J Autom Reasoning (2009) 43:139–172
DOI 10.1007/s10817-009-9140-y

A Formalization of Powerlist Algebra in ACL2

Ruben A. Gamboa

Received: 7 July 2006 / Accepted: 6 January 2009 / Published online: 1 July 2009

Abstract In Misra (ACM Trans Program Lang Syst 16(6):1737–1767, 1994), 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 (Kapur 1997; Kapur and Subramaniam 1995, Form Methods Syst Des 13(2):127–158, 1998). 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. We illustrate our approach with two case studies involving Batcher sorting and prefix sums.

Keywords Powerlists · Verification · ACL2

R. A. Gamboa (✉)
Department of Computer Science, University of Wyoming, Laramie, WY, USA
e-mail: ruben@uwyo.edu

1 Introduction

In [14], 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 [14] 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 [12].

In this paper, we focus not on the discovery or expression of correctness results, but on their mechanical verification. Specifically, we describe the development of a library of provably correct functions on powerlists. To this end, the compositionality of the correctness results is key: The theorems must be stated in ways that make them useful to subsequent (mechanical) proofs. This is a departure from [14], 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.

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, mostly 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 [1, 2, 9–11].

Other researchers have also attempted to use automated theorem provers to reason about powerlists, notably [6, 7] and [8]. While there are some similarities in our respective approaches, there are significant differences as well. In [6], Kapur is interested in extending a theorem prover to facilitate reasoning about regular data structures, such as powerlists. Kapur and Subramaniam [7] uses this structure to prove some of the theorems from [14], 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 [14], 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 [8], 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. Sections 3 and 4 present a formalization in our framework of case studies originally presented in [14], 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 5 summarizes the results.

2 Booking Powerlists

2.1 Regular Powerlists

A powerlist is defined 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 either both scalar or similar powerlists—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, while $x \bowtie y$ 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.

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 [14]):

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

Representing powerlists by directly mapping Misra's powerlist axioms is problematic in ACL2. For starters, ACL2 does not support partial functions, so the definitions of $\mid$ and $\bowtie$ must do *something* for non-similar powerlists, and in fact for non-powerlist operands. One approach is to represent powerlists in ACL2 as lists of length 2^n. The functions `tie` and `zip` take two powerlists and return the appropriate result if the arguments are lists of equal length, and a special error powerlist (e.g., `nil`) if the arguments are not appropriate. A similar approach is taken in [7], with the use of partial constructors. However, this leads to problems in ACL2. To reason about each use of `tie` or `zip`, we have to prove that the arguments are of equal length, and these proof obligations can become expensive.

The second problem is that since ACL2 does not support function definitions over terms, powerlist functions such as

$$rev(\langle x \rangle) = x$$
$$rev(x \mid y) = rev(y) \mid rev(x)$$

need to be turned into the form

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

where $left$ and $right$ are defined so that $left(X) \mid right(X) = X$. But defining and reasoning about these functions in ACL2 is not simple. Intuitively, the problem is that to compute $left(X)$, we must find the midpoint of X, which is an expensive operation for linear structures.

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, i.e., `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 $\langle 1.\langle 2.\langle 3.4\rangle\rangle\rangle$ 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 [13].

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 $\langle nest(\langle 1, 2\rangle), nest(\langle 3, 4\rangle)\rangle$.

2.2.3 The Tie Constructor

We begin the actual implementation with the definition of the data type powerlist. We prefer to define powerlists not directly as `cons`'s, but as record structures:

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

Similar to Common LISP's `defstruct`, ACL2's `defstructure` defines `powerlist`, `powerlist-p`, `powerlist-car`, and `powerlist-cdr`, which respectively construct, recognize, and destruct powerlists. It also proves the relevant "functor" theorems about them, corresponding to Misra's laws *L1a* and *L2b*. Recall that our representation does not differentiate between scalar values and single-element powerlists, so the function `powerlist-p` defined by the `defstructure` recognizes only multi-element lists.

We prefer the function names `p-untie-l` and `p-untie-r` instead of `powerlist-car` and `powerlist-cdr`. We will follow the same convention when introducing `p-zip` below. In the sequel, we will refer to `(p-untie-l x)` as the "left untie" of `x`. Similarly, we will use "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`, as will many other functions defined on powerlists. ACL2 will accept recursive definitions only if it can determine that each recursive invocation decreases some well-founded measure. In this case, ACL2 must determine that both `p-untie-l` and `p-untie-r` decrease some well-founded measure on powerlists, and in fact ACL2's default measure, `acl2-count`, satisfies this property. Since this recursion scheme will be very common with powerlists, we prove the following theorem and add it as a built-in rule, so that ACL2 applies it as early as possible:

```
(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)
```

Note: For integer arguments, such as the value returned by `acl2-count`, the ACL2 primitive `e0-ord-<` is identical to `<`. We use `e0-ord-<` instead of `<` so that the theorem matches exactly the formula generated by ACL2 while processing a function definition. The exact match is the limitation of built-in rules that allows them to be used so efficiently.

