Proving Equivalence Between Imperative and MapReduce Implementations Using Program Transformations

VPT 2018
Thessaloniki

20 April 2018

Bernhard Beckert, Timo Bingman, Moritz Kiefer,
Peter Sanders, **Mattias Ulbrich**, Alexander Weigl

Motivation

- **Simple** algorithm implementations are **understandable**, . . .

Motivation

- **Simple** algorithm implementations are **understandable**, . . .
- . . . but not efficient.

Motivation

- **Simple** algorithm implementations are **understandable**, ...
- ... but not efficient.
- **Efficient** implementations are often **complex**, **error-prone**

Motivation

- **Simple** algorithm implementations are **understandable**, . . .
- . . . but not efficient.
- **Efficient** implementations are often **complex**, **error-prone**

$\Longrightarrow$ Prove equivalence between reference and efficient implementation

Motivation

- **Simple** algorithm implementations are **understandable**, . . .
- . . . but not efficient.
- **Efficient** implementations are often **complex**, **error-prone**

$\Longrightarrow$ Prove equivalence between reference and efficient implementation

Verify **MapReduce** against **imperative** reference implementation.

Motivation

- **Simple** algorithm implementations are **understandable**, ...
- ... but not efficient.
- **Efficient** implementations are often **complex**, **error-prone**

$\Longrightarrow$ Prove equivalence between reference and efficient implementation

Verify **MapReduce** against **imperative** reference implementation.

Challenge for relational reasoning

Programs not (necessarily) structurally close

Idea

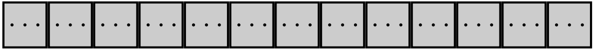
Combine Rewriting and Relational Reasoning

... and be open to automation

MapReduce

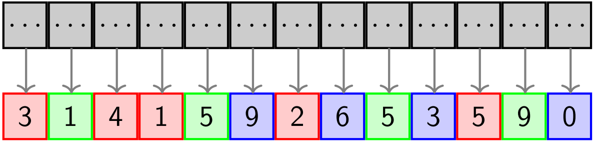
- distributed programming framework / paradigm
- first used large scale by google
- using concepts from functional programming to allow implicit parallisation.
- algorithms are quite different to their IMP counterparts

