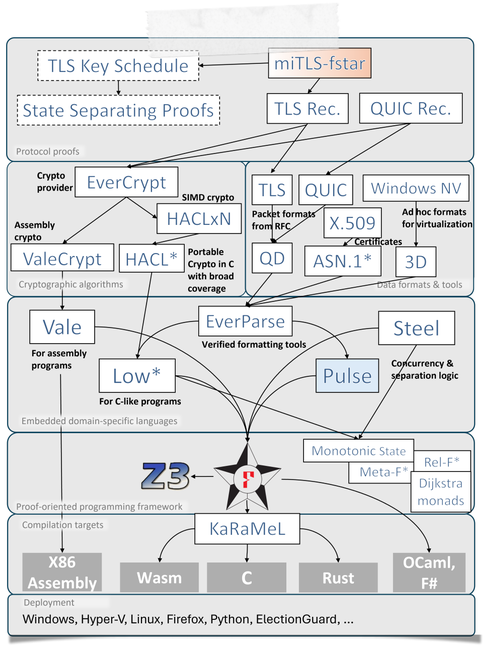
CStar: Unifying Programming and Verification in C

Di Wang
Programming Languages Lab, Peking University

2025 / 11 / 21

Towards Safer Systems Software

formal verification to the rescue?

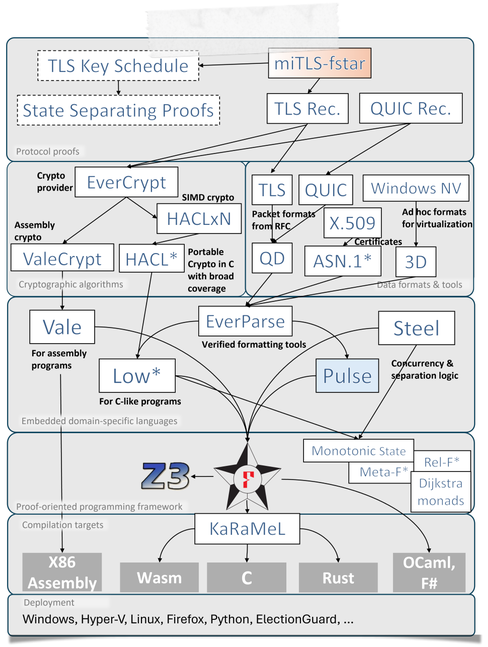


Towards Safer Systems Software

formal verification to the rescue?

seL4
Security. Performance. Proof.

CAKEML
A Verified Implementation of ML



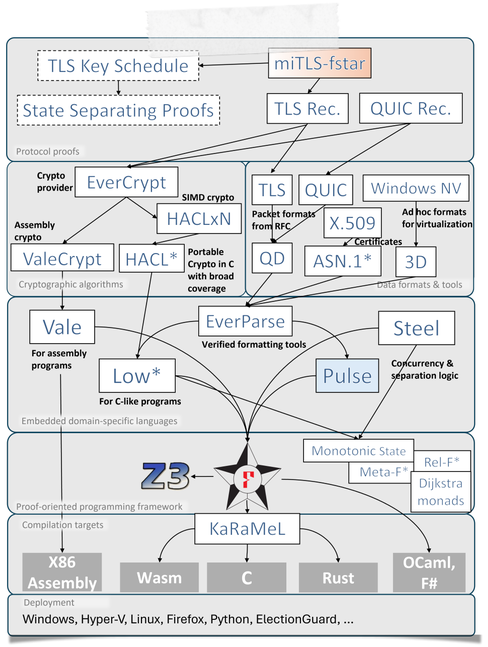
CertiKOS

COMPCERT

verified kernels

Towards Safer Systems Software

formal verification to the rescue?

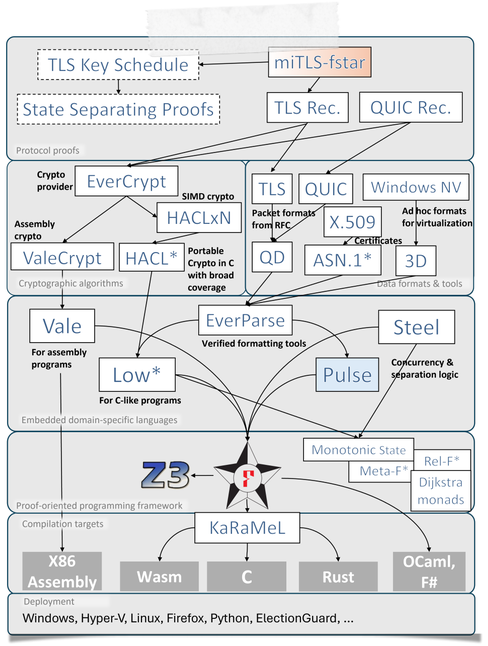


verified kernels

verified compilers

Towards Safer Systems Software

formal verification to the rescue?



verified kernels

verified compilers

verified protocols

Towards Safer Systems Software

formal verification to the rescue?

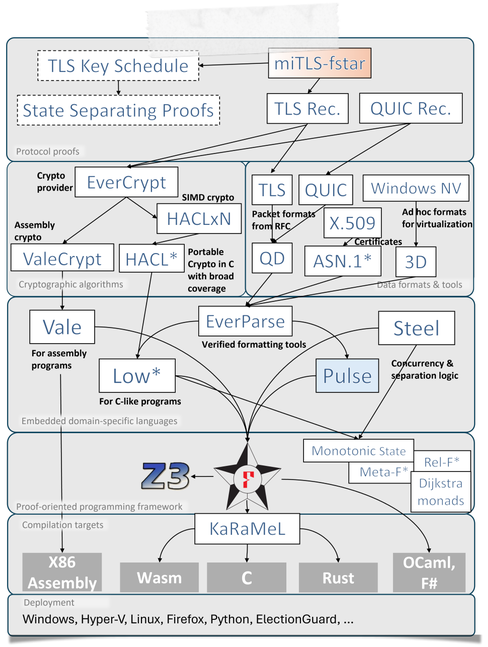
Isabelle/HOL
HOL4

seL4
Security. Performance. Proof.

CAKEML
A Verified Implementation of ML

CertiKOS

COMPCERT



verified kernels

verified compilers

verified protocols

Towards Safer Systems Software

formal verification to the rescue?

Isabelle/HOL
HOL4

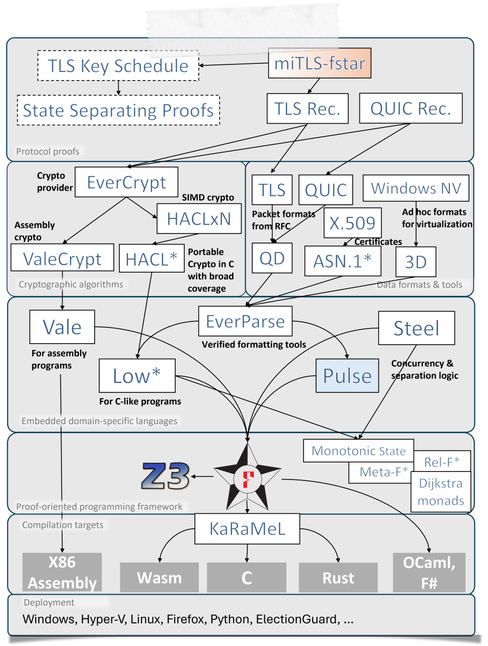
Rocq
(formerly Coq)

seL4
Security. Performance. Proof.

CAKEML
A Verified Implementation of ML

CertiKOS

COMPCERT



verified kernels

verified compilers

verified protocols

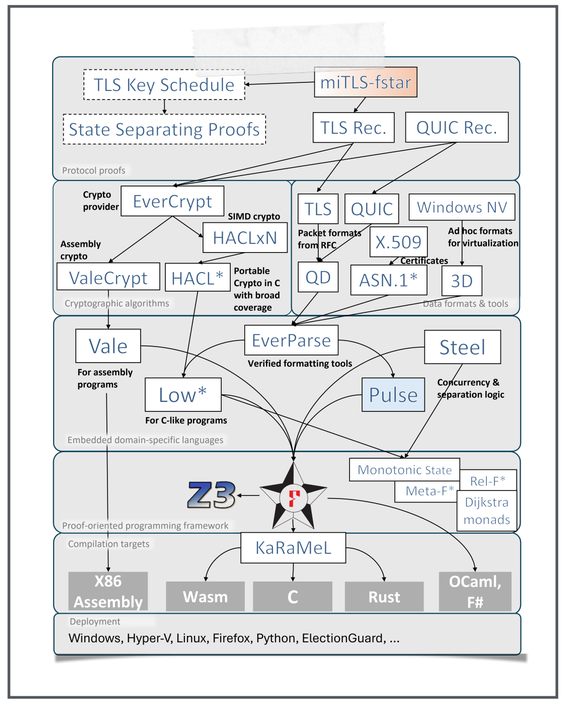
Towards Safer Systems Software

formal verification to the rescue?

Isabelle/HOL
HOL4

Rocq
(formerly Coq)

Dafny
F*



verified kernels

verified compilers

verified protocols

Towards Safer Systems Software

formal verification to the rescue?

