

Defthms About Zip and Tie: Reasoning About Powerlists in ACL2

Ruben A. Gamboa*

Computer Sciences Department
The University of Texas at Austin
Taylor Hall 2.124
Austin, TX 78712-1188

`ruben@lim.com`, `ruben@cs.utexas.edu`

<http://www.lim.com/~ruben/research/acl2/powerlists/>

February 11, 1998

Abstract

In [Mis94], Misra introduced the powerlist data structure, which is well suited to express recursive, data-parallel algorithms. Moreover, Misra and other researchers have shown how powerlists can be used to prove the correctness of several algorithms. This success has encouraged some researchers to pursue automated proofs of theorems about powerlists [Kap97, KS95a, KS95b].

In this paper, we show how ACL2 can be used to verify theorems about powerlists. We depart from previous approaches in two significant ways. First, the powerlists we use are not the regular structures defined by Misra; that is, we do not require powerlists to be balanced trees. As we will see, this complicates some of the proofs, but on the other hand it allows us to state theorems that are otherwise beyond the language of powerlists. Second, we wish to prove the correctness of powerlist algorithms as much as possible within the logic of powerlists. Previous approaches have relied on intermediate lemmas which are unproven (indeed unstated) within the powerlist logic. However, we believe these lemmas must be formalized if the final theorems are to be used as a foundation for subsequent work, e.g., in the verification of system libraries. In our experience, some of these unproven lemmas presented the biggest obstacle to finding an automated proof.

* Author is currently employed by LIM International, Inc.

Contents

1	Introduction	3
2	Booking Powerlists	4
2.1	Regular Powerlists	4
2.2	Defining Powerlists in ACL2	5
2.2.1	A Naive Representation of Powerlists	5
2.2.2	A Better Representation of Powerlists	5
2.2.3	The Tie Constructor	6
2.2.4	The Zip “Constructor”	7
2.3	Similar Powerlists	9
2.4	Regular Powerlists	10
2.5	Functions on Powerlists	12
3	Simple Examples	15
4	Sorting Powerlists	18
4.1	Merge Sorting	20
4.2	Batcher Sorting	22
4.3	A Comparison with the Hand-Proof	30
5	Prefix Sums of Powerlists	31
5.1	Simple Prefix Sums	32
5.2	Ladner-Fischer Prefix Sums	39
5.3	Comparing with the Hand-Proof Again	41
5.4	L’agniappe: A Carry-Lookahead Adder	42
6	Conclusions	43

1 Introduction

In [Mis94], Misra introduced the powerlist data structure and powerlist algebra, which is particularly well-suited to express and reason about recursive parallel algorithms. Of particular interest to Misra is the expressiveness of powerlist algebra and its utility as a logic in which to prove correctness results; much of [Mis94] is devoted to the development of practical examples using powerlists, including Batcher sorting, FFT networks, and prefix sums, as well as the relevant correctness results. In the same spirit, other researchers have used powerlists to find elegant proofs of parallel algorithms, for example odd-even sorting in [Kor97a].

In this paper, we focus not on the discovery or expression of correctness results, but on their mechanical verification. Specifically, we wish to show how a library of provably correct functions on powerlists can be developed. We consider it important, therefore, that the correctness results be in such a form that they can be used in subsequent (mechanical) proofs. This is a departure from [Mis94], where intuition is often used as a guide to transform the original specifications into more tractable forms, in order to simplify the formal proof based on the powerlist algebra. These transformations are justified when the proofs are being generated by hand, since the intuitive arguments can be formalized inside or outside of powerlist algebra.

We will formalize powerlists using the ACL2 theorem prover, the successor to the Boyer-Moore theorem prover. The logic of ACL2 is a first-order, quantifier-free logic of recursive functions with induction on the ordinals up to ϵ_0 , recursive definitions, and witnessed constrain of new function symbols. The theorem prover of ACL2 was designed to be an “industrial-strength” theorem prover, supporting equality rewriting and induction, as well as more esoteric techniques such as equivalence rewriting, congruence reasoning, and reasoning about theorem schemas via functional instantiation. In addition to its reasoning engine, ACL2 provides many amenities to the user. An important one is the abstraction of “books,” which allow the user to construct theories in a modular fashion. For example, we will construct a powerlist “book” with all the commonly used definitions and theorems about powerlists, i.e., the requisite powerlist algebra [KM97].

Other researchers have also attempted to use automated theorem provers to reason about powerlists, notably [Kap97], [KS95a] and [KS95b]. While there are some similarities in our respective approaches, there are significant differences as well. In [Kap97], Kapur is interested in extending a theorem prover to facilitate reasoning about regular data structures, such as powerlists. [KS95a] uses this structure to prove some of the theorems from [Mis94], but the emphasis again is on the theorem prover, and how it can find proofs that rival in elegance those generated by hand. However, the theorems themselves, as in [Mis94], are designed to simplify the powerlist proofs, rather than to certify an algorithm’s correctness with respect to an absolute specification. In spirit, we have more in common with [KS95b], where adder circuits specified using powerlists are proved correct with respect to addition on the natural numbers.

The remainder of the paper is organized as follows. Section 2 develops an ACL2 book about powerlists; the theorems proved there will be used as lemmas in all subsequent work. Section 3 presents some simple examples which provide needed intuition in working with powerlists. These examples also illustrate how the formal foundation laid in section 2 provides a natural basis for reasoning about powerlists. The remaining sections present more substantive examples, namely correctness proofs of Batcher sorting, parallel prefix sum, and carry-lookahead addition. All proofs rest on the foundation of section 2. Moreover, the carry-lookahead proof uses the prefix sum result, thus illustrating a *formal*, modular proof, similar to the informal proof found in most textbooks. Finally, section 6 summarizes the results and gives some pointers for the future.

2 Booking Powerlists

2.1 Regular Powerlists

Misra defines powerlists as follows. For any scalar x , the object $\langle x \rangle$ is a singleton powerlist. If x and y are “similar” powerlists — that is, they have the same number of elements, and corresponding elements are similar — we can construct the new powerlists $x \mid y$ and $x \bowtie y$, called the tie and zip of x and y , respectively. The powerlist $x \mid y$ consists of all elements of x followed by the elements of y . In contrast, $x \bowtie y$ contains the elements of x interleaved with the elements of y . Since tie and zip are defined only for similar powerlists, all powerlists are of length 2^n for some integer n , and moreover all elements of a powerlist are similar to each other. We call these “regular” powerlists.

For example, $\langle 1 \rangle, \langle 1, 2 \rangle, \langle 3, 4 \rangle, \langle 1, 2, 3, 4 \rangle$ and $\langle 1, 3, 2, 4 \rangle$ are all powerlists. Moreover, $\langle 1, 2 \rangle \mid \langle 3, 4 \rangle = \langle 1, 2, 3, 4 \rangle$ and $\langle 1, 2 \rangle \bowtie \langle 3, 4 \rangle = \langle 1, 3, 2, 4 \rangle$.

The theory of powerlists depends on the following axioms (laws in [Mis94]):

L0. For singleton powerlists $\langle x \rangle$ and $\langle y \rangle$, $\langle x \rangle \mid \langle y \rangle = \langle x \rangle \bowtie \langle y \rangle$.

L1a. For any non-singleton powerlist X , there are similar powerlists L, R so that $X = L \mid R$.

L1b. For any non-singleton powerlist X , there are similar powerlists O, E so that $X = O \bowtie E$.

L2a. For singleton powerlists $\langle x \rangle$ and $\langle y \rangle$, $\langle x \rangle = \langle y \rangle$ iff $x = y$.

L2b. For powerlists $X_1 \mid X_2$ and $Y_1 \mid Y_2$, $X_1 \mid X_2 = Y_1 \mid Y_2$ iff $X_1 = Y_1$ and $X_2 = Y_2$.

L2c. For powerlists $X_1 \bowtie X_2$ and $Y_1 \bowtie Y_2$, $X_1 \bowtie X_2 = Y_1 \bowtie Y_2$ iff $X_1 = Y_1$ and $X_2 = Y_2$.

L3. For powerlists X_1, X_2, Y_1 , and Y_2 , $(X_1 \mid X_2) \bowtie (Y_1 \mid Y_2) = (X_1 \bowtie Y_1) \mid (X_2 \bowtie Y_2)$.

2.2 Defining Powerlists in ACL2

2.2.1 A Naive Representation of Powerlists

Choosing the right representation of powerlists in ACL2 is not trivial. One immediate stumbling block is that ACL2 does not support partial functions, so the definitions of `|` and `⊗` must do *something* for non-similar powerlists, and in fact for non-powerlist operands. A first approach might represent powerlists in ACL2 as lists and of length 2^n . The function `tie` would take two powerlists and, if they are of equal length, return their concatenation, otherwise a special error powerlist (e.g., `nil`). Similarly, we could define the function `zip`. A similar approach is taken in [KS95a], though partial constructors are used in that paper.

There are a few problems with taking this approach in ACL2. First of all, each time we make a `tie` or `zip`, we would have to prove that the arguments are of equal length. These proof obligations can become expensive at best, and disastrous if they prevent term simplification. Moreover, the proof obligations propagate into all theorems concerning `tie` and `zip`; we observed these obligations placing a large burden on the ACL2 rewriter. The second problem is that since ACL2 does not support function definitions over terms, powerlist functions such as

$$\begin{aligned} rev(\langle x \rangle) &= x \\ rev(x | y) &= rev(y) | rev(x) \end{aligned}$$

need to be turned into the form

$$rev(X) = \begin{cases} X & \text{if } X \text{ is a singleton} \\ rev(right(X)) | rev(left(X)) & \text{otherwise} \end{cases}$$

where the functions `left` and `right` are defined so that `left(X) | right(X) = X`. But defining these functions in ACL2 — more germanely, reasoning about them — is not simple. Intuitively, the problem is that to compute `left(X)`, we must first count the elements of `X`, divide by two, then walk back through the elements of `X` and return half of them. Reasoning about all these steps is necessary in every function invocation. In our experience, the overhead quickly overwhelmed the prover.

2.2.2 A Better Representation of Powerlists

The observations above led us to pursue an alternative approach. Instead of representing powerlists as lists, we chose to represent them as binary trees, e.g., `cons` trees. Moreover, we remove the restriction that `tie` and `zip` only apply to similar powerlists. The operation `tie` is now replaced by a simple `cons` and `left` and `right` can be defined in terms of `car` and `cdr`. The definition of `zip` requires a recursive function, very similar to the one used when representing powerlists as lists. The result of this representation is that reasoning about powerlists requires much less overhead than before; however, the representation allows objects that were previously not recognized as powerlists, for example

$\langle 1.\langle 2.3 \rangle \rangle$, where we use dotted notation to emphasize the structural nature of the representation. In the sequel, we will use the term “powerlists” to refer to arbitrary “dotted-pair” powerlists as above. When we must refer to the original powerlists explicitly, we will use the term “regular powerlists.”

The generalized notion of powerlists allows us to write some algorithms which cannot be stated in traditional powerlist theory, for example, insertion sort. On the other hand, it presents some new problems. First, it does not retain a 1-to-1 correspondence between linear lists and powerlists. For example, the list $(1, 2, 3, 4)$ can be viewed as either of the powerlists $\langle 1.\langle 2.\langle 3.4 \rangle \rangle$ or $\langle \langle 1.2 \rangle.\langle 3.4 \rangle \rangle$. This does not trouble us, because the theorems we prove will be true of either powerlist representation. Naturally, in parallel processing applications, we would like to choose the powerlist with the smallest maximal branch height. The choice, however, is made in the translation from lists to powerlists, not in the powerlist theory. A second problem is that the operational semantics of certain functions may not carry over to generalized powerlists. For example, the operational semantics of `zip` is that it interleaves the elements from its two powerlist arguments. This is clearly not possible if the arguments have different lengths. We will only insist that our function definitions match the operational semantics for regular powerlists, but that they retain the relevant algebraic properties for all powerlists. For example, we require that our `zip` operator interleave the elements from its two arguments, when these are regular and similar to each other. Furthermore, for all powerlists, we require that `zip` obey the algebraic properties stated in laws *L1b*, *L2c*, and *L3*.