Recap: MapReduce

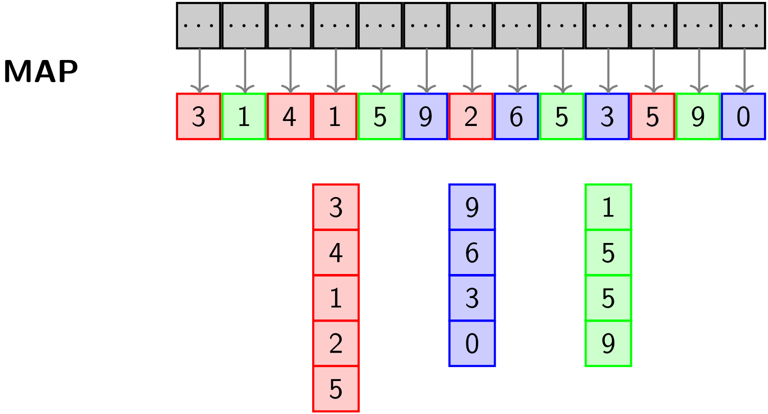


Recap: MapReduce

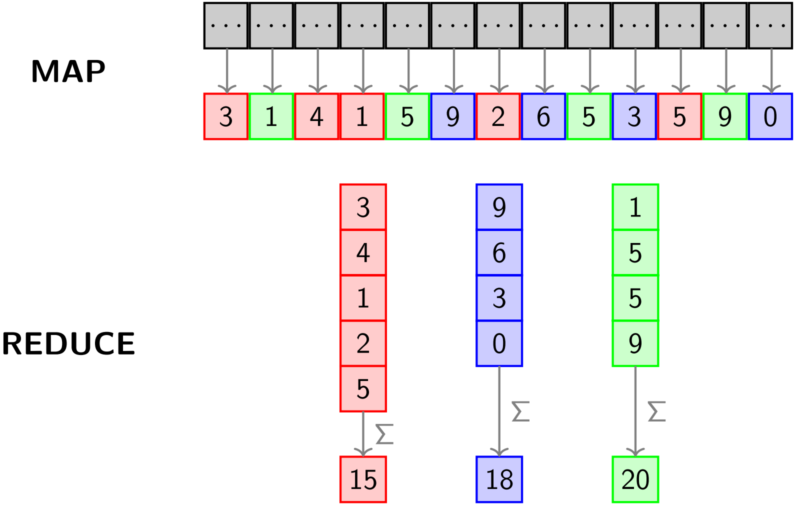
MAP



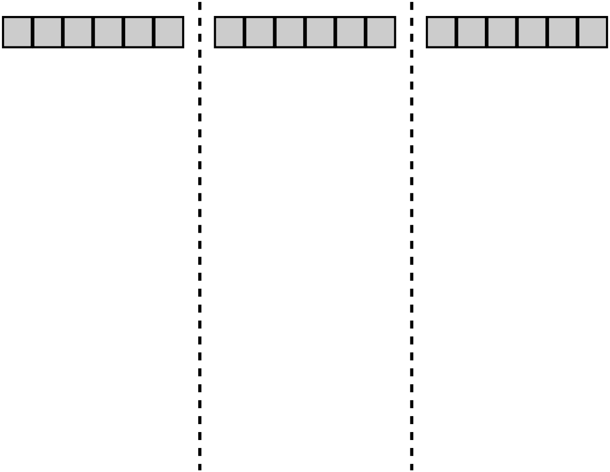
Recap: MapReduce



Recap: MapReduce

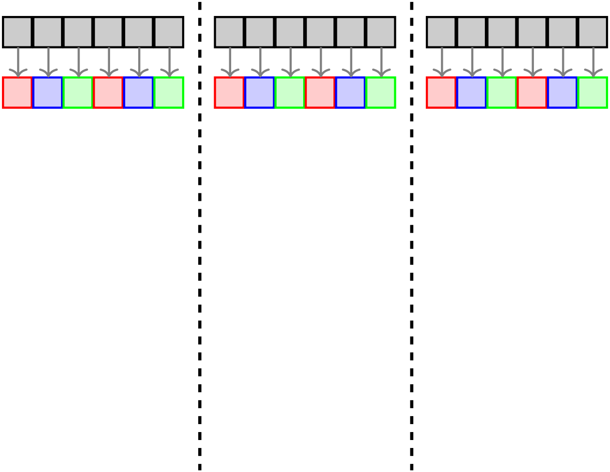


Recap: MapReduce

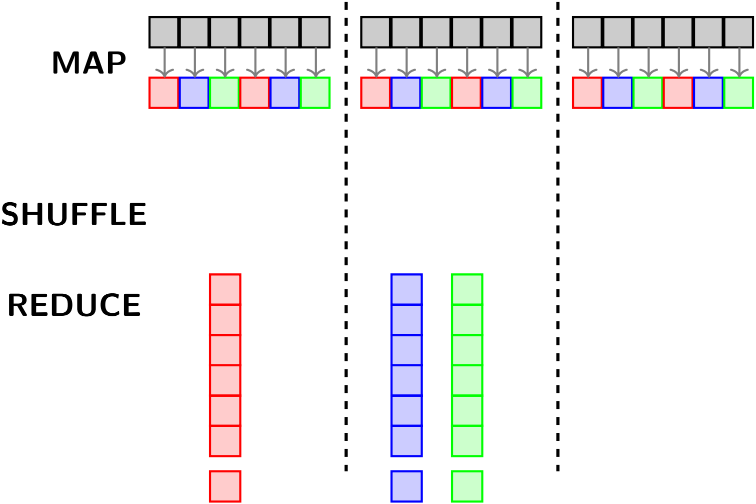


Recap: MapReduce

MAP



Recap: MapReduce



Assumption

Karlsruhe Institute of Technology

The results produced by used reducers do not depend on the order in the array.

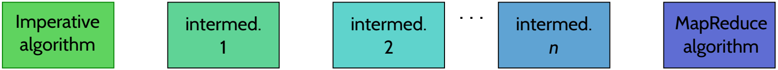
Then we can consider the deterministic non-distributed setting.