Isabelle/HOL
HOL4

Ro
(formerly Coq)

fny
F*

"theorem prover" sounds scary

verified kernels

verified compilers

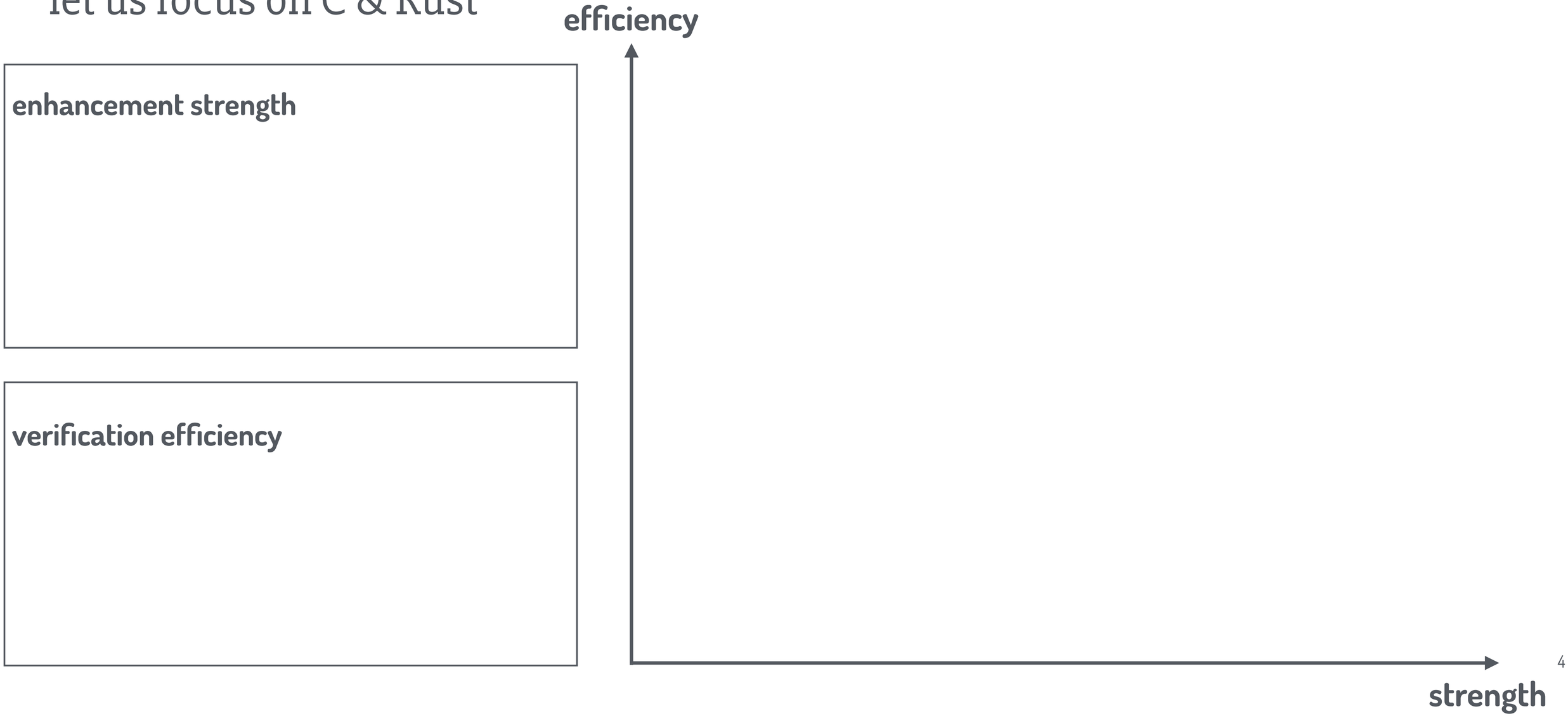
verified protocols

Language Enhancement?

let us focus on C & Rust

Language Enhancement?

let us focus on C & Rust



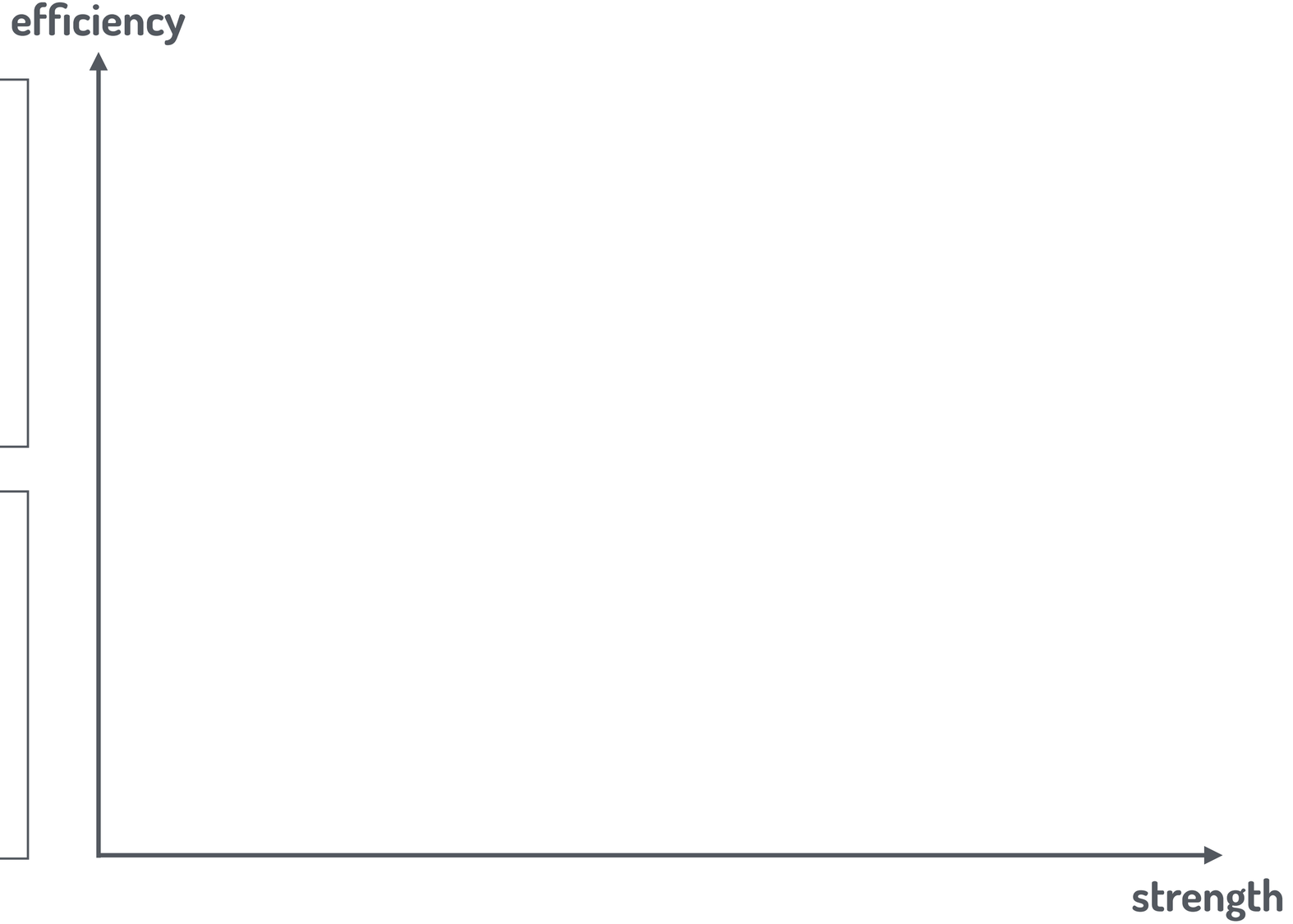
Language Enhancement?

let us focus on C & Rust

enhancement strength

- verified properties: memory / thread safety, functional correctness, security, ...

