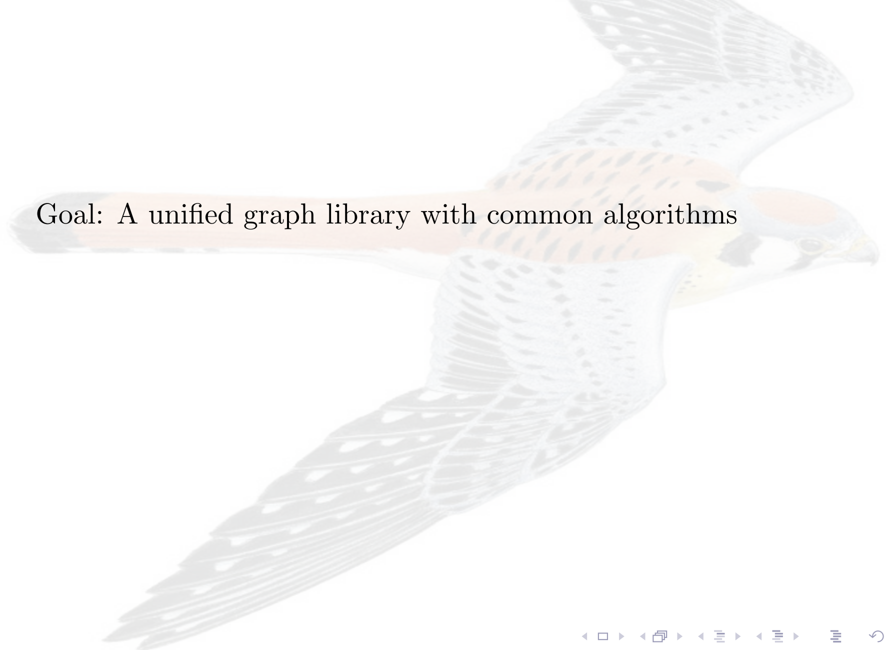
Verified Graph Algorithms in ACL2

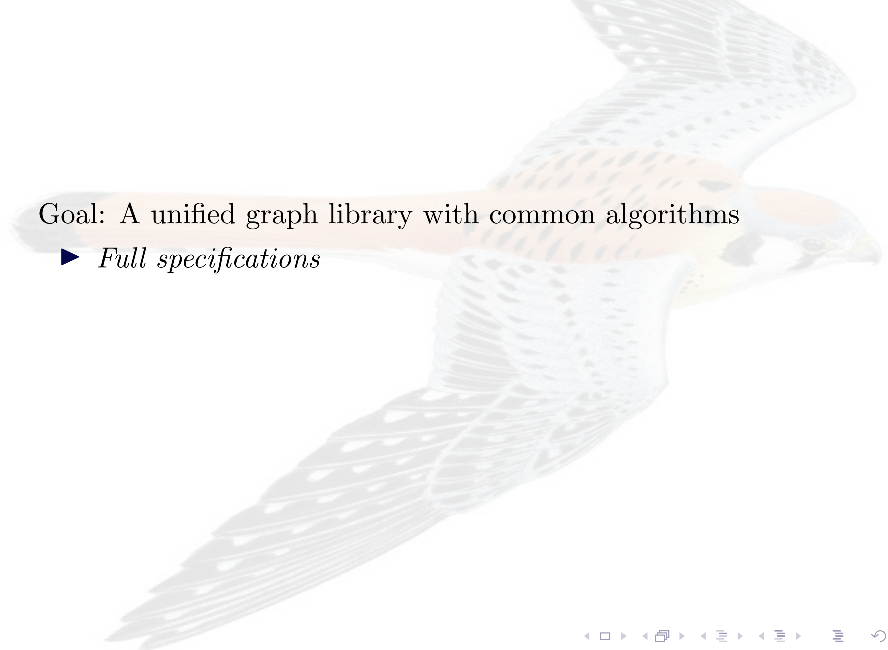
Nathan Guermond
Kestrel Institute
November 5, 2018

Another graph library?