The choice to use generalized powerlists was made taking these tradeoffs into account. Similar tradeoffs can be found in other approaches to generalized powerlists, such as Kornerup’s parlists [Kor97b].

We must be careful here that the resulting theory is nevertheless faithful to the original theory due to Misra. That is, we must ensure that the original axioms of `zip` and `tie` hold in the new theory. At the very least, we must ensure that the theorems about regular powerlists are precisely those of Misra’s theory. We will do so by examining each of Misra’s powerlist axioms in turn.

Observe, since the scalar powerlist $\langle x \rangle$ is simply represented as x in our scheme, law *L2a* is trivially true. A drawback of this approach is that we do not allow nested powerlists, e.g., $\langle \langle 1, 2 \rangle, \langle 3, 4 \rangle \rangle$ is indistinguishable from $\langle 1, 2, 3, 4 \rangle$ in our representation. Where nested powerlists are needed, e.g., for matrices, we suggest adding an explicit *nest* operator, as in $\langle \text{nest}(\langle 1, 2 \rangle), \text{nest}(\langle 3, 4 \rangle) \rangle$.

2.2.3 The Tie Constructor

We begin the actual implementation with the definition of the data type powerlists. For stylistic reasons, we define powerlists not directly as `cons`’s, but as dotted structures:

```
(defstructure powerlist car cdr)
```

The `defstructure` event is similar to Common LISP’s `defstruct`, but there are some key differences. It defines the functions `powerlist`, `powerlist-p`,

`powerlist-car`, and `powerlist-cdr`, which respectively construct, recognize, and destruct powerlists. It also proves the relevant “functor” theorems about them, which correspond to Misra’s laws *L1a* and *L2b*. However, it does *not* introduce a new data type¹.

For style, we rename the functions `powerlist-car` and `powerlist-cdr` into `p-untie-l` and `p-untie-r`, respectively. This will serve to provide more symmetry with `p-zip` below. In the sequel, we will refer to `(p-untie-l x)` as the “left half” or “left untie” of `x`. Similarly, we will say the “right half” or the “right untie” when referring to `(p-untie-r x)`.

The next step is to define the function `p-zip`, by using the laws *L0* and *L3*. This function will use a recursion scheme based on `p-untie-l` and `p-untie-r`. To verify that this recursion is valid, ACL2 needs to verify that each invocation of `p-untie-l` and `p-untie-r` decreases a well-founded measure on powerlists. The measure can be defined using the ACL2 primitive `acl2-count`, which returns an integer measure for all ACL2 objects, so we need only show that the `acl2-count` of a powerlist is decreased appropriately by `p-untie-l` and `p-untie-r`. Since this recursion scheme will be very common with powerlists, we suggest adding a built-in rule to ACL2 to assert that this recursion is valid. This will allow ACL2 to accept definitions recursing on `p-tie` without generating proof obligations that need to be dealt with by the reasoning engine. The only drawback is that because built-in rules are applied before rewriting, we must be careful to prove *exactly* the same theorem that ACL2 will attempt to prove when considering a `defun`. A good way to find this theorem is to define such a recursive function and then yank the induction goal printed by ACL2. This yields the following ACL2 event:

```
(defthm untie-reduces-count-fast
  (implies (powerlist-p x)
    (and (e0-ord-< (acl2-count (p-untie-l x))
                  (acl2-count x))
         (e0-ord-< (acl2-count (p-untie-r x))
                  (acl2-count x))))
  :rule-classes :built-in-clause)
```

For integer arguments, such as the value returned by `acl2-count`, the ACL2 primitive `e0-ord-<` is identical to `<`. As mentioned previously, we have to use `e0-ord-<` instead of `<` so that the theorem matches exactly the formula generated by ACL2 while processing a `defun`.

2.2.4 The Zip “Constructor”

We can now define the function `p-zip` which implements the zip “constructor”:

```
(defun p-zip (x y)
```

¹Behind the scenes, `defstructure` implements a powerlist dotted structure as a regular list with the special form `'(powerlist X Y)`. This is mostly irrelevant, unless one wishes to reason about lists of powerlists or powerlists of lists.

```

(if (and (powerlist-p x) (powerlist-p y))
    (p-tie (p-zip (p-untie-l x) (p-untie-l y))
           (p-zip (p-untie-r x) (p-untie-r y)))
    (p-tie x y)))

```

Note how the definition of `p-zip` mirrors *L0* and *L3*, hence these axioms are satisfied by our definition of `p-tie` and `p-zip`. In order to accept definitions based on `p-zip`, we have to define the functions `p-unzip-l` and `p-unzip-r`, analogous to `p-untie-l` and `p-untie-r`. We can do so as follows:

```

(defun p-unzip-l (x)
  (if (powerlist-p x)
      (if (powerlist-p (p-untie-l x))
          (if (powerlist-p (p-untie-r x))
              (p-tie (p-unzip-l (p-untie-l x))
                     (p-unzip-l (p-untie-r x)))
              (p-untie-l x))
          (p-untie-l x))
      x))

(defun p-unzip-r (x)
  (if (powerlist-p x)
      (if (powerlist-p (p-untie-l x))
          (if (powerlist-p (p-untie-r x))
              (p-tie (p-unzip-r (p-untie-l x))
                     (p-unzip-r (p-untie-r x)))
              (p-untie-r x))
          (p-untie-r x))
      nil))

```

Note, these functions provide the equivalent to Misra's law *L1b*. At this state, it is worthwhile to prove the validity of recursion based on `p-zip`, just as we did for `p-tie`.

Notice that `p-unzip-l` and `p-unzip-r` return every other element of a regular powerlist `x`. If we index the elements of `x` from 1, `(p-unzip-l x)` returns the odd-indexed elements, and `(p-unzip-r x)` the even-indexed ones. Hence, in the sequel we will refer to `p-unzip-l` and `p-unzip-r` as the odd- and even-indexed elements of `x`, respectively. Similarly to `p-untie`, we will also refer to these lists as the "left unzip" and "right unzip" of `x`.

The definitions of `p-unzip-l` and `p-unzip-r` were carefully constructed so that the following theorems are all true:

```

(defthm zip-unzip
  (implies (powerlist-p x)
           (equal (p-zip (p-unzip-l x) (p-unzip-r x)) x)))

(defthm unzip-l-zip

```

```
(equal (p-unzip-l (p-zip x y)) x))
```

```
(defthm unzip-r-zip
  (equal (p-unzip-r (p-zip x y)) y))
```

These three theorems prove the equivalent of law *L2c* for our powerlists. On an implementation note, we make `zip-unzip` an `:elim` rule so that `ACL2` can use it to eliminate the destructors `p-unzip-l` and `p-unzip-r` in favor of `p-zip`, in much the same way it removes `car` and `cdr` and replaces them using `cons`.

2.3 Similar Powerlists

At this point, we have seen how our definitions of `p-tie` and `p-zip` satisfy all of Misra’s powerlist axioms, except for the notion of similarity. Laws *L1a* and *L1b* claim that the `p-untie-l` and `p-untie-r` of a powerlist are similar, i.e. of the same length, and so are its `p-unzip-l` and `p-unzip-r`. This is certainly not the case with our powerlists, since we do not require that powerlists be of length 2^n . We will now add conditions, namely that the given powerlists be regular, that make these theorems true. Later, these regularity conditions will surface as hypotheses in some of the example theorems proved.

In accordance with [Mis94], we define two powerlists as similar if they have the same `tie`-tree structure. We can do so with the following `ACL2` event:

```
(defun p-similar-p (x y)
  (if (powerlist-p x)
      (and (powerlist-p y)
           (p-similar-p (p-untie-l x) (p-untie-l y))
           (p-similar-p (p-untie-r x) (p-untie-r y)))
      (not (powerlist-p y))))
```

We can immediately prove that `p-similar-p` is an equivalence relation. This proves useful, because `ACL2` can use this fact in its generic “equality” reasoning, hence making theorems about `p-similar-p` easier to prove.

Our next task is to show how `p-similar-p` powerlists behave in conjunction with the constructors and destructors based on `p-tie` and `p-zip`. These theorems are trivial for regular powerlists, since powerlists are similar if and only if they have the same length. Moreover, both `zip` and `tie` double the length of a powerlist, and `unzip` and `untie` halve it.

We have to work a little harder in the case of general powerlists; this lost simplicity is the price we pay for not using a regular data structure as suggested by Misra. For starters, we can prove theorems about the destructors, such as the following:

```
(defthm unzip-l-similar
  (implies (p-similar-p x y)
           (p-similar-p (p-unzip-l x) (p-unzip-l y))))
```

We also prove the analogous theorems for `p-unzip-r` as well as for `p-untie`. These theorems will be used most often in proving the antecedent of an inductive hypothesis. For example, with the goal

```
(implies (p-similar-p x y)
         (P x y))
```

where property `P` is defined in terms of `p-zip`, the following subgoal is likely to be generated by induction:

```
(implies (and (powerlist-p x)
              (p-similar-p x y)
              (implies (p-similar-p (p-unzip-1 x)
                                   (p-unzip-1 y))
                       (P (p-unzip-1 x) (p-unzip-1 y))))
         (implies (p-similar-p (p-unzip-r x)
                               (p-unzip-r y))
                 (P (p-unzip-r x) (p-unzip-r y))))
         (P x y))
```

At this point, `p-unzip-1-similar` can be used to establish that `(P (p-unzip-1 x) (p-unzip-1 y))` is true and the proof can proceed. Since this is the intended use, we turned these theorems into `:forward-chaining` rules, which are triggered before ACL2's general rewriting engine (and hence provide a modest performance improvement).

Remaining are the constructors `p-tie` and `p-zip`. We would like to say that when a powerlist is zipped (tied) to one of two similar powerlists, the result is similar to when it is zipped (tied) to the other. ACL2 provides a general way to reason about this type of theorem, namely congruence rewriting. With congruence rewriting, ACL2 will deduce `(p-zip x1 y)` is similar to `(p-zip x2 y)` when `x1` is similar to `x2`, and similarly that `(p-zip x y1)` and `(p-zip x y2)` are similar when `y1` and `y2` are. We can define these congruence rules in ACL2 as follows:

```
(defcong p-similar-p p-similar-p (p-zip x y) 1)
(defcong p-similar-p p-similar-p (p-zip x y) 2)
```

The `defcong` event is simply syntactic sugar for the rules described above.

2.4 Regular Powerlists

Another useful property of powerlists is `p-regular-p` which is true of a perfectly balanced powerlist, that is, one which corresponds to the theory in [Mis94]. This condition is more expensive than `p-similar-p`, because it requires passing information from one half of the powerlist to the other, i.e., not only must the left and right halves of the powerlist be regular, their depth must be the same. Rather than explicitly reasoning about depth, we chose to use `p-similar-p`, since we already have several theorems about it. The result is the following definition:

```

(defun p-regular-p (x)
  (if (powerlist-p x)
      (and (p-similar-p (p-untie-l x) (p-untie-r x))
           (p-regular-p (p-untie-l x))
           (p-regular-p (p-untie-r x)))
      t))

```

Note that both the similarity and regularity conditions of the definition are required to restrict powerlists to be of length 2^n . For example, if the similarity condition were left out, $\langle 1.\langle 2.3\rangle.\langle 4.5\rangle\rangle$ would be considered regular. Likewise, if the regularity conditions were left out, the powerlist $\langle\langle 1.\langle 2.3\rangle\rangle.\langle 4.\langle 5.6\rangle\rangle$ would be considered regular. We shall see later that we do not need to have both regularity conditions in the definition.

As was the case with `p-similar-p`, we must show how `p-regular-p` interacts with the constructors and destructors of `p-tie` and `p-zip`. This results in the following type of theorem:

```

(defthm unzip-regular
  (implies (p-regular-p x)
           (and (p-regular-p (p-unzip-l x))
                (p-regular-p (p-unzip-r x)))))

```

The converse theorem, about the constructor functions requires an extra hypothesis, namely that the powerlists to be tied or zipped be similar. This is the formal equivalent of the restriction that `|` and `⊗` only apply to powerlists of the same length. The theorem can be stated as follows:

```

(defthm zip-regular
  (implies (and (p-regular-p x)
                (p-similar-p x y))
           (p-regular-p (p-zip x y))))

```

Another group of theorems explore the interaction between `p-regular-p` and `p-similar-p` powerlists. For example, we have that the unzips and unties of regular powerlists are similar with the following event:

```

(defthm regular-similar-unzip-untie
  (implies (and (powerlist-p x)
                (p-regular-p x))
           (and (p-similar-p (p-unzip-l x) (p-unzip-r x))
                (p-similar-p (p-unzip-l x) (p-untie-l x))
                (p-similar-p (p-unzip-r x) (p-untie-r x)))))

```

This particular theorem provides the missing similarity assertion of laws *L1a* and *L1b*.

