\, x) \otimes y = \ominus\, (x \otimes y)$, $(\ominus\, x) \otimes (\ominus\, x) = x \otimes x$, and $\texttt{p-similar-p}\,(x,\, y) \rightarrow x \oplus (\ominus\, y) = x \ominus y$. The similarity requirement in the last theorem is needed because the function $\oplus$ is defined to recurse in terms of the structure of its first argument, so it is possible that y will "run out" of terms before x does, in which case $\oplus$ will recurse using the `p-untie-l` and `p-untie-r` of a non-powerlist object.

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

Theorem 2 (eval-poly-u)

$$\begin{array}{lll} & \texttt{powerlist-p}\,(x) & \\ \rightarrow & \quad \texttt{eval-poly}\,(x,\ u \mid \ominus\, u) & \\ = & \qquad \texttt{eval-poly}\,(\texttt{p-unzip-l}\,(x),\ u \otimes u) & \\ & \quad \oplus\ u \otimes \texttt{eval-poly}\,(\texttt{p-unzip-r}\,(x),\ u \otimes u) & \\ & \mid \qquad \texttt{eval-poly}\,(\texttt{p-unzip-l}\,(x),\ u \otimes u) & \\ & \quad \ominus\ u \otimes \texttt{eval-poly}\,(\texttt{p-unzip-r}\,(x),\ u \otimes u) & \end{array}$$

Unfortunately, this proof attempt fails, because the ACL2 rewriter will not use the rewrite rules about unary minus, as it can not relieve the similarity hypothesis. For example, part of the proof requires $\texttt{eval-poly}\,(x,\ x \otimes u)$ to be similar to $u \otimes \texttt{eval-poly}\,(y,\ u \otimes u)$ which, while true, is not obvious to the ACL2 rewriter, and hence the rewrite rule taking $(x\ ep\ u^2) + (-(u \cdot y\ ep\ u^2))$ to the simpler $(x\ ep\ u^2) - (u \cdot y\ ep\ u^2)$ 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, the forced hypotheses are tackled using the full power of the theorem prover, not just the rewriter. To take advantage of this, the similarity conditions in the lemmas about $\ominus$ are marked as forceable. 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. 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 refers to this style of proof as a "structured" proof. It provides several mechanisms designed to support this style of proof, including the encapsulation principle. Using `encapsulate`, it is possible to 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 on $\overline{W}_n$ are as follows:

Constraint 1
`acl2-numberp` (`p-omega` (`0`))

Constraint 2
¬ `zp` (n) → `p-omega` (n) = `p-omega-sqrt` ($n-1$) | ⊖ `p-omega-sqrt` ($n-1$)

Constraint 3
`p-omega-sqrt` (n) ⊗ `p-omega-sqrt` (n) = `p-omega` (n)

Note, the ACL2 idiom ¬ `zp` (n) is used to recognize the positive integers; it is traditional to use `zp` (n) in recursive definitions.

The preceding event introduces the two constrained function symbols `p-omega` and `p-omega-sqrt`; `p-omega`(n) corresponds to $\overline{W}_n$ and `p-omega-sqrt`(n) is its square root. 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. This prevents 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, `p-omega` can be witnessed by a function returning a complete binary tree with zeros in all leaves. Outside of the `encapsulate`, the only known facts about the constrained functions are the actual constraints. Note in particular, 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 3 (eval-poly-omega-n)

```
   powerlist-p(x) ∧ ¬ zp(n)
→    eval-poly(x, p-omega(n))
=       eval-poly(p-unzip-l(x), p-omega(n−1))
     ⊕    p-omega-sqrt(n−1)
        ⊗ eval-poly(p-unzip-r(x), p-omega(n−1))
     |  eval-poly(p-unzip-l(x), p-omega(n−1))
     ⊖    p-omega-sqrt(n−1)
        ⊗ eval-poly(p-unzip-r(x), p-omega(n−1))
```

Proving this theorem requires a hint to encourage ACL2 to use the rule converting `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 3

```
p-depth(x)
=
if powerlist-p(x) then 1 + p-depth(p-untie-l(x))
 else 0
```