Goal: A unified graph library with common algorithms

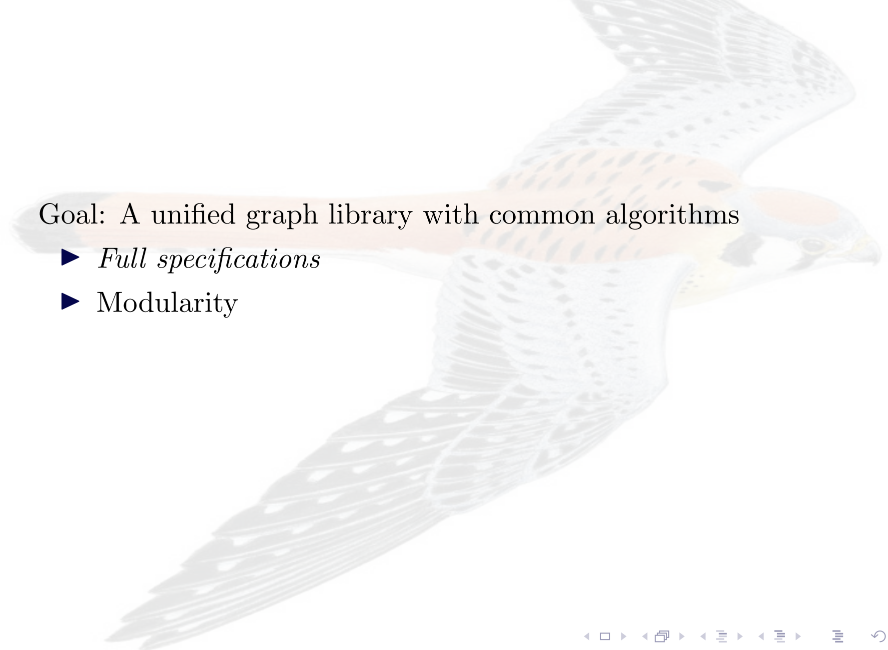
Another graph library?



Goal: A unified graph library with common algorithms

- *Full specifications*

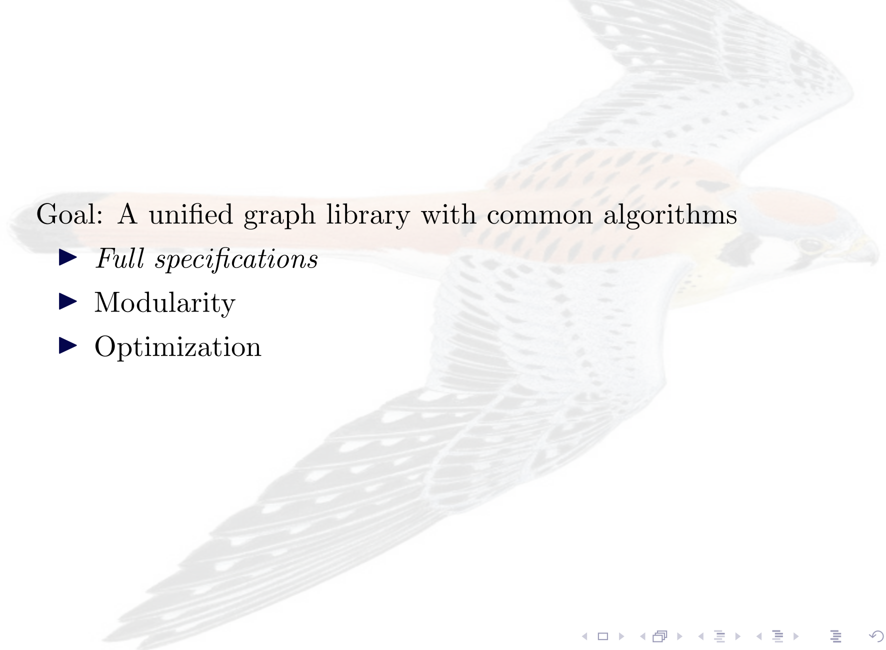
Another graph library?



Goal: A unified graph library with common algorithms

- *Full specifications*
- Modularity

Another graph library?