[Commutativity of Reducers, Chen et al. 2016]

Our Approach

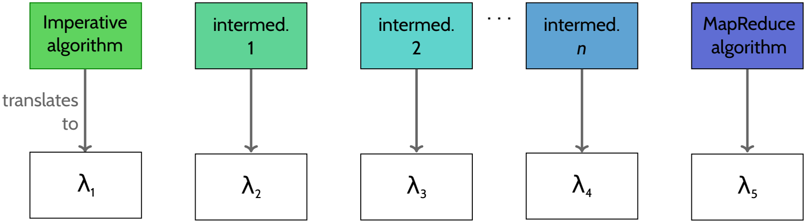
- Goal: Imperative $\cong$ MapReduce Algorithm

Our Approach



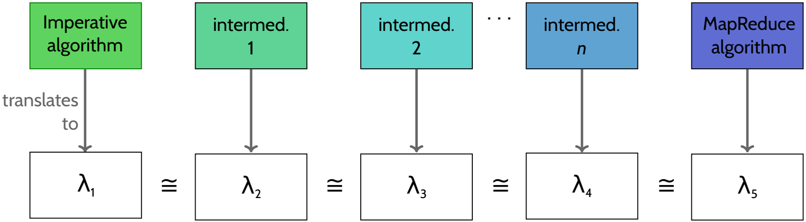
- Goal: Imperative $\cong$ MapReduce Algorithm
- User-provided intermediate steps that guide the proof.

Our Approach



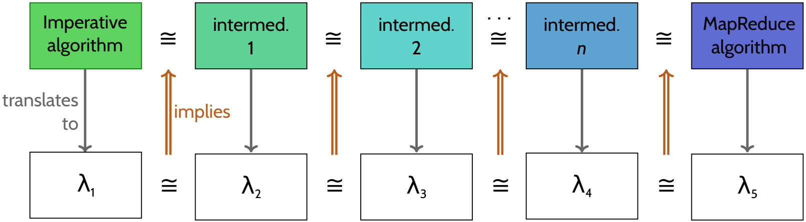
- Goal: Imperative $\cong$ MapReduce Algorithm
- User-provided intermediate steps that guide the proof.
- Translate into equivalent functional expressions

Our Approach



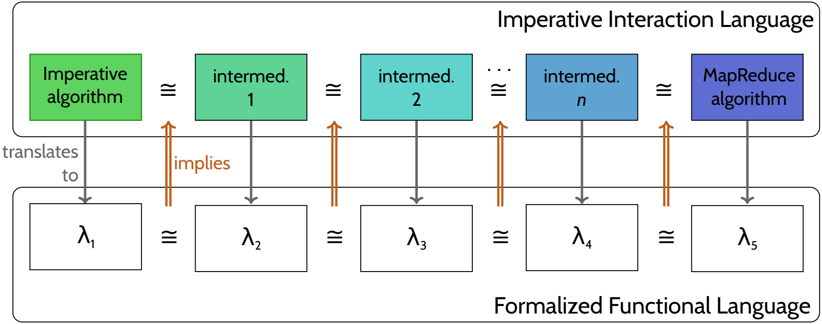
- Goal: Imperative ≅ MapReduce Algorithm
- User-provided intermediate steps that guide the proof.
- Translate into equivalent functional expressions
- Prove equivalences there

Our Approach



- Goal: Imperative ≅ MapReduce Algorithm
- User-provided intermediate steps that guide the proof.
- Translate into equivalent functional expressions
- Prove equivalences there
- Hence, equivalence on the original programs

Our Approach



- Goal: Imperative ≅ MapReduce Algorithm
- User-provided intermediate steps that guide the proof.
- Translate into equivalent functional expressions
- Prove equivalences there
- Hence, equivalence on the original programs

Languages

Interaction (IL)	Formalised Functional (FFL)

Languages

Karlsruhe Institute of Technology

Interaction (IL)	**Formalised Functional (FFL)**
`while`	typed λ calculus $(+\ \mu)$

Languages

Interaction (IL)	**Formalised Functional (FFL)**
`while`	typed λ calculus $(+\ \mu)$
integers, rationals, sequences (arrays)	

Languages

Interaction (IL)	Formalised Functional (FFL)
`while`	typed λ calculus $(+\ \mu)$
integers, rationals, sequences (arrays)	