2.2.4 The Zip "Constructor"

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

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

Note that 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. Using 1-based indexing, `(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. As before, we will 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.

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 [14], we define two powerlists as similar if they have the same `tie`-tree structure:

```
(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, allowing it to be used in congruence (and not just equality) rewriting [3]. The ACL2 event `defequiv` is syntactic sugar that expands to the usual theorems describing equivalence relations:

```
(defequiv p-similar-p)
```

Our next task is to show how the property `p-similar-p` propagates across the powerlist constructors and destructors. For example, we can prove that the property is preserved by the destructors. I.e., if two lists are similar, so is their left unzip:

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

Similar theorems apply to the other destructors. 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-l x)
                                    (p-unzip-l y))
                       (P (p-unzip-l x)
                          (p-unzip-l 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, `unzip-l-similar` can be used to relieve the conditions in the inductive hypothesis and the proof can proceed.

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. These congruence rules can be specified 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` events offer 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, i.e., the ones introduced in [14]. This condition is more expensive to check 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, they must also be similar to each other:

```
(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))
```

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 theorems like the following:

```
(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 requires an extra hypothesis, ensuring that the powerlists to be combined are similar. This is the formal equivalent of the restriction that | and ⋈ only apply to powerlists of the same length:

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

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

```
(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*.

In our experience, similarity is much more useful than regularity, 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 regularity is to show that two powerlists are similar, for example when a single powerlist is split and a pairwise function is applied to the two halves. The regularity condition ensures that the two halves of the powerlist are similar to each other.

The formalization of powerlists presented thus far is sufficient to prove, often automatically, the "simple examples" of powerlists given by Misra in [14]. Interestingly, most of the examples turn out to be true for arbitrary powerlists, though when reasoning about multiple powerlists, we often have to assume the powerlists are similar to each other, as suggested above.

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. ACL2's encapsulation primitive allows us to prove the appropriate theorem schemas, which can later be instantiated with specific functions in mind, simplifying the overall proof effort.

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. Recall that `defcong` and `defequiv` are syntactic sugar for introducing theorems that demonstrate the usual properties of congruence substitution rules and equivalence relations. 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. These functions witness that the constraints are not contradictory, thus ensuring that the resulting theory is sound.

As a motivating example, consider applying `fn1` to all the elements of a powerlist and `fn2-accum` to combine all these results, e.g., to find the sum of the squares of all elements in a powerlist. Both functions can be defined recursively using either `p-tie` or `p-zip`, and the result should be the same. Formally, we can 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-==-b-tie-fn2-accum-fn1
  (equiv (b-tie-fn2-accum-fn1 x)
         (a-zip-fn2-accum-fn1 x)))
```

Other useful 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))))
```

All of these theorem schemas are useful in establishing the antecedent of induction hypotheses, and we will apply these theorem schemas by using specific instances, e.g., where `fn2-accum` is replaced with `max`.

3 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`. For this reason, we define `p-min-elem` using `p-tie`, though it could just as easily have been defined with `p-zip` instead. 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 straightforward consequences of the generic theorems proved in Section 2.5, so ACL2 does not need to perform added work in proving them. Moreover, since different sorting algorithms are likely to require similar theorems about functions such as `p-min-elem` and `p-sorted-p`, 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.

The result of sorting a powerlist should be a permutation of the original powerlist. To specify this, we use the following function, which returns the number of times a given argument appears in a powerlist:

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

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

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

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

Of course, we may allow specific sorting routines to impose additional restrictions, e.g., requiring numeric lists.

3.1 Merge Sorting

Merge sort is a 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-count (p-merge-sort x) m)
                  (p-count x m))))
(defthm merge-sort-sorts-input
  (implies (p-sortable-p x)
           (p-sorted-p (p-merge-sort x))))
```

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*-count-of-splits
   (implies (powerlist-p x)
            (equal (+ (p-count (p-split-1 x) m)
                      (p-count (p-split-2 x) m))
                   (p-count x m))))
 (defthm *obligation*-count-of-merge
   (implies (p-mergeable-p x y)
            (equal (p-count (p-merge x y) m)
                   (+ (p-count x m)
                      (p-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 ACL2 requires a witness for each constrained function; that is, an implementation of such a merging scheme. The easiest is the vacuous merger, i.e., when `p-sortable-p` is always `nil`.