Goal: A unified graph library with common algorithms

- *Full specifications*
- Modularity
- Optimization

Core data structure

A graph is a dependent datastructure with

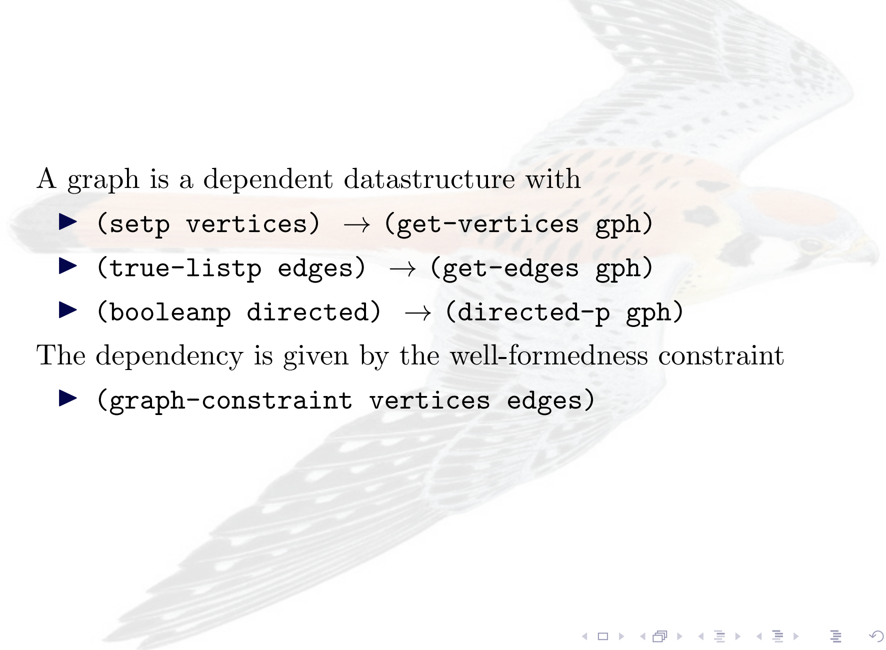
- `(setp vertices)`
- `(true-listp edges)`
- `(booleanp directed)`

Core data structure

A graph is a dependent datastructure with

- `(setp vertices)` → `(get-vertices gph)`
- `(true-listp edges)` → `(get-edges gph)`
- `(booleanp directed)` → `(directed-p gph)`

Core data structure



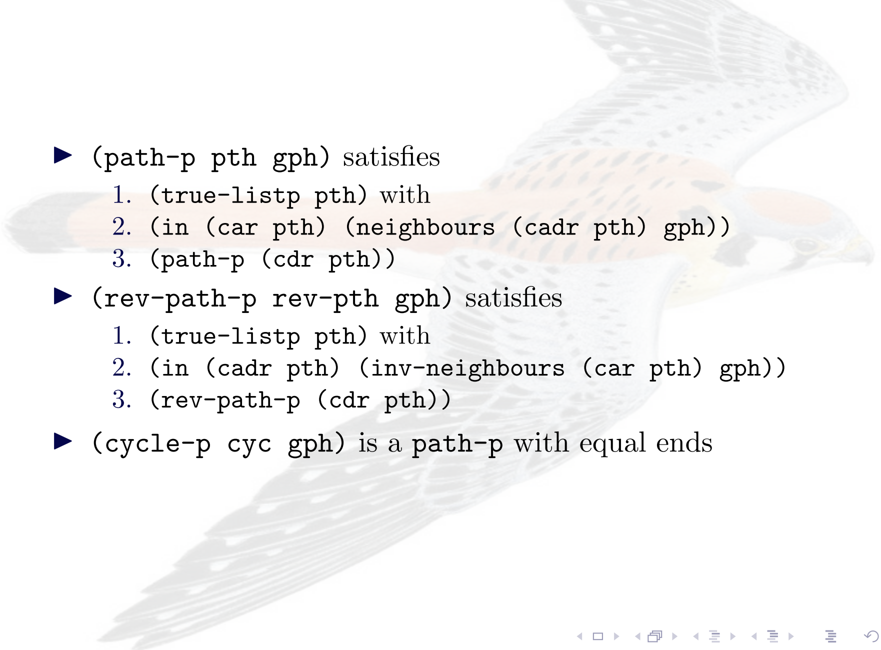
A graph is a dependent datastructure with

- `(setp vertices)` → `(get-vertices gph)`
- `(true-listp edges)` → `(get-edges gph)`
- `(booleanp directed)` → `(directed-p gph)`