verification efficiency



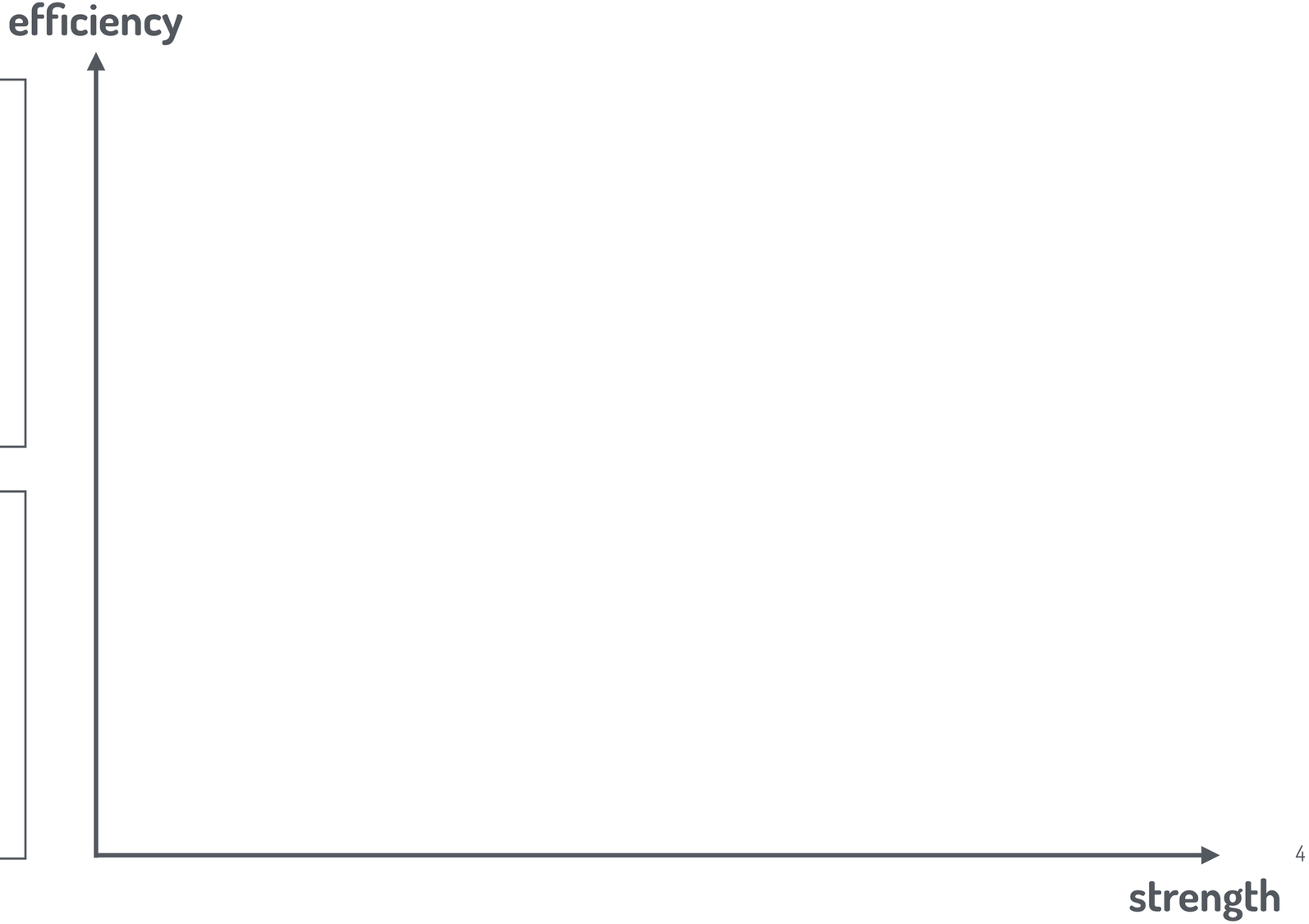
Language Enhancement?

let us focus on C & Rust

enhancement strength

- verified properties: memory / thread safety, functional correctness, security, ...
- trustworthiness: non-foundational / foundational

verification efficiency



Language Enhancement?

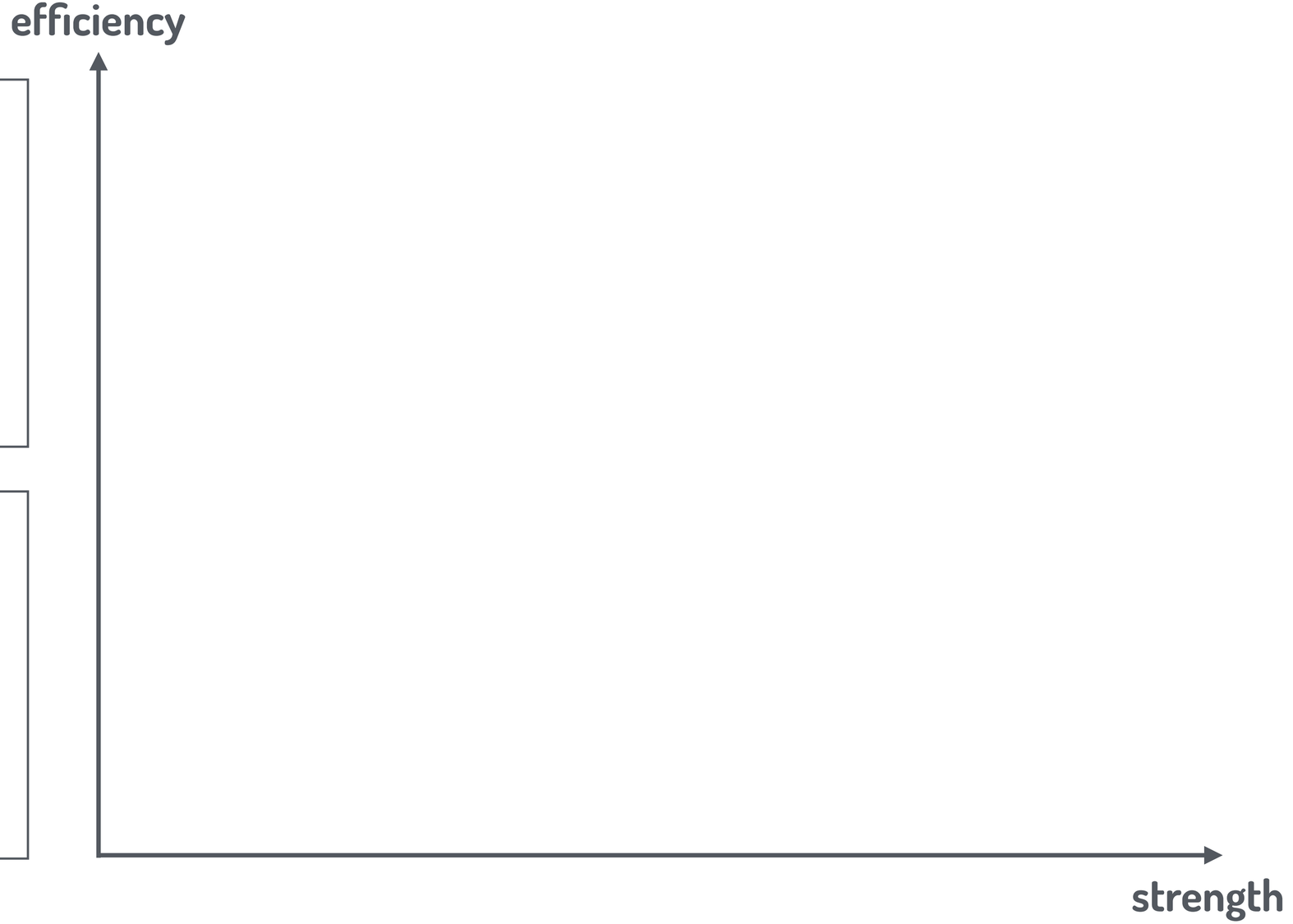
let us focus on C & Rust

enhancement strength

- verified properties: memory / thread safety, functional correctness, security, ...
- trustworthiness: non-foundational / foundational

verification efficiency

- learning curve



Language Enhancement?