3.2 Batcher Sorting

The Batcher 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-batcher-merge` is defined in terms of `p-zip`, we do the same for `p-min` and `p-max`.

Hidden in the definition of `p-batcher-merge` lies considerable complexity. Notice in particular how the the *left* unzip of `x` is merged with the *right* unzip of `y`, and vice versa. The result of this is that the resulting induction scheme is not very natural.

We begin with the proof of the following goal:

```
(equal (p-count (p-batcher-merge x y) m)
       (+ (p-count x m)
          (p-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 count-of-merge
  (implies (and (p-regular-p x)
                (p-similar-p x y)
                (p-number-list x)
                (p-number-list y))
           (equal (p-count (p-batcher-merge x y) m)
                  (+ (p-count x m)
                     (p-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 (pairwise) `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`. This lemma can be extended to `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 function `(lambda (x) (if (= x m) 1 0))` we can prove our original goal, using the equivalence of the `p-tie` and `p-zip` definitions of `p-count`.

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 th‘e 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 these 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. Intuitively, 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. We can replace the "prefix" concept with the following:

```
(defun p-count-<= (x m)
  (if (powerlist-p x)
      (+ (p-count-<= (p-untie-l x) m)
         (p-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-count-<= (p-batcher-merge (p-unzip-l x)
                                  (p-unzip-r y))
                 m)
M2 = (p-count-<= (p-batcher-merge (p-unzip-r x)
                                  (p-unzip-l y))
                 m)
```

so we begin with the following theorem:

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

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

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

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

```
(defthm count-<=-of-sorted-unzips-1
  (implies (and (powerlist-p x)
                (p-regular-p x)
                (p-sorted-p x))
           (<= (p-count-<= (p-unzip-r x) m)
               (p-count-<= (p-unzip-l x) m))))
(defthm count-<=-of-sorted-unzips-2
  (implies (and (powerlist-p x)
                (p-regular-p x)
                (p-sorted-p x))
           (<= (p-count-<= (p-unzip-l x) m)
               (1+ (p-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 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-count-<= (p-batcher-merge
                                   (p-unzip-l x)
                                   (p-unzip-r y))
                                 m))
                 (M2 (p-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)))))
```

To derive a contradiction, we next show that for non p-interleaved-p lists, there is some m so that the respective p-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:

```
(defun find-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)))
            ((find-cutoff (p-untie-l x) (p-untie-l y))
             (find-cutoff (p-untie-l x) (p-untie-l y)))
            ((find-cutoff (p-untie-r x) (p-untie-r y))
             (find-cutoff (p-untie-r x) (p-untie-r y))))
    nil))
```

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

```
(defthm interleaved-if-nil-cutoff
  (implies (and (p-similar-p x y)
                (p-number-list x)
                (p-number-list y)
                (null (find-cutoff x y))
                (null (find-cutoff y x)))
           (p-interleaved-p x y)))
```

In order to establish that find-cutoff finds a valid counterexample when x and y are not p-interleaved-p, notice that it 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 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)
                (find-cutoff x y))
           (< (1+ (p-count-<= y (find-cutoff x y)))
              (p-count-<= x (find-cutoff x y)))))
```

This theorem serves to find the counterexample needed by the two lemmas `count-<=-of-merge-unzips` and `interleaved-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
             (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`.

```
(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 anticlimatically, 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 3.1, 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-count (p-batcher-sort x) m)
                  (p-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))))
```

3.3 A Comparison with the Hand-Proof

It is instructive to compare the machine-verified proof of Section 3.2 with the hand-proof provided in [14] and verified in [7]. That proof starts by defining the function z as follows:

$$z(\langle x \rangle) = 1 \text{ if } x = 0, 0 \text{ otherwise}$$

$$z(p \bowtie q) = z(p) + z(q)$$

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:

$$sorted(\langle x \rangle)$$

$$sorted(p \bowtie q) = sorted(p) \wedge sorted(q) \wedge 0 \leq z(p) - z(q) \leq 1$$

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

$$min(x, y) = x, \text{ if } sorted(x), sorted(y), \text{ and } z(x) \geq z(y)$$

$$max(x, y) = y, \text{ if } sorted(x), sorted(y), \text{ and } z(x) \geq z(y)$$

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 3.2, 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 Section 3.2 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.

