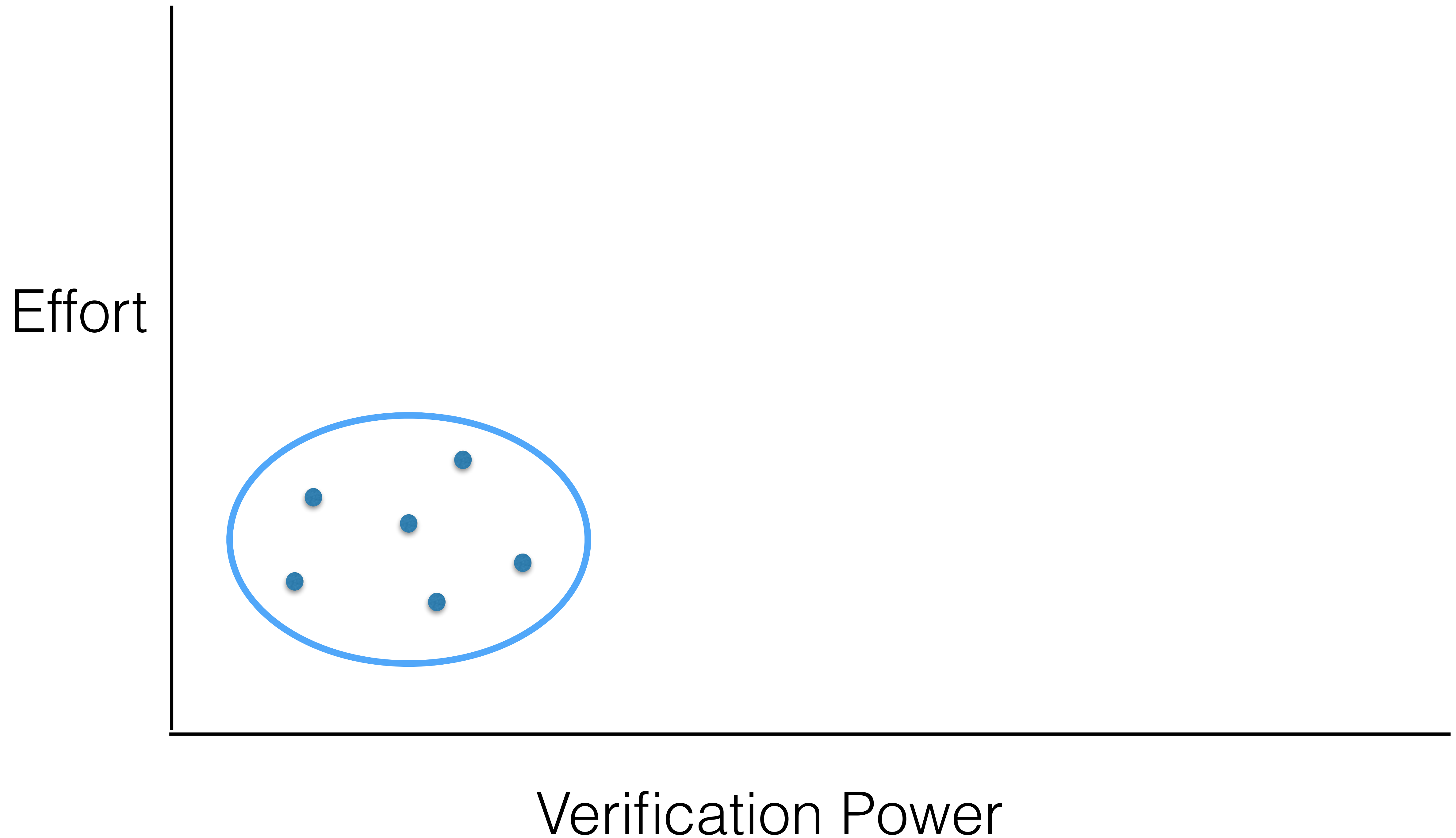
# Verifying Invariants of Lock-free Data Structures with Rely-Guarantee and Refinement Types