The dependency is given by the well-formedness constraint

- `(graph-constraint vertices edges)`

Common data structures



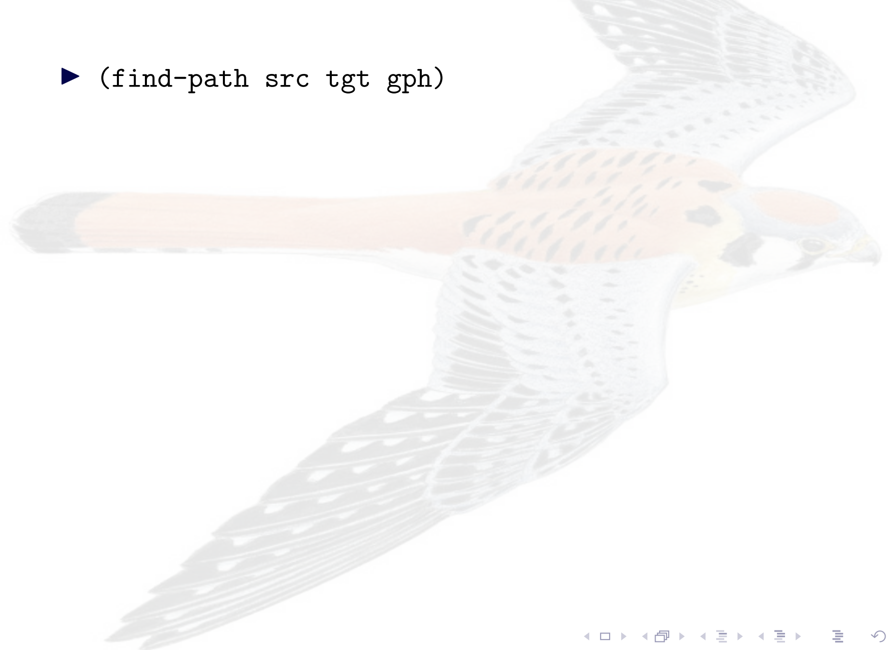
- `(path-p pth gph)` satisfies
  1. `(true-listp pth)` with
  2. `(in (car pth) (neighbours (cadr pth) gph))`
  3. `(path-p (cdr pth))`
- `(rev-path-p rev-pth gph)` satisfies
  1. `(true-listp pth)` with
  2. `(in (cadr pth) (inv-neighbours (car pth) gph))`
  3. `(rev-path-p (cdr pth))`
- `(cycle-p cyc gph)` is a `path-p` with equal ends

Algorithms and specs

- `(find-path src tgt gph)`

```
(defthm path-exists-implies-exists-path-spec
  (implies (and (path-p pth gph)
                (graph-p gph))
    (find-path (get-src pth) (get-tgt pth) gph)))

(defthm exists-path-implies-path-constructible-spec
  (implies (and (graph-p gph)
                (find-path src tgt gph))
    (let ((pth (find-path src tgt gph)))
      (and (path-p pth gph)
           (equal (get-src pth) src)
           (equal (get-tgt pth) tgt)))))
```