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. This becomes especially important if we plan 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 makes it difficult to end up with theorems that can be used compositionally in mechanical verification.

4 Prefix Sums of Powerlists

Prefix sums appear in many applications, e.g., arithmetic circuit design. For a powerlist $X = \langle x_1, x_2, \ldots, x_n \rangle$, its prefix sum is given by $ps(X) = \langle x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \cdots \oplus x_n \rangle$. 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))

 ;; Witnesses omitted...

 (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)))
)
```

Note: ACL2 requires witnesses for these functions, but we are leaving out their local definitions, as they are irrelevant in this discussion.

The function `domain-p` recognizes our intended domain, which is required to be scalar. The function `p-domain-list-p` recognizes powerlists of `domain-p` elements. Note that we require the second argument of `bin-op` to be `domain-p` in `left-zero-identity`, that there is no such requirement in `bin-op-assoc`, and that `domain-p` is always true of the result of `bin-op`. This matches the conventions used by ACL2 built-in binary functions.

There is a natural definition of prefix sums in terms of indices: 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:

```
(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`.

4.1 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)))
```

However, the definition of p-simple-prefix-sum is not obviously admissible, since the recursive call replaces x by (p-unzip-l (p-add (p-star x) x)) which is not obviously "smaller" than x. To resolve this, we can introduce 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 counts the number of elements in a powerlist, and we can easily 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)))
```

This makes the admissibility of p-simple-prefix-sum clear, so we can now concentrate on its correctness. 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

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

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

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

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

$$Y_1 = \langle x_1, x_2 \oplus x_3, \ldots, x_{n-2} \oplus x_{n-1} \rangle$$

$$Y_2 = \langle x_1 \oplus x_2, x_3 \oplus x_4, \ldots, x_{n-1} \oplus x_n \rangle$$

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`. 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)))))
```

An easy induction proves 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 $Y_1 = \langle x_1, x_2 \oplus x_3, \ldots, x_{n-2} \oplus x_{n-1} \rangle$:

```
(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 there is no need for the left half of the computation to pass any information over to the right half in `p-add-right-pairs`; 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 $X = \langle x_1, x_2, x_3, \ldots, x_n \rangle$ again. If we shift this powerlist, getting $X' = \langle 0, x_1, x_2, x_3, \ldots, x_{n-1} \rangle$, and then take the `p-add-right-pairs` of it, we get $\langle x_1, x_2 \oplus x_3, \ldots, x_{n-2} \oplus x_{n-1} \rangle$ 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.

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 order to account for 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 `(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-l x))))
        (p-last (p-untie-l x)))
```

This term can be simplified into

```
(p-last (p-prefix-sum-aux (bin-op c1 c2) (p-untie-l 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 powerlist:

```
(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 to convert 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))))
```

It is easy now to 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 we have a characterization of both *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, our characterization is in terms of `p-add-left-pairs` and `p-add-right-pairs`, while `p-simple-prefix-sum` is defined in terms of `p-star` and `p-add`, instead. 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 the results of `p-add-left-pairs` and `p-add-right-pairs` are constructed inside `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))))
```

4.2 Ladner-Fischer Prefix Sums

Misra [14] 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 4.3, we will see how this theorem, called the "Defining Equation" in [14], plays a key role in the hand proof.

These results imply 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))))
```

4.3 Comparing with the Hand-Proof Again

As was the case with Batcher sorting, the hand proof given in [14] 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 [14] 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":

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

This equation is exactly `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 simplified algebraically to yield the simple prefix sum algorithm.

However, as we saw in Section 4.2, 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 much simpler.

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 $X = \langle x_1, x_2, \ldots, x_n \rangle$ is in fact $ps(X) = \langle x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \cdots \oplus x_n \rangle$. An equivalent correctness result, such as the defining equation above, will not help us. In the next section, we consider an example where having a formal specification for prefix sum is crucial. That is, we can prove the correctness of a carry-lookahead adder by building on the theorem regarding prefix sums. This compositionality is critical in large mechanical verifications.

4.4 L'agniappe: A Carry-Lookahead Adder

Powerlists have been used to represent n-bit registers and to reason about arithmetic operations on them. 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. This is similar to an earlier proof of arithmetic algorithms using powerlists found in [8].