Theorem 4

```
   powerlist-p(x)
→    eval-poly(x, p-omega(p-depth(x)))
=       eval-poly(p-unzip-l (x), p-omega(p-depth(x)−1))
     ⊕    p-omega-sqrt(p-depth(x)−1)
        ⊗ eval-poly(p-unzip-r(x), p-omega (p-depth(x)−1))
     |  eval-poly(p-unzip-l(x), p-omega(p-depth(x)−1))
     ⊖    p-omega-sqrt(p-depth(x)−1)
        ⊗ eval-poly(p-unzip-r(x), p-omega (p-depth(x)−1))
```

This theorem can almost be proved mechanically, but ACL2 fails to use the lemma EVAL-POLY-OMEGA-N because of the hypothesis that N is positive. This hypothesis is clearly satisfied since for powerlist x, `p-depth`(x) is at least 1. Rather than forcing the hypothesis, as before, we will prove this simple lemma first:

Theorem 5

```
powerlist-p(x) → ¬ zp(p-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 4
$\texttt{p-ft-omega}(x) = \texttt{eval-poly}(x, \texttt{p-omega}(\texttt{p-depth}(x)))$

We would like to prove the main result which extends EVAL-POLY-OMEGA-DEPTH into `p-ft-omega`, but this will force us to reason about the `p-depth` of p, given the `p-depth` of $p \bowtie q$. Therefore, we proceed with the following technical lemma:

Theorem 6
$$\begin{array}{ll} & \texttt{powerlist-p}(x) \wedge \texttt{p-regular-p}(x) \\ \rightarrow & \quad \texttt{p-depth}(\texttt{p-unzip-l}(x)) = \texttt{p-depth}(x){-}1 \\ & \wedge\ \texttt{p-depth}(\texttt{p-unzip-r}(x)) = \texttt{p-depth}(x){-}1 \end{array}$$

It may be surprising that this is the only place where we require the powerlist x to be regular.
Finally, we can prove the main result given in the hand-proof of Sect. 3:

Theorem 7
$$\begin{array}{lll} & \texttt{powerlist-p}(x) \wedge \texttt{p-regular-p}(x) & \\ \rightarrow & \quad \texttt{p-ft-omega}(x) & \\ & = \quad\ \texttt{p-ft-omega}(\texttt{p-unzip-l}(x)) & \\ & \quad\ \oplus\ \texttt{p-omega-sqrt}(\texttt{p-depth}(x){-}1) \otimes \texttt{p-ft-omega}(\texttt{p-unzip-r}(x)) \\ & \quad | \quad\ \texttt{p-ft-omega}(\texttt{p-unzip-l}(x)) & \\ & \quad\ \ominus\ \texttt{p-omega-sqrt}(\texttt{p-depth}(x){-}1) \otimes \texttt{p-ft-omega}(\texttt{p-unzip-r}(x)) \end{array}$$

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

Definition 5
$$\begin{array}{l} \texttt{p-fft-omega}(x) \\ = \\ \textbf{if}\ \texttt{powerlist-p}(x) \\ \quad \textbf{then} \quad\ \texttt{p-fft-omega}(\texttt{p-unzip-l}(x)) \\ \qquad\ \oplus\ \texttt{p-omega-sqrt}(\texttt{p-depth}(x){-}1) \otimes \texttt{p-fft-omega}(\texttt{p-unzip-r}(x)) \\ \qquad | \quad\ \texttt{p-fft-omega}(\texttt{p-unzip-l}(x)) \\ \qquad\ \ominus\ \texttt{p-omega-sqrt}(\texttt{p-depth}(x){-}1) \otimes \texttt{p-fft-omega}(\texttt{p-unzip-r}(x)) \\ \quad \textbf{else}\ \texttt{fix}(x) \end{array}$$

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 8 (fft-omega-correctness)
$\texttt{p-regular-p}(x) \rightarrow \texttt{p-fft-omega}(x) = \texttt{p-ft-omega}(x)$

ACL2 needs a subtle hint to use the main lemma in the inductive part of the proof.

This proof is more general than necessary. It proves the correctness of an FFT-like algorithm for any polynomial evaluation at vectors satisfying the constraints on W_N. In the next section, we will refine this proof 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.