`P(xs) { sum := 0`
`for i := 0..n`
`  sum := sum + xs[i]`
`return sum; }`
	$\lambda xs.\ \text{fold}(\lambda sum.\lambda i.sum + xs[i])$
	$0\ \text{range}(0, n)$

Languages

<table>
<tr><th>Interaction (IL)</th><th>Formalised Functional (FFL)</th></tr>
<tr><td><code>while</code></td><td>typed λ calculus $(+\ \mu)$</td></tr>
<tr><td colspan="2">integers, rationals, sequences (arrays)</td></tr>
<tr><td><pre>P(xs) { sum := 0
for i := 0..n
  sum := sum + xs[i]
return sum; }</pre></td><td>$\lambda xs.\ \mathrm{fold}(\lambda sum.\lambda i.sum + xs[i])$
$0\ \mathrm{range}(0, n)$</td></tr>
<tr><td>semantics by translation to FFL</td><td>Big steps semantics $\Rightarrow_{bs}$</td></tr>
</table>

Languages

Interaction (IL)	**Formalised Functional (FFL)**
`while`	typed λ calculus $(+\ \mu)$
integers, rationals, sequences (arrays)	

<pre>P(xs) { sum := 0
for i := 0..n
  sum := sum + xs[i]
return sum; }</pre>
	$\lambda xs.\ \mathrm{fold}(\lambda sum.\lambda i.sum + xs[i])$
	$0\ \mathrm{range}(0, n)$

semantics by translation to FFL	Big steps semantics $\Rightarrow_{bs}$
	Implemented in Coq

Languages

Interaction (IL)	Formalised Functional (FFL)
`while`	typed λ calculus $(+\ \mu)$
integers, rationals, sequences (arrays)	

`P(xs) { sum := 0`
`for i := 0..n`
`  sum := sum + xs[i]`
`return sum; }`
	$\lambda xs.\ \text{fold}(\lambda sum.\lambda i.sum + xs[i])$
	$0\ \text{range}(0, n)$

semantics by translation to FFL	Big steps semantics $\Rightarrow_{bs}$
	Implemented in Coq
$\forall xs.$ `P`$(xs) ==$ `P'`(xs)	$\forall xs\ v.(P\ xs) \Rightarrow_{bs} v \leftrightarrow (P'\ xs) \Rightarrow_{bs} v$

Two Types of Rules

Context-Independent Rules

- local and uniform
- rewriting rules on subexpressions
- for **paradigm shifting**: (e.g., from loop to map)

Context-Dependent Rules

- (more) global and flexible
- relational reasoning using coupling predicates
- **maintaining control structure**, adapt data

Two Types of Rules

Context-Independent Rules

- local and uniform
- rewriting rules on subexpressions
- for **paradigm shifting**: (e.g., from loop to map)

Context-Dependent Rules

- (more) global and flexible
- relational reasoning using coupling predicates
- **maintaining control structure**, adapt data

Context-Independent Rules – Example

Transform parallisable loop to `map`

$$\text{fold}((\lambda(xs, i).\,\text{write}(xs, i, f(xs[i]))), ys, \text{range}(0, \text{length}(xs)))$$
$$\rightsquigarrow\ \text{map}(f, ys)$$

$xs \notin FV(f), i \notin FV(f), xs \notin FV(ys), i \notin FV(ys), f$ is not stuck

Context-Independent Rules – Example

Transform parallisable loop to map

$$\text{fold}((\lambda(xs, i).\,\text{write}(xs, i, f(xs[i]))), ys, \text{range}(0, \text{length}(xs)))$$
$$\rightsquigarrow \quad \text{map}(f, ys)$$

$xs \notin FV(f), i \notin FV(f), xs \notin FV(ys), i \notin FV(ys), f$ is not stuck

IL-level:

```
for i : range(0, length(xs)) {
  xs[i] := f(xs[i])                ⇝    xs := map(f, xs)
}
```

Context-Independent Rules – Example

Transform parallisable loop to `map`

$$\begin{aligned} & \mathrm{fold}((\lambda(xs, i).\, \mathrm{write}(xs, i, f(xs[i]))), ys, \mathrm{range}(0, \mathrm{length}(xs))) \\ \rightsquigarrow\ & \mathrm{map}(f, ys) \end{aligned}$$

$xs \notin FV(f), i \notin FV(f), xs \notin FV(ys), i \notin FV(ys), f$ is not stuck

IL-level:

```
for i : range(0, length(xs)) {
  xs[i] := f(xs[i])                ⇝    xs := map(f, xs)
}
```

Collection of rules

- inspected the examples delivered with *Thrill*.
- identified 13 typical patterns for steps.