**Colin S. Gordon**, Michael D. Ernst, Dan Grossman, and Matthew J. Parkinson

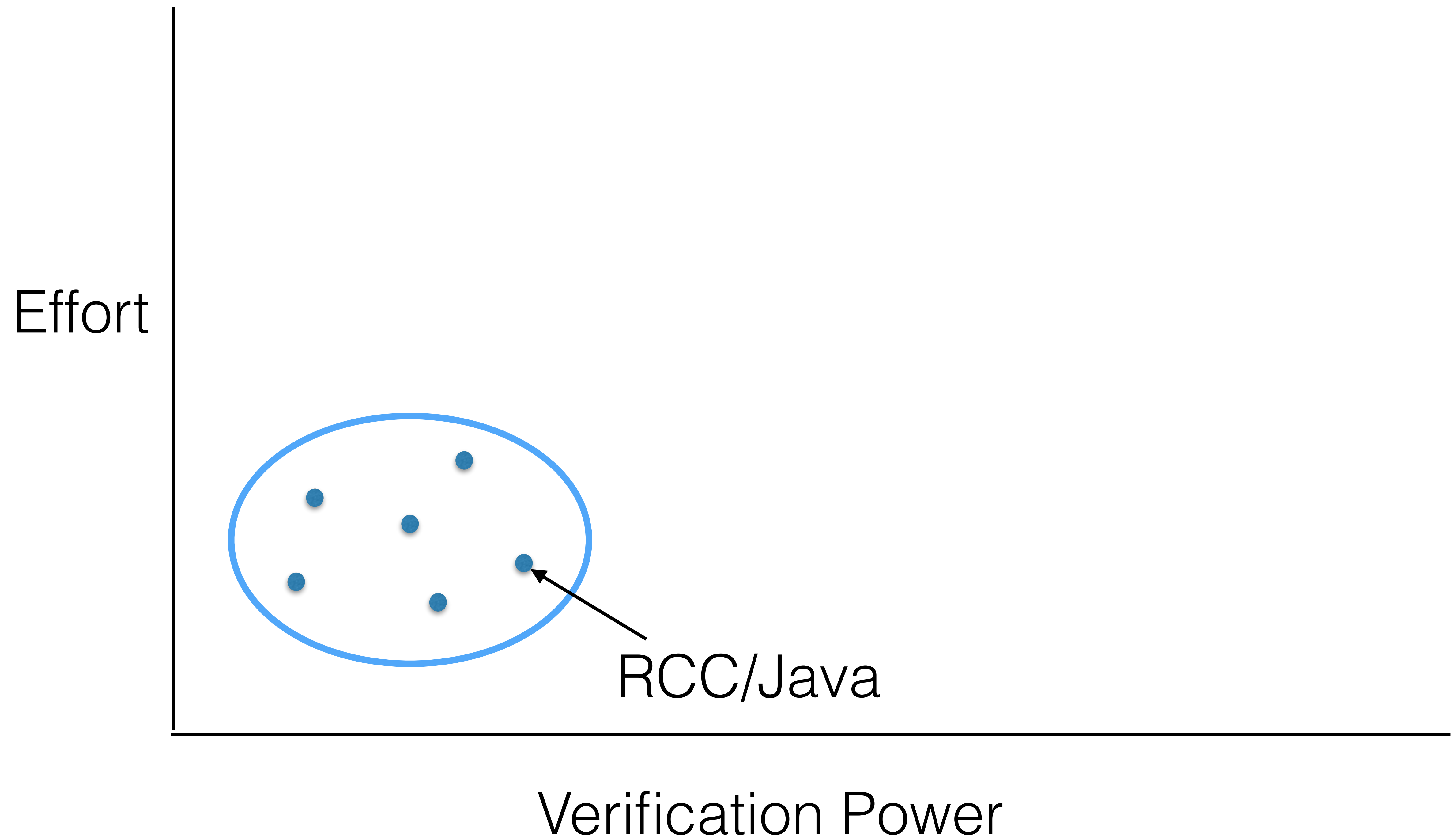
TOPLAS (May 2017) @ PLDI 2017

# Verification Effort vs. Power

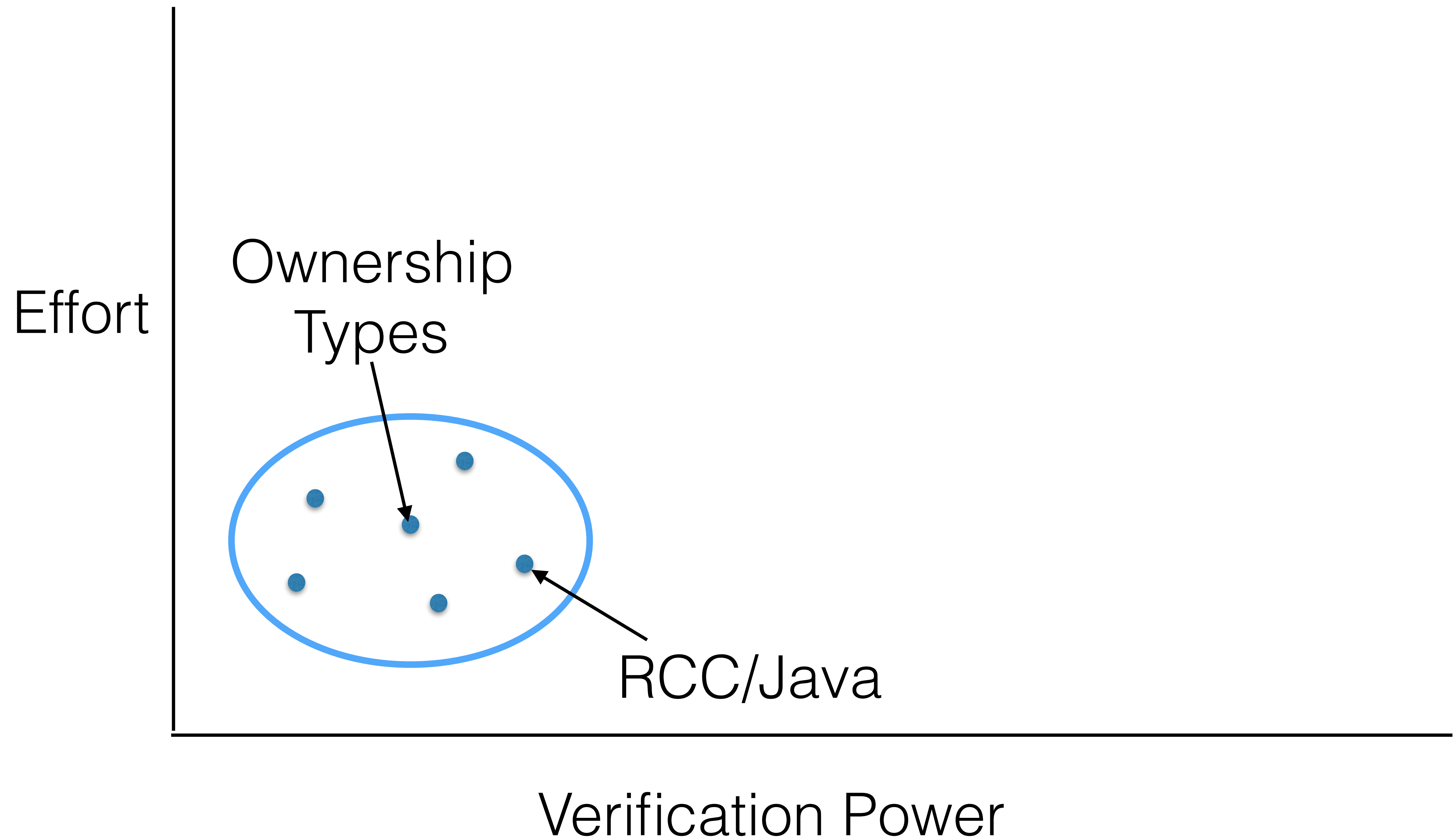
# Verification Effort vs. Power



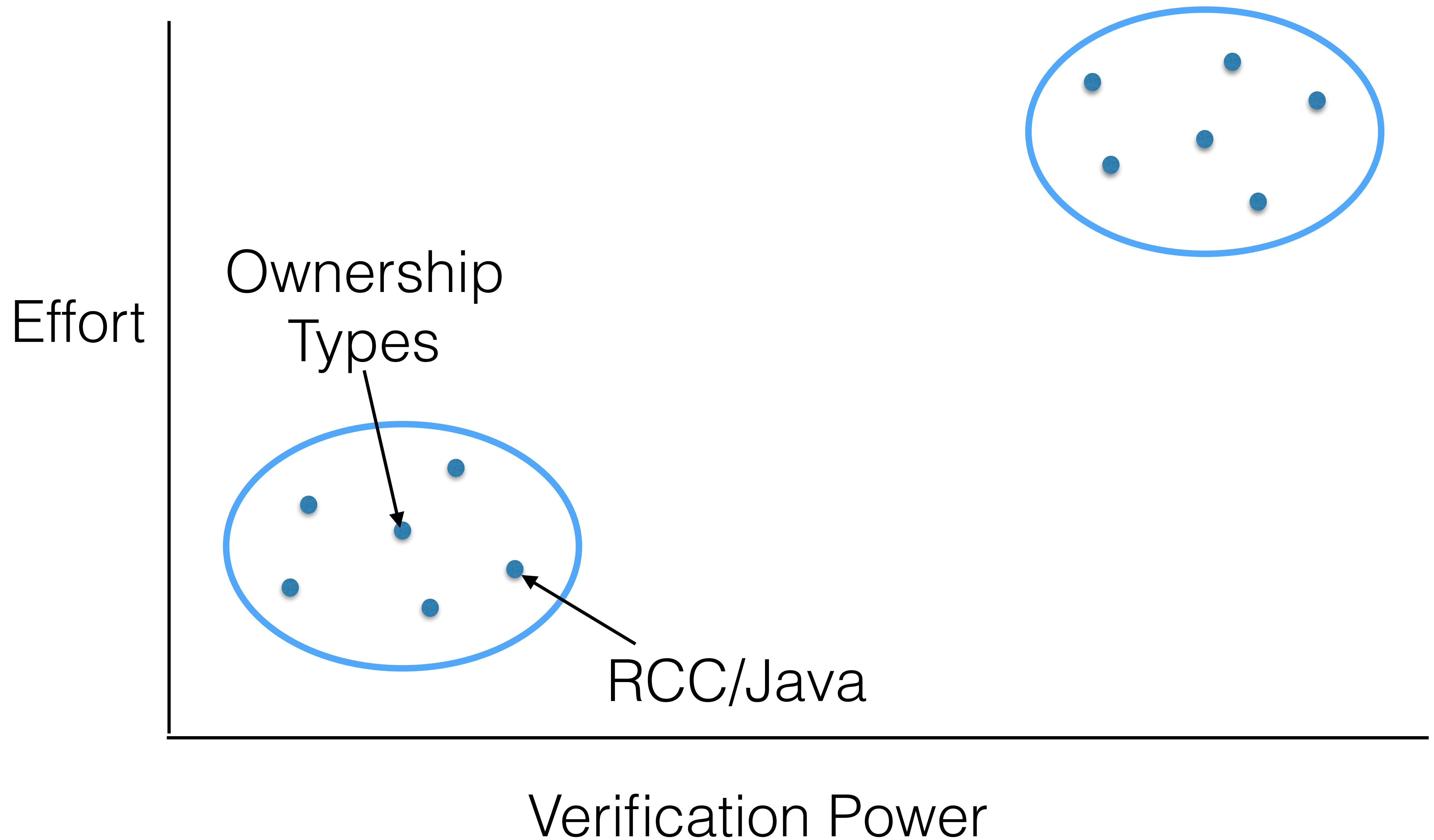
# Verification Effort vs. Power



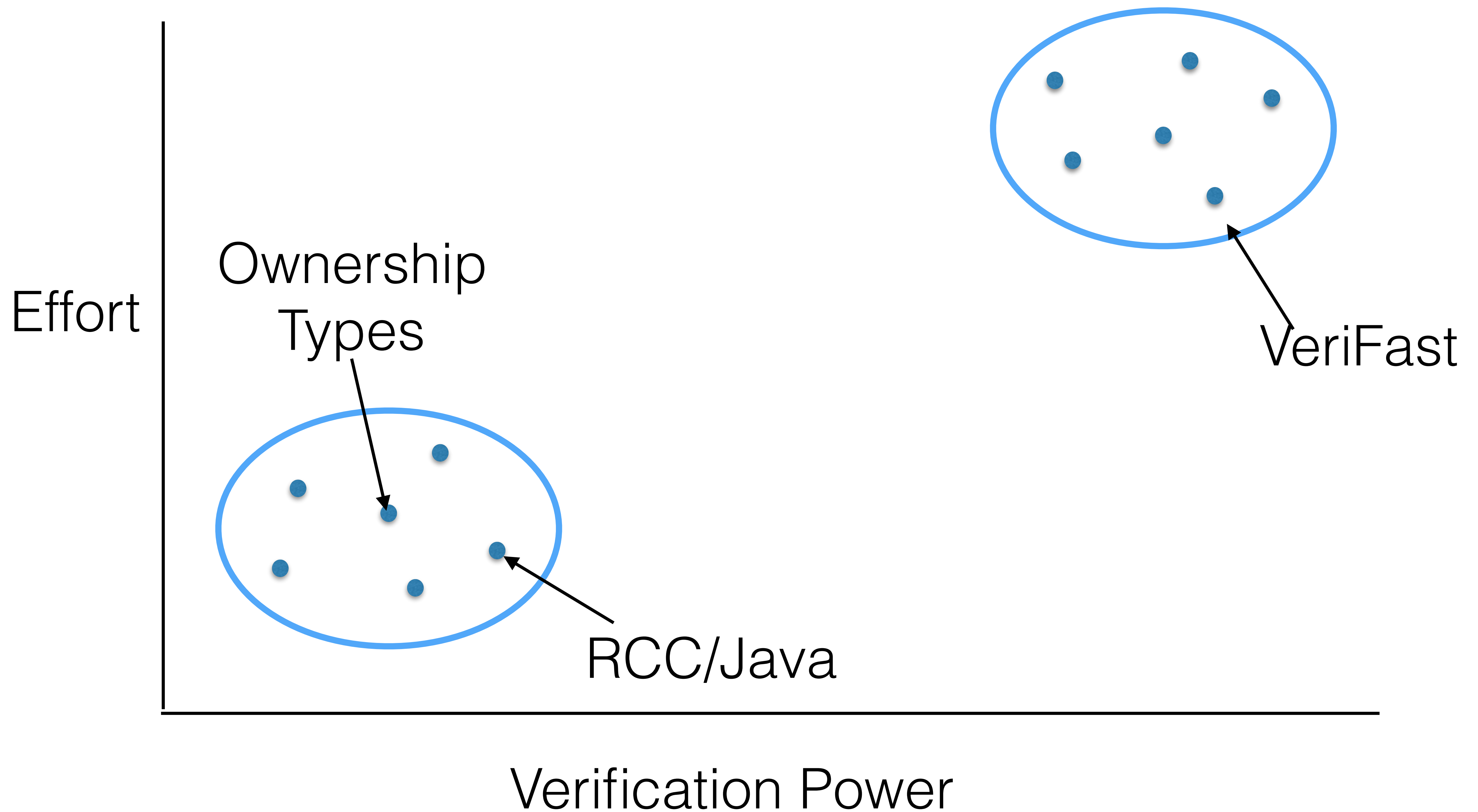
# Verification Effort vs. Power



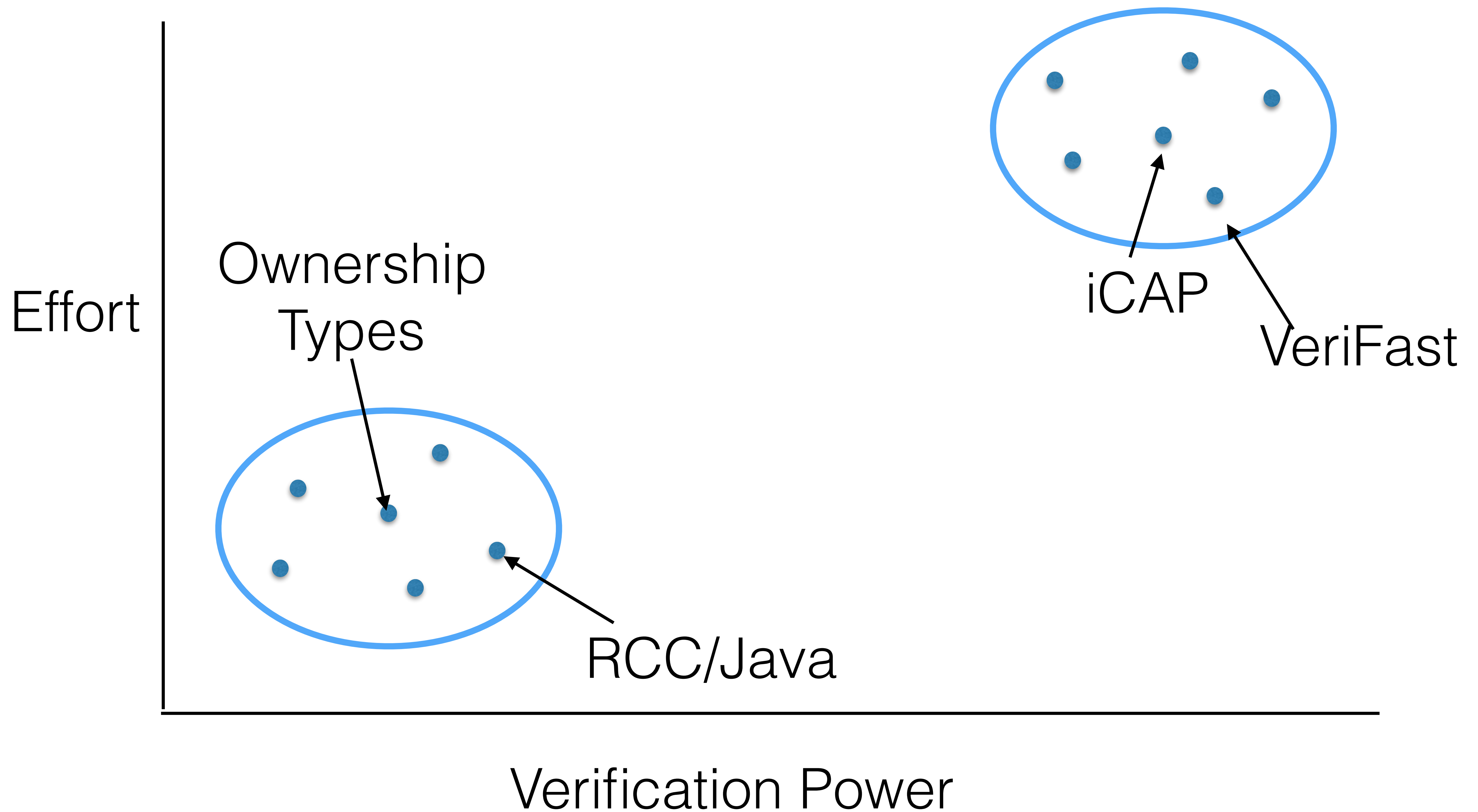
# Verification Effort vs. Power



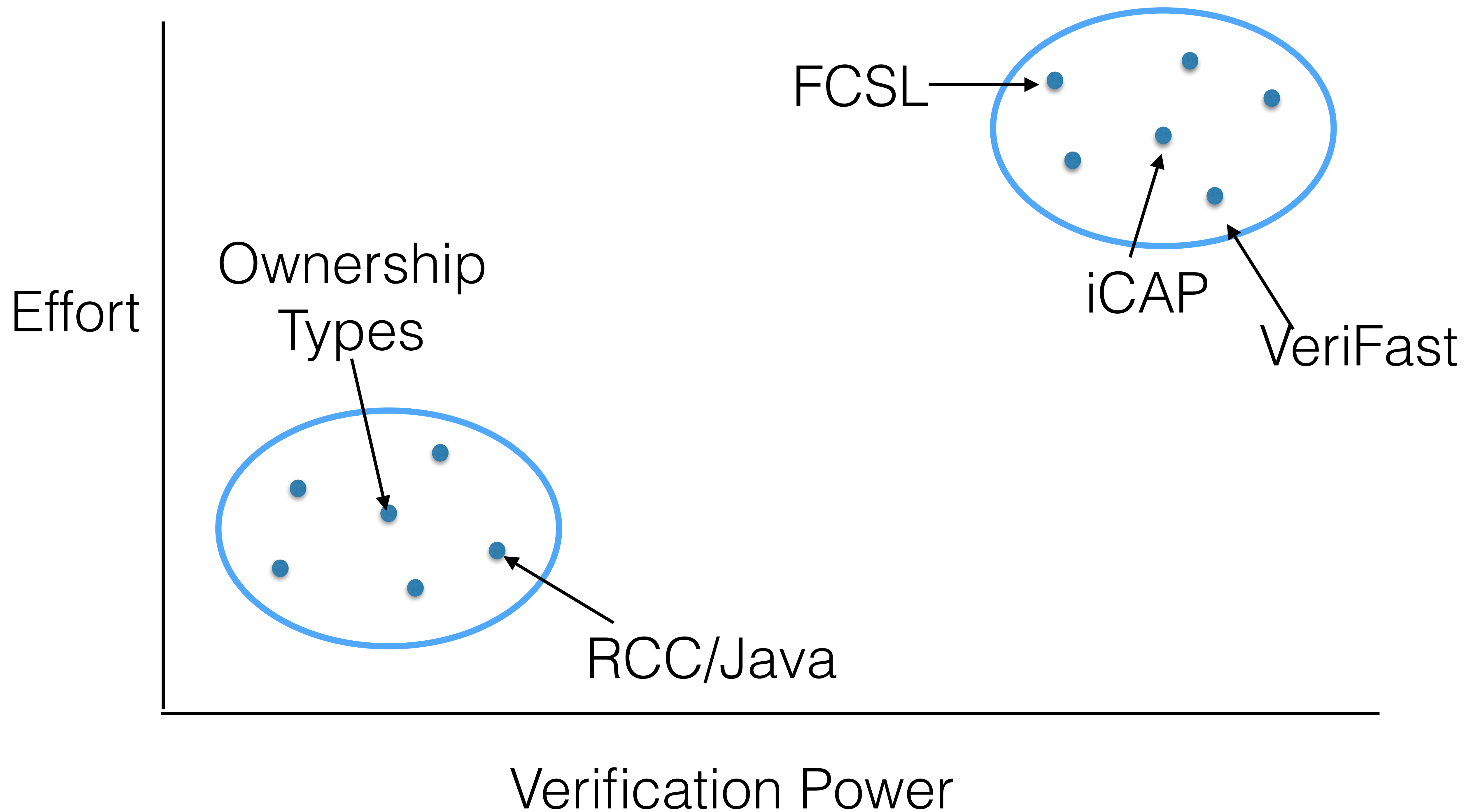
# Verification Effort vs. Power



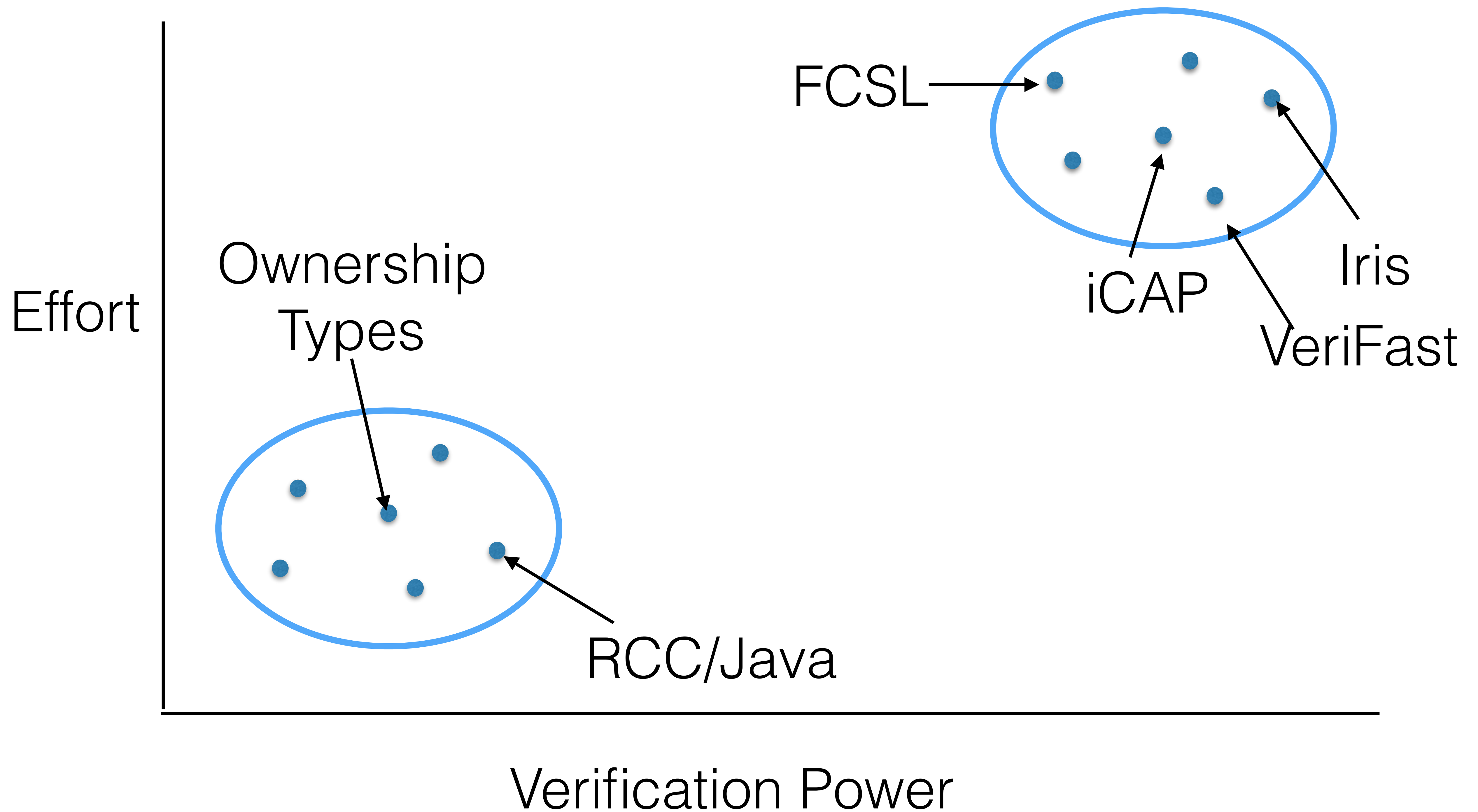
# Verification Effort vs. Power



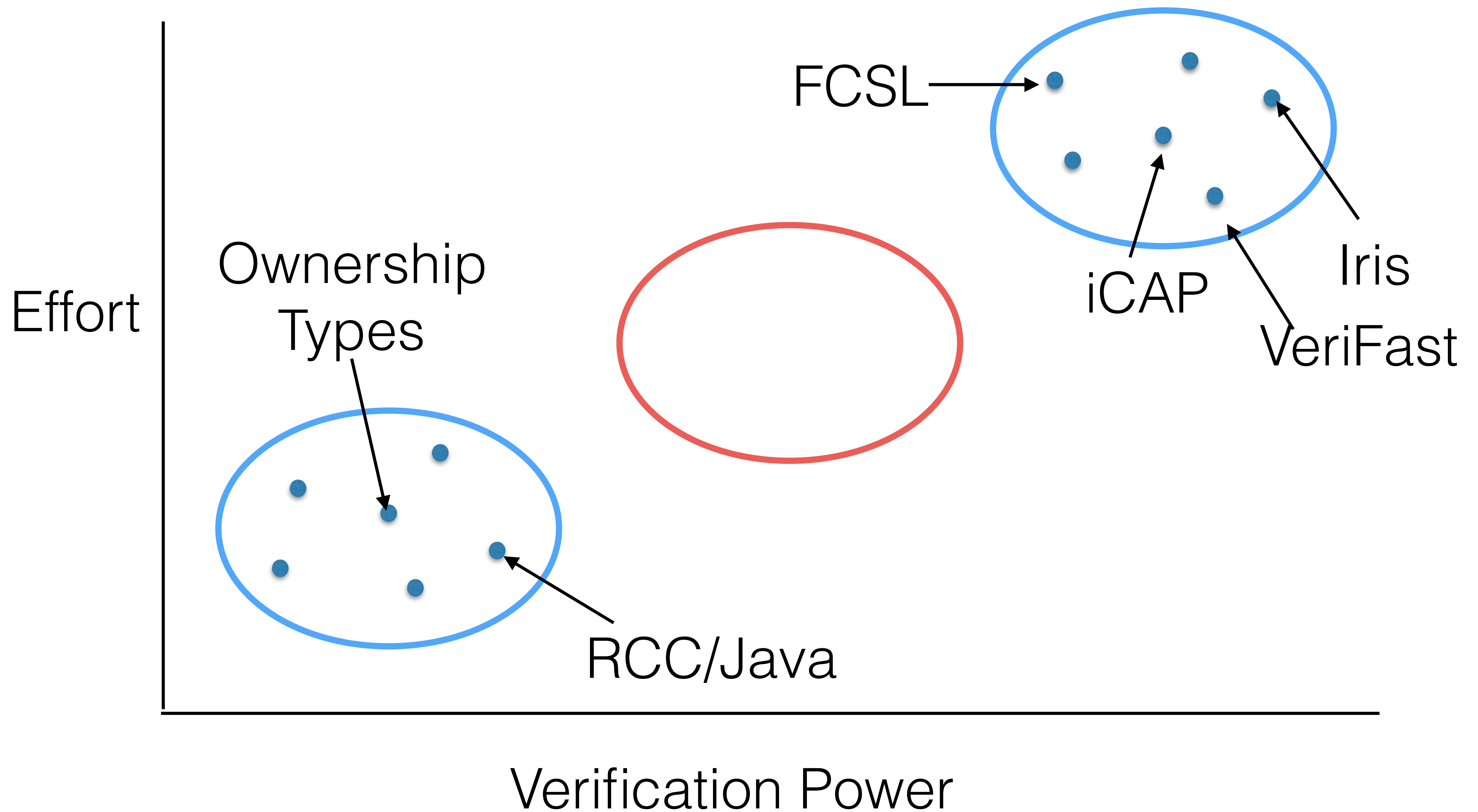
# Verification Effort vs. Power



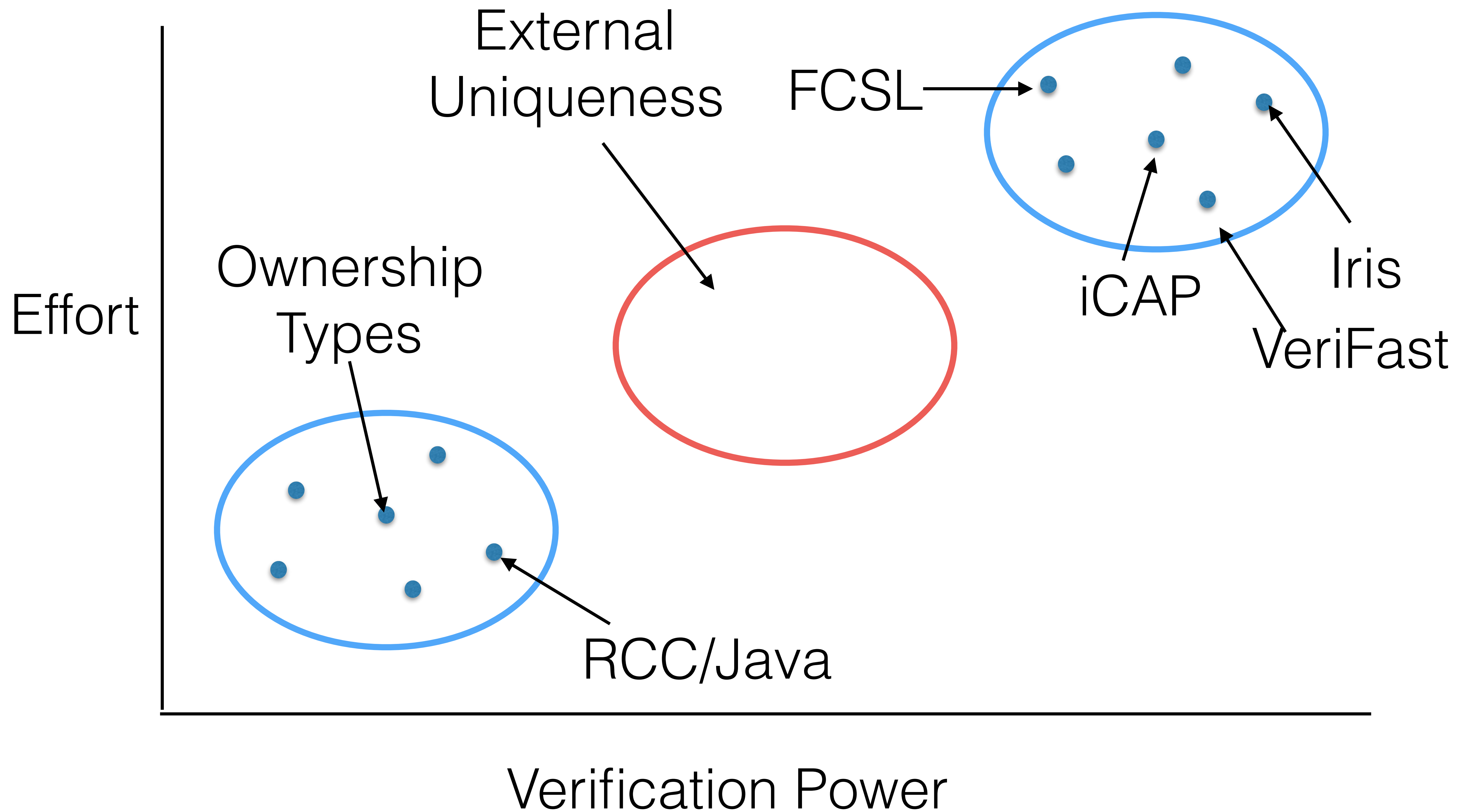
# Verification Effort vs. Power



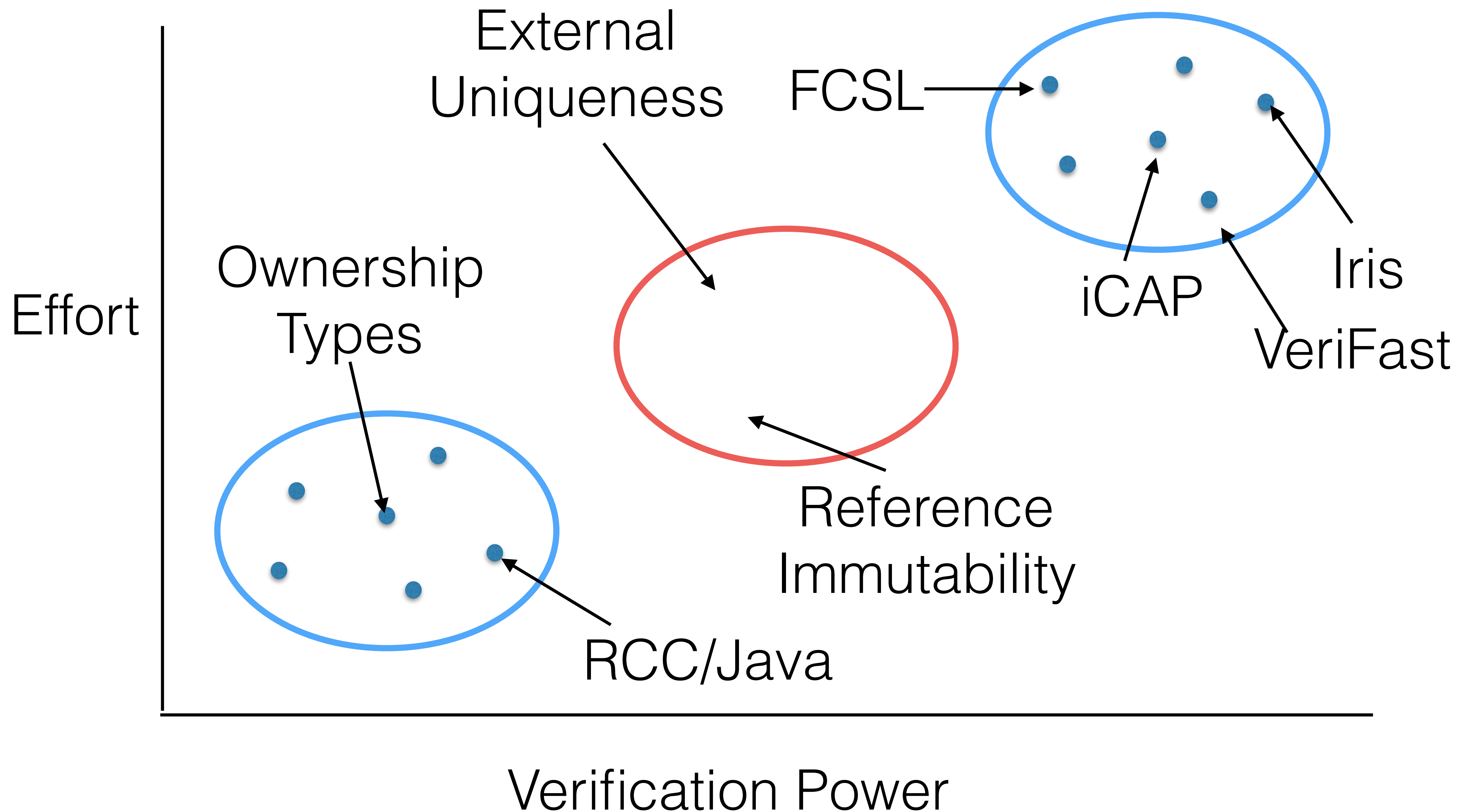
# Verification Effort vs. Power



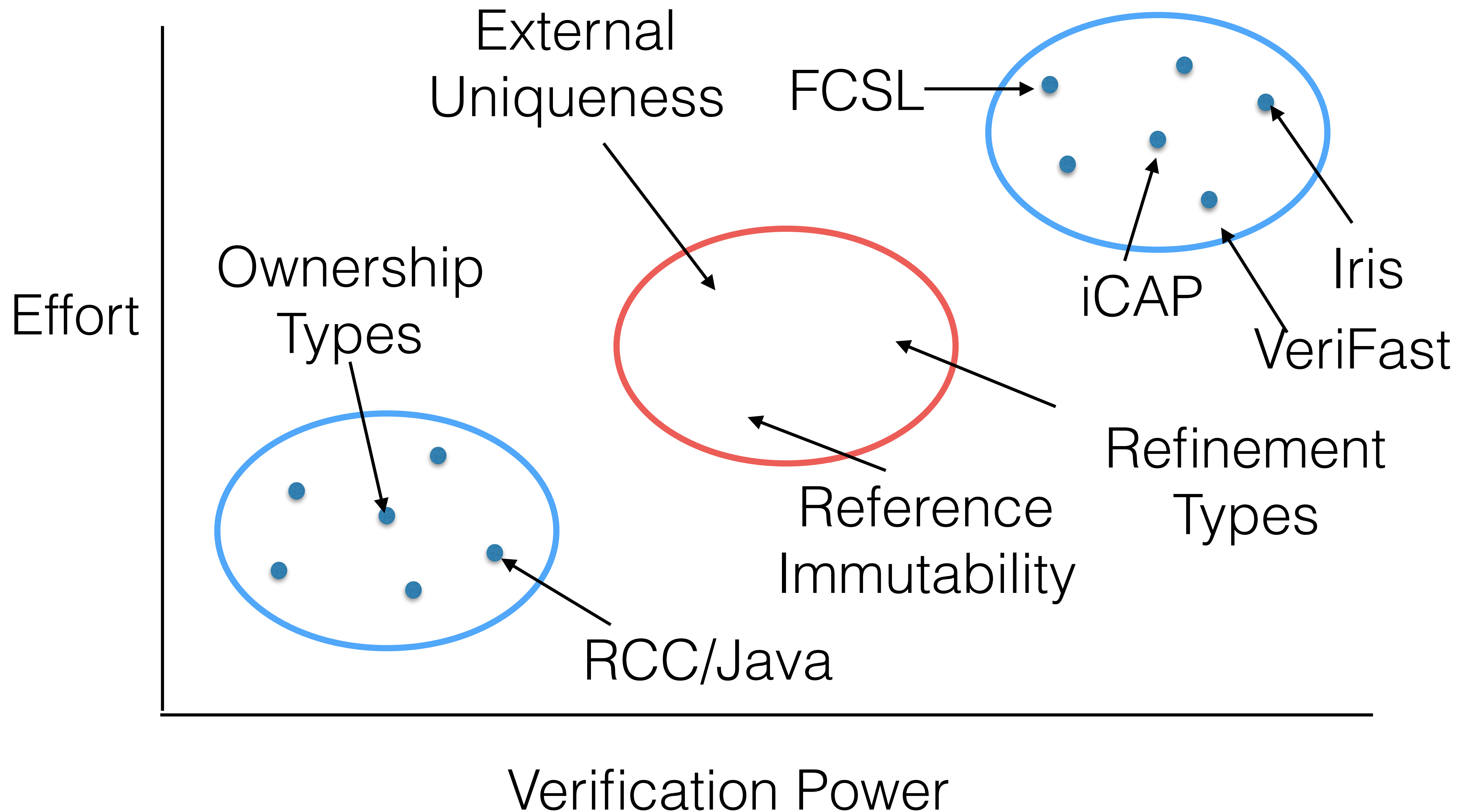
# Verification Effort vs. Power



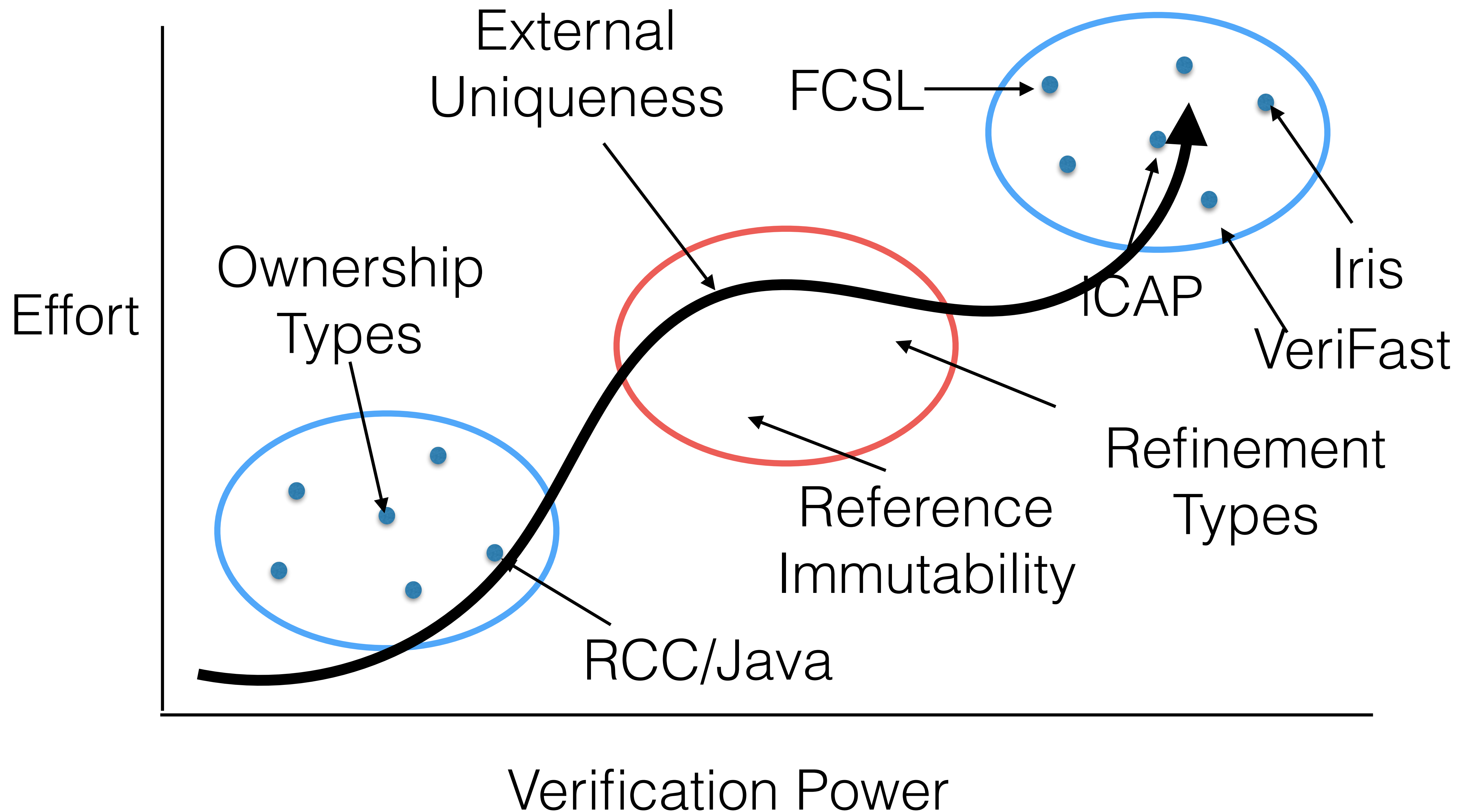