Note that splitting the proof into these two parts has two main benefits. First, it serves as an aid in a presentation of the proof. Notice, for example, that the first part of the proof closely follows Misra's hand proof. Second, it helps ACL2 find a mechanical proof of the result because it isolates the theories needed to reason about each part of the proof. I.e., the proof presented in this section does not depend at all on the theory of trigonometry. On the other hand, trigonometry will play a central role in the next section, whereas polynomial arithmetic will not. This reduces the number of lemmas that must be considered at each step, so it greatly reduces the size of the search space needed to find a proof.

6. Specializing the ACL2 Proof

In the previous section, we showed how the function

$$FT(x) = x \; ep \; \overline{W}_n$$

where n is the depth of x can be quickly computed for any family of vectors $\overline{W}_n$ such that

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

for some u, possibly depending on n. The actual Fourier Transform uses powers of the $(2^n)^{\text{th}}$ principal root of 1 in place of $\overline{W}_n$. In this section, we will try to prove that this particular vector satisfies the needed properties.

The n^{th} principal root of 1 is given by the complex number $e^{2\pi i/n}$. Using the standard definition of complex exponentiation, this gives

$$\begin{aligned} w_n &= e^{2\pi i/n} \\ &= \cos(2\pi/n) + i \sin(2\pi/n) \end{aligned}$$

The properties of the vector $\overline{W}_n = (w_{2^n}, w_{2^n}^2, \ldots, w_{2^n}^{2^n})$ can be derived from the basic properties of sine, cosine, and π. We will take this approach.

Note, the currently released version of ACL2 does not support the real numbers, so the trigonometric functions cannot be defined in it. The theorems in this section were verified using the most current beta version of ACL2, enhanced by the author to support the real numbers as well as the rationals.

In order to establish the multiplicative properties of e^{ix}, we will require the formulas for $\sin(x+y)$ and $\cos(x+y)$. Moreover, in order to establish that $W_n = u \mid -u$, we will need the facts that $\sin \pi = 0$ and $\cos \pi = -1$. ACL2 does not have built-in

knowledge of these trigonometric facts. These functions can be defined in ACL2 using the recent ACL2 features supporting non-standard analysis. The actual construction is beyond the scope of this paper, so we will simply assume these basic trigonometric facts in this section.

We next consider the definition of $\overline{W}_n$ from $w_{2^n} = e^{2\pi i/2^n}$. This is one place where it would be simpler, programmatically, to process the elements of $\overline{W}_n$ serially than in parallel, i.e., where it would be easier to use linear lists than powerlists. The reason is that it is not simple to do a "for i from 1 to n" type of loop in powerlists because their recursive structure is always a split down the middle. The solution is to think of our defining properties as a recurrence relation:

$$\begin{aligned} \overline{W}_n &= \sqrt{\overline{W}_{n-1}} \;\Big|\; -\sqrt{\overline{W}_{n-1}} \\ \overline{W}_0 &= 1 \end{aligned}$$

This gives a recurrence relation for the exponents as follows:

$$\begin{aligned} E_n &= E_{n-1}/2 \mid (E_{n-1}/2 + \pi) \\ E_0 &= 2\pi \end{aligned}$$

where the arithmetic operators are defined over pointwise powerlists. We can then derive $\overline{W}_n = e^{iE_n}$.

We begin with the definition of the scalar operators:

Definition 6

