Coverset Induction with Partiality and Subsorts: A Powerlist Case Study

Joe Hendrix[1] Deepak Kapur[2] José Meseguer[3]

[1]Galois, Inc [2]University of New Mexico

[3]University of Illinois at Urbana-Champaign

Presented at ITP 2010

Overview

Motivation

Prove properties about recursive data parallel algorithms in a natural way using an interactive theorem prover.

Algorithms include:

- Parallel prefix sum
- Variable-size addition circuits
- Fast Fourier Transform and inverse
- Batcher sort

Overview

Motivation

Prove properties about recursive data parallel algorithms in a natural way using an interactive theorem prover.

Algorithms include:

- Parallel prefix sum
- Variable-size addition circuits
- Fast Fourier Transform and inverse
- Batcher sort

Our Approach

Main design decisions:

- Formalize algorithms using Misra's Powerlists.
- Use Maude Inductive Theorem Prover (ITP) for verification.
  - Added generalized form of coverset induction to Maude ITP.
  - Coverset induction generates induction schemes from recursive terminating equations.
  - Reflective architecture that uses Maude's rewriter for simplification in theorems.

Powerlists

- Powerlists (Misra 94) are an algebraic type designed specifically for formalizing parallel data-recursive algorithms.
- Existing work by Misra, Kornerup, and Adams uses powerlists to formalize many different data parallel algorithms to be elegantly defined.
- Results have been shown with ACL2 and RRL, but existing automated proofs lack the elegance of the hand proofs.
- Our work is an effort to recapture the elegance within an inductive theorem prover.

Powerlists

- Powerlists (Misra 94) are an algebraic type designed specifically for formalizing parallel data-recursive algorithms.
- Existing work by Misra, Kornerup, and Adams uses powerlists to formalize many different data parallel algorithms to be elegantly defined.
- Results have been shown with ACL2 and RRL, but existing automated proofs lack the elegance of the hand proofs.
- Our work is an effort to recapture the elegance within an inductive theorem prover.

Powerlist definition

- Powerlists are lists with length 2^n for any $n \in \mathbb{N}$.
- Constructed in several ways:
  - Single elements $[e]$ of an arbitrary type,
  - Concatenating two similar powerlists p tie q, or
  - Interleaving two similar powerlists p zip q.
  - Nesting a powerlist as an element inside another $\langle p \rangle$.

  Powerlists p and q are similar $(\mathrm{sim?}(p, q))$ if they have the same number of nestings and the same number of elements at each nested level.
- In this presentation, we will only consider powerlists without nesting.

Powerlist definition

- Powerlists are lists with length 2^n for any $n \in \mathbb{N}$.
- Constructed in several ways:
  - Single elements $[e]$ of an arbitrary type,
  - Concatenating two similar powerlists p tie q, or
  - Interleaving two similar powerlists p zip q.
  - Nesting a powerlist as an element inside another $\langle p \rangle$.

  Powerlists p and q are similar $(\mathrm{sim?}(p, q))$ if they have the same number of nestings and the same number of elements at each nested level.
- In this presentation, we will only consider powerlists without nesting.

Examples

$$[0\,1\,2\,3]\,\texttt{tie}\,[4\,5\,6\,7] = [0\,1\,2\,3\,4\,5\,6\,7]$$

$$[0\,1\,2\,3]\,\texttt{zip}\,[4\,5\,6\,7] = [0\,4\,1\,5\,2\,6\,3\,7]$$

Membership Equational Logic (MEL)

- Membership Equational Logic is a Horn logic with two levels of typing:

  Kinds Defined by the signature

  Sorts More refined mechanism defined by memberships in the theory.

  Maude statically type-checks at the kind level, while sorts are interpreted as predicates at runtime.
- The kind of a sort s is denoted by $[s]$.
- Sort and function definitions may be mutually recursive.

Membership Equational Logic (MEL)

- Membership Equational Logic is a Horn logic with two levels of typing:

  Kinds Defined by the signature

  Sorts More refined mechanism defined by memberships in the theory.

  Maude statically type-checks at the kind level, while sorts are interpreted as predicates at runtime.
- The kind of a sort s is denoted by $[s]$.
- Sort and function definitions may be mutually recursive.