We can also prove similar theorems, such as a powerlist similar to a regular powerlist is also regular. This is why we could remove one of the recursive `p-regular-p` instances in the definition of `p-regular-p`. We choose not to

because of symmetry, and also because having the extra condition immediately available may be useful when `p-regular-p` is found as a hypothesis in a theorem.

In our experience, `p-similar-p` is much more useful than `p-regular-p`, since similarity ensures that a function taking more than one argument can recurse on one of the arguments and still visit all the nodes of the other argument, e.g., for pairwise addition of powerlists. In fact, the main use of `p-regular-p` is to show that two powerlists are similar. This occurs when a single powerlist is split and a function applied to the two halves. It also occurs when two powerlists are traversed in a non-standard ordering, e.g., by splitting them into left and right halves and then combining the left half of one with the right half of the other or by splitting with `unzip` and combining with `tie`. In these cases, we use the `p-regular-p` condition to ensure that all of the pieces that can be split are `p-similar-p` to each other, and we can use whatever function of two lists we wish to process them.

2.5 Functions on Powerlists

When working with powerlists, many similar functions, usually small and incidental to the main theorem, are encountered. For example, we may have to add all the elements of a powerlist, or find their minimum or maximum, etc. We may also have to take two powerlists and return their pairwise sum, product, etc. Moreover, we often wish to prove similar theorems about these functions, such as the sum (maximum, minimum) of the sum (maximum, minimum) of two powerlists is the same as the sum (maximum, minimum) of their zip. This is a perfect opportunity to use ACL2's encapsulation primitive to prove the appropriate theorem schemas, which can later be instantiated with specific functions in mind.

To illustrate our approach, consider the following encapsulation:

```
(encapsulate
  ((fn1      (x)  t)
   (fn2-accum (x y) t)
   (equiv    (x y) t))

  (local (defun fn1      (x) (fix x)))
  (local (defun fn2-accum (x y) (+ (fix x) (fix y))))
  (local (defun equiv    (x y) (equal x y)))

  (defthm fn1-scalar
    (implies (not (powerlist-p x))
             (not (powerlist-p (fn1 x)))))

  (defthm fn2-accum-commutative
    (equiv (fn2-accum x y) (fn2-accum y x)))

  (defthm fn2-accum-associative
```

```

(equiv (fn2-accum (fn2-accum x y) z)
       (fn2-accum x (fn2-accum y z))))

(defcong equiv equiv (fn2-accum x y) 1)
(defcong equiv equiv (fn2-accum x y) 2)

(defequiv equiv))

```

This defines `fn1` as a scalar function, `fn2-accum` as an associative-commutative binary function, and `equiv` as an equivalence relation. Outside of the encapsulation, nothing is known about the functions other than the constraints proved in the encapsulate. Hence, any theorems that can be proved about these functions could also be proved about any functions that satisfy the constraints. In effect, theorems about `fn1`, `fn2-accum`, and `equiv` are theorem schemas, which can be instantiated for any suitable function. This allows the basic proof pattern to be derived once and to be used in multiple instances thereafter.

Note that the functions must be defined locally in the encapsulate event. This simple requirement allows ACL2 to verify that the constraints about the functions are not contradictory, thus ensuring that the resulting theory is sound. This is a departure from the use of `functional-instantiate` in Nqthm, which relied on possibly unsound applications of `add-axiom`.

As a motivating example, consider applying `fn1` to all the elements of a powerlist, e.g., squaring all values in a powerlist. Another example uses `fn2-accum` to accumulate all the values in the powerlist into an aggregate. Both of these functions can be defined in two obvious ways, namely recursing in terms of either `p-tie` or `p-zip`. Naturally, we expect the result to be the same, regardless of which way the function is defined. So for example, we would expect to prove the following:

```

(defun a-zip-fn2-accum-fn1 (x)
  (if (powerlist-p x)
      (fn2-accum (a-zip-fn2-accum-fn1 (p-unzip-l x))
                 (a-zip-fn2-accum-fn1 (p-unzip-r x)))
      (fn1 x)))

(defun b-tie-fn2-accum-fn1 (x)
  (if (powerlist-p x)
      (fn2-accum (b-tie-fn2-accum-fn1 (p-untie-l x))
                 (b-tie-fn2-accum-fn1 (p-untie-r x)))
      (fn1 x)))

(defthm a-zip-fn2-accum-fn1-same-as-b-tie-fn2-accum-fn1
  (equiv (b-tie-fn2-accum-fn1 x)
         (a-zip-fn2-accum-fn1 x)))

```

At this point, it is not clear that we have done anything important. After all, we have proved an abstract theorem which seems a bit contrived. How often,

we can ask, will one define a function first in terms of `p-zip`, then in terms of `p-tie`? And if we do not define such functions, say by arbitrarily choosing to define them in terms of `p-tie` always, the above is wasted effort.

It is difficult at this time to adequately address this issue, though it will become clear when we look at the examples. For now, the following intuition may suffice. While simple functions, such as the above, are just as easily defined in terms of `p-tie` as `p-zip`, this is not the case for more complex functions. For example, consider the function `p-ascending-p` which is true for an ascending powerlist. This is much more easily expressed in terms of `p-tie`, since it is simpler to decide when the `p-tie` of two ascending powerlists is ascending than to decide when their `p-zip` is ascending. On the other hand the function `p-batcher-merge` is naturally expressed in terms of `p-zip`, since it works by successively merging the odd- and even-indexed elements of a powerlist. Naturally, when proving theorems about `p-ascending-p`, we will wish to use functions defined in terms of `p-tie`. Such a function may find the minimum of a powerlist. But when reasoning about `p-batcher-merge`, we will need the same function, only this time we may prefer to write it in terms of `p-zip`, so that it “opens up” the same way in an inductive proof. What is left then is the glue to tie these two definitions of the function together. This is an explicit instance of the theorem schema above.

In fact, it should be pointed out that the creation of these theorem schemas came as a direct result of having proved a seemingly endless stream of similar small theorems. It is these theorems that formed the basis of the theorem schema above; i.e., all these abstract theorems were constructed by “unifying” needed lemmas in one specific proof of another. To reinforce this, consider the accumulators above. The scalar function `fn1` seems unnecessary, as does the equivalence relation `equiv`. It would be simpler to state the theorems purely in terms of `fn2-accum` which is the binary operator we are trying to abstract and `equal`. However, the forms above were suggested by the specific instances we wished to create. One such instance is `minimum` where the accumulator is the `min` function and `equiv` and `fn1` are the equality and identity functions, respectively. Another instance is `list-of-type` where the accumulator is the `and` function, `equiv` the `iff` function, and `fn1` a scalar `type-p` function.

Accepting for now that this effort is not wasted, we can consider some of the theorems we found useful. As expected by now, a key series of lemmas show how the functions `a-zip-fn2-accum-fn1` and `b-tie-fn2-accum-fn1` behave with respect to the constructors and destructors of `p-tie` and `p-zip`; for example, the following theorems relate `b-tie-fn2-accum-fn1` to `p-zip`:

```
(defthm zip-b-tie-fn2-accum-fn1
  (equiv (b-tie-fn2-accum-fn1 (p-zip x y))
        (fn2-accum (b-tie-fn2-accum-fn1 x)
                  (b-tie-fn2-accum-fn1 y))))
(defthm unzip-b-tie-fn2-accum-fn1
  (implies (powerlist-p x)
           (equiv
```

```
(fn2-accum (b-tie-fn2-accum-fn1 (p-unzip-l x))
           (b-tie-fn2-accum-fn1 (p-unzip-r x)))
(b-tie-fn2-accum-fn1 x)))
```

Both of these theorems are useful in establishing the antecedent of induction hypotheses.

ACL2 provides a special type of rule which seems tailor made for this purpose. The `:definition` rule type allows alternate definitions to be specified for a given function; e.g., it allows the function `fn2-accum-fn1` to be given a definition in terms of `p-tie` and one in terms of `p-zip`, provided their equivalence can be proved. The ACL2 rewriter decides which definition to use when expanding `fn2-accum-fn1` depending on the other terms in the theorem to be proved. So for example, if a theorem contains many instances of `p-zip`, the `p-zip` definition of `fn2-accum-fn1` would be chosen. While this approach seems promising, we observed a significant performance degradation when we used it.

3 Simple Examples

In this section, we take various examples from [Mis94] and prove them in ACL2. Our goal is to show how the primitives defined in section 2 are sufficient for ACL2 to prove theorems about powerlists.

We start with the `p-reverse` function, which reverses a powerlist. The definition, a straight transliteration from [Mis94], is as follows:

```
(defun p-reverse (p)
  (if (powerlist-p p)
      (p-tie (p-reverse (p-untie-r p))
            (p-reverse (p-untie-l p)))
      p))
```

Similarly, we can define `p-reverse-zip`, which reverses in terms of `p-zip` instead of `p-tie`. ACL2 can immediately verify that `p-reverse` is its own inverse. That is, it trivially accepts the following theorem:

```
(defthm reverse-reverse
  (equal (p-reverse (p-reverse x)) x))
```

Before proving that `p-reverse` and `p-reverse-zip` are equal, however, we need the following lemma:

```
(defthm reverse-zip
  (equal (p-zip (p-reverse x) (p-reverse y))
         (p-reverse (p-zip y x))))
```

This lemma, typical of both Nqthm and ACL2 lemmas, tells ACL2 how to “push” `p-zip` into a `p-reverse`. Given this lemma, ACL2 can now easily verify the following:

```
(defthm reverse-reverse-zip
  (equal (p-reverse-zip x) (p-reverse x)))
```

It is interesting to note that the theorem above does not depend on the structure of the powerlist x . Specifically, there is no requirement that x is regular.

The functions `p-rotate-right` and `p-rotate-left` are easily defined in terms of `p-zip`; indeed their simplicity is a tribute to the `p-zip` constructor:

```
(defun p-rotate-right (x)
  (if (powerlist-p x)
      (p-zip (p-rotate-right (p-unzip-r x)) (p-unzip-l x))
      x))
(defun p-rotate-left (x)
  (if (powerlist-p x)
      (p-zip (p-unzip-r x) (p-rotate-left (p-unzip-l x)))
      x))
```

Again, ACL2 can prove a number of theorems unassisted. For example, it can show that `p-rotate-right` and `p-rotate-left` are inverses with the following theorem:

```
(defthm rotate-left-right
  (equal (p-rotate-left (p-rotate-right x)) x))
```

Notice, again, that the theorem remains true even for arbitrary powerlists, not just regular powerlists. ACL2 can also prove the analogous theorem where we rotate to the left first.

In addition, ACL2 proves the following surprising identity:

```
(defthm rotate-reverse-rotate
  (equal (p-rotate-right (p-reverse-zip (p-rotate-right x)))
         (p-reverse-zip x)))
```

This theorem can be used to prove the following “amusing identity” due to Misra:

```
(defthm reverse-rotate-reverse-rotate
  (equal (p-reverse-zip
          (p-rotate-right
           (p-reverse-zip
            (p-rotate-right x))))
         x))
```

Next, we consider repeated shifts. The function `p-rotate-right-k` loops over `p-rotate-right` k times:

```
(defun p-rotate-right-k (x k)
  (if (zp k)
      x
      (p-rotate-right (p-rotate-right-k x (1- k)))))
```

A subtler definition shifts the odd-indexed and even-indexed elements by about half of k , then joins the result. This is given below:

```
(defun p-rotate-right-k-fast (x k)
  (if (powerlist-p x)
      (if (integerp (/ k 2))
          (p-zip (p-rotate-right-k-fast (p-unzip-l x)
                                         (/ k 2))
                 (p-rotate-right-k-fast (p-unzip-r x)
                                         (/ k 2)))
          (p-zip (p-rotate-right-k-fast (p-unzip-r x)
                                         (1+ (/ (1- k) 2)))
                 (p-rotate-right-k-fast (p-unzip-l x)
                                         (/ (1- k) 2))))
      x))
```

ACL2 can prove the equality of these two functions, but only with a certain amount of help, partly because ACL2 has a hard time reasoning about the values in k above.