Algorithms and specs



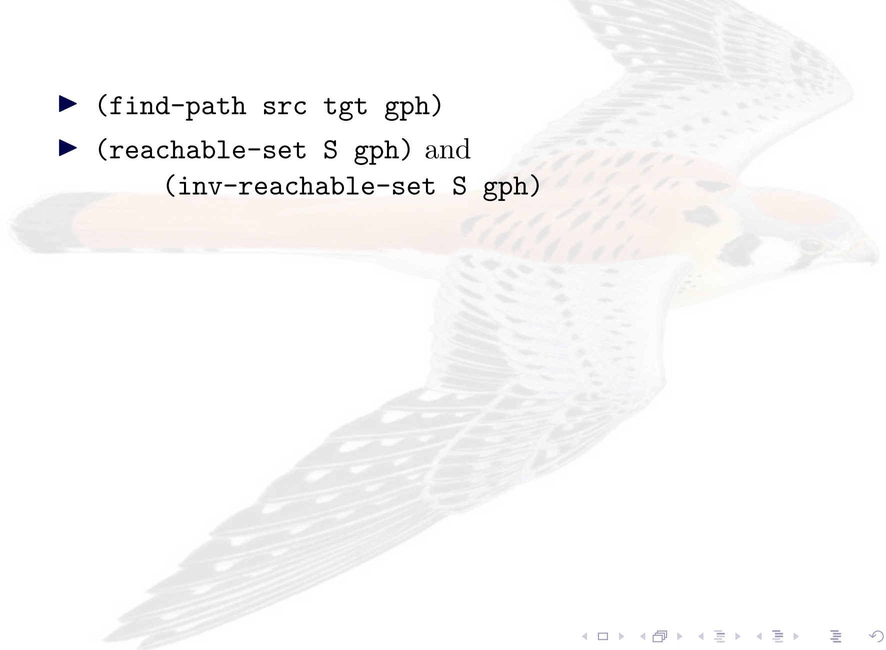
- `(find-path src tgt gph)`

Algorithms and specs

- `(find-path src tgt gph)`
- `(reachable-set S gph)`

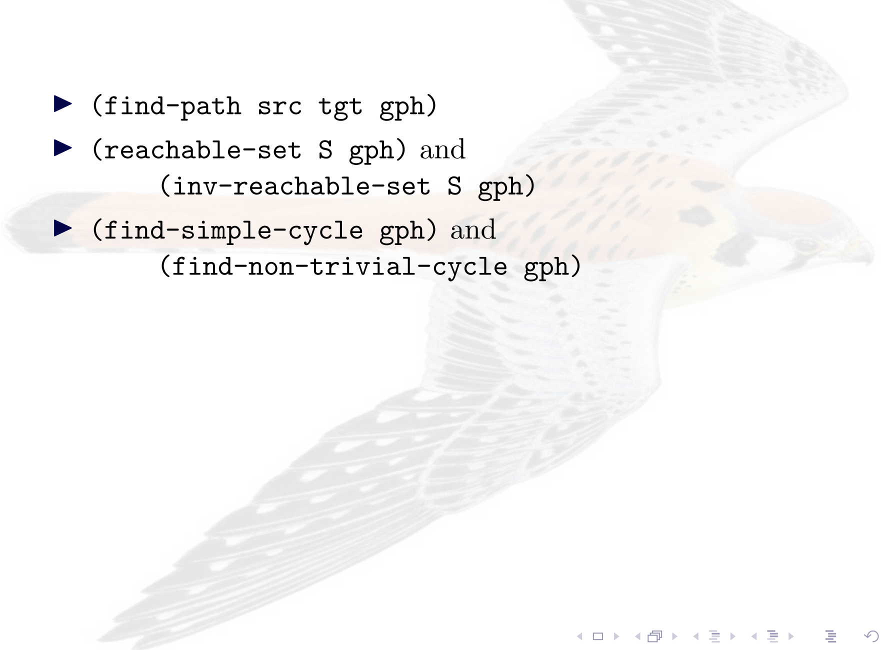
```
(defthm exists-path-implies-reachable-spec
  (implies (and (graph-p gph)
                (path-p pth gph))
    (in (get-tgt pth)
        (reachable-set
          (singleton (get-src pth)) gph))))

(defthm exists-path-from-src-to-reachable-set-spec
  (implies (and (graph-p gph)
                (in src (get-vertices gph))
                (in tgt (reachable-set
                          (singleton src) gph)))
    (find-path src tgt gph)))
```