Powerlists over Natural Numbers in Maude

```
fmod POWERLIST is protecting NAT .
  sort Pow .
  op [_] : Nat -> Pow [ctor].
  op _tie_ : [Pow] [Pow] -> [Pow] .

  vars M N : Nat .     vars P Q R S : Pow .

  cmb P tie Q : Pow if sim?(P, Q) = true .

  op sim? : Pow Pow -> Bool [comm].
  eq sim?([M], [N]) = true .
  eq sim?(P tie Q, [N]) = false .
  eq sim?(P tie Q, R tie S) = sim?(P, R) .

  op _zip_ : [Pow] [Pow] -> [Pow] .
  eq [M] zip [N] = [M] tie [N] .
  eq (P tie Q) zip (R tie S) = (P zip R) tie (Q zip S) .
endfm
```

Powerlists over Natural Numbers in Maude

```
fmod POWERLIST is protecting NAT .
  sort Pow .
  op [_] : Nat -> Pow [ctor].
  op _tie_ : [Pow] [Pow] -> [Pow] .

  vars M N : Nat .    vars P Q R S : Pow .

  cmb P tie Q : Pow if sim?(P, Q) = true .

  op sim? : Pow Pow -> Bool [comm].
  eq sim?([M], [N]) = true .
  eq sim?(P tie Q, [N]) = false .
  eq sim?(P tie Q, R tie S) = sim?(P, R) .

  op _zip_ : [Pow] [Pow] -> [Pow] .
  eq [M] zip [N] = [M] tie [N] .
  eq (P tie Q) zip (R tie S) = (P zip R) tie (Q zip S) .
endfm
```

Powerlists over Natural Numbers in Maude

```
fmod POWERLIST is protecting NAT .
  sort Pow .
  op [_] : Nat -> Pow [ctor].
  op _tie_ : [Pow] [Pow] -> [Pow] .

  vars M N : Nat .     vars P Q R S : Pow .

  cmb P tie Q : Pow if sim?(P, Q) = true .

  op sim? : Pow Pow -> Bool [comm].
  eq sim?([M], [N]) = true .
  eq sim?(P tie Q, [N]) = false .
  eq sim?(P tie Q, R tie S) = sim?(P, R) .

  op _zip_ : [Pow] [Pow] -> [Pow] .
  eq [M] zip [N] = [M] tie [N] .
  eq (P tie Q) zip (R tie S) = (P zip R) tie (Q zip S) .
endfm
```

Powerlists over Natural Numbers in Maude

```
fmod POWERLIST is protecting NAT .
  sort Pow .
  op [_] : Nat -> Pow [ctor].
  op _tie_ : [Pow] [Pow] -> [Pow] .

  vars M N : Nat .     vars P Q R S : Pow .

  cmb P tie Q : Pow if sim?(P, Q) = true .

  op sim? : Pow Pow -> Bool [comm].
  eq sim?([M], [N]) = true .
  eq sim?(P tie Q, [N]) = false .
  eq sim?(P tie Q, R tie S) = sim?(P, R) .

  op _zip_ : [Pow] [Pow] -> [Pow] .
  eq [M] zip [N] = [M] tie [N] .
  eq (P tie Q) zip (R tie S) = (P zip R) tie (Q zip S) .
endfm
```

Syntactic Sugar

This definition uses syntactic sugar:

```
op [_] : Nat -> Pow [ctor] .

op sim? : Pow Pow -> Bool [comm].

eq sim?(P tie Q, R tie S) = sim?(P, R) .
```

Semantically, these resolve into:

```
op [_] : [Nat] -> [Pow] .
cmb [N] : Pow if N : Nat.

op sim? : [Pow] [Pow] -> [Bool] [comm].
cmb sim?(P, Q) : Bool if P : Pow /\ Q : Pow .

ceq sim?(P tie Q, R tie S) = sim?(P, R)
 if P : Pow /\ Q : Pow /\ R : Pow /\ S : Pow .
```

Syntactic Sugar

This definition uses syntactic sugar:

```
op [_] : Nat -> Pow [ctor] .

op sim? : Pow Pow -> Bool [comm].

eq sim?(P tie Q, R tie S) = sim?(P, R) .
```

Semantically, these resolve into:

```
op [_] : [Nat] -> [Pow] .
cmb [N] : Pow if N : Nat.

op sim? : [Pow] [Pow] -> [Bool] [comm].
cmb sim?(P, Q) : Bool if P : Pow /\ Q : Pow .

ceq sim?(P tie Q, R tie S) = sim?(P, R)
 if P : Pow /\ Q : Pow /\ R : Pow /\ S : Pow .
```