Context-Independent Rules

1. Extract independent part of loop body to `map`
2. Group accesses to the same index of an array
3. Group accesses to the same key
4. Fuse consecutive calls to `map` into a single call of `map`
5. Separate arrays that are read from and written to
6. Flatten `fold` over array of arrays
7. Transform `iter` to `fold`
8. Transform `fold` to `map`
9. `fold` over the values in an array instead of over index range
10. `map` over the values in an array instead of over the index range
11. Commute writing back updates to an array and applying `map` to the result
12. Commute read and zip
13. Commute read and map

Context-Independent Rules

1. Extract independent part of loop body to `map`
2. Group accesses to the same index of an array
3. Group accesses to the same key
4. Fuse consecutive calls to `map` into a single call of `map`
5. Separate arrays that are read from and written to
6. Flatten `fold` over array of arrays
7. Transform `iter` to `fold`
8. Transform `fold` to `map`
9. `fold` over the values in an array instead of over index range
10. `map` over the values in an array instead of over the index range
11. Commute writing back updates to an array and applying `map` to the result
12. Commute read and zip
13. Commute read and map

Formalised in Coq

mostly proved
(work in progress)

Two Types of Rules

Context-Independent Rules

- local and uniform
- rewriting rules on subexpressions
- for **paradigm shifting**: (e.g., from loop to map)

Context-Dependent Rules

- (more) global and flexible
- relational reasoning using coupling predicates
- **maintaining control structure**, adapt data

Relational While Rule

`while(c1) { B1 }` $\rightsquigarrow$ `while(c2) { B2 }`

1. a loop (plus surrounding statements) can be rewritten ...

Relational While Rule

$$\frac{[x_1 = x_2]\ \texttt{while(c1)\{B1\}} \parallel \texttt{while(c2)\{B2\}}\ [x_1 = x_2]}{\texttt{while(c1) \{ B1 \}} \rightsquigarrow \texttt{while(c2) \{ B2 \}}}$$

1. a loop (plus surrounding statements) can be rewritten ...
2. if the two programs can be proved equivalent ...

Relational While Rule

$$\frac{\dfrac{[x_1 = x_2]\ \texttt{while(c1 or c2) \{ if(c1) B1; if(c2) B2; \}}\ [x_1 = x_2]}{[x_1 = x_2]\ \texttt{while(c1)\{B1\}} \parallel \texttt{while(c2)\{B2\}}\ [x_1 = x_2]}}{\texttt{while(c1) \{ B1 \}} \rightsquigarrow \texttt{while(c2) \{ B2 \}}}$$

1. a loop (plus surrounding statements) can be rewritten ...
2. if the two programs can be proved equivalent ...
3. which can be shown using product programs

Example: PageRank

$$ranks_0(p) = \frac{1}{\#\texttt{links}}$$

Example: PageRank

$$ranks_0(p) = \frac{1}{\#\texttt{links}}$$

$$\Delta_k(p) = \sum_{(o,p)\in\texttt{links}} \frac{ranks_{k-1}(o)}{\#\texttt{links}[\texttt{o}]}$$

Example: PageRank

Karlsruhe Institute of Technology

$$ranks_0(p) = \frac{1}{\#\texttt{links}}$$

$$\Delta_k(p) = \sum_{(o,p)\in\texttt{links}} \frac{ranks_{k-1}(o)}{\#\texttt{links}[\texttt{o}]}$$

$$ranks_k(p) = \alpha \cdot \Delta_k(p) + \frac{1-\alpha}{\#\texttt{links}}$$

Example: PageRank

$$ranks_0(p) = \frac{1}{\#\texttt{links}}$$

$$\Delta_k(p) = \sum_{(o,p)\in\texttt{links}} \frac{ranks_{k-1}(o)}{\#\texttt{links[o]}}$$

$$ranks_k(p) = \alpha \cdot \Delta_k(p) + \frac{1-\alpha}{\#\texttt{links}}$$

Sidenote

a special case of sparse matrix-vector multiplication;
broader applications in scientific computing.