# Verification Effort vs. Power



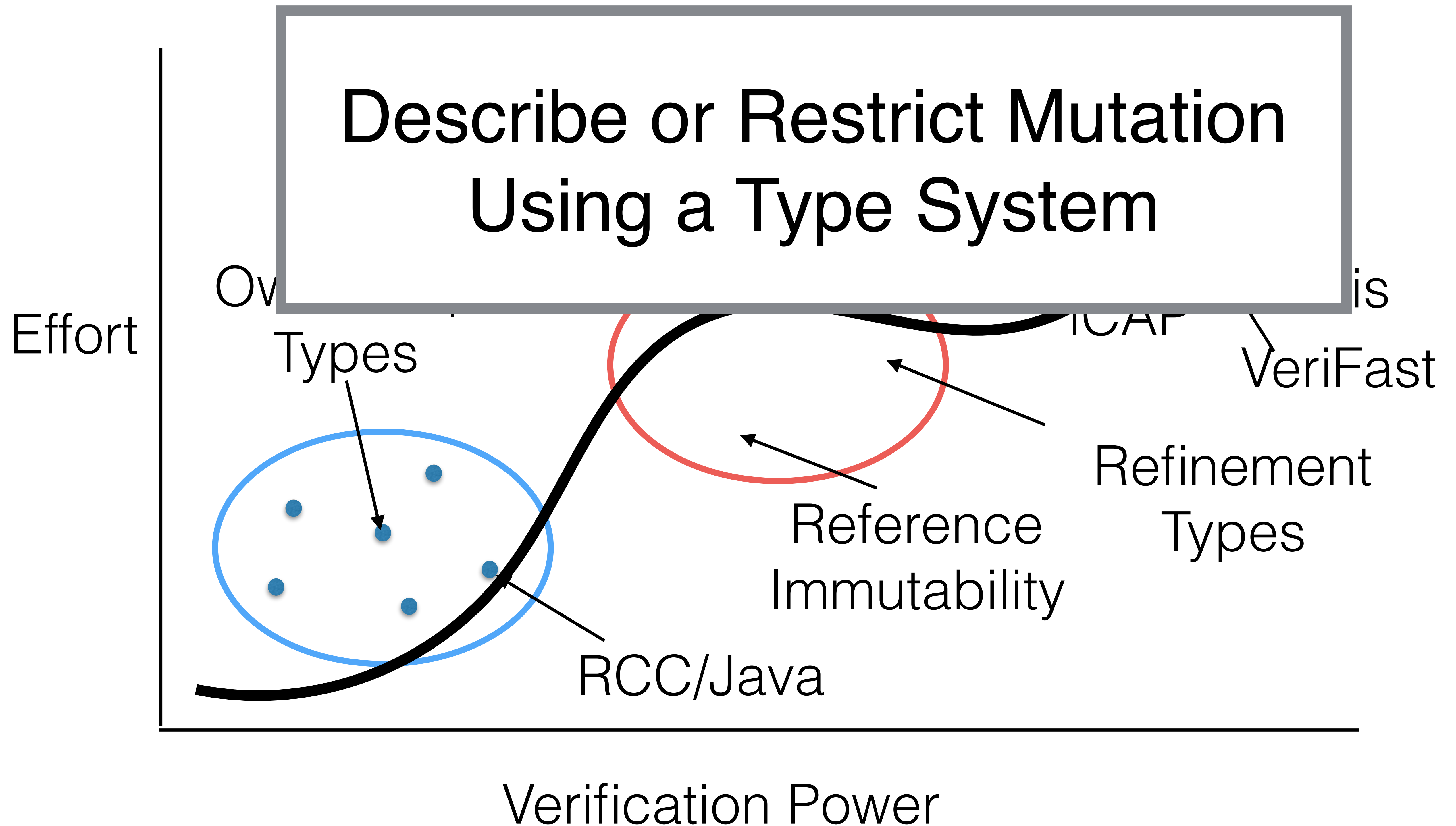
# Verification Effort vs. Power



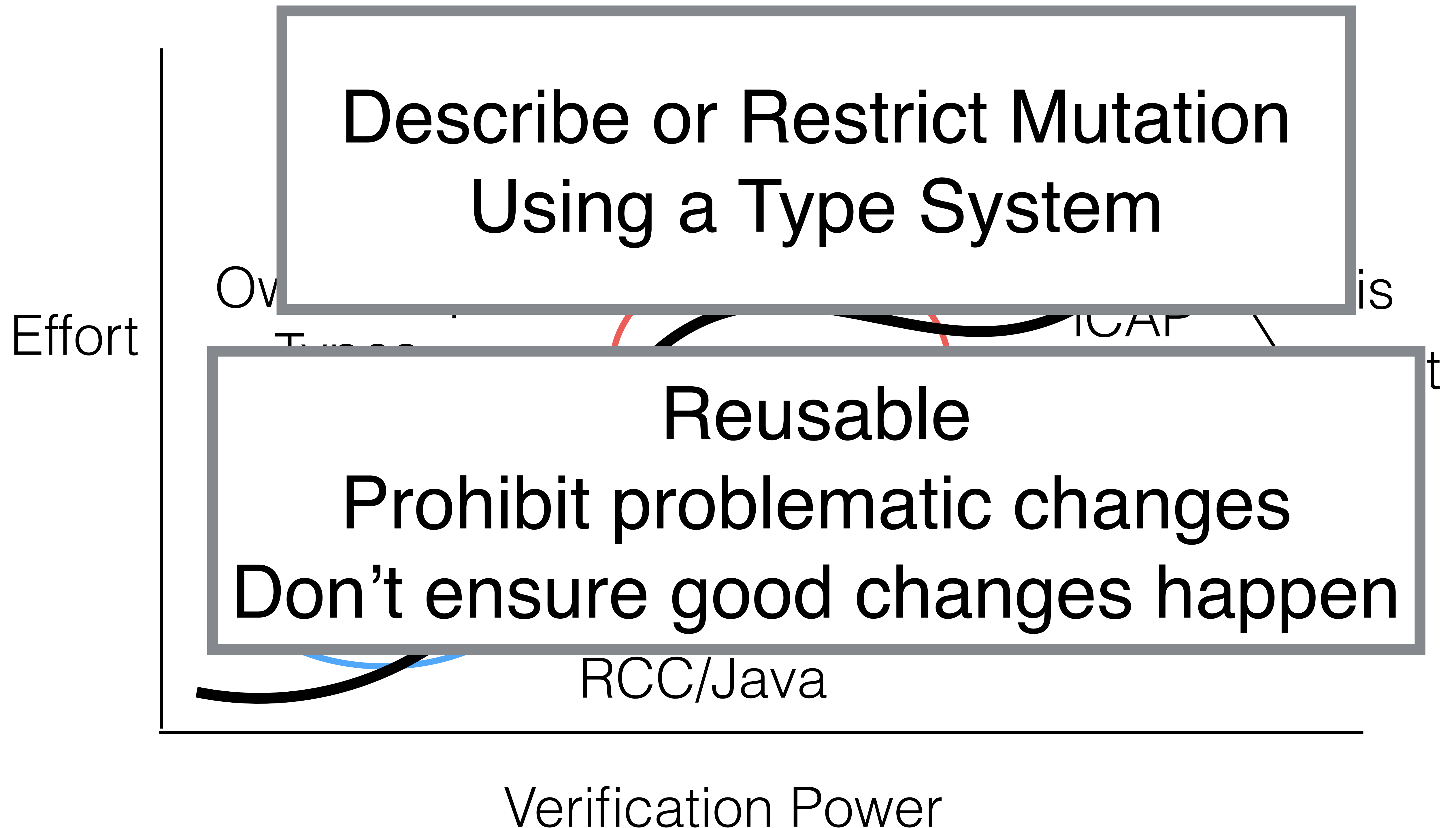
# Verification Effort vs. Power



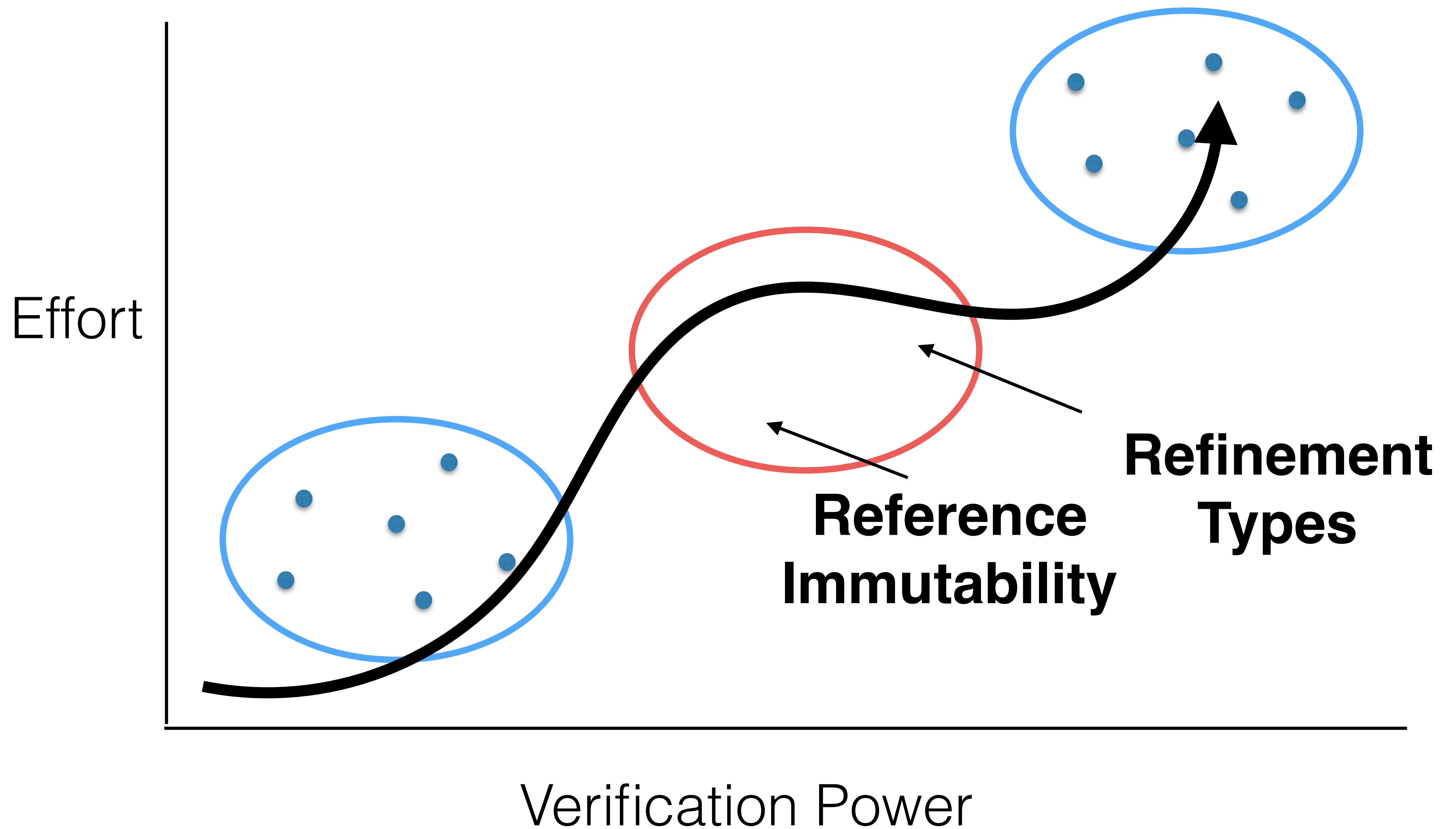
# Verification Effort vs. Power



# Verification Effort vs. Power



# Verification Effort vs. Power



# New Approach to Invariants of Lock-Free Data Structures

# New Approach to Invariants of Lock-Free Data Structures

- **Goals**:

# New Approach to Invariants of Lock-Free Data Structures

- **Goals**:

  1. Verify one- and two-state invariants of lock-free data structures

# New Approach to Invariants of Lock-Free Data Structures

- **Goals**:

  1. Verify one- and two-state invariants of lock-free data structures

     - Includes encoding (asymmetric) protocols for state change

# New Approach to Invariants of Lock-Free Data Structures

- **Goals**:

    1. Verify one- and two-state invariants of lock-free data structures