Prefix Sums

Given an associative binary operation $+$ over scalars,
the prefix sum $\mathrm{ps}(p)$ of a powerlist p over $+$ is:

$$\mathrm{ps}([x_0, x_1, \ldots, x_{n-1}]) = [x_0, x_0 + x_1, \ldots, x_0 + x_1 + \cdots + x_{n-1}]$$

Sequential Prefix Sums

Defining prefix sums in Maude.

```
--- Return prefix sum of list.
op ps : Pow -> Pow .
eq ps([ N ]) = [ N ] .
eq ps(P tie Q) = ps(P) tie (last(ps(P)) + ps(Q)) .

--- Return last element in list.
op last : Pow -> Nat .
eq last([N]) = N .  eq last(P tie Q) = last(Q) .

--- Add value to each element in Powerlist.
op _+_ : Nat Pow -> Pow .
eq M + [ N ] = [ M + N ] .
eq M + (P tie Q) = (M + P) tie (M + Q) .
```

Sequential Prefix Sums

Defining prefix sums in Maude.

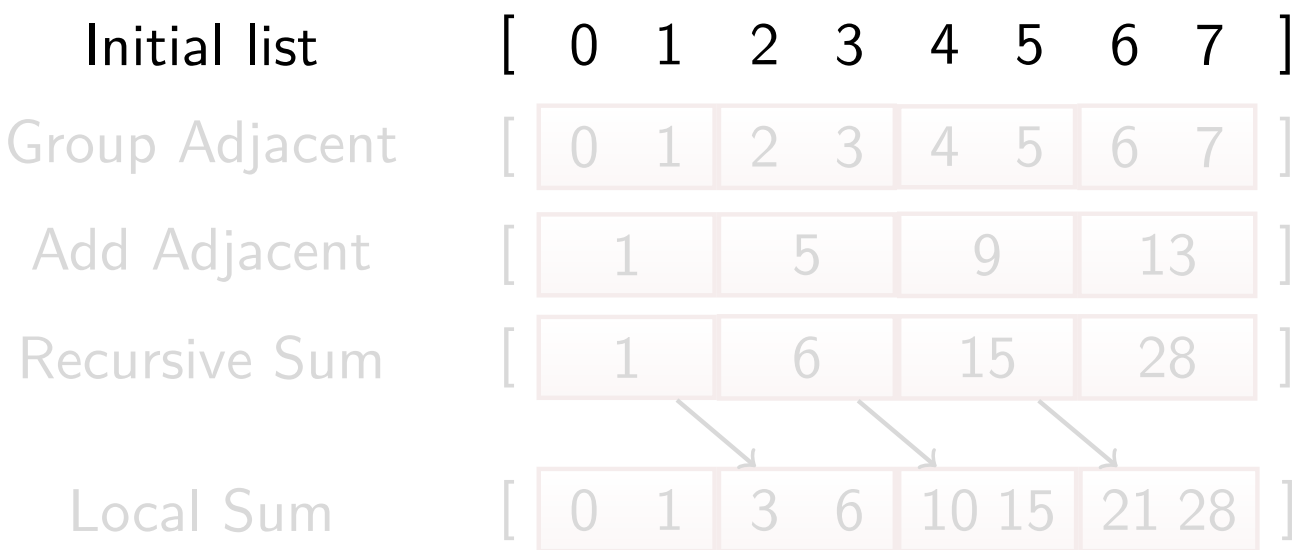
```
--- Return prefix sum of list.
op ps : Pow -> Pow .
eq ps([ N ]) = [ N ] .
eq ps(P tie Q) = ps(P) tie (last(ps(P)) + ps(Q)) .

--- Return last element in list.
op last : Pow -> Nat .
eq last([N]) = N .  eq last(P tie Q) = last(Q) .

--- Add value to each element in Powerlist.
op _+_ : Nat Pow -> Pow .
eq M + [ N ] = [ M + N ] .
eq M + (P tie Q) = (M + P) tie (M + Q) .
```