Another function suggested by Misra is the shuffle function, which rotates not the elements of a powerlist, but their index, based on zero-indexing. For example, the low-order bit of the index becomes the high-order bit, and hence the even-indexed elements will appear at the front of the result. This function can be defined as follows:

```
(defun p-right-shuffle (x)
  (if (powerlist-p x)
      (p-tie (p-unzip-l x) (p-unzip-r x))
      x))
```

It is especially interesting, because it mixes the `p-zip` destructors with the `p-tie` constructor. Once more, ACL2 is able to prove without assistance that `p-left-shuffle` and `p-right-shuffle` are inverses:

```
(defthm left-right-shuffle
  (equal (p-left-shuffle (p-right-shuffle x)) x))
```

Notice again that the theorem is true regardless of whether the powerlist x is regular. This is slightly surprising when we consider that the functions were defined precisely with a regular powerlist in mind.

Another interesting permutation function is `p-invert` which reverses the bit vector of the index of a powerlist. This function is used, for example, in the Fast Fourier Transform algorithm. It can be defined as follows:

```
(defun p-invert (x)
  (if (powerlist-p x)
      (p-zip (p-invert (p-untie-l x))
             (p-invert (p-untie-r x)))
      x))
```

Following [Mis94], we can prove the following lemma:

```
(defthm invert-zip
  (equal (p-invert (p-zip x y))
         (p-tie (p-invert x) (p-invert y))))
```

It is interesting that this lemma, although typical of ACL2 lemmas, was actually needed in Misra's original hand proof. As in [Mis94], ACL2 can now prove, without user intervention, that `p-invert` is its own inverse. Moreover, it can prove that `p-invert` and `p-reverse` commutes:

```
(defthm invert-invert
  (equal (p-invert (p-invert x)) x))
```

```
(defthm invert-reverse
  (equal (p-invert (p-reverse x))
         (p-reverse (p-invert x))))
```

Finally, we can show that given an arbitrary binary function `fn2` (similar to the one encapsulated in section 2.5) applied pairwise to the elements of two lists, `p-invert` and `fn2` commute:

```
(defthm invert-zip-fn2
  (implies (p-similar-p x y)
           (equal (p-invert (a-zip-fn2 x y))
                  (a-zip-fn2 (p-invert x) (p-invert y)))))
```

4 Sorting Powerlists

We turn our attention to the problem of sorting a powerlist. Our specification is as follows:

```
(defun p-sorted-p (x)
  (if (powerlist-p x)
      (and (p-sorted-p (p-untie-l x))
           (p-sorted-p (p-untie-r x))
           (<= (p-max-elem (p-untie-l x))
              (p-min-elem (p-untie-r x))))
      t))
```

where the functions `p-min-elem` and `p-max-elem` return the minimum and maximum elements of a list respectively. We show how `p-min-elem` is defined.

```
(defun p-min-elem (x)
  (if (powerlist-p x)
      (if (<= (p-min-elem (p-untie-l x))
            (p-min-elem (p-untie-r x)))
          (p-min-elem (p-untie-l x))
          (p-min-elem (p-untie-r x)))
      (rfix x)))
```

Notice how `p-sorted-p` is most naturally expressed in terms of `p-tie`; in fact, it is not immediately obvious how an equivalent definition can be written in terms of `p-zip`. For this reason, we choose to define `p-min-elem` in terms of `p-tie`, though it could just as easily have been defined in terms of `p-zip`. However, since it is likely that we will want to reason about `p-zip` in the future, we can prepare by proving theorems such as the following:

```
(defthm min-elem-zip
  (equal (p-min-elem (p-zip x y))
    (if (<= (p-min-elem x) (p-min-elem y))
      (p-min-elem x)
      (p-min-elem y))))

(defthm min-elem-unzip
  (implies (powerlist-p x)
    (and (>= (p-min-elem (p-unzip-l x))
      (p-min-elem x))
      (>= (p-min-elem (p-unzip-r x))
      (p-min-elem x)))))
```

Both of these theorems are instances of generic theorems proved in section 2.5, so ACL2 does not need to perform added work in proving them (given an appropriate hint to instantiate the generic theorems). Moreover, since different sorting algorithms are likely to require similar theorems about `p-min-elem`, `p-sorted-p`, and so on, it pays to prove these up front. For example, we can establish once and for all that the minimum of a powerlist is no larger than its maximum. We can also prove how `p-sorted` behaves in the presence `p-zip`, etc.

An oft forgotten requirement of sorting is that it not only return a sorted list, but that it return a permutation of its argument. To ensure this, we can define the following function, which returns the number of times a given argument appears in a powerlist:

```
(defun p-member-count (x m)
  (if (powerlist-p x)
    (+ (p-member-count (p-untie-l x) m)
      (p-member-count (p-untie-r x) m))
    (if (equal x m) 1 0)))
```

Again, we can prove basic theorems about `p-member-count`, such as how it behaves with `p-zip`, since these lemmas will likely prove useful to any sorting algorithm.

In summary, we will require that a proposed sorting algorithm `p-sort` satisfy the following theorems:

- `(p-sorted-p (p-sort x))`
- `(equal (p-member-count (p-sort x) m) (p-member-count x m))`

Of course, we may allow specific sorting routines to impose restrictions on the original powerlist x , e.g., a routine may only work with numeric lists.

4.1 Merge Sorting

Merge sort is the most natural parallel sorting algorithm. We can write an abstract merge sort over powerlists as follows:

```
(defun my-merge-sort (x)
  (if (powerlist-p x)
      (p-merge (my-merge-sort (p-split-1 x))
               (my-merge-sort (p-split-2 x)))
      x))
```

The functions `p-merge`, and `p-split-1` and `p-split-2` instantiate specific merge sort algorithms. Classically, `p-merge` will be a complicated function and the split functions will be trivial. What we would like to do is to encapsulate these functions and their relevant theorems and then prove the correctness of this generic merge sort. In particular, we wish to establish the following theorems:

```
(defthm merge-sort-is-permutation
  (implies (p-sortable-p x)
            (equal (p-member-count (p-merge-sort x) m)
                   (p-member-count x m))))

(defthm merge-sort-sorts-input
  (implies (p-sortable-p x)
            (p-sorted-p (p-merge-sort x))))
```

The `p-sortable-p` goal lets us specify merge algorithms that only work for a subclass of powerlists; the forthcoming Batcher merge, which only works for regular powerlists, is an example of such an algorithm.

In order to prove the theorems above, we need the following assumptions about the generic merge functions:

```
(encapsulate
  ((p-sortable-p (x) t)
   (p-mergeable-p (x y) t)
   (p-split-1 (x) t)
   (p-split-2 (x) t)
   (p-merge (x y) t)
   (p-merge-sort (x) x))

(defthm *obligation*-split-reduces-count
  (implies (powerlist-p x)
            (and (e0-ord-< (acl2-count (p-split-1 x))
                           (acl2-count x))
```

```

(e0-ord-< (acl2-count (p-split-2 x))
          (acl2-count x))))

(defthm *obligation*-member-count-of-splits
  (implies (powerlist-p x)
    (equal (+ (p-member-count (p-split-1 x) m)
              (p-member-count (p-split-2 x) m))
            (p-member-count x m))))

(defthm *obligation*-member-count-of-merge
  (implies (p-mergeable-p x y)
    (equal (p-member-count (p-merge x y) m)
            (+ (p-member-count x m)
               (p-member-count y m)))))

(defthm *obligation*-sorted-merge
  (implies (and (p-mergeable-p x y)
                (p-sorted-p x)
                (p-sorted-p y))
    (p-sorted-p (p-merge x y))))

(defthm *obligation*-merge-sort
  (equal (p-merge-sort x)
    (if (powerlist-p x)
        (p-merge (p-merge-sort (p-split-1 x))
                  (p-merge-sort (p-split-2 x)))
        x)))

(defthm *obligation*-sortable-split
  (implies (and (powerlist-p x)
                (p-sortable-p x))
    (and (p-sortable-p (p-split-1 x))
          (p-sortable-p (p-split-2 x)))))

(defthm *obligation*-sortable-mergeable
  (implies (and (powerlist-p x)
                (p-sortable-p x))
    (p-mergeable-p (p-merge-sort (p-split-1 x))
                    (p-merge-sort (p-split-2 x)))))

```

Recall, however, that before ACL2 accepts such an `encapsulate` event, it must be given a witness function; that is, an implementation of such a merging scheme. The easiest route is to use a vacuous merger, i.e., by locally defining `p-sortable-p` to be `nil`.

4.2 Batchers Sorting

The Batchers merging algorithm can be defined as follows:

```
(defun p-batcher-merge (x y)
  (if (powerlist-p x)
      (p-zip (p-min (p-batcher-merge (p-unzip-l x)
                                     (p-unzip-r y))
                (p-batcher-merge (p-unzip-r x)
                                  (p-unzip-l y)))
            (p-max (p-batcher-merge (p-unzip-l x)
                                    (p-unzip-r y))
                    (p-batcher-merge (p-unzip-r x)
                                       (p-unzip-l y))))
      (p-zip (p-min x y) (p-max x y))))
```

The functions `p-min` and `p-max` return respectively the pairwise minimum and maximum of two powerlists. Since `p-zip` features prominently in the definition of `p-batcher-merge`, we expect to find `p-min` and `p-max` similarly defined.

At first glance, the definition of `p-batcher-merge` looks straight-forward. Certainly, it seems that a straight-forward structural induction should be sufficient to prove all the properties about it one would wish. Such a blissful perspective will most likely be short-lived. There are two imposing challenges ahead. The first is that `p-batcher-merge` is defined in terms of `p-zip`, whereas our target predicate `p-sorted-p` is defined in terms of `p-tie`. This is usually enough to make even a simple proof a little challenging. But in this case it is especially troublesome, because `p-batcher-merge` does not recurse evenly through its arguments. Notice in particular how the *left* unzip of `x` is merged with the *right* unzip of `y`, and vice versa.

Upon further consideration, the definition of `p-batcher-merge` seems to pose an unsurmountable challenge to verification. An induction scheme based on `p-batcher-merge` will provide assertions about the left half of `x` mixed with the right half of `y`. But to complete the proof, we will also need assertions about corresponding halves of `x` and `y`. One readily envisions nests of left halves of right unzips of left halves. . .

Clearly, more caution than usual is required to verify this function.

Consider first the proof of the following goal:

```
(equal (p-member-count (p-batcher-merge x y) m)
       (+ (p-member-count x m)
          (p-member-count y m)))
```

Since `p-min` and `p-max` operate on the pairwise points of `x` and `y`, it is reasonable to require that `x` and `y` be similar. Moreover, since `p-batcher-merge` is recursing on opposite halves of `x` and `y`, we can expect that the powerlists must also be regular. It turns out that we will also need to constrain the powerlist to be numeric. This is because the ordering imposed by `p-max` is only well-defined

over this domain. Of course, we will have to prove the theorems that all intermediate results satisfy the structural requirements of the hypothesis; i.e., we must establish that for similar x and y their p -min and p -max are also similar, etc.