Algorithms and specs



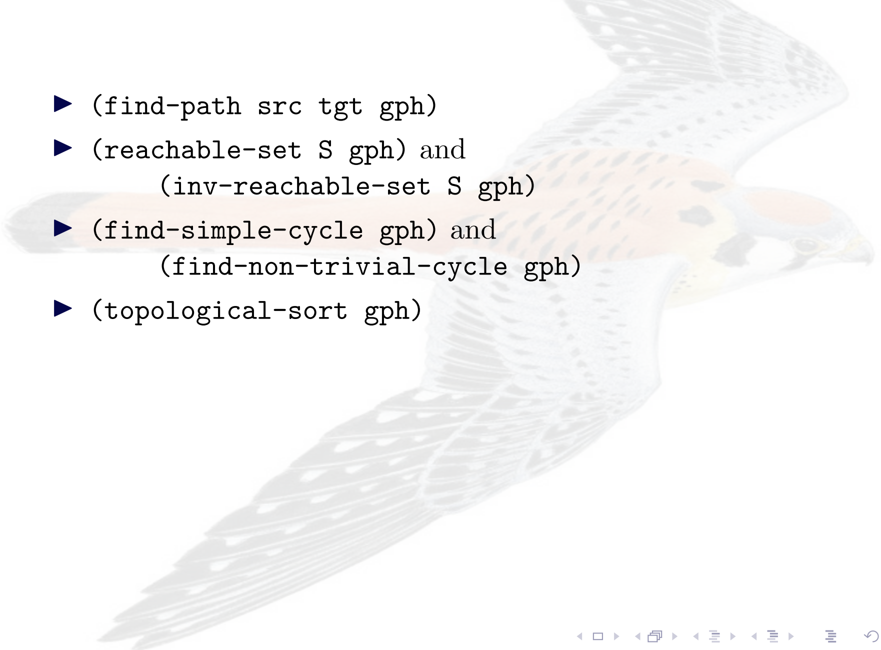
- `(find-path src tgt gph)`
- `(reachable-set S gph)` and
  `(inv-reachable-set S gph)`

Algorithms and specs



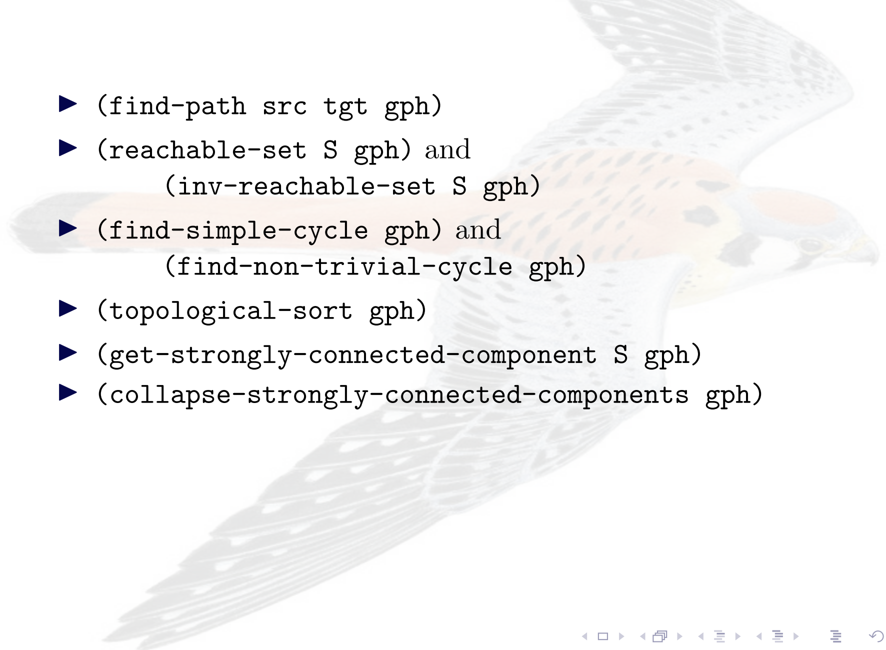
- `(find-path src tgt gph)`
- `(reachable-set S gph)` and
  `(inv-reachable-set S gph)`
- `(find-simple-cycle gph)` and
  `(find-non-trivial-cycle gph)`

Algorithms and specs



- `(find-path src tgt gph)`
- `(reachable-set S gph)` and
  `(inv-reachable-set S gph)`
- `(find-simple-cycle gph)` and
  `(find-non-trivial-cycle gph)`
- `(topological-sort gph)`

Algorithms and specs



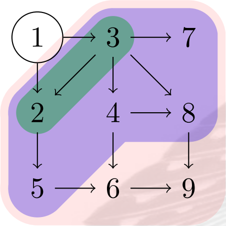
- `(find-path src tgt gph)`
- `(reachable-set S gph)` and
  `(inv-reachable-set S gph)`
- `(find-simple-cycle gph)` and
  `(find-non-trivial-cycle gph)`
- `(topological-sort gph)`
- `(get-strongly-connected-component S gph)`
- `(collapse-strongly-connected-components gph)`