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 could be computed in a single parallel step (using n full-adders). Moreover, given inputs $X = x_n x_{n-1} \ldots x_1$ and $Y = y_n y_{n-1} \ldots y_1$, the carry vector $C = c_n c_{n-1} \ldots c_1$ 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 [4], 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 4.3. 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 4. 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 3 and 4.

5 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 [14]. Moreover, by completing the theorems—that is, by mechanically verifying their correctness with respect to a natural correctness specification—we achieved compositional theories: complex proofs can be simplified by referring to earlier theorems. 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 4.4 depends on the correctness proof of the prefix sum algorithm in Section 4.

The formalization of powerlist algebra demonstrates the usefulness of reasoning about constrained functions with `encapsulate`. However, while `encapsulate` supports reasoning about functions, it is a strictly first-order inference rule. Moreover, its application in ACL2 requires extensive use of hints to direct the theorem prover in the mapping of actual functions to constrained functions. Automation in this process would be enormously useful, and it is something that is under partial

File	Definitions	Theorems	Hints
Powerlist definitions	42	105	38
Simple theorems	10	22	6
Sortings pecification	8	10	6
Merge sort	8	18	3
Batcher merge	10	71	38
Prefix sum	20	74	31
Carry lookahead adder	17	20	9

Fig. 1 Effort of work

development. Congruence rewriting turned out to be less useful in this project, although its usefulness has been demonstrated elsewhere [3, 5].

Figure 1 gives an idea of the proof effort required in the formalization of powerlists. The hints tell an interesting story. Out of 320 theorems, 131 required user intervention in the form of hints to the theorem prover. These 131 hints can be further classified into four categories. 51 of them instruct ACL2 to use a specific lemma in a key spot, usually with bindings for the universally quantified variables in the lemma. 57 of the hints are used to prove a theorem by invoking one of the meta-theorems created with `encapsulate`. This illustrates both the utility of `encapsulate` and the effort required to use it. 19 hints are required to suggest an appropriate induction scheme, and the final 4 hints instruct ACL2 to ignore certain lemmas that it knows about, as they steer ACL2 away from the actual proof. As can be seen from this breakdown, the majority of the human intervention that is required involves choosing specific lemma instances or instances involving constrained functions.

The formalization of powerlists described in this paper is included in the ACL2 distribution. Also included are several examples based on this formalization, including all the examples and case studies described here.

Acknowledgements 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, and later suggesting the correctness of a carry-lookahead adder as a follow-up to prefix sums.

References

1. Boyer, R.S., Moore, J.S.: A Computational Logic. Academic, Orlando (1979)
2. Boyer, R.S., Moore, J.S.: A Computational Logic Handbook. Academic, San Diego (1988)
3. Brock, B., Kaufmann, M., Moore, J.S.: Rewriting with equivalence relations in ACL2. J. Autom. Reason. **40**(4), 293–306 (2008)
4. Corman, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms, chap. 32. McGraw-Hill, New York (1990)
5. Greve, D.: Parameterized congruences in ACL2. In: Proceedings of the Sixth International Workshop of the ACL2 Theorem Prover and its Applications (ACL2-2006) (2006)
6. Kapur, D.: Constructors can be partial too. In: Veroff, R. (ed.) Automated Reasoning and Its Applications: Essays in Honor of Larry Wos. MIT, Cambridge (1997)
7. Kapur, D., Subramaniam, M.: Automated reasoning about parallel algorithms using powerlists. Technical Report TR-95-14, State University of New York at Albany (1995)
8. Kapur, D., Subramaniam, M.: Mechanical verification of adder circuits using rewrite rulelaboratory. Form. Methods Syst. Des. **13**(2), 127–158 (1998)
9. Kaufmann, M., Moore, J.S.: The ACL2 home page. http://www.cs.utexas.edu/users/moore/acl2/acl2-doc.html
10. Kaufmann, M., Moore, J.S.: Design goals for ACL2. Technical Report 101, Computational Logic, Inc. http://www.cs.utexas.edu/-users/moore/publications/acl2-papers.html#Overviews (1994)
11. Kaufmann, M., Moore, J.S.: An industrial strength theorem prover for a logic based on common lisp. IEEE Trans. Softw. Eng. **23**(4), 203–213 (1997)
12. Kornerup, J.: Odd-even sort in powerlists. Inf. Process. Lett. **61**(1), 15–24 (1997)
13. Kornerup, J.: Parlists: a generalization of powerlists. In: Proceedings of Euro-Par'97 (1997)
14. Misra, J.: Powerlists: a structure for parallel recursion. ACM Trans. Program. Lang. Syst. **16**(6), 1737–1767 (1994)