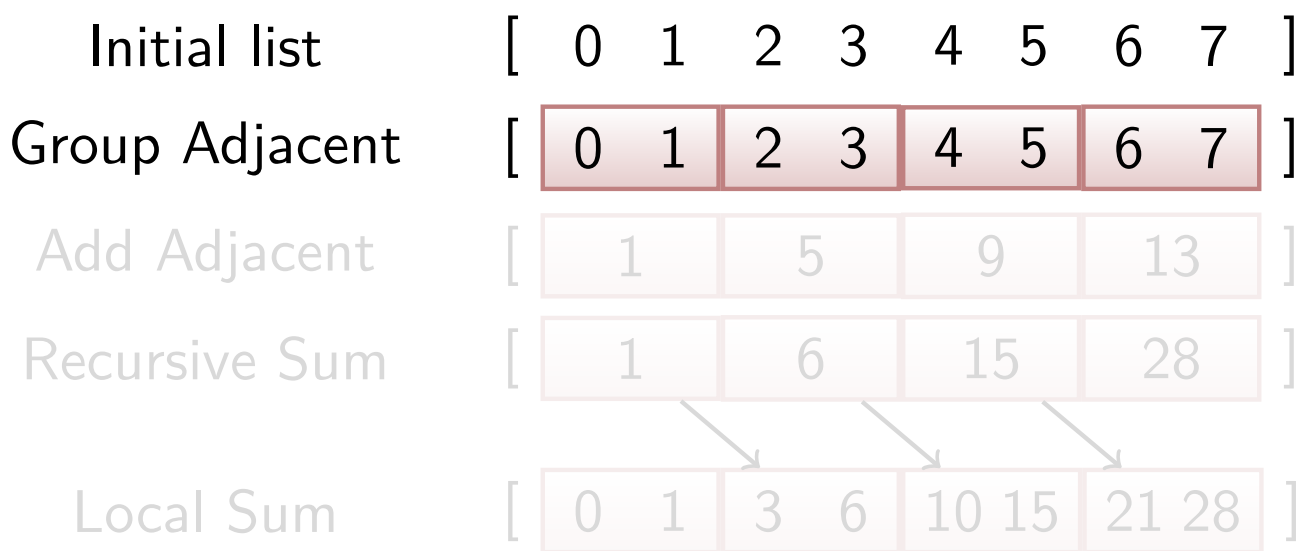
Ladner and Fischer Prefix Sums

Ladner and Fischer's approach operates on adjacent elements.



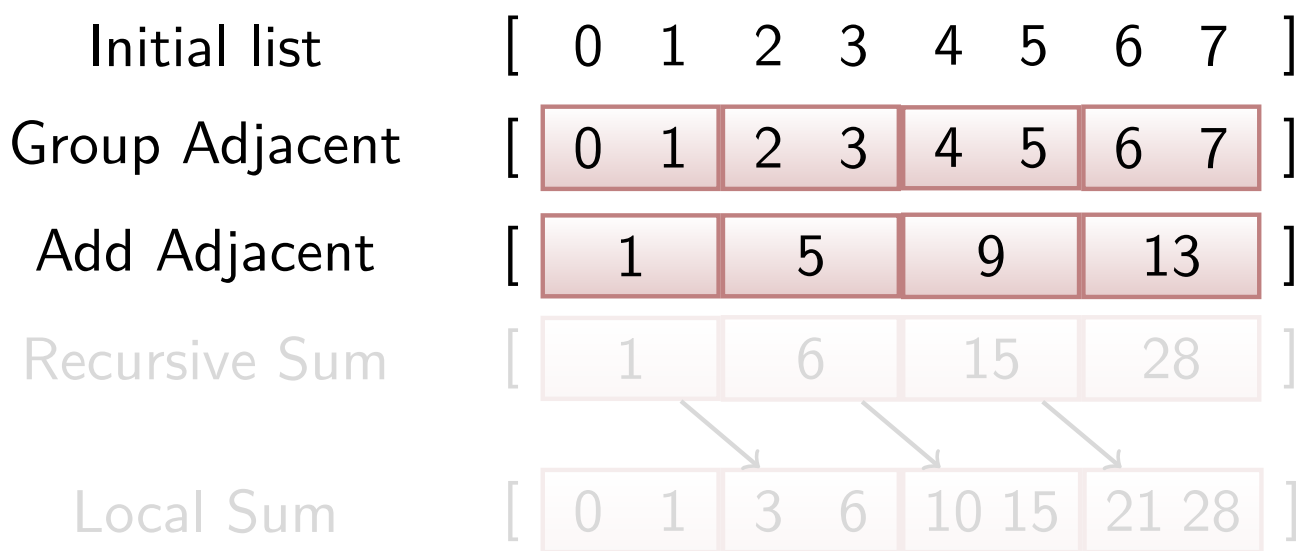
Ladner and Fischer Prefix Sums

Ladner and Fischer's approach operates on adjacent elements.