Our goal becomes the following:

```
(defthm member-count-of-merge
  (implies (and (p-regular-p x)
                (p-similar-p x y)
                (p-number-list x)
                (p-number-list y))
           (equal (p-member-count (p-batcher-merge x y) m)
                  (+ (p-member-count x m)
                     (p-member-count y m))))))
```

To prove the above claim, we must first establish that all the values of x and y can be found somewhere in their p -min and p -max. We can prove this generically; that is, we can prove that the sum of any scalar function over x and y is unaffected by p -min and p -max:

```
(defthm a-zip-plus-fn1-of-min-max
  (implies (and (p-similar-p x y)
                (p-number-list x)
                (p-number-list y))
           (equal (+ (a-zip-plus-fn1 (p-max x y))
                    (a-zip-plus-fn1 (p-min x y)))
                  (+ (a-zip-plus-fn1 x)
                     (a-zip-plus-fn1 y)))))
```

Notice how we are extending the generic theorems defined in section 2.5 to include specific functions, such as p -min and p -max. With this lemma, we can prove the similar result for p -batcher-merge:

```
(defthm a-zip-plus-fn1-of-merge
  (implies (and (p-regular-p x)
                (p-similar-p x y)
                (p-number-list x)
                (p-number-list y))
           (equal (a-zip-plus-fn1 (p-batcher-merge x y))
                  (+ (a-zip-plus-fn1 x)
                     (a-zip-plus-fn1 y)))))
```

Instantiating `fn1` with the pseudo-function `(lambda (x) (if (= x m) 1 0))` and using the equivalence of `a-zip-plus-fn1` and `b-tie-plus-fn1`, we can prove our original goal.

Notice above how all the reasoning was done with respect to p -zip, and only in the last step did we appeal to the equivalence of p -member-count as defined in terms of p -zip and p -tie to complete the proof.

We must now tackle the question of when `p-batcher-merge` returns a sorted powerlist. The recursive step returns a powerlist of the form

```
(p-zip (p-min (p-batcher-merge X1 Y2)
              (p-batcher-merge X2 Y1))
       (p-max (p-batcher-merge X1 Y2)
              (p-batcher-merge X2 Y1)))
```

We know from the inductive hypothesis it will be easy to establish that both `(p-batcher-merge X1 Y2)` and `(p-batcher-merge X2 Y1)` are sorted. It is natural to ask, therefore, whether `(p-zip (p-min X Y) (p-max X Y))` is sorted, given sorted `X` and `Y`. Unfortunately, this is not the case, as the powerlists $\langle 1, 2 \rangle$ and $\langle 3, 4 \rangle$ demonstrate. The problem is that the `p-min` of 2 and 4 is 2, which is smaller than the `p-max` of 1 and 3. What we need is to ensure that the elements of the lists are not only sorted independently, but that one lists does not “grow” too much faster than the other.

Consider $X = \langle x_1, x_2, x_3, x_4 \rangle$ and $Y = \langle y_1, y_2, y_3, y_4 \rangle$. Our condition amounts to the following:

$$\max(x_i, y_i) \leq \min(x_j, y_j)$$

for all indices $i < j$. This condition automatically implies that X and Y are sorted. We can write this in ACL2 as follows:

```
(defun p-interleaved-p (x y)
  (if (powerlist-p x)
      (and (powerlist-p y)
           (p-interleaved-p (p-untie-l x) (p-untie-l y))
           (p-interleaved-p (p-untie-r x) (p-untie-r y))
           (<= (p-max-elem (p-untie-l x))
              (p-min-elem (p-untie-r x)))
           (<= (p-max-elem (p-untie-l x))
              (p-min-elem (p-untie-r y)))
           (<= (p-max-elem (p-untie-l y))
              (p-min-elem (p-untie-r x)))
           (<= (p-max-elem (p-untie-l y))
              (p-min-elem (p-untie-r y))))
      (not (powerlist-p y))))
```

So now, if `(p-interleaved-p x y)` is true, we would like to show that `(p-zip (p-min x y) (p-max x y))` is sorted. Intuitively, this is a simple result. In our example above, the first two elements of Z will be x_1 and y_1 , in ascending order. Moreover, the hypothesis assures us these two numbers are the smallest of the x_j and y_j for $j \geq 2$. Similarly, we can reason about x_2 and y_2 , and so on.

To prove the claim in ACL2, we have to reason about the interaction of `p-min` and `p-min-elem`, as well as their `max` counterparts. Since `p-min` is defined in terms of `p-zip` and `p-min-elem` in terms of `p-tie`, it is easier to prove this theorems in terms of a single recursive scheme, say `p-tie` and then use the bridging lemmas to prove the result:

```

(defthm zip-min-max-sorted-if-interleaved
  (implies (and (p-interleaved-p x y)
                (p-similar-p x y)
                (p-number-list x)
                (p-number-list y))
           (p-sorted-p (p-zip (p-min x y) (p-max x y)))))

```

Again, it is easier at first to prove this for `p-min-tie` and `p-max-tie`, since `p-sorted-p` is defined in terms of `p-tie`.

We have only to show that the recursive calls to `p-batcher-merge` return `p-interleaved-p` lists. That is, given sorted `X` and `Y`,

```

L1 = (p-batcher-merge (p-unzip-l X) (p-unzip-r Y))
L2 = (p-batcher-merge (p-unzip-r X) (p-unzip-l Y))

```

are `p-interleaved-p`. We can use our intuition to see why this must be the case. We can assume that both `L1` and `L2` are sorted, since this fact will follow from the induction hypothesis. Any prefix of `L1` will have some values from `X` and some from `Y`, say i and j values respectively. Moreover, since `L1` has only odd-indexed elements of `X` and `L2` only the even-indexed elements of `X`, no prefix of `L1` can have more elements from `X` than the corresponding prefix of `L2`, and similarly for the elements from `Y`. For example, suppose that `L1` starts with x_1 and x_3 , but the corresponding prefix of `L2` does not contain x_2 . In this case, `L2` must start with y_1 and y_3 , which means that $y_3 < x_2$, since `L2` is sorted and its prefix does not contain x_2 . But, we can conclude from `L1` that $x_3 \leq y_2$, since `L1` is also sorted. We have then that $x_3 \leq y_2 \leq y_3 < x_2$ and so $x_3 < x_2$. But this is a contradiction, since `X` is sorted.

Formalizing the argument given above places a severe challenge on the powerlist paradigm, since the reasoning involves indices so explicitly, whereas powerlists do away with the index concept. In fact, the whole concept of “prefix” is strange, since these prefixes will by definition be irregular, and we have already observed how `p-batcher-merge` requires regular arguments. This calls for a little subtlety in our approach. We can replace the “prefix” concept with the following:

```

(defun p-member-count-<= (x m)
  (if (powerlist-p x)
      (+ (p-member-count-<= (p-untie-l x) m)
         (p-member-count-<= (p-untie-r x) m))
      (if (<= (rfix x) m) 1 0)))

```

This returns the number of elements in `x` which are less than or equal to `m`; that is, for an element `m` in `x`, it returns its (largest) index in `x`. With this notion, we can formalize our argument involving the “prefix” of a powerlist.

We are interested in expressions of the form

```

M1 = (p-member-count-<= (p-batcher-merge (p-unzip-l x)
                                          (p-unzip-r y))

```

```

m)
M2 = (p-member-count-<= (p-batcher-merge (p-unzip-r x)
                                           (p-unzip-l y))
m)

```

so we begin with the following theorem:

```

(defthm member-count-<=of-merge
  (implies (and (p-regular-p x)
                (p-similar-p x y)
                (p-number-list x)
                (p-number-list y))
    (equal (p-member-count-<= (p-batcher-merge x y) m)
           (+ (p-member-count-<= x m)
              (p-member-count-<= y m))))))

```

This theorem allows us to remove `p-batcher-merge` from the computation of `p-member-count`. We are left with the following:

```

M1 = (+ (p-member-count-<= (p-unzip-l x) m)
        (p-member-count-<= (p-unzip-r y) m))
M2 = (+ (p-member-count-<= (p-unzip-r x) m)
        (p-member-count-<= (p-unzip-l y) m))

```

So the next step will be to compare the `p-member-count-<=` of the `p-unzip-l` and `p-unzip-r` of a powerlist, specifically a *sorted* powerlist. Intuitively, we expect these to differ by no more than 1; moreover, since the `p-unzip-r` starts counting from the second position, we expect its `p-member-count-<=` to be smaller than that of the `p-unzip-l`. In fact, we can prove the following theorems:

```

(defthm member-count-<=of-sorted-unzips-1
  (implies (and (powerlist-p x)
                (p-regular-p x)
                (p-sorted-p x))
    (<= (p-member-count-<= (p-unzip-r x) m)
        (p-member-count-<= (p-unzip-l x) m))))

```

```

(defthm member-count-<=of-sorted-unzips-2
  (implies (and (powerlist-p x)
                (p-regular-p x)
                (p-sorted-p x))
    (<= (p-member-count-<= (p-unzip-l x) m)
        (1+ (p-member-count-<= (p-unzip-r x) m))))))

```

Putting it all together, we end up with the following theorem, which states `M1` and `M2` differ by no more than 1:

```

(defthm member-count-<=-of-merge-unzips
  (implies (and (powerlist-p x)
                (p-regular-p x)
                (p-similar-p x y)
                (p-number-list x)
                (p-number-list y)
                (p-sorted-p x)
                (p-sorted-p y))
    (let ((M1 (p-member-count-<= (p-batcher-merge
                                  (p-unzip-l x)
                                  (p-unzip-r y))
                                  m))
          (M2 (p-member-count-<= (p-batcher-merge
                                  (p-unzip-r x)
                                  (p-unzip-l y))
                                  m)))
      (or (equal M1 M2)
          (equal (1+ M1) M2)
          (equal (1+ M2) M1))))))

```

The next step is to show that for non `p-interleaved-p` lists, there is some `m` so that the respective `p-member-count-<=` differ by more than 1. We can find this `m` by making a “cut” through the two lists at the precise spot where they fail the `p-interleaved-p` test. The following function performs such a “cut”:

```

(defun interleaved-p-cutoff (x y)
  (if (and (powerlist-p x) (powerlist-p y))
      (cond ((< (p-min-elem (p-untie-r x))
                (p-max-elem (p-untie-l x)))
             (p-min-elem (p-untie-r x)))
          ((< (p-min-elem (p-untie-r x))
                (p-max-elem (p-untie-l y)))
             (p-min-elem (p-untie-r x)))
          ((interleaved-p-cutoff (p-untie-l x)
                                  (p-untie-l y))
             (interleaved-p-cutoff (p-untie-l x)
                                  (p-untie-l y)))
          ((interleaved-p-cutoff (p-untie-r x)
                                  (p-untie-r y))
             (interleaved-p-cutoff (p-untie-r x)
                                  (p-untie-r y)))
          (p-untie-r y)))
      nil))

```

When `x` and `y` are `p-interleaved-p`, the function `interleaved-p-cutoff` will return `nil`. In all other cases, it returns a valid choice of `m` as a counterexample to `member-count-<=-of-merge-unzips`. We can trivially show the first observation as follows:

```

(defthm interleaved-p-if-nil-cutoff
  (implies (and (p-similar-p x y)
                (p-number-list x)
                (p-number-list y)
                (not (numericp (interleaved-p-cutoff x y)))
                (not (numericp (interleaved-p-cutoff y x))))
            (p-interleaved-p x y)))

```

In order to establish that `interleaved-p-cutoff` finds a valid counterexample when `x` and `y` are not `p-interleaved-p`, notice that `interleaved-p-cutoff` always returns an element of `x`, and furthermore for sorted `x` this value `m` is such that its “index” in `x` is at least one more than its “index” in `y`, since it must satisfy

```

(< (p-min-elem (p-untie-r x)) (p-max-elem (p-untie-l y)))

```

for some corresponding subtree of `x` and `y`. In `ACL2`, we can prove the following theorem:

```

(defthm member-count-diff-2-if-interleaved-cutoff-sorted
  (implies (and (p-similar-p x y)
                (p-number-list x)
                (p-number-list y)
                (p-sorted-p x)
                (p-sorted-p y)
                (interleaved-p-cutoff x y))
            (< (1+ (p-member-count-<=
                  y
                  (interleaved-p-cutoff x y)))
              (p-member-count-<=
                x
                (interleaved-p-cutoff x y)))))

```

This theorem serves to find the counterexample needed by the two lemmas `member-count-<=of-merge-unzips` and `interleaved-p-if-nil-cutoff`, so we can now establish the following key theorem:

```

(defthm inner-batcher-merge-call-is-interleaved-p
  (implies (and (powerlist-p x)
                (p-regular-p x)
                (p-similar-p x y)
                (p-number-list x)
                (p-number-list y)
                (p-sorted-p x)
                (p-sorted-p y)
                (p-sorted-p (p-batcher-merge (p-unzip-l x)
                                              (p-unzip-r y)))
                (p-sorted-p (p-batcher-merge (p-unzip-r x)
                                              (p-unzip-l y))))
            (p-interleaved-p x y)))