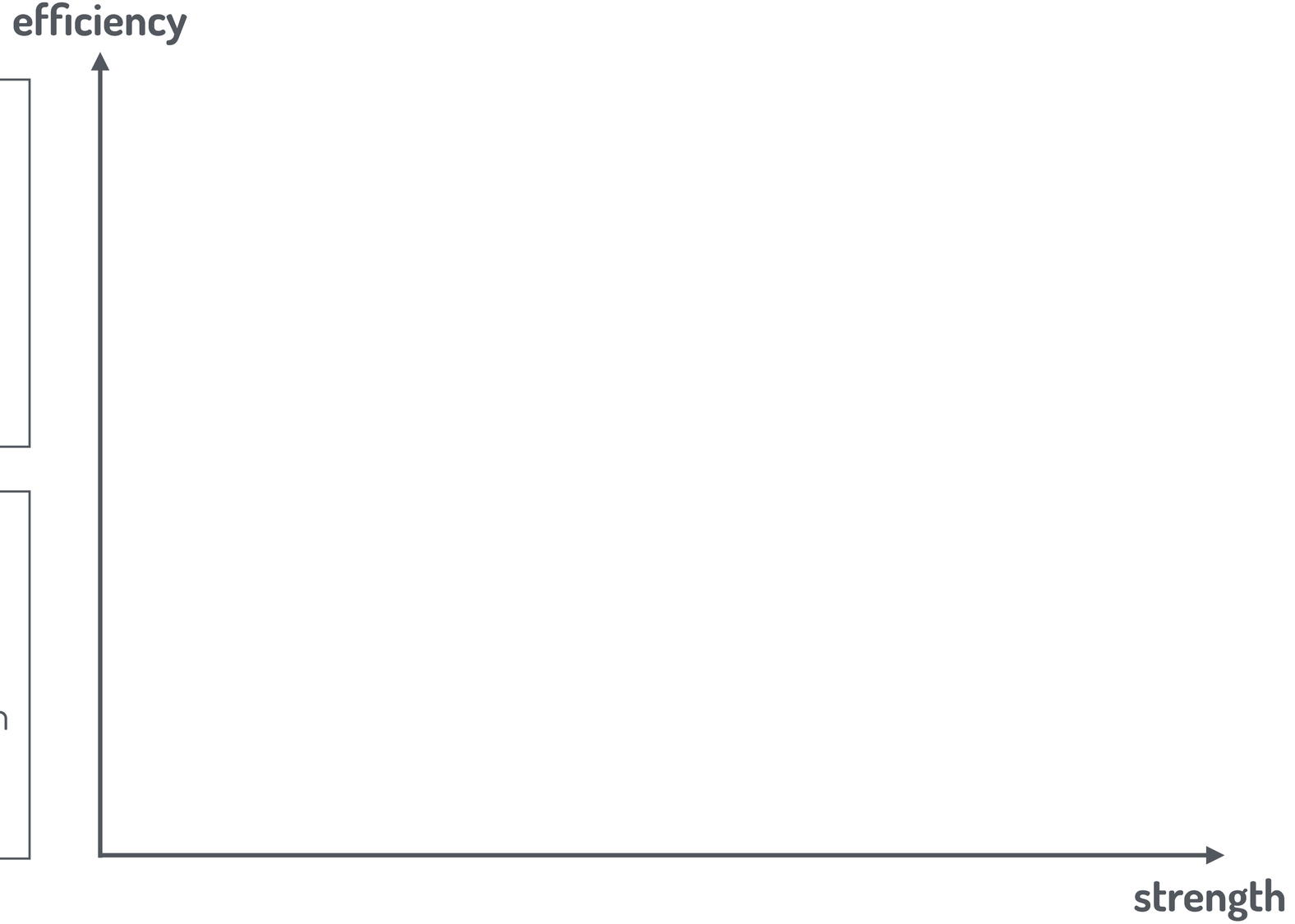
let us focus on C & Rust

enhancement strength

- verified properties: memory / thread safety, functional correctness, security, ...
- trustworthiness: non-foundational / foundational

verification efficiency

- learning curve
- automation, modularity, reusability, evolution



Language Enhancement?

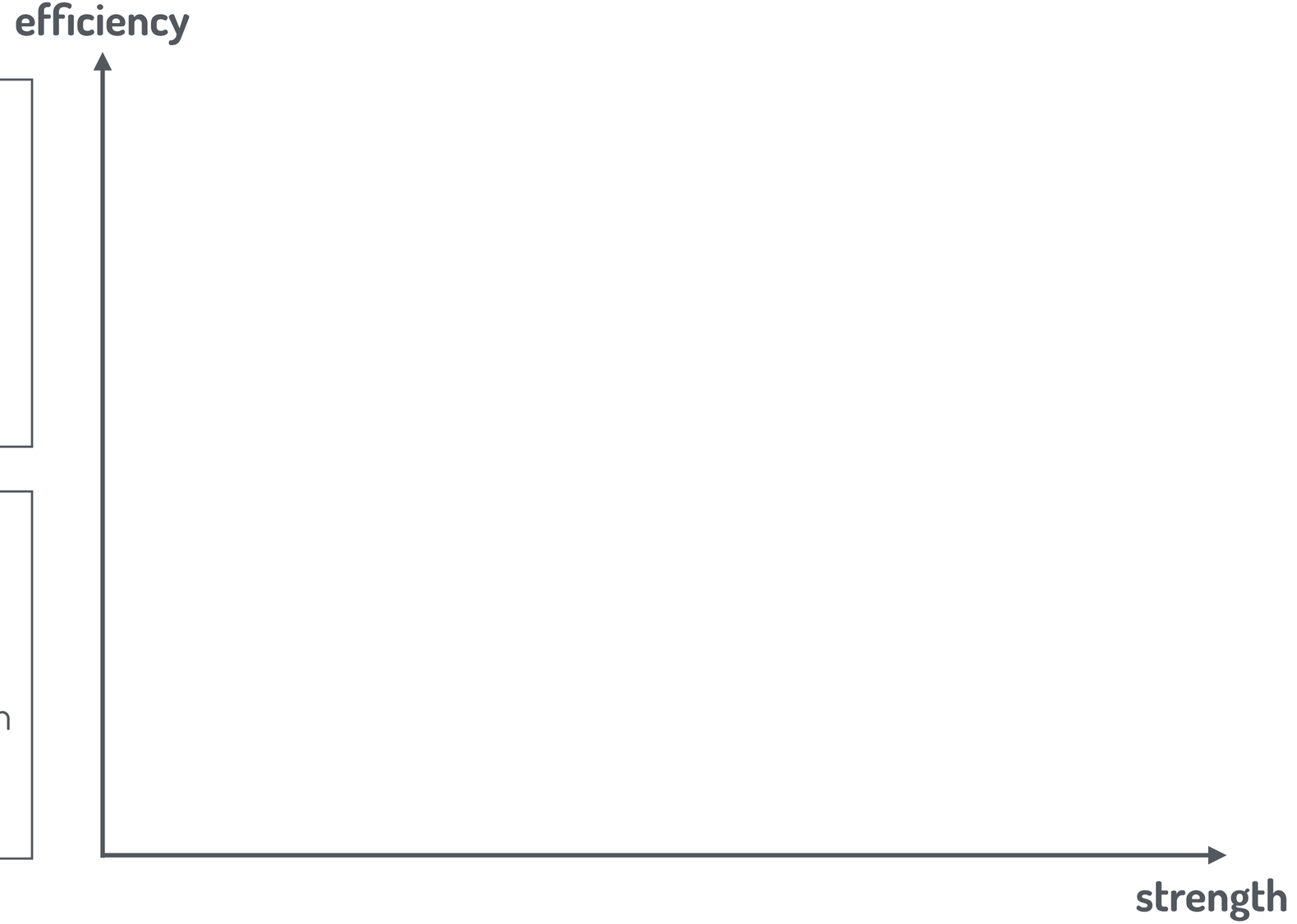
let us focus on C & Rust

enhancement strength

- verified properties: memory / thread safety, functional correctness, security, ...
- trustworthiness: non-foundational / foundational

verification efficiency

- learning curve
- automation, modularity, reusability, evolution
- synergy with existing tooling



Language Enhancement?

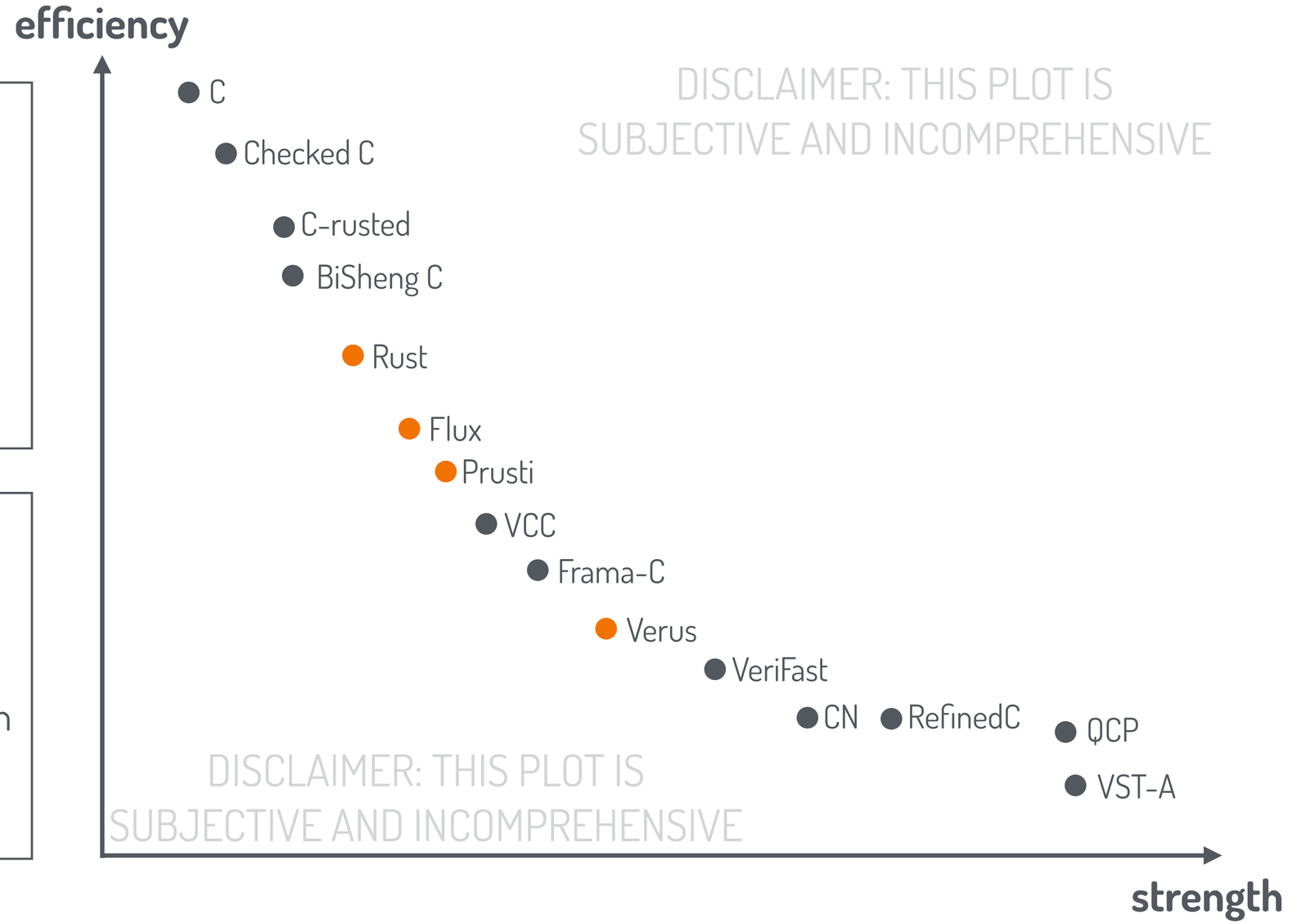
let us focus on C & Rust

enhancement strength

- verified properties: memory / thread safety, functional correctness, security, ...
- trustworthiness: non-foundational / foundational