        - Includes encoding (asymmetric) protocols for state change

    2. *Intermediate* verification burden

# New Approach to Invariants of Lock-Free Data Structures

- **Goals**:

  1. Verify one- and two-state invariants of lock-free data structures

     - Includes encoding (asymmetric) protocols for state change

  2. *Intermediate* verification burden

  3. Compatible with unverified code

# New Approach to Invariants of Lock-Free Data Structures

- **Goals**:
  1. Verify one- and two-state invariants of lock-free data structures
     - Includes encoding (asymmetric) protocols for state change
  2. *Intermediate* verification burden
  3. Compatible with unverified code
- **Approach**: Extend RGRefs (PLDI'13) for safe concurrency

# New Approach to Invariants of Lock-Free Data Structures

- **Goals**:
  1. Verify one- and two-state invariants of lock-free data structures
     - Includes encoding (asymmetric) protocols for state change
  2. *Intermediate* verification burden
  3. Compatible with unverified code
- **Approach**: Extend RGRefs (PLDI'13) for safe concurrency
- **Validation**: Axiomatic Coq DSL *and* Liquid Haskell library

# Rely-Guarantee References

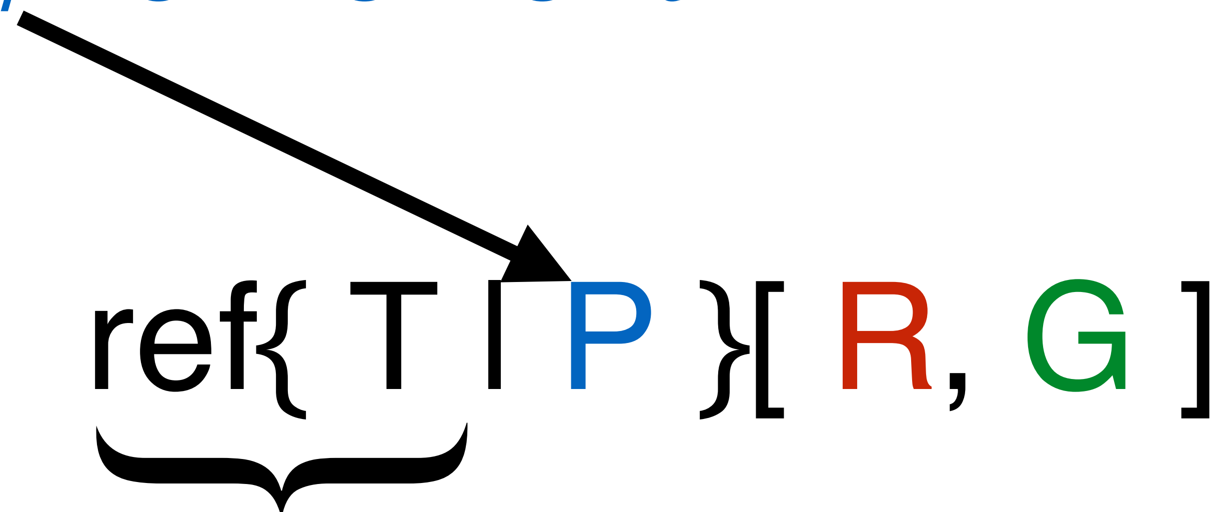
ref{ T | P }[ R, G ]

# Rely-Guarantee References

ref{ T | P }[ R, G ]

Regular ML Reference

# Rely-Guarantee References



# Rely-Guarantee References

Predicate/Refinement

# Rely-Guarantee References

Predicate/Refinement

ref{ T | P }[ R, G ]

Regular ML Reference

Rely (e.g. ==)

Guarantee (e.g., $\leq$)

Two extra requirements:

1. P is *stable* w.r.t. R
2. Aliased references are *compatible*: $x.G \subseteq y.R$, $y.G \subseteq x.R$

# Rely-Guarantee References

Predicate/Refinement

By default, P/R/G range over referent *and reachable heap fragment!*

Regular ML Reference

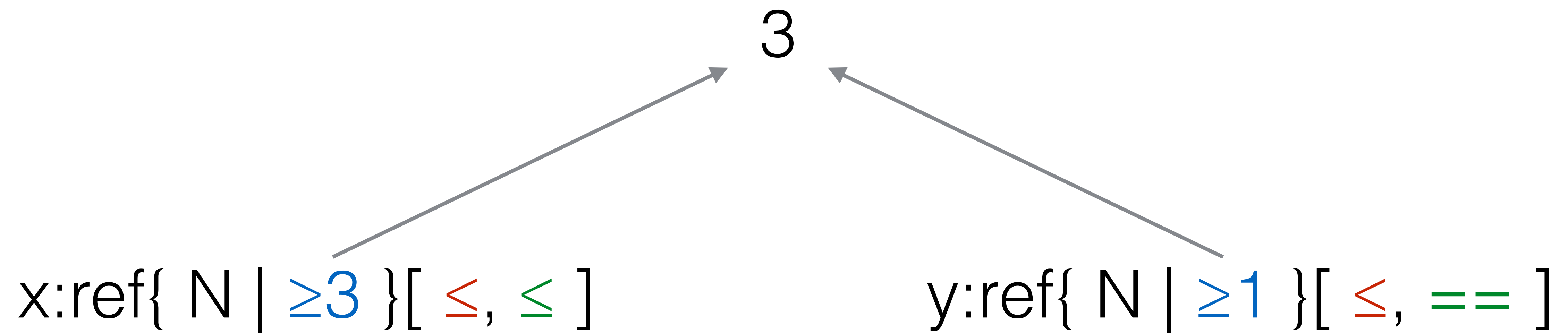
Rely (e.g. ==)

Guarantee (e.g., ≤)

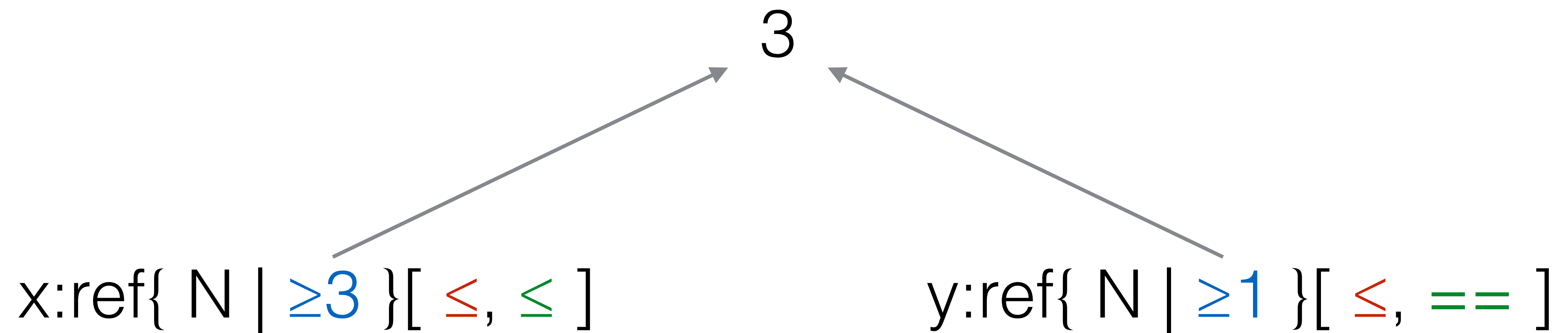
Two extra requirements:

1. P is *stable* w.r.t. R
2. Aliased references are *compatible*: x.G ⊆ y.R, y.G ⊆ x.R

# RGRef Monotonic Counter

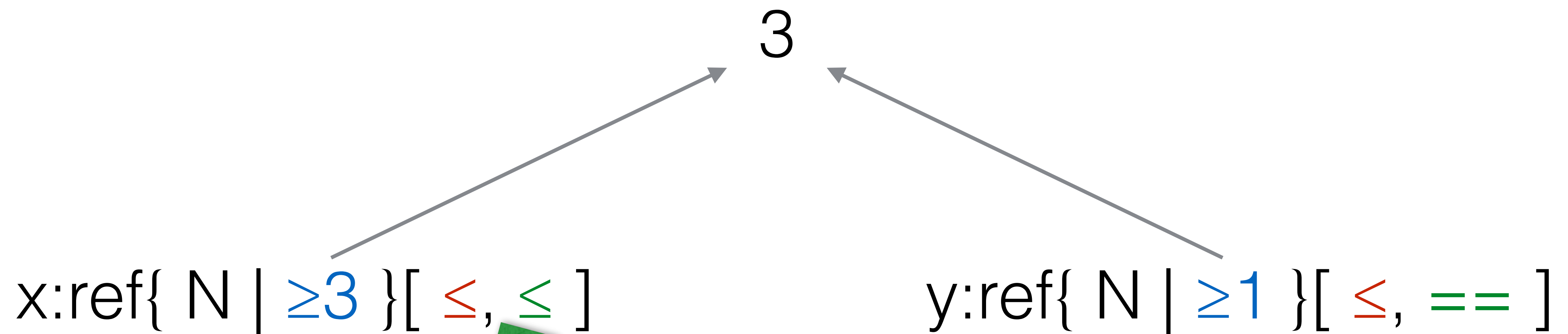


# RGRef Monotonic Counter




x := !x + 1;

# RGRef Monotonic Counter

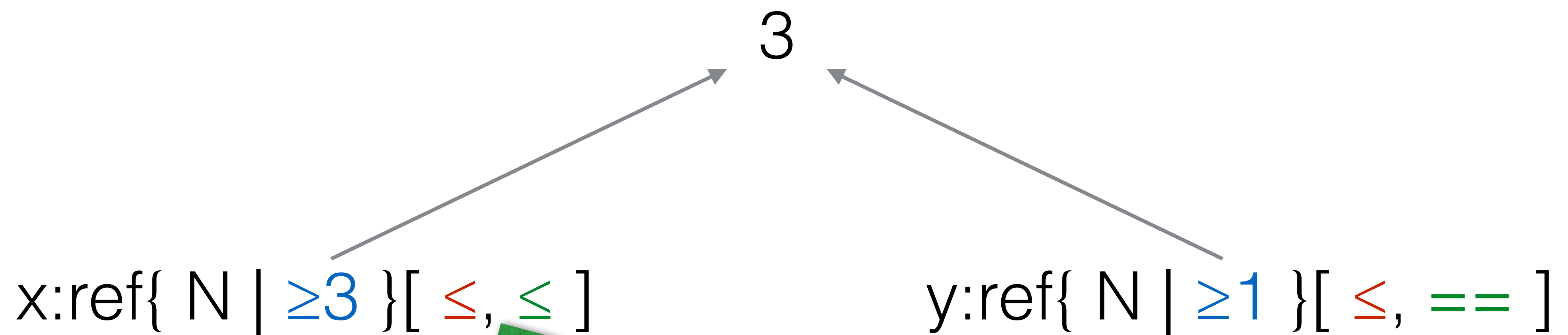



x := !x + 1;

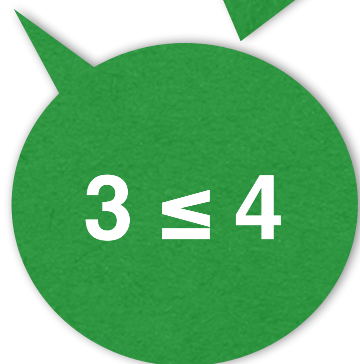
**3 ≤ 4**

Proof: (!x) <= !x + 1

# RGRef Monotonic Counter



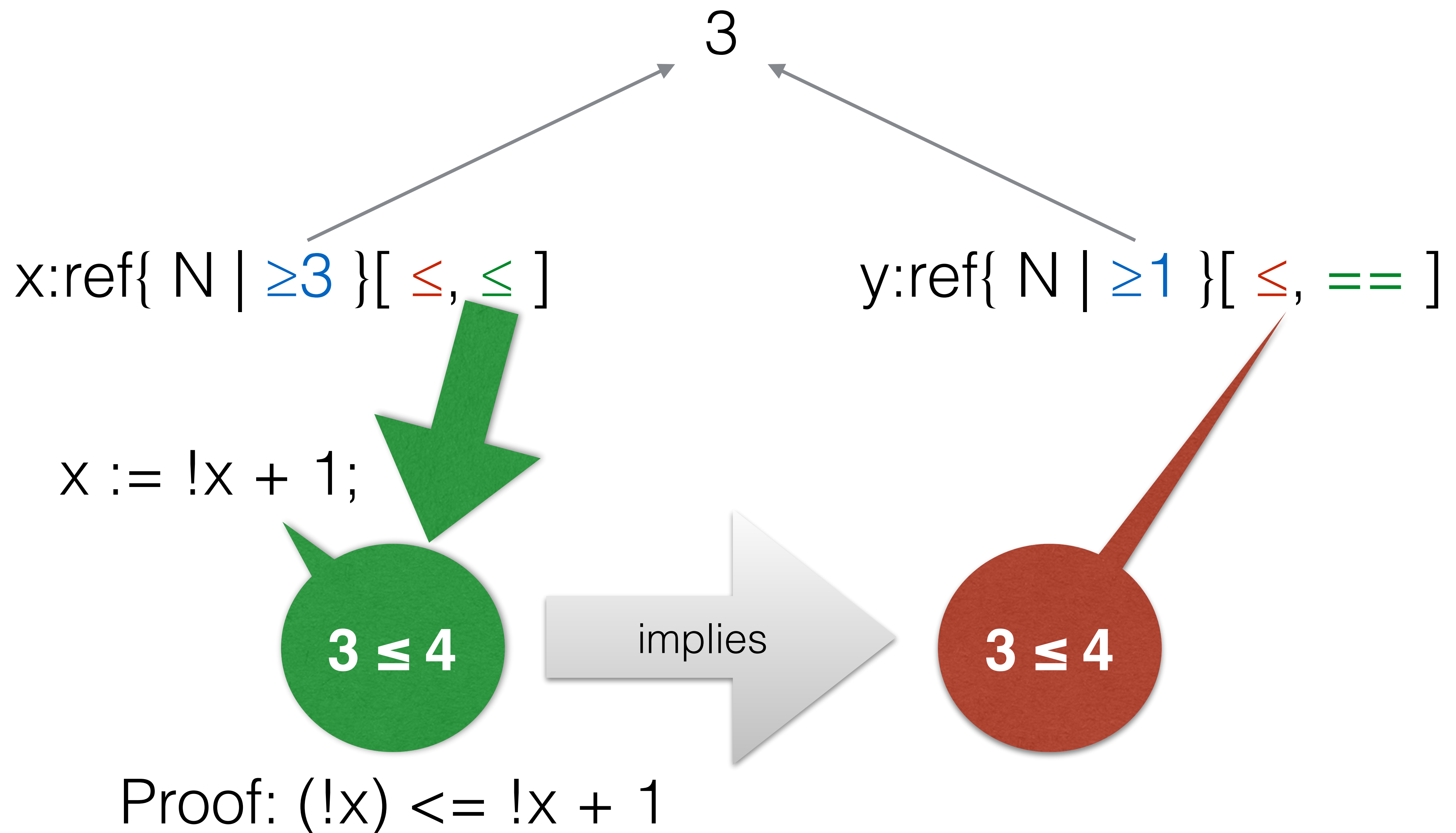

x := !x + 1;



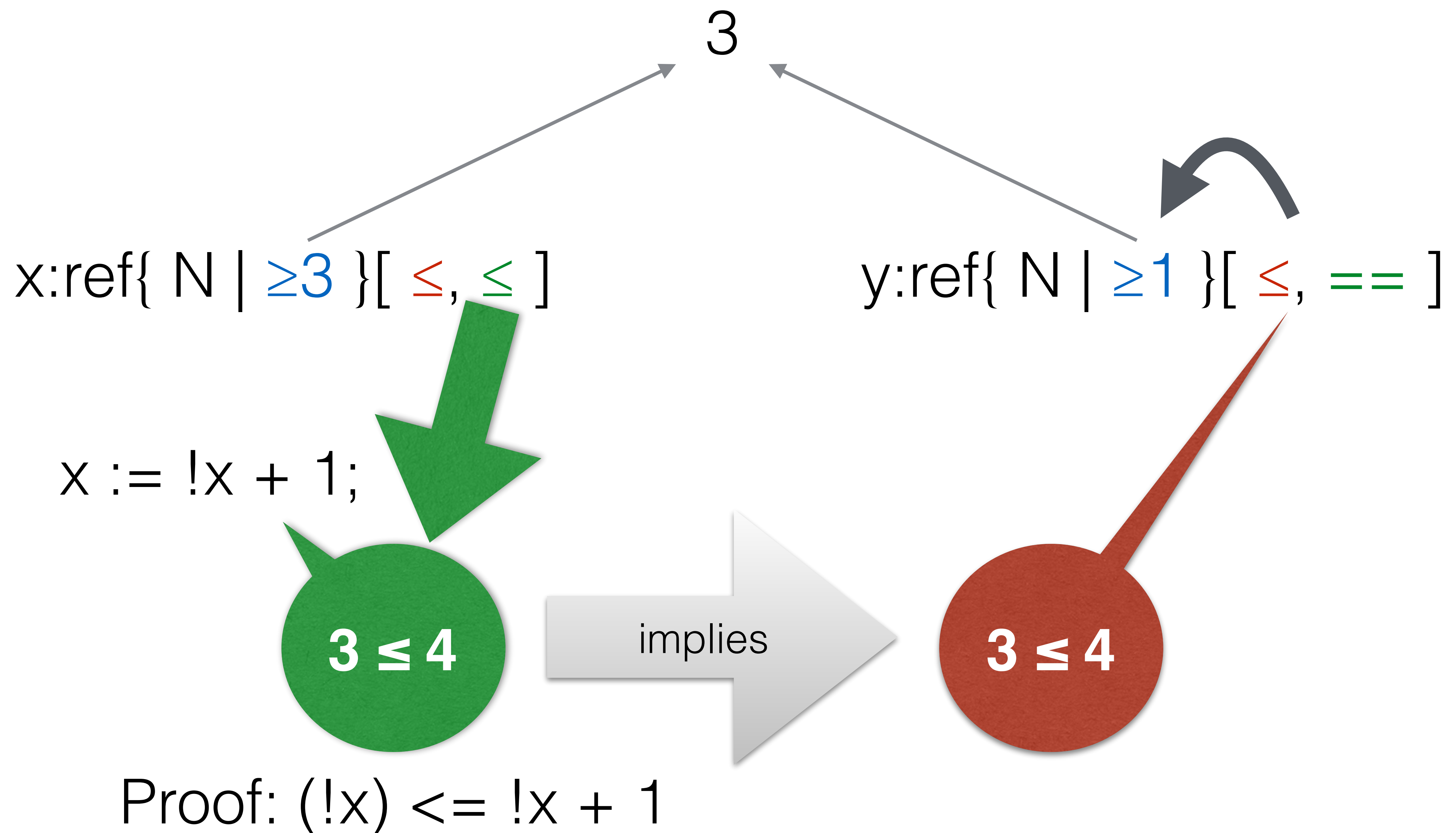

implies

Proof: (!x) <= !x + 1

# RGRef Monotonic Counter



# RGRef Monotonic Counter



# RGRefs are Well-Suited for Fine-Grained Concurrency

# RGRefs are Well-Suited for Fine-Grained Concurrency

- Rely-Guarantee originated in concurrent program verification

# RGRefs are Well-Suited for Fine-Grained Concurrency

- Rely-Guarantee originated in concurrent program verification
- Many structures have natural one- and two-state per-node invariants (e.g., O’Hearn et al.’s Hindsight paper, PODC’10)

# RGRefs are Well-Suited for Fine-Grained Concurrency

- Rely-Guarantee originated in concurrent program verification
- Many structures have natural one- and two-state per-node invariants (e.g., O'Hearn et al.'s Hindsight paper, PODC'10)
- RGRefs "play nice" with unverified code

# RGRefs are Well-Suited for Fine-Grained Concurrency

- Rely-Guarantee originated in concurrent program verification
- Many structures have natural one- and two-state per-node invariants (e.g., O'Hearn et al.'s Hindsight paper, PODC'10)
- RGRefs "play nice" with unverified code
  - Subsume regular ML references: ref{ $\tau$ | True }[ True , True ]

# RGRefs are Well-Suited for Fine-Grained Concurrency

- Rely-Guarantee originated in concurrent program verification
- Many structures have natural one- and two-state per-node invariants (e.g., O'Hearn et al.'s Hindsight paper, PODC'10)
- RGRefs "play nice" with unverified code
  - Subsume regular ML references: ref{ $\tau$ | True }[ True , True ]
  - Often we want to verify "critical" code (e.g., lock-free algorithms) without verifying everything