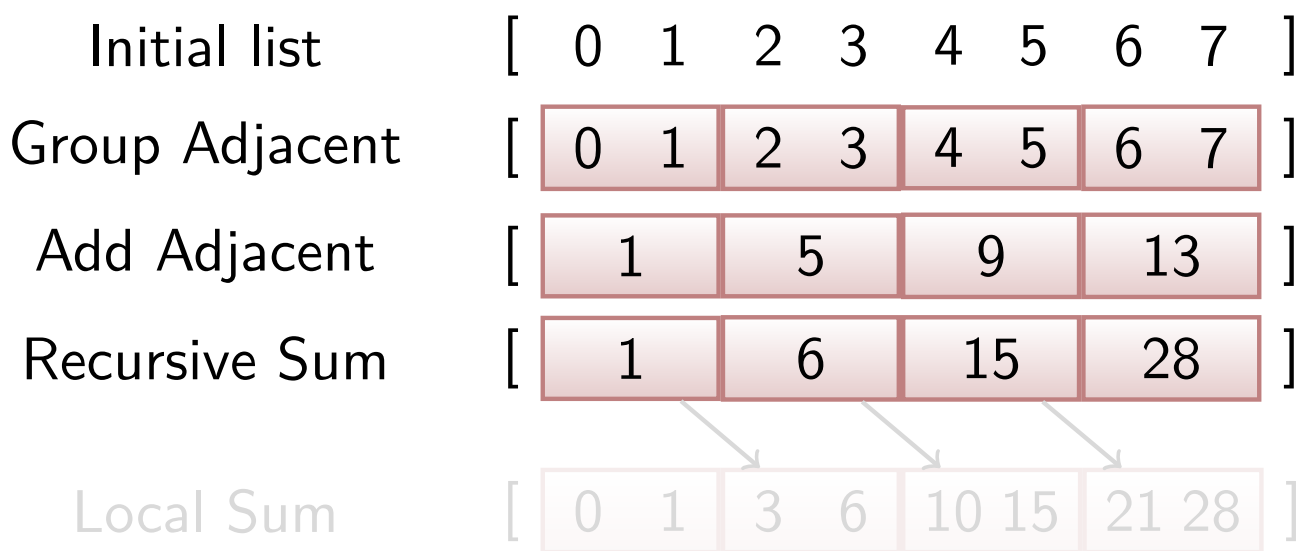
Ladner and Fischer Prefix Sums

Ladner and Fischer's approach operates on adjacent elements.



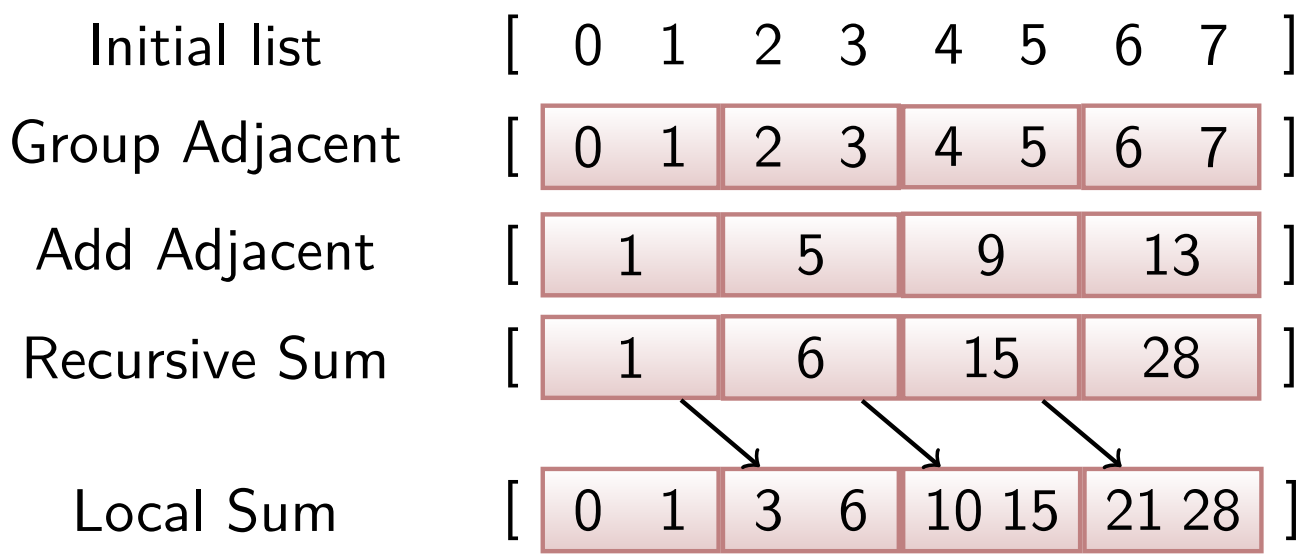
Ladner and Fischer Prefix Sums

Ladner and Fischer's approach operates on adjacent elements.