Algorithms and specs

- `(find-path src tgt gph)`
- `(reachable-set S gph)` and
  `(inv-reachable-set S gph)`
- `(find-simple-cycle gph)` and
  `(find-non-trivial-cycle gph)`
- `(topological-sort gph)`
- `(get-strongly-connected-component S gph)`
- `(collapse-strongly-connected-components gph)`
  - constructed from `find-non-trivial-cycle`, `reachable-set`, and `inv-reachable-set`
  - A strongly connected compoment is given by `(Reach cyc)` $\cap$ `(InvReach cyc)`

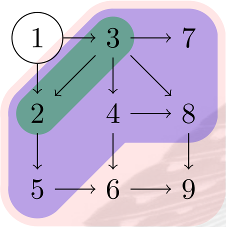
Reachable and finite differencing

- Specification is proven by a two step refinement
  - Compute set reachable in k steps
    - $S \cup (\texttt{Neighs}\ S) \cup \ldots \cup (\texttt{Neighs}\ (\ldots(\texttt{Neighs}\ S))\ldots)$



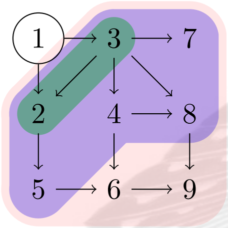
Reachable and finite differencing

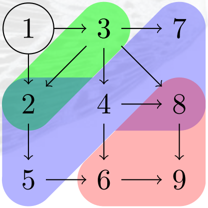
- Specification is proven by a two step refinement
  - Compute set reachable in k steps
    - $S \cup (\texttt{Neighs}\ S) \cup \ldots \cup (\texttt{Neighs}\ (\ldots(\texttt{Neighs}\ S))\ldots)$
  - Compute reachable set by iterative unioning
    - $S \cup (\texttt{Neighs}\ S) \cup (\texttt{Neighs}\ (\texttt{Neighs}\ S))\ldots$



Reachable and finite differencing

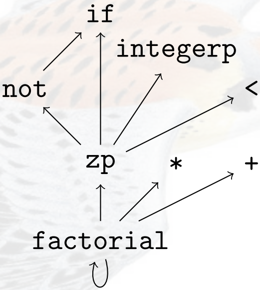
- Specification is proven by a two step refinement
  - Compute set reachable in k steps
    - $S \cup (\texttt{Neighs}\ S) \cup \ldots \cup (\texttt{Neighs}\ (\ldots(\texttt{Neighs}\ S))\ldots)$
  - Compute reachable set by iterative unioning
    - $S \cup (\texttt{Neighs}\ S) \cup (\texttt{Neighs}\ (\texttt{Neighs}\ S))\ldots$
  - Compute reachable set by finite difference
    - $S_0 = S$, $S_1 = (\texttt{Neighs}\ S_0)$
    - $D_{i+1} = S_{i+1} - S_i$, $S_{i+1} = S_i \cup (\texttt{Neighs}\ D_i)$





Applications

- Call-graphs



if

integerp

not

<

zp

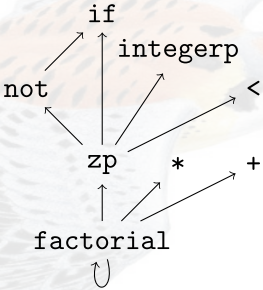
*

+

factorial

Applications

- Call-graphs
- Guard verification



if

integerp

not

<

zp

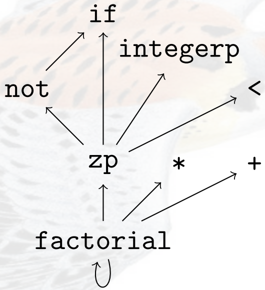
*

+

factorial

Applications

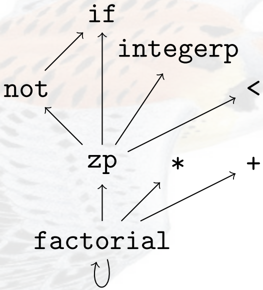
- Call-graphs
- Guard verification
- Getting ordered guard obligations



`if`
`integerp`
`not`
`<`
`zp`
`*`
`+`
`factorial`

Applications

- Call-graphs
- Guard verification
- Getting ordered guard obligations
- Your next project!