Example: PageRank

```
fn pageRank(links : [[Int]], α : Rat, iterations : Int) -> [Rat] {
  var iter : Int := 0;
  var ranks : [Rat] := replicate(#links, 1. / #links);
  while (iter < iterations) {
    var Δ : [Rat] := replicate(#links, 0);
    for (pageId : range(0, #links)) {
      var contribution : Rat := ranks[pageId] / #links[pageId];
      for (outgoingId : links[pageId]) {
        Δ[outgoingId] := Δ[outgoingId] + contribution;
      }
    }
    for (pageId : range(0, #links)) {
      ranks[pageId] := α * Δ[pageId] + (1-α)/#links;
    }
    iter := iter + 1;
  }
  return ranks;
} 1
```

Example: PageRank

```
fn pageRank(links : [[Int]], α : Rat, iterations : Int) -> [Rat] {
  var iter : Int := 0;
  var ranks : [Rat] := replicate(#links, 1. / #links);
  while (iter < iterations) {
    var Δ : [Rat] := replicate(#links, 0);
    for (pageId : range(0, #links)) {
      var contribution : Rat := ranks[pageId] / #links[pageId];
      for (outgoingId : links[pageId]) {
        Δ[outgoingId] := Δ[outgoingId] + contribution;
      }
    }
    ranks :=
      map((rank : Rat) => α * rank + (1 - α) / #links, Δ);
    iter := iter + 1;
  }
  return ranks;
} 2
```

Rewrite rule: transform loop to map

```
for (i : range(0, #as)) {
  b[i] := g(as[i]);              ⇝    b := map(g, as)
}
```

Example: PageRank

```
fn pageRank(links : [[Int]], α : Rat, iterations : Int) -> [Rat] {
  var iter : Int := 0;
  var ranks : [Rat] := replicate(#links, 1. / #links);
  while (iter < iterations) {
    var Δ : [Rat] := replicate(#links, 0);
    for (pageId : range(0, #links)) {
      var contribution : Rat := ranks[pageId] / #links[pageId];
      for (outgoingId : links[pageId]) {
        Δ[outgoingId] := Δ[outgoingId] + contribution;
      }
    }
    ranks :=
      map((rank : Rat) => α * rank + (1 - α) / #links, Δ);
    iter := iter + 1;
  }
  return ranks;
} 2
```

Example: PageRank

```
fn pageRank(links : [[Int]], α : Rat, iterations : Int) -> [Rat] {
  var iter : Int := 0;
  var ranks : [Rat] := replicate(#links, 1 / #links);
  while (iter < iterations) {
    var Δ : [Rat] := replicate(#links, 0);
    var outRanks : [[Int] * Rat] := zip(links, ranks);
    for (pageId : range(0, #links)) {
      var contribution : Rat := snd outRanks[pageId] / #(fst outRanks[pageId]);
      for (outgoingId : fst outRanks[pageId]) {
        Δ[outgoingId] := Δ[outgoingId] + contribution;
      }
    }
    ranks :=
      map((rank : Rat) => α * rank + (1 - α) / #links, Δ);
    iter := iter + 1;
  }
  return ranks;
} 3
```

Example: PageRank

```
fn pageRank(links : [[Int]], α : Rat, iterations : Int) -> [Rat] {
  var iter : Int := 0;
  var ranks : [Rat] := replicate(#links, 1 / #links);
  while (iter < iterations) {
    var Δ : [Rat] := replicate(#links, 0);
    var outRanks : [[Int] * Rat] := zip(links, ranks);
    for (pageId : range(0, #links)) {
      var contribution : Rat := snd outRanks[pageId] / #(fst outRanks[pageId]);
      for (outgoingId : fst outRanks[pageId]) {
        Δ[outgoingId] := Δ[outgoingId] + contribution;
      }
    }
    ranks :=
      map((rank : Rat) => α * rank + (1 - α) / #links, Δ);
    iter := iter + 1;
  }
  return ranks;
} 3
```

Relational loop invariant

$$\Delta_1 = \Delta_2 \quad \wedge \quad \mathtt{outRanks}_2 = \mathtt{zip}(\mathtt{links}_1, \mathtt{ranks}_1)$$