```
p-halve(x)
=
if powerlist-p(x) then p-halve(p-untie-l(x)) | p-halve(p-untie-r(x))
 else x / 2
```

Definition 7

```
p-offset(x, p)
=
if powerlist-p(p)
 then p-offset(x, p-untie-l(p)) | p-offset(x, p-untie-r(p))
 else x + p
```

Definition 8

```
p-exponents(n)
=
if zp(n) then 2 · acl2-pi
 else let sqrt-expnts be p-halve(p-exponents(n−1))
           in
      sqrt-expnts | p-offset(acl2-pi, sqrt-expnts)
```

Definition 9

```
complex-expt(x) = complex(acl2-cosine(x), acl2-sine(x))
```

Definition 10

```
p-complex-expt (x)
=
if powerlist-p (x)
 then p-complex-expt (p-untie-l (x)) | p-complex-expt (p-untie-r (x))
 else complex-expt (x)
```

It is now possible to define the functions `p-expt-omega` and `p-expt-omega-sqrt` which generate $\overline{W}_n$ and $\sqrt{\overline{W}_n}$, respectively.

Definition 11

```
p-expt-omega (n) = p-complex-expt (p-exponents (n))
```

Definition 12

```
p-expt-omega-sqrt (n)
=
p-complex-expt (p-halve (p-exponents (n)))
```

We now show that these functions satisfy all the constraints associated with `p-omega` and `p-omega-sqrt`. We begin with the simplest constraint, namely that $\overline{W}_0$ is numeric:

Theorem 9

```
acl2-numberp (p-expt-omega (0))
```

We will next show that `p-expt-omega-sqrt` is the square root of `p-expt-omega`. To do this, we need the fact that $e^{x/2}e^{x/2} = e^x$. We can prove this in ACL2 with the following theorem:

Theorem 10 (complex-expt-/-2)

```
    acl2-numberp (x)
→   complex-expt (1/2 · x) · complex-expt (1/2 · x)
  = complex-expt (x)
```

ACL2 needs hints to generate good instances of the sine of sums and cosine of sums axioms. This is the most common type of hint we have to give ACL2, since its heuristics are not powerful enough to pick good lemma instances in general.

The next step is to generalize the lemma COMPLEX-EXPT-/-2 to powerlists in general. We do this as follows:

Theorem 11