Ladner and Fischer Prefix Sums

Ladner and Fischer's approach operates on adjacent elements.



Ladner and Fischer Prefix Sums

```
op lf : Pow -> Pow .
eq lf([ N ]) = [ N ] .
eq lf(P zip Q) = (rsh(0, lf(P ++ Q)) ++ P) zip lf(P ++ Q) .

--- Right shift.
op rsh : Nat Pow -> Pow .
eq rsh(M, [ N ]) = [ M ] .
eq rsh(M, P tie Q) = rsh(M, P) tie rsh(last(P), Q) .

--- Sum each element.
op _++_ : [Pow] [Pow] -> [Pow] .
eq [ M ] ++ [ N ] = [ M + N ] .
eq (P1 tie P2) ++ (Q1 tie Q2) = (P1 ++ Q1) tie (P2 ++ Q2) .
```

Coverset Induction

- As an example, we want to show this theorem.

```
(goal ps-lf : POWERLIST-PREFIX |-
  A{P:Pow}   ((ps(P)) = (lf(P))) .)
```

- Will need to use recursive calls in the definition to generate induction schemes.

```
op lf : Pow -> Pow .
eq lf([ N ]) = [ N ] .
eq lf(P zip Q) = (rsh(0, lf(P ++ Q)) ++ P) zip lf(P ++ Q) .
```

Generalization to MEL

- Coverset induction was originally developed for unsorted and total functions.
- Prefix sum is only defined on well-structured powerlists.
- Our idea is to iteratively refine a *pattern* against equations. unifying with that pattern.
- Instantiate variables using memberships associated with variable sort constraint.

Example Induction

- Suppose we want to show the following lemma:

```
(lem last-zip :
  A{P:Pow ; Q:Pow}
    ((sim?(P,Q)) = (true)
      => (last(P zip Q)) = (last(Q))) .)
```

- Can show using coverset induction on `P zip Q`

```
op _zip_ : [Pow] [Pow] -> [Pow] .
eq (P tie Q) zip (R tie S) = (P zip R) tie (Q zip S) .
eq [M] zip [N] = [M] tie [N] .
```

Example Induction

- We start with pattern `P zip Q` which unifies with the left-hand sides of both equations.

```
op _zip_ : [Pow] [Pow] -> [Pow] .
eq (P tie Q) zip (R tie S) = (P zip R) tie (Q zip S) .
eq [M] zip [N] = [M] tie [N] .
```

- We can then instantiate these variables with the constructor memberships to yield 4 patterns.

```
cmb P tie Q : Pow if sim?(P, Q) = true .
cmb [N] : Pow if N : Nat.
```

- For each pattern, we generate a separate case. Induction hypotheses are added when the specialized pattern matches a recursive call.

Induction Cases 1/2

```
A{M:Nat ; N:Nat}
  ((sim?([M], [N])) = (true)
    => (last([M] zip [N])) = (last([N])))

A{P1:Pow ; P2:Pow ; N:Nat}
  ((sim?(P1 tie P2, [N])) = (true)
    => (last((P1 tie P2) zip [N])) = (last([N])))

A{M:Nat ; Q1:Pow ; Q2:Pow}
  ((sim?([M], Q1 tie Q2)) = (true)
    => (last([M] zip (Q1 tie Q2))) = (last(Q1 tie Q2)))
```

Induction Cases 2/2

```
A{P1:Pow ; P2:Pow ; Q1:Pow ; Q2:Pow}
  ((sim?(P1 tie P2, Q1 tie Q2)) = (true)
   & (sim?(P1, P2)) = (true)
   & (sim?(Q1, Q2)) = (true)
   & ((sim?(P1, Q1)) = (true)
         => (last(P1 zip Q1)) = (last(Q1)))
   & ((sim?(P2, Q2)) = (true)
         => (last(P2 zip Q2)) = (last(Q2)))
   => (last((P1 tie P2) zip (Q1 tie Q2))) = (last(Q1 tie Q2)))
```