```

```

(p-interleaved-p (p-batcher-merge (p-unzip-l y)))
(p-interleaved-p (p-batcher-merge (p-unzip-l x)
                                   (p-unzip-r y))
                 (p-batcher-merge (p-unzip-r x)
                                   (p-unzip-l y))))

```

From this point, the remainder of the proof is almost propositional. We can use `inner-batcher-merge-call-is-interleaved-p` to prove the inductive case of the correctness of `batcher-merge`. It is no accident that the inductive hypothesis shares the antecedent of `inner-batcher-merge-call-is-interleaved-p`.

```

(defthm recursive-batcher-merge-is-sorted
  (implies (and (powerlist-p x)
                (p-regular-p x)
                (p-similar-p x y)
                (p-number-list x)
                (p-number-list y)
                (p-sorted-p x)
                (p-sorted-p y)
                (p-sorted-p (p-batcher-merge (p-unzip-l x)
                                              (p-unzip-r y)))
                (p-sorted-p (p-batcher-merge (p-unzip-r x)
                                              (p-unzip-l y))))
            (p-sorted-p (p-batcher-merge x y))))

```

Almost anticlimactically, we can now prove the main result, which establishes the correctness of Batcher merging:

```

(defthm sorted-merge
  (implies (and (p-regular-p x)
                (p-similar-p x y)
                (p-number-list x)
                (p-number-list y)
                (p-sorted-p x)
                (p-sorted-p y))
            (p-sorted-p (p-batcher-merge x y))))

```

With the theorem above and the meta-theorems about merge sorts proved in section 4.1, we can prove the correctness of Batcher sorting:

```

(defthm batcher-sort-is-permutation
  (implies (and (p-regular-p x)
                (p-number-list x))
            (equal (p-member-count (p-batcher-sort x) m)
                   (p-member-count x m))))
(defthm batcher-sort-sorts-inputs
  (implies (and (p-regular-p x)
                (p-number-list x))
            (p-sorted-p (p-batcher-sort x))))

```

4.3 A Comparison with the Hand-Proof

It is instructive to compare the machine-verified proof of section 4.2 with the hand-proof provided in [Mis94] and verified in [KS95a].

The proof starts by defining the function z as follows:

$$\begin{aligned} z(\langle x \rangle) &= 1 \text{ if } x = 0, 0 \text{ otherwise} \\ z(p \bowtie q) &= z(p) + z(q) \end{aligned}$$

That is, $z(x)$ counts the number of zeros in x . Assuming that all powerlists range only over 0's and 1's, we have the following characterization of sorted powerlists:

$$\begin{aligned} &sorted(\langle x \rangle) \\ sorted(p \bowtie q) &= sorted(p) \wedge sorted(q) \wedge 0 \leq z(p) - z(q) \leq 1 \end{aligned}$$

The 0-1 assumption completely characterizes the pairwise minimum and maximum of two sorted lists as follows:

$$\begin{aligned} min(x, y) &= x, \text{ if } sorted(x), sorted(y), \text{ and } z(x) \geq z(y) \\ max(x, y) &= y, \text{ if } sorted(x), sorted(y), \text{ and } z(x) \leq z(y) \end{aligned}$$

Moreover, the following key lemma can be established:

$$sorted(min(x, y) \bowtie max(x, y)) \text{ if } sorted(x), sorted(y), \text{ and } |z(x) - z(y)| \leq 1$$

With some algebraic reasoning, this yields the main correctness result:

$$sorted(pbm(x, y)) \text{ if } sorted(x) \text{ and } sorted(y)$$

where pbm is the Batcher merge function on powerlists.

This proof is much simpler than that given in section 4.2, and that may be taken as an indication that ACL2 is ineffective in reasoning about powerlists. However, such a conclusion is premature. In fact, ACL2 can verify the reasoning given above without too much difficulty. But the end result would not be as satisfying as the main theorems proven in 4.2 for a number of reasons. First, the hand proof relies on the 0/1 principle, which states that any comparison based sorting algorithm which correctly sorts all lists consisting exclusively of zeros and ones will correctly sort an arbitrary list. The formal proof in the powerlist logic proves the correctness only for lists of zeros and ones, and then uses the 0/1 principle to “lift” this proof to the arbitrary case. But the 0/1 principle is certainly not obvious; if anything, it is more surprising than the proof of Batcher merge itself. For instance, proving the 0/1 principle in ACL2 would be extremely difficult, if not impossible, because the principle references all comparator-based sorting functions, and ACL2 is strictly a first-order logic.

Another problem with the hand proof is that the definition of *sorted* used is not the same as the “standard” definition of a sorted list. It is *only* true for lists of 0's and 1's, and it is not immediately clear how this property compares to

our usual notion of sorted lists. The definition supplied, however, is extremely useful, since it is based on `zip` instead of `tie`, and so it works more naturally with the definition of Batcher merge. However, the proof of the equivalence of the two definitions is missing, and that serves to reinforce the feeling of unease and sense of incompleteness in the final proof. This is especially important if we were to use Batcher sorting as part of a more complex function, e.g. a search routine, since the key property we require in the complex function — that Batcher sort correctly sorts its input — has not been established yet.

In fact, it is fair to say that the hand proof presents a mixture of formal reasoning and informal arguments. Such a mixture is extremely convenient when generating the proof by hand, but it can also be the source of subtle errors, such as the failure to identify needed hypothesis.

5 Prefix Sums of Powerlists

Prefix sums appear in many applications, e.g., arithmetic circuit design. For a powerlist $X = \langle x_1, x_2, \dots, x_n \rangle$, its prefix sum is given by $ps(X) = \langle x_1, x_1 \oplus x_2, \dots, x_1 \oplus x_2 \oplus \dots \oplus x_n \rangle$. The operator \oplus is an arbitrary binary operator; for our purposes, we will assume it to be associative, and to have a left-identity 0.

We use the functions `bin-op` and `left-zero` to encapsulate the binary operator \oplus and its left identity, respectively. We use ACL2's `encapsulate` so that the following theorems are all theorem schemas which can be instantiated with any suitable operator, e.g, `plus`, `and`, `min`, etc. The required axioms are as follows:

```
(encapsulate
  ((domain-p (x) t)
   (bin-op (x y) t)
   (left-zero () t))

  (defthm booleanp-domain-p
    (booleanp (domain-p x)))

  (defthm scalar-left-zero
    (domain-p (left-zero)))

  (defthm domain-powerlist
    (implies (domain-p x)
              (not (powerlist-p x))))

  (defthm left-zero-identity
    (implies (domain-p x)
              (equal (bin-op (left-zero) x) x)))

  (defthm bin-op-assoc
    (equal (bin-op (bin-op x y) z)
```

```

(bin-op x (bin-op y z)))

(defthm scalar-bin-op
  (domain-p (bin-op x y)))
)

```

The function `domain-p` recognizes our intended domain, which is required to be scalar, i.e. non-powerlist. The function `p-domain-list-p` extends `domain-p` over powerlists; i.e., it recognizes powerlists of `domain-p` elements. Note that we require the second argument to be `domain-p` in `left-zero-identity`, but that `domain-p` is not a requirement of `bin-op-assoc`, and furthermore that `domain-p` is always true of the result of `bin-op`. This turns out to be important, in that ACL2 defines many binary operators that meet these requirements precisely. Moreover, we need at least one of these theorems to have `domain-p` as a hypothesis. For example, if we remove the hypothesis from `left-zero-identity`, then for a powerlist x , we would have that $0 \oplus x = x$ and so \oplus would not always return a scalar.

There is a natural definition of prefix sums in terms of indices. That is, entry y_j in the prefix sum of X is equal to the sum of all the x_i up to x_j . However, this definition does not extend nicely to powerlists, since the two halves of a prefix sum are not themselves prefix sums. The trick is to generalize prefix sums to allow an arbitrary value to be added to the first element, in a manner analogous to a carry-in bit. This leads to the following definitions:

```

(defun p-prefix-sum-aux (prefix x)
  (if (powerlist-p x)
      (p-tie (p-prefix-sum-aux prefix (p-untie-l x))
            (p-prefix-sum-aux (p-last (p-prefix-sum-aux
                                      prefix
                                      (p-untie-l x)))
                              (p-untie-r x)))
      (bin-op prefix x)))
(defmacro p-prefix-sum (x)
  '(p-prefix-sum-aux (left-zero) ,x))

```

where `p-last` returns the last element of a powerlist. In the sequel, most of the theorems will be about `p-prefix-sum-aux`, though a few will have to be proved exclusively for `p-prefix-sum`. Alternatively, we could have defined `p-prefix-sum-aux` to pass the sum of the left half of x instead of the last element of the left prefix sum. We chose the current definition, simply because it is closer to the usual way we compute powerlists.

5.1 Simple Prefix Sums

The definition of `p-prefix-sum` is inherently sequential. Our first goal will be to prove that the following definition, more amenable to a parallel implementation, is equivalent:

```

(defun p-simple-prefix-sum (x)
  (if (powerlist-p x)
      (let ((y (p-add (p-star x) x)))
        (p-zip (p-simple-prefix-sum (p-unzip-l y))
                (p-simple-prefix-sum (p-unzip-r y))))
      x))

```

The function `p-add` returns the sum of two powerlists, and `p-star` shifts a powerlist to the right, prefixing the result with `left-zero`:

```

(defun p-star (x)
  (if (powerlist-p x)
      (p-zip (p-star (p-unzip-r x)) (p-unzip-l x))
      (left-zero)))
(defun p-add (x y)
  (if (powerlist-p x)
      (p-zip (p-add (p-unzip-l x) (p-unzip-l y))
              (p-add (p-unzip-r x) (p-unzip-r y)))
      (bin-op x y)))

```

The first problem is that ACL2 does not accept the definition given above for `p-simple-prefix-sum`. The difficulty is that the definition recurses with `x` changing to `(p-unzip-l (p-add (p-star x) x))` and the latter term is not obviously “smaller” than `x`. Therefore, ACL2 can not prove that the recursive definition is well-founded. To circumvent this, we define the following “measure” on powerlists:

```

(defun p-measure (x)
  (if (powerlist-p x)
      (+ (p-measure (p-unzip-l x))
         (p-measure (p-unzip-r x)))
      1))

```

The measure, in effect, counts the number of elements in a powerlist. We next prove theorems showing how `p-star` and `p-add` preserve measures:

```

(defthm measure-star
  (equal (p-measure (p-star x)) (p-measure x)))

(defthm measure-add
  (<= (p-measure (p-add x y)) (p-measure x)))

```

Finally, we provide ACL2 with the hint to use `p-measure` when proving the definition of `p-simple-prefix-sum` is well-founded.