verification efficiency

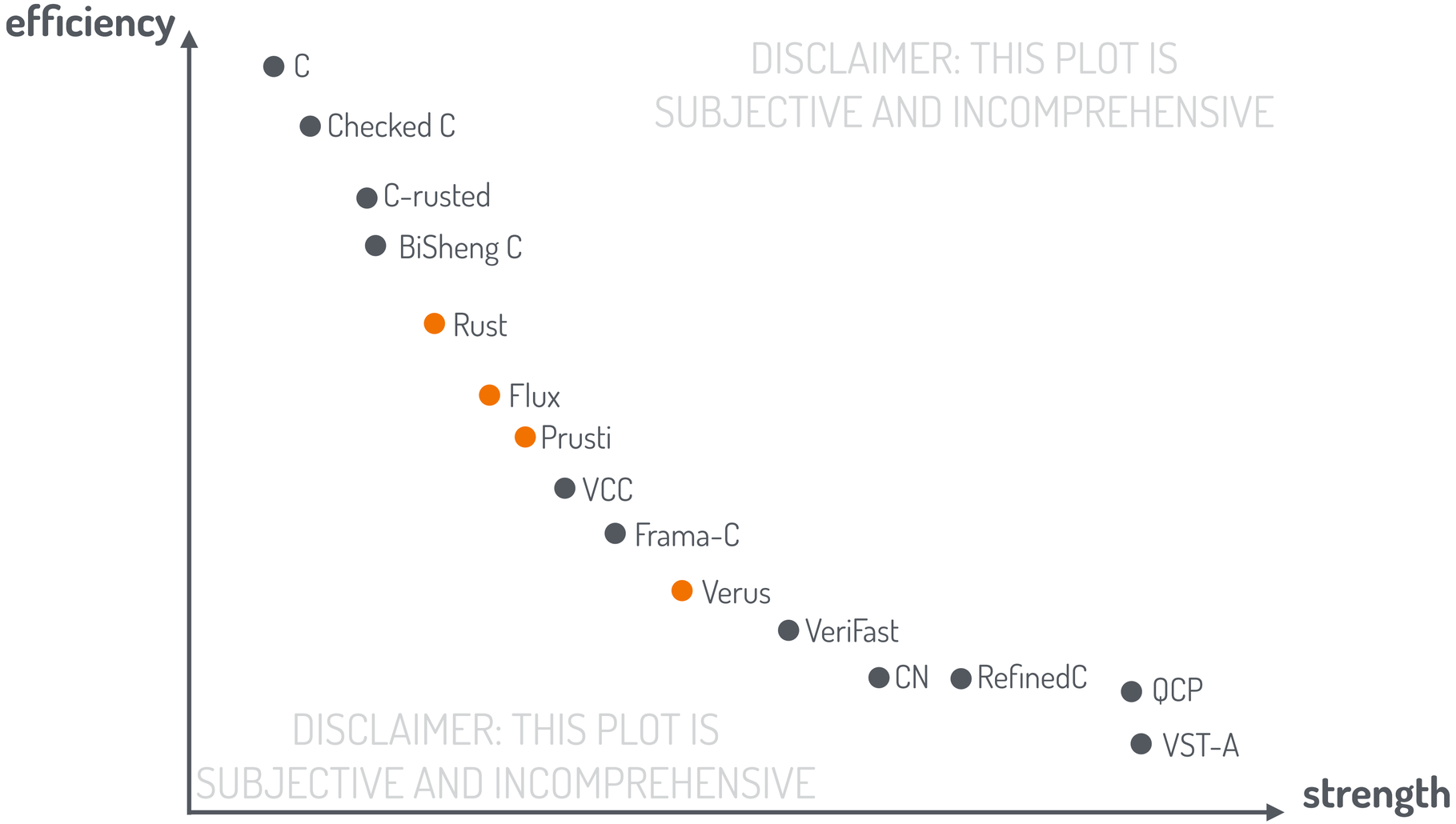
- learning curve
- automation, modularity, reusability, evolution
- synergy with existing tooling