```
    p-acl2-number-list (x)
→   p-complex-expt (p-halve (x)) ⊗ p-complex-expt (p-halve (x))
  = p-complex-expt (x)
```

The hypothesis `p-acl2-number-list` is needed, because COMPLEX-EXPT-/-2 requires x to be a number. A required hint disables `complex-expt` so that the rewriter does not expand its body and fail to see the rewrite rule COMPLEX-EXPT-/-2.

With the new rule, it is easy to prove the next constraint required, namely that `p-expt-omega-sqrt` is the square root of `p-expt-omega`:

Theorem 12

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

Here again, ACL2 requires a hint to guide the selection of a proper lemma instance.

We now turn our attention to the final constraint, which deals with unary minus. The only lemma we need is the following:

Theorem 13

$\texttt{p-complex-expt}(\texttt{p-offset}(\texttt{acl2-pi}, \mathit{expnts})) = \ominus\ \texttt{p-complex-expt}(\mathit{expnts})$

This follows from the facts that $e^{x+y} = e^x e^y$ and $e^{i\pi} = -1$ — Euler's beautiful identity. ACL2 can then immediately extend this result to powerlists, which is our third and last constraint on `p-omega` and `p-omega-sqrt`:

Theorem 14

$$\begin{array}{ll} & \neg\ \texttt{zp}(n) \\ \rightarrow & \quad \texttt{p-expt-omega}(n) \\ & = \texttt{p-expt-omega-sqrt}(n{-}1) \mid \ominus\ \texttt{p-expt-omega-sqrt}(n{-}1) \end{array}$$

What this means is that we can now instantiate the theorems proved in Sect. 5 about the Fast Fourier Transform. Hence, we proceed with the new definitions of FT and FFT, these based on the trigonometric version of W_n:

Definition 13

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

Definition 14

$$\begin{array}{l} \texttt{p-fft-expt-omega}(x) \\ = \\ \textbf{if}\ \texttt{powerlist-p}(x) \\ \quad \textbf{then} \quad \texttt{p-fft-expt-omega}(\texttt{p-unzip-l}(x)) \\ \qquad\qquad \oplus \quad \texttt{p-expt-omega-sqrt}(\texttt{p-depth}(x){-}1) \\ \qquad\qquad\quad \otimes\ \texttt{p-fft-expt-omega}(\texttt{p-unzip-r}(x)) \\ \qquad\quad \mid \quad \texttt{p-fft-expt-omega}(\texttt{p-unzip-l}(x)) \\ \qquad\qquad \ominus \quad \texttt{p-expt-omega-sqrt}(\texttt{p-depth}(x){-}1) \\ \qquad\qquad\quad \otimes\ \texttt{p-fft-expt-omega}(\texttt{p-unzip-r}(x)) \\ \quad \textbf{else}\ \texttt{fix}(x) \end{array}$$

ACL2 immediately verifies that the new definition of FFT correctly computes the Fourier Transform:

Theorem 15

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

A hint is required to prove this by instantiating the meta-theorem FFT-OMEGA-CORRECTNESS, because ACL2 does not attempt to use meta-theorems by itself. It is a pity that ACL2 can not do this directly, but we have already seen how difficult it can be to pick good *instances* of a regular lemma, so perhaps this final automation would be expecting too much of the theorem prover.

7. 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 [16]. 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 [6], where powerlists were introduced into ACL2.

Another important aspect of the formal proof is that it was structured using ACL2's encapsulation principle. As seen from the hand proof, the correctness of the FFT follows from the key properties of the powers of roots of unity. ACL2 allows the formal proof to be split into two parts, the first establishing that the key properties do indeed prove the correctness of the FFT, and the second showing that the vectors of powers of roots of unity have the key properties. By allowing proofs to be decomposed in this fashion, it is easier to find mechanical proofs of complex theorems. Put another way, it is easier to prove two "medium-sized" theorems than one "large" theorem, especially in the context of a mechanical theorem prover. Support for structured theory development is one of the ways ACL2 addresses the requirements of industrial-strength theorem proving [10, 14].

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 may 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.

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 URL `http://www.lim.com/~ruben/research/acl2/powerlists`. This code was processed with ACL2 version 2.1(r), an enhanced version of ACL2 2.1 with support for the real numbers.

References

1. B. Brock, M. Kaufmann, and J Moore, "ACL2 theorems about commercial microprocessors," in *Formal Methods in Computer-Aided Design (FMCAD'96)*, pages 275–293, 1996.
2. R. Boyer and J Moore, *A Computational Logic,* Academic Press, Orlando, 1979.
3. R. Boyer and J Moore, *A Computational Logic Handbook,* Academic Press, San Diego, 1988.
4. R. Churchill and J. Brown, *Complex Variables and Applications,* McGraw-Hill, 1984.
5. T. Corman, C. Leiserson, R. Rivest, *Introduction to Algorithms,* McGraw-Hill, New York, 1990, chapter 32.
6. R. Gamboa, "Defthms about zip and tie: Reasoning about powerlists in ACL2," Univ. of Texas Comp. Sci. Tech. Rep. TR97-02, 1997.
7. D. Kapur, "Constructors can be Partial too," in *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*, MIT Press, 1997.

8. D. Kapur and M. Subramaniam, "Automated reasoning about parallel algorithms using powerlists", State University of New York at Albany Tech. Rep. TR-95-14, 1995.
9. D. Kapur and H. Zhang, "An Overview of Rewrite Rule Laboratory (RRL)", in *Computers in Mathematics with Applications*, 1994.
10. M. Kaufmann, "ACL2 Support for Verification Projects," invited talk in *Proc. 15th Intl. Conf. on Automated Deduction*, pages 220–238, 1998.
11. M. Kaufmann and J Moore, "ACL2 Version 2.1 Documentation," available in the world-wide web at `http://www.cs.utexas.edu/users/moore/acl2`.
12. M. Kaufmann and J Moore, "Design goals for ACL2," Computational Logic, Inc. Tech. Rep. 101, 1994.
13. M. Kaufmann and J Moore, "An industrial strength theorem prover for a logic based on common lisp," in *IEEE Transactions on Software Engineering*, **23(4)**, pages 203–213, 1997.
14. M. Kaufmann and J Moore, "Structured Theory Development for a Mechanized Logic," manuspript under preparation.
15. M. Kaufmann, and P. Pecchiari, "Interaction with the Boyer-Moore theorem prover: A tutorial study using the arithmetic-geometric mean theorem," Computational Logic, Inc. Tech. Rep. 100, 1994.
16. J Misra, "Powerlists: A structure for parallel recursion," in *ACM Trans. on Prog. Lang. and Systems*, **16(6)**, pages 1737–1767, 1994.

Contributing Authors

R. Gamboa holds a B.S. degree from Angelo State University in 1984, an M.C.S. from Texas A&M University in 1986, and a Ph.D. from the University of Texas at Austin in 1999. All degrees are in computer science. As part of his dissertation he modified ACL2 to support the irrational numbers, using the principles of non-standard analysis. The modified ACL2 was used to prove the results contained in this paper.

His interests range over many issues in computation, encompassing deductive databases, computability, artificial intelligence, and automated theorem proving. He has been with Logical Information Machines (LIM) since 1990, where he is the chief architect for its financial database server. Before joining LIM, he was a member of the technical staff at MCC, a research consortium located in Austin, TX.