We can now concentrate on the correctness of `p-simple-prefix-sum`. The definition of this function suggests two approaches: we can explore the powerlist given by `(p-add (p-star x) x)`, or we can consider what happens when we `unzip` the prefix sum of `x`. We will take the first approach. Recall that `p-star`

shifts its argument to the right, and that `p-add` returns a pairwise sum. Thus, for `x` given by

$$X = \langle x_1, x_2, x_3, \dots, x_n \rangle$$

`(p-add (p-star x) x)` is

$$Y = X^* \oplus X = \langle x_1, x_1 \oplus x_2, x_2 \oplus x_3, \dots, x_{n-1} \oplus x_n \rangle$$

Taking the `p-unzip` of this powerlist, gives the following:

$$\begin{aligned} Y_1 &= \langle x_1, x_2 \oplus x_3, \dots, x_{n-2} \oplus x_{n-1} \rangle \\ Y_2 &= \langle x_1 \oplus x_2, x_3 \oplus x_4, \dots, x_{n-1} \oplus x_n \rangle \end{aligned}$$

It is clear now that indeed the prefix sum of Y_1 yields precisely the odd-indexed elements of the prefix sum of X and, similarly, the prefix sum of Y_2 yields the even-indexed elements. Thus we can, intuitively at least, verify the correctness of `p-simple-prefix-sum`. To formalize this, it will be convenient to think of Y_1 and Y_2 not as components of Y , but as two separate lists in their own right. This removes the awkward reference to `p-unzip` and allows us to rederive Y_1 and Y_2 in a way more amenable to reasoning about `p-prefix-sum`. We begin with a new characterization of Y_2 :

```
(defun p-add-right-pairs (x)
  (if (powerlist-p x)
      (if (powerlist-p (p-untie-l x))
          (p-tie (p-add-right-pairs (p-untie-l x))
                 (p-add-right-pairs (p-untie-r x)))
              (bin-op (p-untie-l x) (p-untie-r x)))
      x))
```

This redefinition of Y_2 is especially useful, because it is in terms of `p-tie`, not `p-zip`, so it will be easier to reason about its `p-prefix-sum`. To begin with, it is trivial to characterize the prefix sum of the `p-add-right-pairs` of a two-element powerlist — note that a two-element powerlist is the natural base case for an induction, since `p-add-right-pairs` is only reasonable over non-singleton arguments. In particular, we can prove that for a powerlist $X = \langle x_1, x_2 \rangle$, both the prefix sum of its `p-add-right-pairs` and the right unzip of its prefix sum are equal to $x_1 \oplus x_2$:

```
(defthm prefix-sum-p-add-right-pairs-base
  (implies (and (domain-p val)
                 (powerlist-p x)
                 (not (powerlist-p (p-untie-l x)))
                 (p-regular-p x)
                 (p-domain-list-p x))
            (and (equal (p-prefix-sum-aux
                        val
```

```

      (p-add-right-pairs x))
    (bin-op val
      (bin-op (p-untie-l x)
        (p-untie-r x))))
  (equal (p-unzip-r (p-prefix-sum-aux val x))
    (bin-op val
      (bin-op (p-untie-l x)
        (p-untie-r x))))))

```

The definition of `p-prefix-sum` uses the last element of the left prefix sum to compute the right prefix sum. This suggests the following important lemma:

```

(defthm p-last-p-prefix-sum-p-add-right-pairs
  (implies (and (domain-p val)
    (p-regular-p x)
    (p-domain-list-p x))
    (equal (p-last (p-prefix-sum-aux
      val
      (p-add-right-pairs x)))
      (p-last (p-prefix-sum-aux val x)))))

```

Using induction, it is easy to establish that the prefix sum of `p-add-right-pairs` computes the right unzip of the prefix sum of a powerlist:

```

(defthm prefix-sum-p-add-right-pairs
  (implies (and (domain-p val)
    (powerlist-p x)
    (p-regular-p x)
    (p-domain-list-p x))
    (equal (p-prefix-sum-aux val
      (p-add-right-pairs x))
      (p-unzip-r (p-prefix-sum-aux val x)))))

```

The second half of the proof is similar. Consider `p-add-left-pairs`, which is a new characterization of $Y_1 = \langle x_1, x_2 \oplus x_3, \dots, x_{n-2} \oplus x_{n-1} \rangle$:

```

(defun p-add-left-pairs (val x)
  (if (powerlist-p x)
    (if (powerlist-p (p-untie-l x))
      (p-tie (p-add-left-pairs val (p-untie-l x))
        (p-add-left-pairs (p-last (p-untie-l x))
          (p-untie-r x)))
      (bin-op val (p-untie-l x)))
    (bin-op val x)))

```

Unfortunately, the function `p-add-left-pairs` is considerably more complicated than `p-add-right-pairs`. The reason is that in `p-add-right-pairs` there was no need for the left half of the computation to pass any information

over to the right half; i.e., the two recursive calls were completely independent of each other. The net effect is that reasoning about **p-add-left-pairs** is much more difficult than reasoning about **p-add-right-pairs**. However, there is a simple way around this. Consider the powerlist $X = \langle x_1, x_2, x_3, \dots, x_n \rangle$ again. If we shift this powerlist, getting $X' = \langle 0, x_1, x_2, x_3, \dots, x_{n-1} \rangle$, and then take the **p-add-right-pairs** of it, we get $\langle x_1, x_2 \oplus x_3, \dots, x_{n-2} \oplus x_{n-1} \rangle$ which is precisely the **p-add-left-pairs** of X . Moreover, it is clear that the prefix sum of X and the prefix sum of X' are related; specifically, the prefix sum of the shift is the shift of the prefix sum. So, if we can formalize the above intuition, we can use the theorem about **p-add-right-pairs** and not have to reason about **p-add-left-pairs** at all.

Since both **p-add-left-pairs** and **p-add-right-pairs** are defined in terms of **p-tie**, we define **p-shift** in terms of **p-tie**, rather than using the equivalent function **p-star**:

```
(defun p-shift (val x)
  (if (powerlist-p x)
      (p-tie (p-shift val (p-untie-l x))
             (p-shift (p-last (p-untie-l x))
                      (p-untie-r x)))
      val))
```

We first establish that the prefix sum and shift operators commute. That is, we prove the following theorem:

```
(defthm p-prefix-sum-p-shift
  (implies (and (domain-p c1)
                (domain-p c2)
                (p-domain-list-p x))
            (equal (p-prefix-sum-aux c1 (p-shift c2 x))
                   (p-shift (bin-op c1 c2)
                             (p-prefix-sum-aux (bin-op c1 c2)
                                                x))))))
```

The proof of this theorem, requires a subtle induction scheme. In particular, to conclude the theorem, we must consider the two partial prefix sums

```
PS1 = (p-prefix-sum-aux c1 (p-shift c2 (p-untie-l x)))
PS2 = (p-prefix-sum-aux (p-last PS1)
                        (p-shift (p-last (p-untie-l x))
                                (p-untie-r x)))
```

Hence, in the second instance, the term $(\text{bin-op } c1 \ c2)$ in the theorem becomes

```
(bin-op (p-last (p-prefix-sum-aux c1 (p-shift c2 (p-untie-l x))))
        (p-last (p-untie-l x)))
```

which using the inductive hypothesis is equal to the following:

```

(bin-op (p-last (p-shift (bin-op c1 c2)
                        (p-prefix-sum-aux (bin-op c1 c2)
                                           (p-untie-1 x))))
      (p-last (p-untie-1 x)))

```

This term can be simplified into

```

(p-last (p-prefix-sum-aux (bin-op c1 c2) (p-untie-1 x)))

```

using the following technical lemma:

```

(defthm binop-last-shift-prefix-sum
  (implies (domain-p c)
    (equal (bin-op (p-last
                  (p-shift c (p-prefix-sum-aux c x)))
            (p-last x))
          (p-last (p-prefix-sum-aux c x))))))

```

This simplification is the key step in the proof.

Now that we have established that prefix sum and shift commute, we can return to `p-add-left-pairs`. In particular, we will convert `p-add-left-pairs` into `p-add-right-pairs` of a shifted powerlists as follows:

```

(defthm p-add-left-pairs->p-add-right-pairs-p-shift
  (implies (and (domain-p val)
                (powerlist-p x)
                (p-regular-p x)
                (p-domain-list-p x))
    (equal (p-add-left-pairs val x)
          (p-add-right-pairs (p-shift val x)))))

```

It is now trivial to establish that

```

(p-prefix-sum-aux val (p-add-left-pairs val2 x))

```

is equal to

```

(p-prefix-sum-aux val (p-add-right-pairs (p-shift val2 x)))

```

and hence to

```

(p-unzip-r (p-prefix-sum val (p-shift val2 x)))

```

and

```

(p-unzip-r (p-shift (bin-op val val2)
                    (p-prefix-sum (bin-op val val2) x)))

```

To complete the proof, we need only the following technical lemma, which converts the right unzip of a shift to the left unzip of the powerlist:

```

(defthm p-unzip-r-p-shift
  (implies (and (powerlist-p x)
                (p-regular-p x)
                (not (powerlist-p val)))
            (equal (p-unzip-r (p-shift val x))
                   (p-unzip-l x))))

```

Trivially now, we can characterize the prefix sum of the `p-add-left-pairs` of a powerlist:

```

(defthm prefix-sum-p-add-left-pairs
  (implies (and (p-regular-p x)
                (p-domain-list-p x)
                (powerlist-p x)
                (domain-p val1)
                (domain-p val2))
            (equal (p-prefix-sum-aux val1
                                     (p-add-left-pairs val2 x))
                   (p-unzip-l (p-prefix-sum-aux (bin-op val1
                                                         val2)
                                                x))))))

```

This is an important moment, because `prefix-sum-p-add-left-pairs` and `prefix-sum-p-add-right-pairs` together give a characterization of the *unzips* of `p-prefix-sum`. That is, we have taken the original definition of `p-prefix-sum`, which was inherently sequential, and we have replaced it with an independent characterization of its unzips, which will make it much easier to prove the correctness of `p-simple-prefix-sum`.

However, `p-simple-prefix-sum` is defined in terms of `p-star` and `p-add`, and our new characterization uses `p-add-left-pairs` and `p-add-right-pairs`. The next step is to show how these are related. To start with, we give alternative definitions of `p-star` and `p-add` which use `tie` instead of `zip`; this will make it easier to reason about them and `p-add-left-pairs/p-add-right-pairs` together. Recall that `p-star` performs a shift operation and `p-add` a pairwise addition. We have already defined `p-shift`. Pairwise addition can be defined as follows:

```

(defun p-add-tie (x y)
  (if (powerlist-p x)
      (p-tie (p-add-tie (p-untie-l x) (p-untie-l y))
             (p-add-tie (p-untie-r x) (p-untie-r y)))
      (bin-op x y)))

```

ACL2 can easily prove the equivalence of these definitions with the original ones. For our purposes, we only need the following theorem:

```

(defthm add-star-add-tie-shift
  (implies (p-regular-p x)

```

```
(equal (p-add (p-star x) x)
      (p-add-tie (p-shift (left-zero) x) x))))
```

Using `p-shift` and `p-add-tie`, we can now prove how `p-add-left-pairs` and `p-add-right-pairs` are constructed in `p-simple-prefix-sum`:

```
(defthm zip-add-left-pairs-add-right-pairs
  (implies (and (powerlist-p x)
                (p-regular-p x)
                (p-domain-list-p x))
    (equal (p-zip (p-add-left-pairs (left-zero) x)
                (p-add-right-pairs x))
           (p-add (p-star x) x))))
```

At this point, the proof is almost complete. We know that the term

```
(p-add (p-star x) x)
```

can be rewritten as

```
(p-add-tie (p-shift (left-zero) x) x)
```

Moreover, we know how this term is unzipped into the two terms

```
(p-add-left-pairs (left-zero) x)
(p-add-right-pairs x)
```

And, finally, we know that the prefix sum of these terms can be zipped back together to get the prefix sum of `x`. Putting all this together, we can prove the correctness of `p-simple-prefix-sum`:

```
(defthm simple-prefix-sum-prefix-sum
  (implies (and (p-regular-p x)
                (p-domain-list-p x))
    (equal (p-simple-prefix-sum x)
           (p-prefix-sum x))))
```

5.2 Ladner-Fischer Prefix Sums

[Mis94] gives another algorithm for computing prefix sums, this one due to Ladner and Fischer:

```
(defun p-lf-prefix-sum (x)
  (if (powerlist-p x)
      (let ((y (p-lf-prefix-sum
                (p-add (p-unzip-l x) (p-unzip-r x)))))
        (p-zip (p-add (p-star y) (p-unzip-l x)) y))
      x))
```

The complexity of this algorithm is what justifies the previous usage of the name `p-simple-prefix-sum`!

As before, we proceed by considering the correctness of the left and right unzips separately. The right unzip is immediate:

```
(defthm unzip-r-lf-prefix-sum
  (implies (and (powerlist-p x)
                (p-regular-p x)
                (p-domain-list-p x)
                (equal (p-lf-prefix-sum (p-add (p-unzip-l x)
                                              (p-unzip-r x)))
                      (p-prefix-sum (p-add (p-unzip-l x)
                                              (p-unzip-r x)))))
            (equal (p-lf-prefix-sum (p-add (p-unzip-l x)
                                              (p-unzip-r x)))
                  (p-unzip-r (p-prefix-sum x)))))
```

It is only necessary to recognize that

```
(p-add (p-unzip-l x) (p-unzip-r x))
```

is the same as `(p-add-right-pairs x)`.