DISCLAIMER: THIS PLOT IS SUBJECTIVE AND INCOMPREHENSIVE

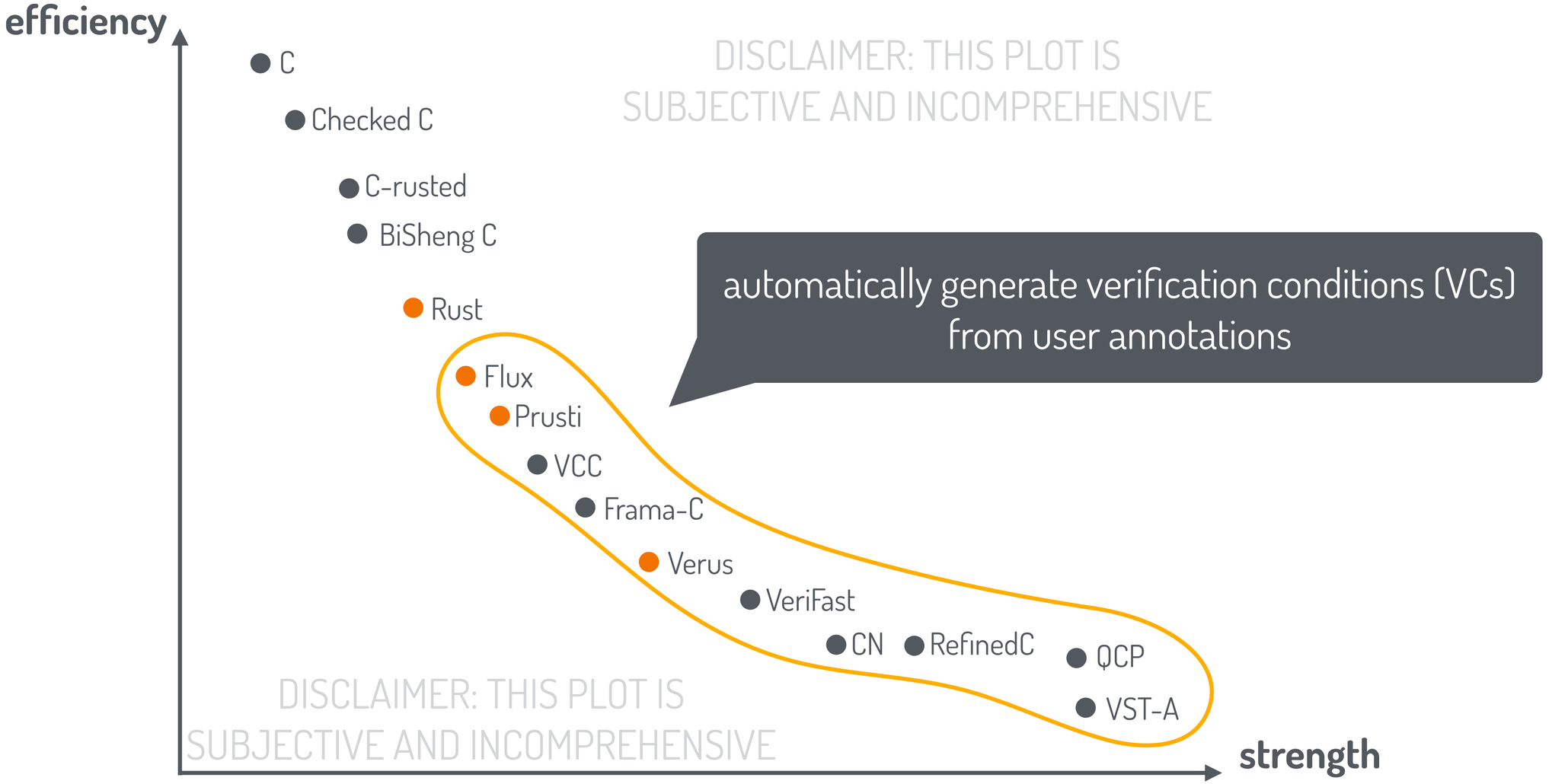
Obs I: VC Generation is Required

if aiming for high efficiency



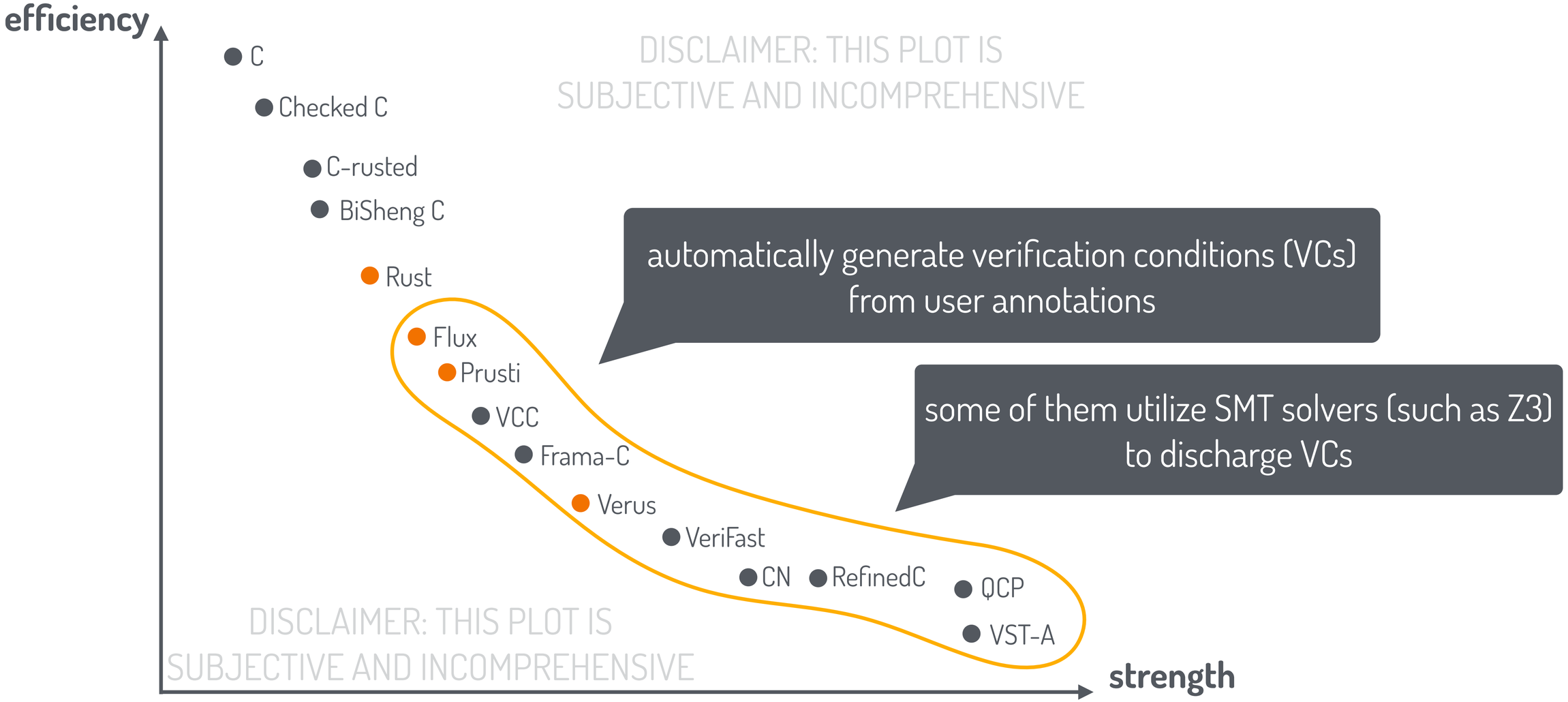
Obs I: VC Generation is Required

if aiming for high efficiency



Obs I: VC Generation is Required

if aiming for high efficiency

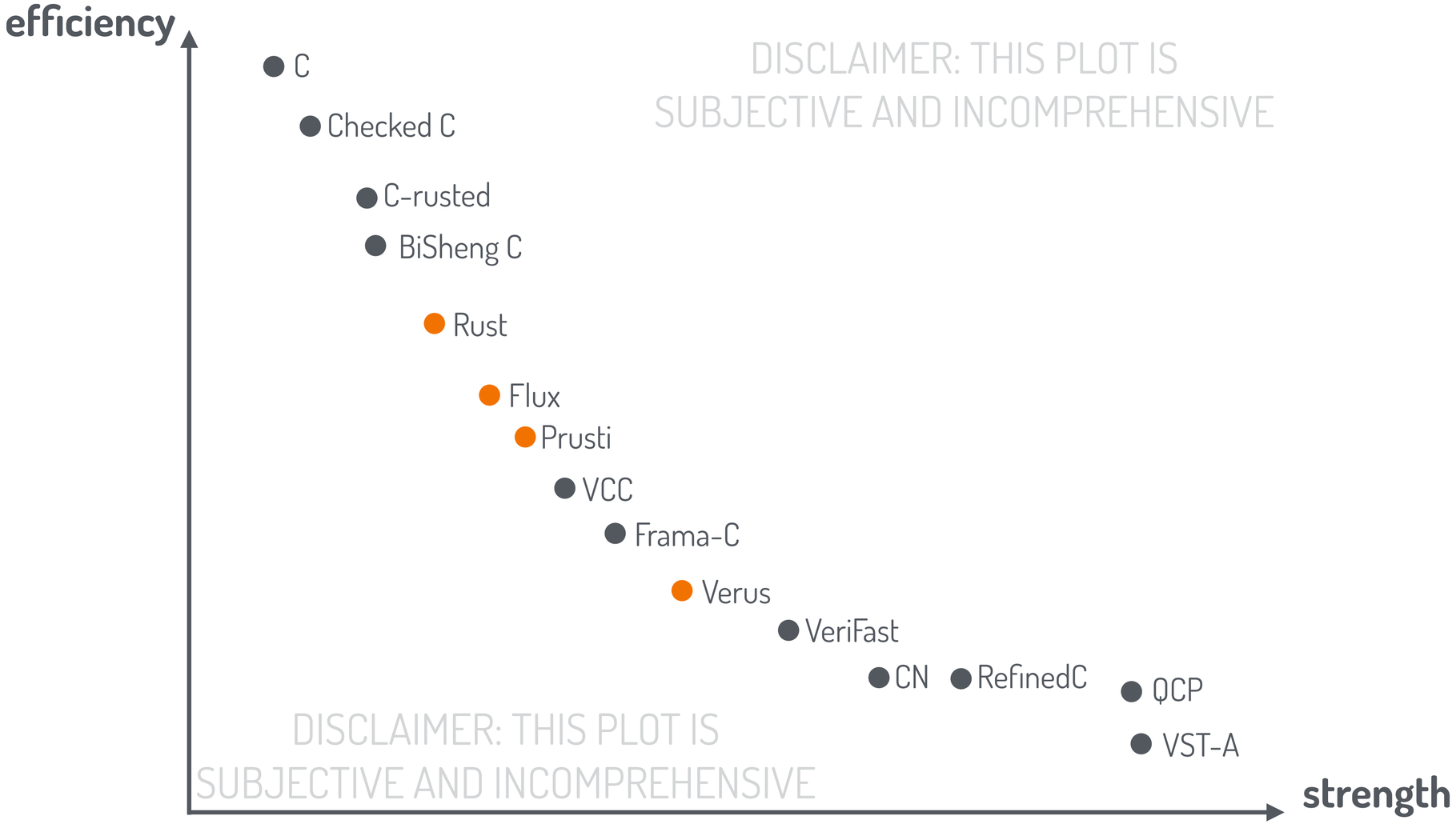


CStar Demo: VC Generation

nothing new yet

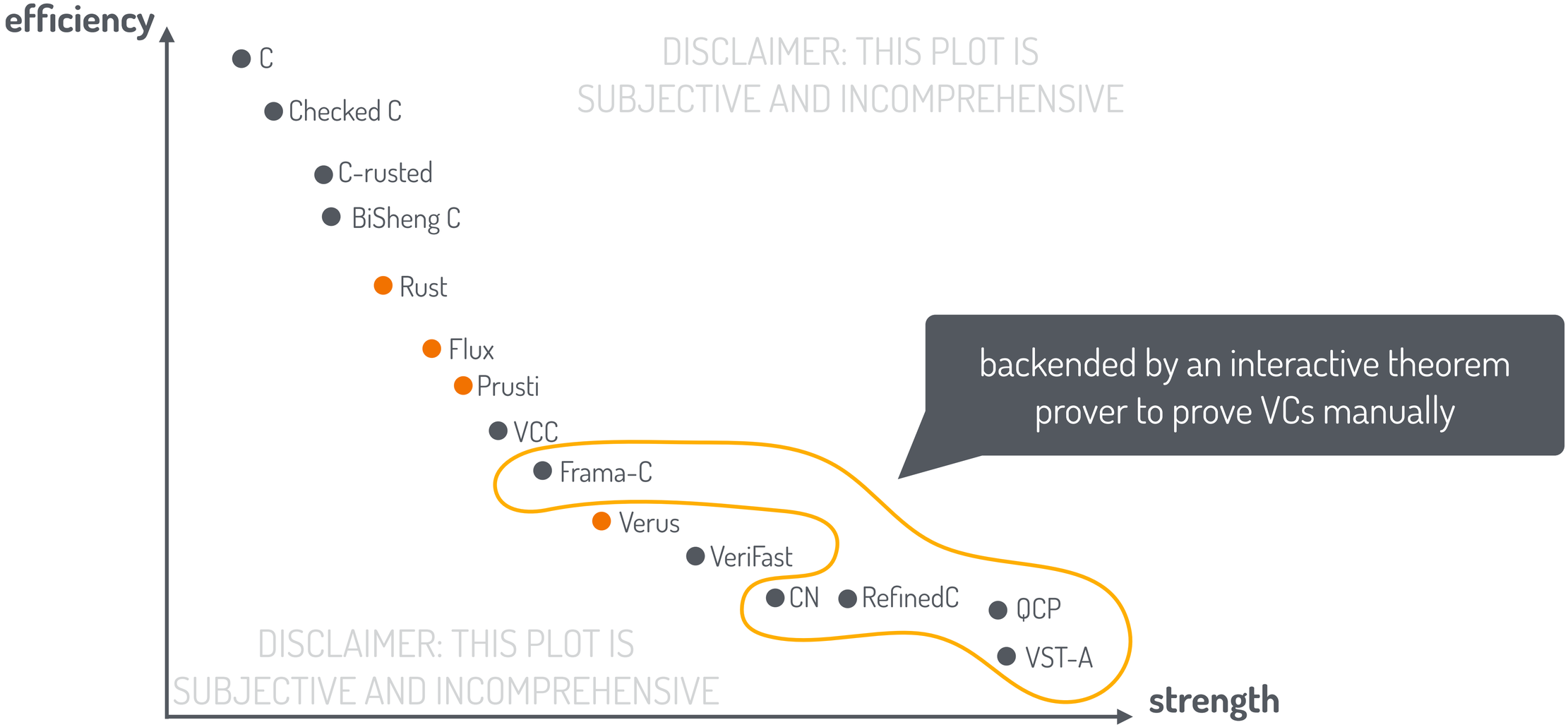
Obs: Manual Proof is Unavoidable

if aiming at strong strength



Obs: Manual Proof is Unavoidable

if aiming at strong strength



Prob I: Proof Encapsulation

theorem provers are nice, but ...

current status in CN (and many others):
proofs are delegated to Rocq

```
lemma append_nil (datatype seq l1)
  requires true;
  ensures append(l1, Nil {}) == l1;

lemma append_cons (datatype seq l1, i32 x, datatype seq l2)
  requires true;
  ensures append(l1, Cons {head: x, tail: l2})
          == append(snoc(l1, x), l2);
```

```
struct node *tmp = cur->tail;
cur->tail = last;
last = cur;
cur = tmp;
/*@ unfold rev(L2); @*/
/*@ apply append_cons (rev (tl(L2)), hd(L2), L1); @*/
```

Prob I: Proof Encapsulation

theorem provers are nice, but ...

- obs: libraries constitute the hidden substrate of modern (systems) software

current status in CN (and many others): proofs are delegated to Rocq

```
lemma append_nil (datatype seq l1)
  requires true;
  ensures append(l1, Nil {}) == l1;

lemma append_cons (datatype seq l1, i32 x, datatype seq l2)
  requires true;
  ensures append(l1, Cons {head: x, tail: l2})
          == append(snoc(l1, x), l2);
```

```
struct node *tmp = cur->tail;
cur->tail = last;