Example: PageRank

```
fn pageRank(links : [[Int]], α : Rat, iterations : Int) -> [Rat] {
  var iter : Int := 0;
  var ranks : [Rat] := replicate(#links, 1 / #links);
  while (iter < iterations) {
    var Δ : [Rat] := replicate(#links, 0);
    var outRanks : [[Int] * Rat] := zip(links, ranks);
    for (links_rank : outRanks) {
      var contribution : Rat := snd links_rank / #fst links_rank;
      for (outgoingId : fst links_rank) {
        Δ[outgoingId] := Δ[outgoingId] + contribution;
      }
    }
    ranks :=
      map((rank : Rat) => α * rank + (1 - α) / #links, Δ);
    iter := iter + 1;
  }
  return ranks;
} 4
```

Example: PageRank

```
fn pageRank(links : [[Int]], α : Rat, iterations : Int) -> [Rat] {
  var iter : Int := 0;
  var ranks : [Rat] := replicate(#links, 1 / #links);
  while (iter < iterations) {
    var Δ : [Rat] := replicate(#links, 0);
    var outRanks : [[Int] * Rat] := zip(links, ranks);
    var linksAndContrib : [[Int * Rat]] :=
      map((links_rank : [Int] * Rat) =>
             map((link : Int) =>
                    (link, snd links_rank / #fst links_rank),
                 fst links_rank),
          outRanks);
    for (link_contribs : linksAndContrib) {
      for (link_contrib : link_contribs) {
        Δ[fst link_contrib] :=
          Δ[fst link_contrib] + snd link_contrib;
      }
    }
    ranks :=
      map((rank : Rat) => α * rank + (1 - α) / #links, Δ);
    iter := iter + 1;
  }
  return ranks;
} 5
```

Example: PageRank

```
fn pageRank(links : [[Int]], α : Rat, iterations : Int) -> [Rat] {
  var iter : Int := 0;
  var ranks : [Rat] := replicate(#links, 1 / #links);
  while (iter < iterations) {
    var Δ : [Rat] := replicate(#links, 0);
    var outRanks : [[Int] * Rat] := zip(links, ranks);
    var linksAndContrib : [[Int * Rat]] :=
      map((links_rank : [Int] * Rat) =>
            map((link : Int) =>
                  (link, snd links_rank / #fst links_rank),
                fst links_rank),
          outRanks);
    for (link_contribs : linksAndContrib) {
      for (link_contrib : link_contribs) {
        Δ[fst link_contrib] :=
          Δ[fst link_contrib] + snd link_contrib;
      }
    }
```

Relational loop invariant

$\Delta_1 = \Delta_2 \wedge$

$\forall ij.\ \texttt{fst linksAndContrib}_2[i][j] = (\texttt{fst outRanks}_1[i])[j] \wedge$

$\texttt{snd linksAndContrib}_2[i][j] = \texttt{snd outRanks}_1[i] / \#(\texttt{fst outRanks}_1[i])$

Example: PageRank

```
fn pageRank(links : [[Int]], α : Rat, iterations : Int) -> [Rat] {
  var iter : Int := 0;
  var ranks : [Rat] := replicate(#links, 1 / #links);
  while (iter < iterations) {
    var Δ : [Rat] := replicate(#links, 0);
    var outRanks : [[Int] * Rat] := zip(links, ranks);
    var linksAndContrib : [[Int * Rat]] :=
      map((links_rank : [Int] * Rat) =>
             map((link : Int) =>
                   (link, snd links_rank / #fst links_rank),
                 fst links_rank),
        outRanks);
    for (link_contrib : concat(linksAndContrib)) {
      Δ[fst link_contrib] :=
        Δ[fst link_contrib] + snd link_contrib;
    }
    ranks :=
      map((rank : Rat) => α * rank + (1 - α) / #links, Δ);
    iter := iter + 1;
  }
  return ranks;
} 6
```