The left unzip is a little more subtle. It is equal to

```
(p-add (p-star (p-prefix-sum (p-add (p-unzip-l x)
                                    (p-unzip-r x))))
      (p-unzip-l x))
```

which we know can be reduced to

```
(p-add (p-star (p-unzip-r (p-prefix-sum x)))
      (p-unzip-l x))
```

We can reduce this further using the following trivial lemma:

```
(defthm unzip-l-star
  (equal (p-unzip-l (p-star x)) (p-star (p-unzip-r x))))
```

Thus, we have the term

```
(p-add (p-unzip-l (p-star (p-prefix-sum x)))
      (p-unzip-l x))
```

which we hope simplifies to

```
(p-unzip-l (p-prefix-sum x))
```

It is natural to generalize the above conjecture into the following theorem, which is provable by ACL2:

```

(defthm add-star-prefix-sum
  (implies (and (p-regular-p x)
                (p-domain-list-p x))
            (equal (p-add (p-star (p-prefix-sum x)) x)
                   (p-prefix-sum x))))

```

In the following section 5.3, we will see how this theorem, called the “Defining Equation” in [Mis94], plays a key role in the hand proof.

With the results above, it is now easy to establish that `p-lf-prefix-sum` equals `p-prefix-sum`, and thus we have demonstrated its correctness:

```

(defthm lf-prefix-sum-prefix-sum
  (implies (and (p-regular-p x)
                (p-domain-list x))
            (equal (p-lf-prefix-sum x)
                   (p-prefix-sum x))))

```

5.3 Comparing with the Hand-Proof Again

As was the case with Batchier sorting, the hand proof given in [Mis94] is much simpler than the machine-verified proof given above for the correctness of the prefix sum algorithms. Part of the reason is that in [Mis94] the proof begins in *media res*, as it were. Instead of providing a constructive definition, the prefix sum $ps(x)$ of a powerlist x is defined as the solution to the following “defining equation”:

$$z = z^* \oplus x$$

The perceptive reader will recognize this equation as `add-star-prefix-sum`.

The proof then proceeds by applying the defining equation to derive formulas for the left and right unzip of a prefix sum. Specifically, the derivation yields the Ladner-Fischer scheme. From there, it is shown how this scheme can be algebraically simplified to yield the simple prefix sum algorithm.

However, as we saw in section 5.2, establishing the correctness of the defining equation requires a fair amount of effort, and once it is established the remainder of the Ladner-Fischer proof is relatively simple.

The extra difficulty observed in the previous sections is a direct result of insisting the specifications, i.e., defining axioms, be constructive and readily accepted. In the interest of rigor, we believe this insistence is justified, so that our faith in a mechanically verified proof is not undermined by the necessity for a large unstated theory which has only been verified by human hands.

Moreover, requiring that correctness be established with respect to generally accepted specifications is a necessity if the proof is to be used in part of a larger project. For example, we stated that prefix sums appear in many applications, and so one expects to find a prefix sum computation in the middle of a complex algorithm. However, in establishing the correctness of the embedding algorithm, we must have that the prefix sum of $X = \langle x_1, x_2, \dots, x_n \rangle$ is in fact $ps(X) = \langle x_1, x_1 \oplus x_2, \dots, x_1 \oplus x_2 \oplus \dots \oplus x_n \rangle$. An equivalent correctness result, such as

the defining equation above, will not help us. In the next section, we consider one such example.

5.4 L’agniappe: A Carry-Lookahead Adder

Powerlists have been used to represent n -bit registers and to reason about arithmetic operations on them[KS95b]. In this section, we discuss how a carry-lookahead adder can be proved correct, using the correctness of a parallel prefix sum algorithm, i.e., the Ladner-Fischer scheme.

The “ripple-carry” or “schoolbook” algorithm for adding two n -bit registers is inherently sequential. Beginning with the least-significant bit, the algorithm progresses by adding corresponding bits. In so doing, it generates the carry bit for the next significant bit, and so on. This algorithm serves as our specification of addition.

The carry-lookahead adder uses the following observation. If it were only possible to compute all the carry bits a priori, the result of adding two n -bit registers can be computed in a single parallel step (using n full-adders). Moreover, given inputs $X = x_n x_{n-1} \dots x_1$ and $Y = y_n y_{n-1} \dots y_1$, the carry vector $C = c_n c_{n-1} \dots c_1$ can be computed as follows. Consider c_j . If x_j and y_j are both 0, then c_j must also be 0. Moreover, if x_j and y_j are both 1, then c_j is equal to 1. In all other cases, c_j is equal to c_{j-1} , where c_0 is the original carry bit.

The essential remaining point is that this computation is actually a prefix sum over an associative operator with left-identity. The prefix sum runs over the domain $\{0, 1, p\}$ with intuitive meaning of *no-carry*, *carry*, and *propagate carry* respectively. In constant time, the carry bit for c_i can be estimated as either 0, 1, or p , depending on whether x_i and y_i are both 0, both 1, or otherwise. The prefix sum over this vector of the operator \odot with $x \odot 0 = 0$, $x \odot 1 = 1$ and $x \odot p = x$ will generate the required carry bits. It is easily seen that the operator \odot is associative, with left-identity p .

This informal argument, as described for example in [CLR90], can be made precise in ACL2. In doing so, we found that the formal proof follows the informal one rather closely. That is, the hardest step in the proof is the establishment that the prefix sum computation — based on a linear algorithm similar to **p-prefix-sum-aux** — actually computes the correct carry vector. Both formal and informal proofs are made simpler by the fact the linear prefix sum algorithm is very similar to the ripple-carry adder algorithm. This would not be the case, of course, with a more complex version of prefix sum, e.g., one based on the Ladner-Fischer scheme, or with an abstract definition of prefix sum, such as the “defining equation” described in section 5.3. However, once the basic correctness results are established, it is trivial to extend this result to a carry-lookahead algorithm based on a fast prefix sum: the “hard” part of the proof is a simple instance of the generic theorems proved in section 5. We are encouraged that the formal proof for carry-lookahead was so easy to establish — it took no more than a single session with ACL2. We feel this illustrates the power of the powerlist formalism in general, the specific powerlist formalization presented in section 2,

and the usefulness of mechanically establishing correctness results with respect to “natural” specifications, as in sections 4 and 5.

6 Conclusions

In this paper, we set out to formalize powerlists in ACL2. Although powerlists are designed as regular data structures, we found it advantageous to generalize them in ACL2 to encompass non-regular powerlists. This is more in keeping with ACL2’s style, where even arithmetic and boolean operators can apply to any ACL2 object.

An unexpected contribution was the complete formalization of algorithms using powerlists. Previously, it had been shown how powerlists could be used to reason informally about software, but the reasoning was performed with a mixture of arguments inside as well as outside of powerlist algebra. In this paper, we showed the completion, using powerlists, of many of the example theorems in [Mis94]. Moreover, by completing the theorems — that is, by mechanically verifying their correctness with respect to a natural correctness specification — we were able to show how the resulting theories can be built on top of each other. Specifically, the theory of powerlists, described in section 2, was used in all the other theorems, and moreover the verification of the carry-lookahead adder in section 5.4 depends on the correctness proof of the prefix sum algorithm in section 5.

The more significant portion of this research was devoted to working with ACL2. In particular, we have shown how a complex theory can be developed in ACL2 by someone who is *outside* of the ACL2 development effort. We believe this shows a high degree of maturity in ACL2 and illustrates how it can be used to prove large theories. More importantly, we showed how many of ACL2’s “new” features — e.g., books, congruence rules, equivalence rewriting, encapsulations, forward chaining — can be successfully combined in a large project. Other, more obscure, features also played a role, though they were unmentioned in this paper. Readers interested in using ACL2 can find these instances in the available source code.

We also found some short-comings in ACL2 that suggest further improvements. For example, the arithmetic reasoning was a major stumbling block in proving the correctness of the fast rotate functions described in section 3. Encapsulation also presented us with some minor problems. For example, a great convenience of ACL2 is that its logic is computational. Thus, when “debugging” a new function, it is possible to execute it and see the results. However, this is not possible when using encapsulated functions. It would be useful if such functions could be used, perhaps by allowing the user to provide “sample” definitions for printing, or by simply printing them as called, e.g. (`bin-op 2 4`).

A final observation concerns the development of large ACL2 theories. While it would be nice if they were developed fully grown, most of these theories are developed through a process of iterative refinement. So, for example, while

deep in the middle of a proof concerning Batcher merge, we may discover an important lemma about powerlists that should have been proved in the powerlist book. However, theories that arise in this fashion can produce disaster, much the same way that a program that is hacked over a long period of time can become unmaintainable. Among the pitfalls are circular rewrite rules, which drive the theorem prover into infinite loops. More subtle problems involve rules which prematurely rewrite a term, preventing another rule from firing and thus “breaking” a previously proved theorem. It would be nice if a tool were available which could predict the ramifications of such a “small” change.

Moreover, even when a change is logically harmless, that is all the previous theorems are still provable by ACL2, it may have drastic consequences on the performance of the proof. For example, adding a rewrite rule can “hide” a former rule, and thus a proof that was previously a few lines long now involves a nested recursive proof, perhaps with a large number of cases. This suggests the opportunity for another type of tool. This tool would take a theory and return an “optimized” version, perhaps one including a few “Knuth-Bendix” style rewrite rules, or one in which the rewrite rules are reordered. Such a tool could use a mixture of automated and interactive processing; e.g., “why was this rule used here?” or “why didn’t you use this theorem here?” While writing this tool would be a significant task, we believe it would greatly enhance the use of ACL2. After all, most portions of an ACL2 theory are devoted to guiding ACL2 towards a certain proof. This tool, then, would be roughly analogous to a program debugger in interactive mode, and to an optimizer when used non-interactively.

Source Code

The source code for all the ACL2 examples listed here can be found in our web page at the URL <http://www.lim.com/~ruben/research/ac12/powerlists>. This code was processed with ACL2 version 1.8, and it has since been ported to ACL2 version 2.1, the current ACL2. As new versions of ACL2 become available, we intend to port the code to them.

Acknowledgments

We would like to extend our immense gratitude to Robert S. Boyer for suggesting the key data structures and definitions which made this work possible. We would also like to thank him for reviewing early drafts of this paper and offering many insightful comments. Our thanks also go to Jay Misra for first suggesting the mechanical verification of Batcher sort as a worthy challenge (we did not fully appreciate at the time how worthy the challenge would turn out to be,) and later suggesting the correctness of a carry-lookahead adder as a follow-up to prefix sums.

References

- [BKM96] B. Brock, M. Kaufmann, and J S. Moore. ACL2 theorems about commercial microprocessors. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD'96)*, pages 275–293. Springer-Verlag, November 1996.
- [BM79] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, Orlando, 1979.
- [BM88] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, San Diego, 1988.
- [CLR90] T. H. Corman, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*, chapter 32. McGraw-Hill, New York, 1990.
- [Kap97] D. Kapur. Constructors can be partial too. In R. Veroff, editor, *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*. MIT Press, 1997.
- [KM] M. Kaufmann and J S. Moore. *ACL2: A Computational Logic for Applicative Common Lisp, The User's Manual*.
- [KM94] M. Kaufmann and J S. Moore. Design goals for ACL2. Technical Report 101, Computational Logic, Inc., 1994.
- [KM97] M. Kaufmann and J S. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.
- [Kor97a] J. Kornerup. Odd-even sort in powerlists. *Information Processing Letters*, (61):15–24, 1997.
- [Kor97b] J. Kornerup. Parlists: A generalization of powerlists. In *Proceedings of Euro-Par'97*, 1997.
- [KP96] M. Kaufmann and P. Pecchiari. Interaction with the Boyer-Moore theorem prover: A tutorial study using the arithmetic-geometric mean theorem. *Journal of Automated Reasoning*, 16(1-2):181–222, 1996.
- [KS95a] D. Kapur and M. Subramaniam. Automated reasoning about parallel algorithms using powerlists. Technical Report TR-95-14, State University of New York at Albany, 1995.
- [KS95b] D. Kapur and M. Subramaniam. Mechanical verification of adder circuits using powerlists. Technical report, State University of New York at Albany, 1995.
- [Mis94] J. Misra. Powerlists: A structure for parallel recursion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1737–1767, November 1994.