last = cur;
cur = tmp;
/*@ unfold rev(L2); @*/
/*@ apply append_cons (rev (tl(L2)), hd(L2), L1); @*/
```

Prob I: Proof Encapsulation

theorem provers are nice, but ...

- obs: libraries constitute the hidden substrate of modern (systems) software
- obs: multilingual libraries, on the other hand, might cause cross-language-boundary issues

current status in CN (and many others): proofs are delegated to Rocq

```
lemma append_nil (datatype seq l1)
  requires true;
  ensures append(l1, Nil {}) == l1;

lemma append_cons (datatype seq l1, i32 x, datatype seq l2)
  requires true;
  ensures append(l1, Cons {head: x, tail: l2})
          == append(snoc(l1, x), l2);
```

```
struct node *tmp = cur->tail;
cur->tail = last;
last = cur;
cur = tmp;
/*@ unfold rev(L2); @*/
/*@ apply append_cons (rev (tl(L2)), hd(L2), L1); @*/
```

Prob I: Proof Encapsulation

theorem provers are nice, but ...

- obs: libraries constitute the hidden substrate of modern (systems) software
- obs: multilingual libraries, on the other hand, might cause cross-language-boundary issues

principle: integrate **proof-specification-implementation** together into one language

current status in CN (and many others): proofs are delegated to Rocq

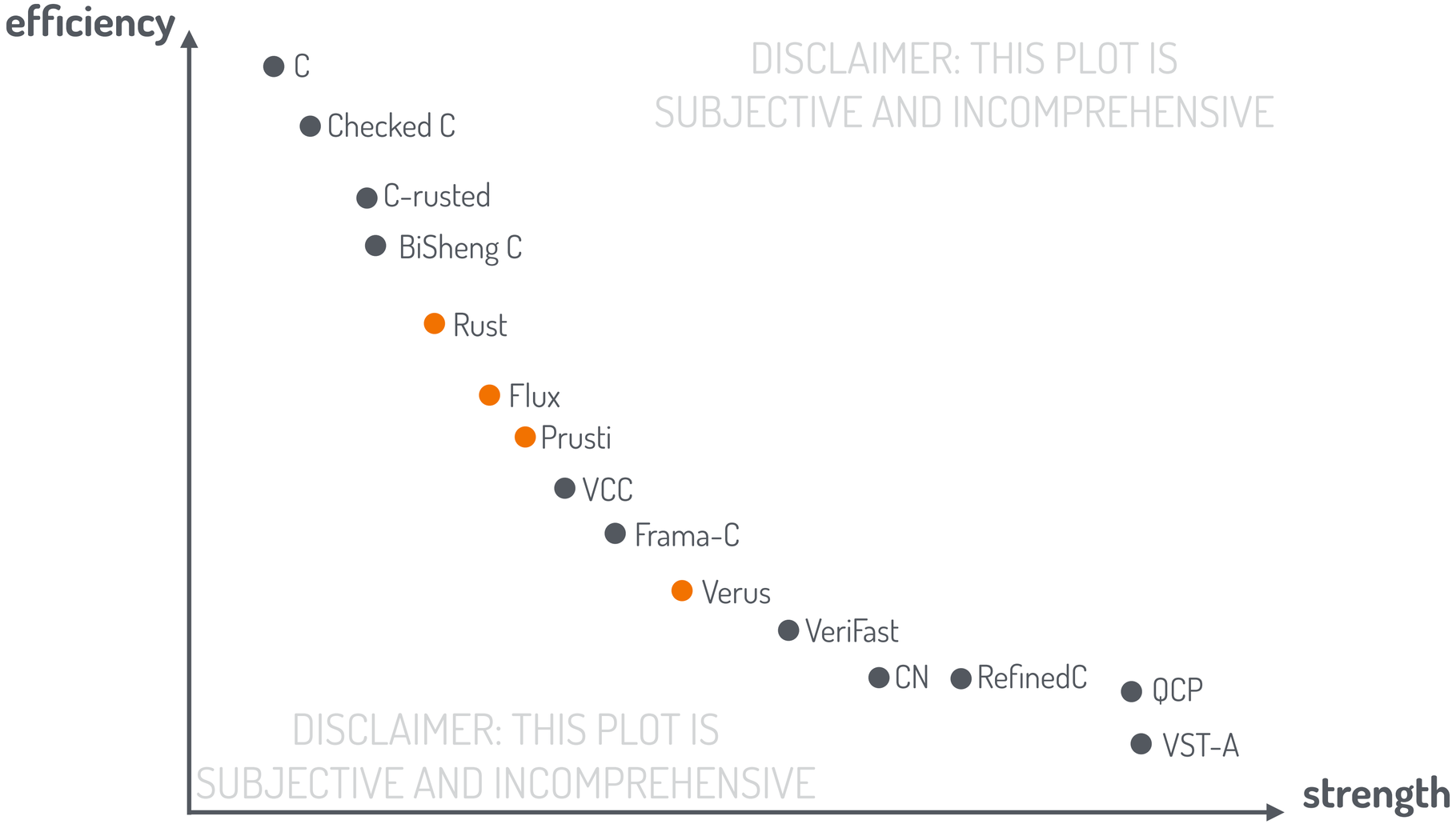
```
lemma append_nil (datatype seq l1)
  requires true;
  ensures append(l1, Nil {}) == l1;

lemma append_cons (datatype seq l1, i32 x, datatype seq l2)
  requires true;
  ensures append(l1, Cons {head: x, tail: l2})
          == append(snoc(l1, x), l2);
```

```
struct node *tmp = cur->tail;
cur->tail = last;
last = cur;
cur = tmp;
/*@ unfold rev(L2); @*/
/*@ apply append_cons (rev (tl(L2)), hd(L2), L1); @*/
```