Extensions

- With coverset induction, one can perform induction by matching against any set of complete memberships.

```
cmb P zip Q : Pow if sim?(P, Q) = true .
cmb [N] : Pow if N : Nat.
```

- Can further instantiate variables to match additional left-hand sides and enable more simplification via rewriting.

```
sim?(P zip Q, R)
```

Prefix sum example

- For the theorem

$$(\forall\, x\, :\, \mathrm{Pow})\ \mathrm{lf}(x) = \mathrm{ps}(x),$$

coverset induction using $\mathrm{lf}(x)$ using zip generates two cases:

```
eq lf([ N ]) = [ N ] .
eq lf(P zip Q) = (rsh(0, lf(P ++ Q)) ++ P) zip lf(P ++ Q) .
```

- Base case:

$$(\forall x : \mathrm{Nat})\ \mathrm{lf}([x]) = \mathrm{ps}([x]).$$

- Induction case:

$$(\forall p, q : \mathrm{Pow})\ \mathrm{sim?}(p, q) \wedge$$
$$\mathrm{lf}(p +\!\!+ q) = \mathrm{ps}(p +\!\!+ q) \implies \mathrm{lf}(p\ \mathrm{zip}\ q) = \mathrm{ps}(p\ \mathrm{zip}\ q).$$

Maude ITP Proof for Fischer-Ladner closure

- Required 8 basic lemmas about `sim?`, `tie`, and `zip`.
- One "typing" lemma about each defined operation:
  - `last`, element-wise addition, `rsh`, `ps`, and `lf`.
- Requires 6 lemmas to simplify terms with different operations.

Typing Lemma for Element-wise Addition

```
(lem list-sum-pow :
  A{P:Pow ; Q:Pow ; R:Pow}
   ((sim?(P,Q)) = (true)
     => (P ++ Q) : Pow
      & (sim?(P ++ Q, R)) = (sim?(P, R))) .)
(cov-split* on (P ++ Q) split (sim?(P, R)) .)
```

Proof Script 1/2

```
(lem last-zip :
  A{P:Pow ; Q:Pow}
    ((sim?(P,Q)) = (true) => (last(P zip Q)) = (last(Q))) .)
(cov* on P zip Q .)

(lem last-elt-sum :
  A{N:Nat ; P:Pow} ((last(N + P)) = (N + last(P))) .)
(cov* on last(P) .)

(lem elt-sum-zip :
  A{M:Nat ; P:Pow ; Q:Pow}
   (  (sim?(P, Q)) = (true)
    => (M + (P zip Q)) = ((M + P) zip (M + Q))) .)
(cov* on P zip Q .)
```

Proof Script 2/2

```
(lem rsh-self-elt-plus-general :
  A{M:Nat ; N:Nat ; P:Pow}
   ((rsh(N + M, N + P)) = (N + rsh(M, P))) .)
(cov* on N + P .)

(lem rsh-self-elt-plus :
  A{N:Nat ; P:Pow}
   ((rsh(N, N + P)) = (N + rsh(0, P))) .)
(cov* on N + P .)

(lem ps-zip :
  A{P:Pow ; Q:Pow}
   ((sim?(P,Q)) = (true)
    => (ps(P zip Q))
         = ((rsh(0, ps(P ++ Q)) ++ P) zip ps(P ++ Q))) .)
(cov* on P ++ Q .)
```

Comparison with Related Work

- Misra's definition of prefix sum requires 1 journal page.
- Misra derives Ladner-Fischer from definition in 2.5 journal pages.
- Maude ITP Proof differs from Misra's, but uses same functions.
- Complete prefix sum spec is ~ 60 lines, and proof scripts are ~ 120 lines.
- ACL2 proof involves ~ 400 lines of proof, and several auxiliary definitions.

Summary

- We have made significant improvements to the Maude ITP including
  - Coverset induction
  - Equivalence propagation
  - Additional commands for developing a proof strategy and debugging proofs.
- We have validated these extensions with an extensive case study on formalizing parallel algorithms using powerlists.
  - Correctness of Fast Fourier transform, Batcher sort, prefix sums, and adder circuits.
  - Theorems require these new features in ITP.
  - Total case study: ~ 560 lines of specification and ~ 1320 lines of proof.

Thanks for listening