Example: PageRank

```
fn pageRank(links : [[Int]], α : Rat, iterations : Int) -> [Rat] {
  var iter : Int := 0;
  var ranks : [Rat] := replicate(#links, 1 / #links);
  while (iter < iterations) {
    var Δ : [Rat] := replicate(#links, 0);
    var outRanks : [[Int] * Rat] := zip(links, ranks);
    var linksAndContrib : [Int * Rat] :=
      flatMap((links_rank : [Int] * Rat) =>
             map((link : Int) =>
                   (link, snd links_rank / #fst links_rank),
                 fst links_rank),
          outRanks);
    for (link_contrib : linksAndContrib) {
      Δ[fst link_contrib] :=
        Δ[fst link_contrib] + snd link_contrib;
    }
    ranks :=
      map((rank : Rat) => α * rank + (1 - α) / #links, Δ);
    iter := iter + 1;
  }
  return ranks;
} 7
```

Example: PageRank

```
fn pageRank(links : [[Int]], α : Rat, iterations : Int) -> [Rat] {
  var iter : Int := 0;
  var ranks : [Rat] := replicate(#links, 1 / #links);
  while (iter < iterations) {
    var outRanks : [[Int] * Rat] := zip(links, ranks);
    var contribs : [Int * Rat] :=
      flatMap((links_rank : [Int] * Rat) =>
                 map((link : Int) => (link,
                                         snd links_rank / #fst links_rank),
                     fst links_rank),
              outRanks);
    var rankUpdates : [Int * Rat] :=
      map((link : Int) (contribs : [Rat]) =>
            (link, fold((x: Rat) (y : Rat) => x + y, 0, contribs)),
          group(contribs));
    var Δ : [Rat] := replicate(#links, 0);
    for (link_rank : rankUpdates) {
      Δ[fst link_rank] := snd link_rank;
    }
    ranks :=
      map((rank : Rat) => α * rank + (1 - α) / Δ);
    iter := iter + 1;
  }
  return ranks;
} 8
```

Example: PageRank

```
fn pageRank(links : [[Int]], α : Rat, iterations : Int) -> [Rat] {
  var iter : Int := 0;
  var ranks : [Rat] := replicate(#links, 1 / #links);
  while (iter < iterations) {
    var outRanks : [[Int] * Rat] := zip(links, ranks);
    var contribs : [Int * Rat] :=
      flatMap((links_rank : [Int] * Rat) =>
                 map((link : Int) => (link,
                                      snd links_rank / #fst links_rank),
                     fst links_rank),
              outRanks);
    var rankUpdates : [Int * Rat] :=
      map((link : Int) (contribs : [Rat]) =>
           (link, fold((x: Rat) (y : Rat) => x + y, 0, contribs)),
         group(contribs));
    var Δ : [Rat] := replicate(#links, 0);
    for (link_rank : rankUpdates) {
      Δ[fst link_rank] := snd link_rank;
    }
```

Rule group-intro

```
for ((i,v) : xs) {
  acc[i] := f(acc[i], v);
}
```

$\rightsquigarrow$

```
var xss := map((i,vs) => fold(f, acc[i], vs),
               group(acc));
for (x : concat(xss)) {
  acc := f(acc, x);
}
```

Example: PageRank

```
fn pageRank(links : [[Int]], α : Rat, iterations : Int) -> [Rat] {
  var iter : Int := 0;
  var ranks : [Rat] := replicate(#links, 1 / #links);
  while (iter < iterations) {
    var outRanks : [[Int] * Rat] := zip(links, ranks);
    var contribs : [Int * Rat] :=
      flatMap((links_rank : [Int] * Rat) =>
                map((link : Int) => (link,
                                     snd links_rank / #fst links_rank),
                    fst links_rank),
             outRanks);
    var rankUpdates : [Int * Rat] := reduceByKey('+', 0, contribs);
    var Δ : [Rat] := replicate(#links, 0);
    for (link_rank : rankUpdates) {
      Δ[fst link_rank] := snd link_rank;
    }
    ranks :=
      map((rank : Rat) => α * rank + (1 - α) / #links, Δ);
    iter := iter + 1;
  }
  return ranks;
} 9
```

Automation

Manual proof:

c. 3700 LOC in Coq, mostly handwritten, partially generated

Designed with automation in mind:

- user interaction on programming language level only.
- side conditions of rewrite rules are easily syntactically checked

Future work:

An automated tool as combination of relational reasoning and rewriting

Conclusion

Equivalence Between Imperative and MapReduce Algorithms

Deductive equivalence verification using a combination of rewriting and relational reasoning

- equivalence proofs of structurally not so similar programs
- rewriting rules: change control structure
- relational reasoning: change data structure
- next step: automation