# RGRefs are Well-Suited for Fine-Grained Concurrency

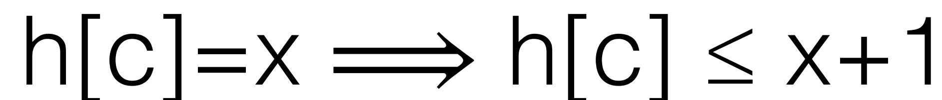
- Rely-Guarantee originated in concurrent program verification
- Many structures have natural one- and two-state per-node invariants (e.g., O'Hearn et al.'s Hindsight paper, PODC'10)
- RGRefs "play nice" with unverified code
  - Subsume regular ML references: ref{ $\tau$ | True }[ True , True ]
  - Often we want to verify "critical" code (e.g., lock-free algorithms) without verifying everything
  - Unverified code can treat RGRefs as opaque type

# Atomic Increment

```
let rec atom_inc (c:monotonic_counter) : IO () =
    x <- !c;
    done <- CAS(c, x, x+1);
    if done then return () else atom_inc c;;
```

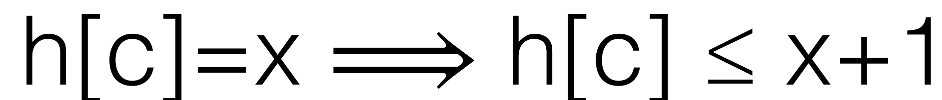
# Atomic Increment

```
let rec atom_inc (c:monotonic_counter) : IO () =
    x <- !c;
    done <- CAS(c, x, x+1);
    if done then return () else atom_inc c;;
```



# Atomic Increment

```
let rec atom_inc (c:monotonic_counter) : IO () =
    x <- !c;
    done <- CAS(c, x, x+1);
    if done then return () else atom_inc c;;
```




Other structures require more extensions

# Refiners

# Refiners

- Some proofs need to observe a heap cell, and *later* rely on a fact the observation established

# Refiners

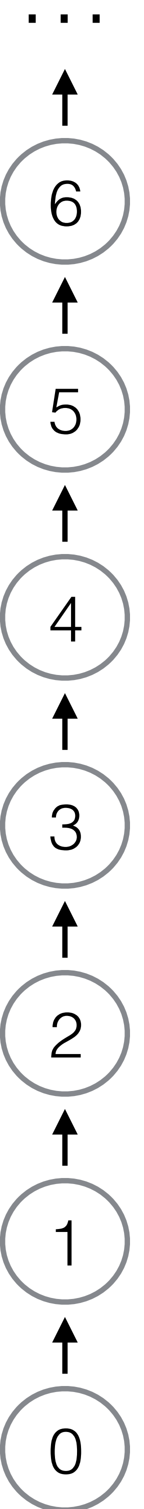
- Some proofs need to observe a heap cell, and *later* rely on a fact the observation established

- Refiners are field access that *relate the value read to new stable refinements*

# Refiners

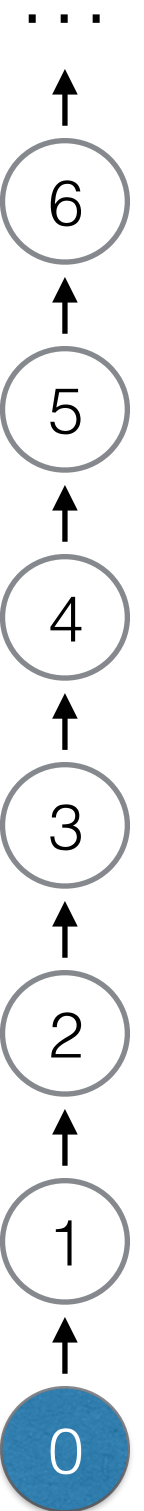
- Some proofs need to observe a heap cell, and *later* rely on a fact the observation established

- Refiners are field access that *relate the value read to new stable refinements*



# Refiners

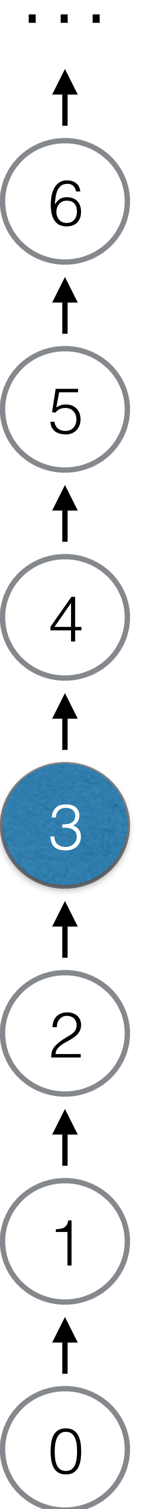
- Some proofs need to observe a heap cell, and *later* rely on a fact the observation established
- Refiners are field access that *relate the value read to new stable refinements*



# Refiners

- Some proofs need to observe a heap cell, and *later* rely on a fact the observation established

- Refiners are field access that *relate the value read to new stable refinements*

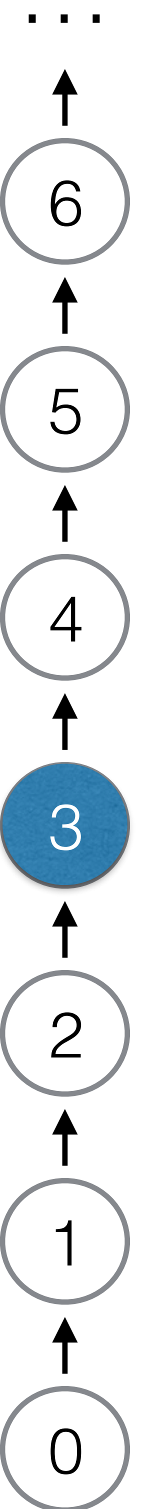


# Refiners

- Some proofs need to observe a heap cell, and *later* rely on a fact the observation established

- Refiners are field access that *relate the value read to new stable refinements*

  - $x:\mathrm{ref}\{\mathbb{N} \mid \geq 0\}[\leq,\leq] \Longrightarrow n:\mathbb{N}, x:\mathrm{ref}\{\mathbb{N} \mid \geq n\}[\leq,\leq]$



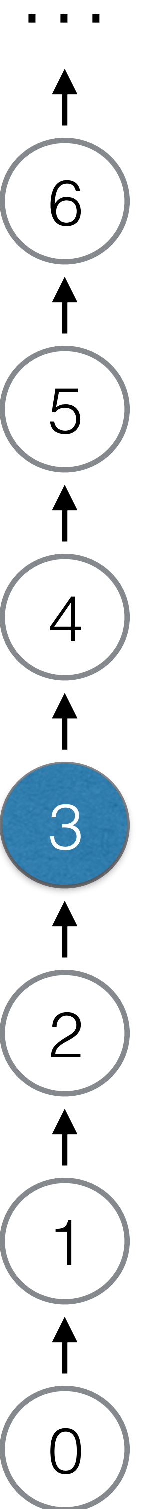
# Refiners

- Some proofs need to observe a heap cell, and *later* rely on a fact the observation established

- Refiners are field access that *relate the value read to new stable refinements*

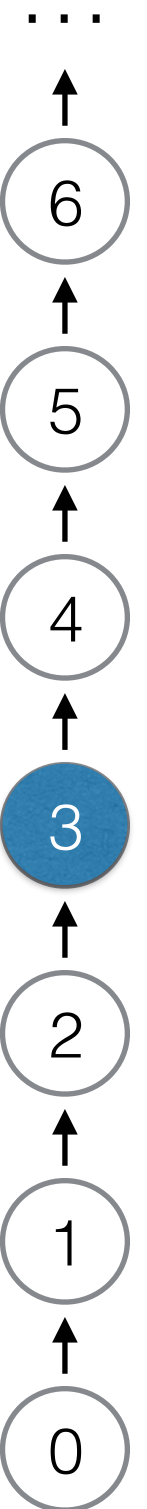
    - x:ref{N|≥0}[≤,≤] ⟹ n:N,x:ref{N|≥n}[≤,≤]

    - observe x as n in (λv,h. n ≤ v)



# Refiners

- Some proofs need to observe a heap cell, and *later* rely on a fact the observation established
- Refiners are field access that *relate the value read to new stable refinements*
  - $x\text{:ref}\{\mathbb{N} \mid \geq 0\}[\leq,\leq] \Longrightarrow n\text{:}\mathbb{N}, x\text{:ref}\{\mathbb{N} \mid \geq n\}[\leq,\leq]$
  - observe x as n in (λv,h. n ≤ v)
- Supports idiom of *accumulating invariants* during data structure traversal



# Example Refiner Usages

# Example Refiner Usages

- Observing new lower bound on counter

# Example Refiner Usages

- Observing new lower bound on counter
- Observe the final state of a protocol

# Example Refiner Usages

- Observing new lower bound on counter
- Observe the final state of a protocol
- Observing fixed field value
  - Used to prove Treiber stack pops exactly 1 node

# Example Refiner Usages

- Observing new lower bound on counter
- Observe the final state of a protocol
- Observing fixed field value
  - Used to prove Treiber stack pops exactly 1 node
- Observe predicates on rank, order or set membership for union find

# Lock-Free Union Find

# Lock-Free Union Find

- Anderson & Woll, STOC'91
    - Ranks & path compression

# Lock-Free Union Find

- Anderson & Woll, STOC'91
    - Ranks & path compression
- We give first mechanized proof of invariants

# Lock-Free Union Find

- Anderson & Woll, STOC'91
  - Ranks & path compression
- We give first mechanized proof of invariants

```
let rec Find {n:nat} (r:ref{uf (S n)|φ}[δ,δ]) (f:Fin.t (S n))
    : IO (Fin.t (S n)) :=
  observe-field r → f as c in (λ x h, sameset f ((h[c]).parent) x h /\
                               rankSorted (h[c]) (h[x<|((h[c]).parent)|>]));
  observe-field c → parent as p in (λ x h, x.parent = p);
  if (p == f) then (return f) (* found root --- f's parent = f *)
  else (observe-field c → rank as rnk in (λ x h, x.rank = rnk);
        observe-field r → p as p_ptr in
     (*A*) (λ x h, sameset p ((h[p_ptr]).parent) x h
     (*B*) /\ (rankSorted (h[p_ptr]) (h[x<|((h[p_ptr]).parent)|>]))
     (*C*) /\ (rankSorted (h[c]) (h[p_ptr]))
     (*D*) /\ ((h[p_ptr]).parent ≠ p ->
                nonroot_rank p ((h[p_ptr]).rank) x h)
     (*E*) /\ ((h[p_ptr]).parent ≠ p ->
                ( rankSorted_strict (h[p_ptr]) (h[x<|((h[p_ptr]).parent)|>])
                 \/ fin_lt p ((h[p_ptr]).parent))));
        observe-field p_ptr → parent as gp in (λ x h, x.parent = gp);
        gp_cell <- Alloc! (mkCell _ rnk gp ) ;
        _ <- fCAS( r → f, c, convert gp_cell);
        Find p);;
```

# Lock-Free Union Find

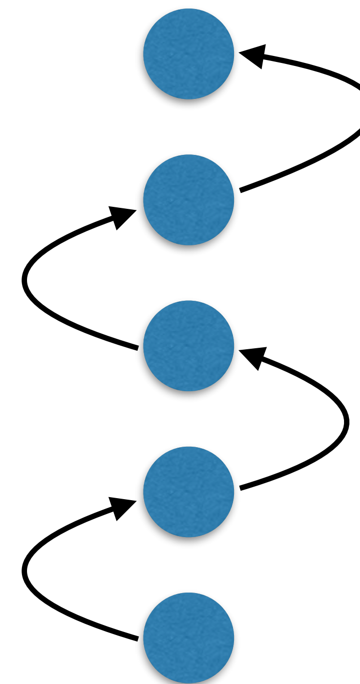
- Anderson & Woll, STOC'91
  - Ranks & path compression
- We give first mechanized proof of invariants
- Our proof reveals subtleties:
  - Path "compression" writes sometimes *extend* the chains momentarily!

```
let rec Find {n:nat} (r:ref{uf (S n)|φ}[δ,δ]) (f:Fin.t (S n))
   : IO (Fin.t (S n)) :=
  observe-field r → f as c in (λ x h, sameset f ((h[c]).parent) x h /\
                             rankSorted (h[c]) (h[x<|((h[c]).parent)|>]));
  observe-field c → parent as p in (λ x h, x.parent = p);
  if (p == f) then (return f) (* found root --- f's parent = f *)
  else (observe-field c → rank as rnk in (λ x h, x.rank = rnk);
        observe-field r → p as p_ptr in
     (*A*) (λ x h, sameset p ((h[p_ptr]).parent) x h
     (*B*) /\ (rankSorted (h[p_ptr]) (h[x<|((h[p_ptr]).parent)|>]))
     (*C*) /\ (rankSorted (h[c]) (h[p_ptr]))
     (*D*) /\ ((h[p_ptr]).parent ≠ p ->
                nonroot_rank p ((h[p_ptr]).rank) x h)
     (*E*) /\ ((h[p_ptr]).parent ≠ p ->
                ( rankSorted_strict (h[p_ptr]) (h[x<|((h[p_ptr]).parent)|>])
                 \/ fin_lt p ((h[p_ptr]).parent))));
        observe-field p_ptr → parent as gp in (λ x h, x.parent = gp);
        gp_cell <- Alloc! (mkCell _ rnk gp ) ;
        _ <- fCAS( r → f, c, convert gp_cell);
        Find p);;
```

# Lock-Free Union Find

- Anderson & Woll, STOC'91
  - Ranks & path compression
- We give first mechanized proof of invariants
- Our proof reveals subtleties:
  - Path "compression" writes sometimes *extend* the chains momentarily!