if

integerp

not

<

zp

*

+

factorial

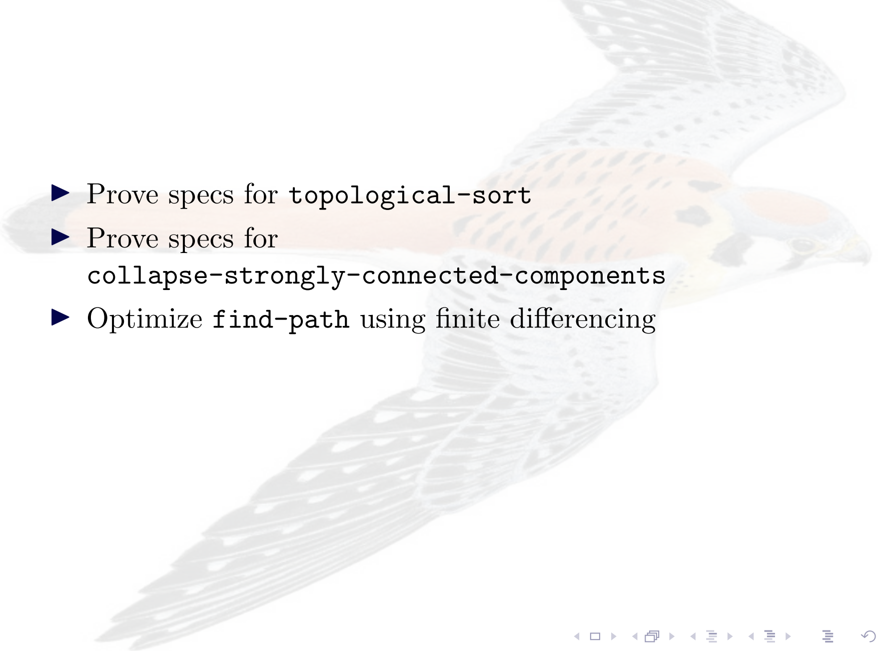
Future work

- Prove specs for `topological-sort`

Future work

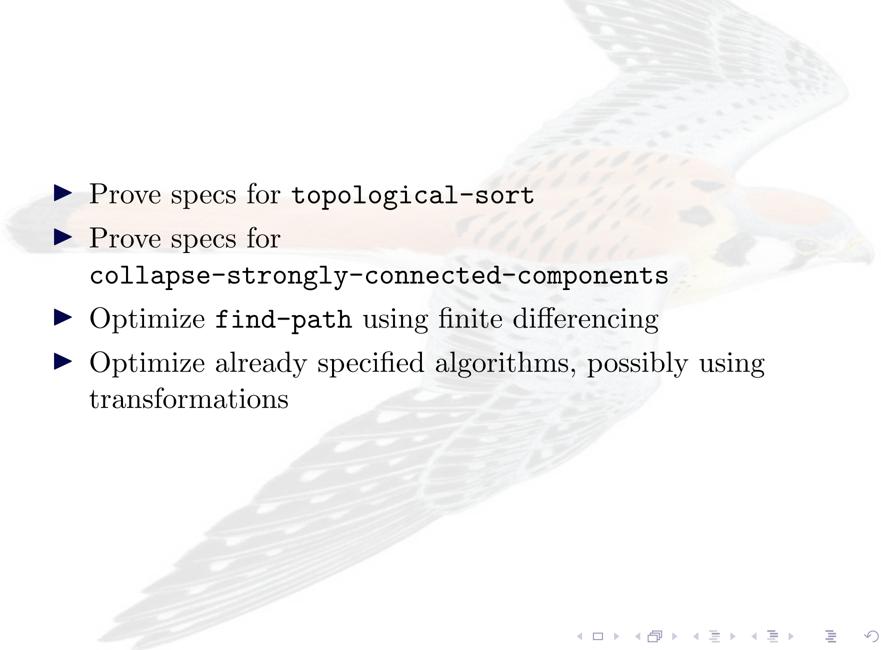
- Prove specs for `topological-sort`
- Prove specs for `collapse-strongly-connected-components`

Future work



- Prove specs for `topological-sort`
- Prove specs for `collapse-strongly-connected-components`
- Optimize `find-path` using finite differencing

Future work



- Prove specs for `topological-sort`
- Prove specs for `collapse-strongly-connected-components`
- Optimize `find-path` using finite differencing
- Optimize already specified algorithms, possibly using transformations