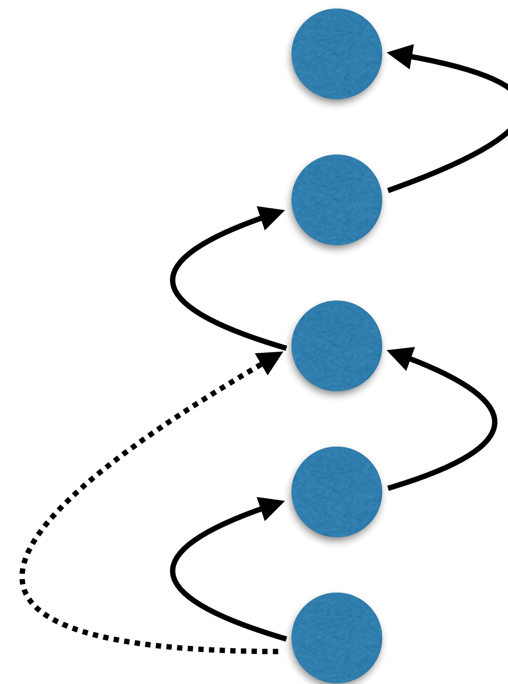
```
let rec Find {n:nat} (r:ref{uf (S n)|φ}[δ,δ]) (f:Fin.t (S n))
    : IO (Fin.t (S n)) :=
  observe-field r → f as c in (λ x h, sameset f ((h[c]).parent) x h /\
                                rankSorted (h[c]) (h[x<|((h[c]).parent)|>]));
  observe-field c → parent as p in (λ x h, x.parent = p);
  if (p == f) then (return f) (* found root --- f's parent = f *)
  else (observe-field c → rank as rnk in (λ x h, x.rank = rnk);
        observe-field r → p as p_ptr in
     (*A*) (λ x h, sameset p ((h[p_ptr]).parent) x h
     (*B*) /\ (rankSorted (h[p_ptr]) (h[x<|((h[p_ptr]).parent)|>]))
     (*C*) /\ (rankSorted (h[c]) (h[p_ptr]))
     (*D*) /\ ((h[p_ptr]).parent ≠ p ->
                nonroot_rank p ((h[p_ptr]).rank) x h)
     (*E*) /\ ((h[p_ptr]).parent ≠ p ->
                ( rankSorted_strict (h[p_ptr]) (h[x<|((h[p_ptr]).parent)|>])
                 \/ fin_lt p ((h[p_ptr]).parent))));
        observe-field p_ptr → parent as gp in (λ x h, x.parent = gp);
        gp_cell <- Alloc! (mkCell _ rnk gp ) ;
        _ <- fCAS( r → f, c, convert gp_cell);
        Find p);;
```



# Lock-Free Union Find

- Anderson & Woll, STOC'91
  - Ranks & path compression
- We give first mechanized proof of invariants
- Our proof reveals subtleties:
  - Path "compression" writes sometimes *extend* the chains momentarily!

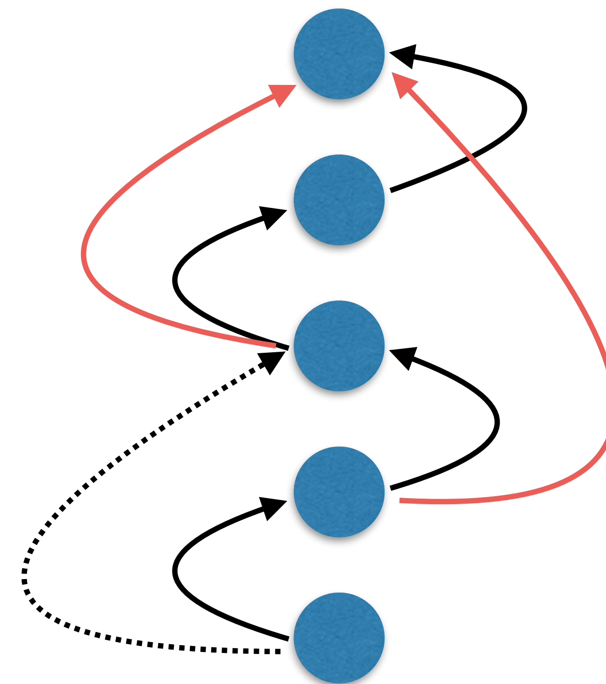
```
let rec Find {n:nat} (r:ref{uf (S n)|φ}[δ,δ]) (f:Fin.t (S n))
    : IO (Fin.t (S n)) :=
  observe-field r → f as c in (λ x h, sameset f ((h[c]).parent) x h /\
                              rankSorted (h[c]) (h[x<|((h[c]).parent)|>]));
  observe-field c → parent as p in (λ x h, x.parent = p);
  if (p == f) then (return f) (* found root --- f's parent = f *)
  else (observe-field c → rank as rnk in (λ x h, x.rank = rnk);
        observe-field r → p as p_ptr in
     (*A*) (λ x h, sameset p ((h[p_ptr]).parent) x h
     (*B*) /\ (rankSorted (h[p_ptr]) (h[x<|((h[p_ptr]).parent)|>]))
     (*C*) /\ (rankSorted (h[c]) (h[p_ptr]))
     (*D*) /\ ((h[p_ptr]).parent ≠ p ->
                nonroot_rank p ((h[p_ptr]).rank) x h)
     (*E*) /\ ((h[p_ptr]).parent ≠ p ->
                ( rankSorted_strict (h[p_ptr]) (h[x<|((h[p_ptr]).parent)|>])
                 \/ fin_lt p ((h[p_ptr]).parent))));
        observe-field p_ptr → parent as gp in (λ x h, x.parent = gp);
        gp_cell <- Alloc! (mkCell _ rnk gp ) ;
        _ <- fCAS( r → f, c, convert gp_cell);
        Find p);;
```



# Lock-Free Union Find

- Anderson & Woll, STOC'91
  - Ranks & path compression
- We give first mechanized proof of invariants
- Our proof reveals subtleties:
  - Path "compression" writes sometimes *extend* the chains momentarily!

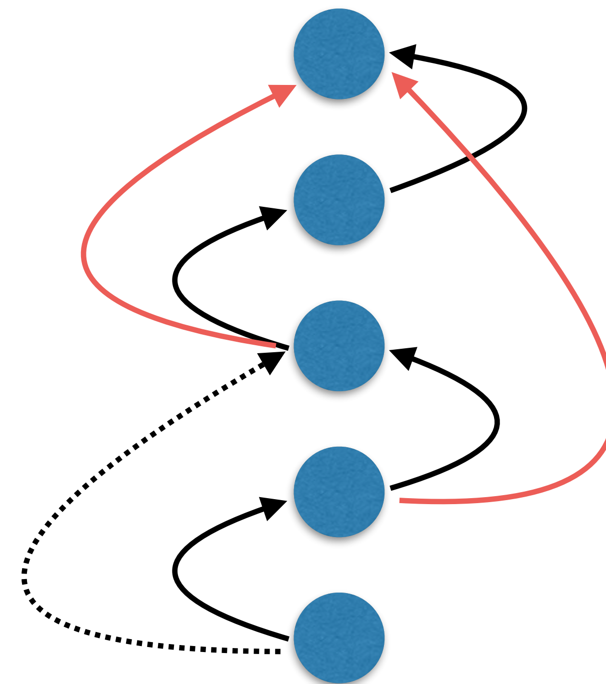
```
let rec Find {n:nat} (r:ref{uf (S n)|φ}[δ,δ]) (f:Fin.t (S n))
    : IO (Fin.t (S n)) :=
  observe-field r → f as c in (λ x h, sameset f ((h[c]).parent) x h /\
                                rankSorted (h[c]) (h[x<|((h[c]).parent)|>]));
  observe-field c → parent as p in (λ x h, x.parent = p);
  if (p == f) then (return f) (* found root --- f's parent = f *)
  else (observe-field c → rank as rnk in (λ x h, x.rank = rnk);
       observe-field r → p as p_ptr in
     (*A*) (λ x h, sameset p ((h[p_ptr]).parent) x h
     (*B*) /\ (rankSorted (h[p_ptr]) (h[x<|((h[p_ptr]).parent)|>]))
     (*C*) /\ (rankSorted (h[c]) (h[p_ptr]))
     (*D*) /\ ((h[p_ptr]).parent ≠ p ->
                nonroot_rank p ((h[p_ptr]).rank) x h)
     (*E*) /\ ((h[p_ptr]).parent ≠ p ->
                ( rankSorted_strict (h[p_ptr]) (h[x<|((h[p_ptr]).parent)|>])
                 \/ fin_lt p ((h[p_ptr]).parent))));
       observe-field p_ptr → parent as gp in (λ x h, x.parent = gp);
       gp_cell <- Alloc! (mkCell _ rnk gp ) ;
       _ <- fCAS( r → f, c, convert gp_cell);
       Find p);;
```



# Lock-Free Union Find

- Anderson & Woll, STOC'91
  - Ranks & path compression
- We give first mechanized proof of invariants
- Our proof reveals subtleties:
  - Path "compression" writes sometimes *extend* the chains momentarily!
- Good example for your next verification paper…

```
let rec Find {n:nat} (r:ref{uf (S n)|ϕ}[δ,δ]) (f:Fin.t (S n))
    : IO (Fin.t (S n)) :=
  observe-field r → f as c in (λ x h, sameset f ((h[c]).parent) x h /\
                              rankSorted (h[c]) (h[x<|((h[c]).parent)|>]));
  observe-field c → parent as p in (λ x h, x.parent = p);
  if (p == f) then (return f) (* found root --- f's parent = f *)
  else (observe-field c → rank as rnk in (λ x h, x.rank = rnk);
        observe-field r → p as p_ptr in
     (*A*) (λ x h, sameset p ((h[p_ptr]).parent) x h
     (*B*) /\ (rankSorted (h[p_ptr]) (h[x<|((h[p_ptr]).parent)|>]))
     (*C*) /\ (rankSorted (h[c]) (h[p_ptr]))
     (*D*) /\ ((h[p_ptr]).parent ≠ p ->
                nonroot_rank p ((h[p_ptr]).rank) x h)
     (*E*) /\ ((h[p_ptr]).parent ≠ p ->
                ( rankSorted_strict (h[p_ptr]) (h[x<|((h[p_ptr]).parent)|>])
                 \/ fin_lt p ((h[p_ptr]).parent))));
        observe-field p_ptr → parent as gp in (λ x h, x.parent = gp);
        gp_cell <- Alloc! (mkCell _ rnk gp ) ;
        _ <- fCAS( r → f, c, convert gp_cell);
        Find p);;
```



# Automating RGRefs

# Automating RGRefs

- Because RGRef P/R/G range over referent *and heaps*, they lead to complex side-conditions

# Automating RGRefs

- Because RGRef P/R/G range over referent *and heaps*, they lead to complex side-conditions
  - Annoying to specify, hard to automate or infer

# Automating RGRefs

- Because RGRef P/R/G range over referent *and heaps*, they lead to complex side-conditions
  - Annoying to specify, hard to automate or infer
- A slight simplification: Drop heaps from predicates and relations!

# Automating RGRefs

- Because RGRef P/R/G range over referent *and heaps*, they lead to complex side-conditions
  - Annoying to specify, hard to automate or infer
- A slight simplification: Drop heaps from predicates and relations!
- The simplification doesn't cost much useful expressivity

# Automating RGRefs

- Because RGRef P/R/G range over referent *and heaps*, they lead to complex side-conditions
  - Annoying to specify, hard to automate or infer
- A slight simplification: Drop heaps from predicates and relations!
- The simplification doesn't cost much useful expressivity
  - Deep heap access rarely used

# Automating RGRefs

- Because RGRef P/R/G range over referent *and heaps*, they lead to complex side-conditions
  - Annoying to specify, hard to automate or infer
- A slight simplification: Drop heaps from predicates and relations!
- The simplification doesn't cost much useful expressivity
  - Deep heap access rarely used
- Heapless RGRefs lead to a concise Liquid Haskell Library

# Automating RGRefs

- Because RGRef P/R/G range over referent *and heaps*, they lead to complex side-conditions
    - Annoying to specify, hard to automate or infer
- A slight simplification: Drop heaps from predicates and relations!
- The simplification doesn't cost much useful expressivity
    - Deep heap access rarely used
- Heapless RGRefs lead to a concise Liquid Haskell Library
    - Other Liquid Haskell mechanisms recover some heap access

# Examples Considered

- Atomic Counter (Coq DSL + Liquid Haskell)
- Treiber Stack (Coq)
- Michael-Scott Queue (no tail, Coq)
- Lock-free linked list (Liquid Haskell)
- Lock-free list-based Set (Liquid Haskell)
- Lock-free Union Find (Coq)

# More in the Paper

# More in the Paper

- Detailed examples

# More in the Paper

- Detailed examples
  - Very detailed discussion of verifying path compression

# More in the Paper

- Detailed examples
  - Very detailed discussion of verifying path compression
  - Discussion of idioms for using concurrent RGRefs

# More in the Paper

- Detailed examples
  - Very detailed discussion of verifying path compression
  - Discussion of idioms for using concurrent RGRefs
- Formal type system

# More in the Paper

- Detailed examples
  - Very detailed discussion of verifying path compression
  - Discussion of idioms for using concurrent RGRefs
- Formal type system
- Views Framework-based soundness proof
  (see Dinsdale-Young et al., POPL'13)

# More in the Paper

- Detailed examples
  - Very detailed discussion of verifying path compression
  - Discussion of idioms for using concurrent RGRefs
- Formal type system
- Views Framework-based soundness proof
  (see Dinsdale-Young et al., POPL'13)
- Qualitative comparison of Coq DSL vs. Liquid RGRefs

# More in the Paper

- Detailed examples
  - Very detailed discussion of verifying path compression
  - Discussion of idioms for using concurrent RGRefs
- Formal type system
- Views Framework-based soundness proof
  (see Dinsdale-Young et al., POPL'13)
- Qualitative comparison of Coq DSL vs. Liquid RGRefs
- Proof burden comparison to FCSL and VeriFast

# Thanks!

- And thanks to the Liquid Haskell team for lots of bug fixes, clarifications, and general tool support!
- VM & Source: http://csgordon.github.io/rgref