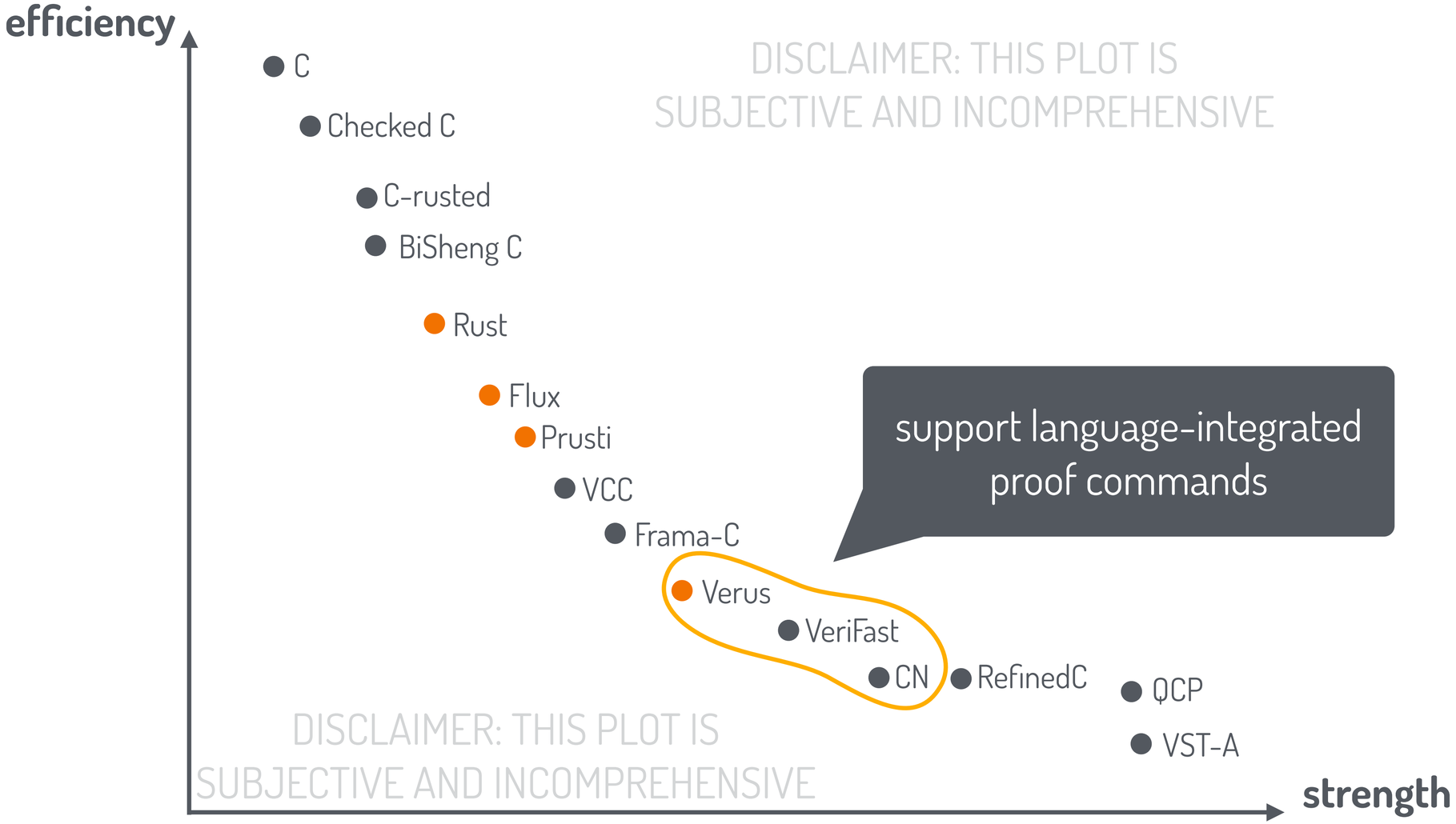
Obs: Manual Proof is Unavoidable

if aiming at strong strength



Obs: Manual Proof is Unavoidable

if aiming at strong strength



```
struct node *reverse(struct node *xs)
/*@ requires take L = IntList(xs);
    ensures  take L_ = IntList(return);
             L_ == rev(L);
@*/
{

  struct node *last = 0;
  struct node *cur = xs;
  /*@ apply append_nil(rev(L)); @*/
  while(1)
  /*@ inv take L1 = IntList(last);
          take L2 = IntList(cur);
          append(rev(L2), L1) == rev(L);
  @*/
  {
    if (cur == 0) {
      /*@ unfold rev(Nil {}); @*/
      /*@ unfold append(Nil {}, L1); @*/

      return last;
    }
    struct node *tmp = cur->tail;
    cur->tail = last;
    last = cur;
    cur = tmp;
    /*@ unfold rev(L2); @*/
    /*@ apply append_cons (rev (tl(L2)), hd(L2), L1); @*/
  }
}
```

example program with proof commands in CN

```
struct node *reverse(struct node *xs)
/*@ requires take L = IntList(xs);
    ensures  take L_ = IntList(return);
             L_ == rev(L);
@*/
{

  struct node *last = 0;
  struct node *cur = xs;
  /*@ apply append_nil(rev(L)); @*/
  while(1)
  /*@ inv take L1 = IntList(last);
          take L2 = IntList(cur);
          append(rev(L2), L1) == rev(L);
  @*/
  {
    if (cur == 0) {
      /*@ unfold rev(Nil {}); @*/
      /*@ unfold append(Nil {}, L1); @*/

      return last;
    }
    struct node *tmp = cur->tail;
    cur->tail = last;
    last = cur;
    cur = tmp;
    /*@ unfold rev(L2); @*/
    /*@ apply append_cons (rev (tl(L2)), hd(L2), L1); @*/
  }
}
```

example program with proof commands in CN

```
public boolean remove(Object o)
/*: requires " init"
    modifies content, csize
    ensures
 "( result  → (∃ i. (i,o)∈old content ∧
          (¬∃ j. j<i ∧ (j,o)∈old content) ∧
          (∀ j e. 0≤j∧j<i→(j,e)∈content=(j,e)∈old content) ∧
          (i≤j∧j<csize→(j,e)∈content=(j+1,e)∈old content)) ∧
  (¬result → (content=old content∧¬∃i. (i,o)∈old content))" */
{
 int index = 0;
 while /*: inv "(∀j. 0≤j∧j<index→o≠elements.[j]) ∧
                0≤index ∧ size=old size" */
       (index < size) {
  if (elements[index] == o) {
   shift (index);
   /*: note ObjectRemoved:
    "∀j e. (0≤j∧j<index→(j,e)∈content=(j,e)∈old content) ∧
       (index≤j∧j<csize→(j,e)∈content=(j+1,e)∈old content)"
            from shift_Postcondition , LoopInv, LoopCondition
                                        content_def , csize_def ;
       witness index for
    "∃i. (i,o)∈old content ∧ (¬∃j. j<i ∧ (j,o)∈old content) ∧
      (∀j e. (0≤j∧j<i→(j,e)∈content=(j,e)∈old content) ∧
       (i≤j∧j<csize→(j,e)∈content=(j+1,e)∈old content))" */
   return true;
  }
  index = index + 1;
 }
 return false ;
}
```

example program with proof commands in Jahob

Prob II: Programmable Proof

proof commands are nice, but ...

$$
\begin{array}{lcl}
p & ::= & p_1 \,;\, p_2 \\
 & | & \text{assert } l\text{: } F \text{ from } \vec{h} \\
 & | & \text{note } l\text{:}F \text{ from } \vec{h} \\
 & | & \text{localize in } (p \,;\, \text{note } l\text{:}F) \\
 & | & \text{mp } l\text{:}(F \rightarrow G) \\
 & | & \text{assuming } l_F\text{:}F \text{ in } (p \,;\, \text{note } l_G\text{:}G) \\
 & | & \text{cases } \vec{F} \text{ for } l\text{:}G \\
 & | & \text{showedCase } i \text{ of } l : F_1 \vee \ldots \vee F_n \\
 & | & \text{byContradiction } l\text{:}F \text{ in } p \\
 & | & \text{contradiction } l\text{:}F \\
 & | & \text{instantiate } l\text{:}\forall \vec{x}.F \text{ with } \vec{t} \\
 & | & \text{witness } \vec{t} \text{ for } l\text{:}\exists \vec{x}.F \\
 & | & \text{pickWitness } \vec{x} \text{ for } l_F\text{:}F \text{ in } (p \,;\, \text{note } l_G\text{:}G) \\
 & | & \text{pickAny } \vec{x} \text{ in } (p \,;\, \text{note } l\text{:}F) \\
 & | & \text{induct } l\text{:}F \text{ over } n \text{ in } p
\end{array}
$$

integrated proof commands in Jahob

Prob II: Programmable Proof

proof commands are nice, but ...

- obs: manual proofs are inevitable for complex reasoning tasks

$$
\begin{array}{lcl}
p & ::= & p_1 \,;\, p_2 \\
& | & \text{assert } l{:}\, F \text{ from } \vec{h} \\
& | & \text{note } l{:}F \text{ from } \vec{h} \\
& | & \text{localize in } (p \,;\, \text{note } l{:}F) \\
& | & \text{mp } l{:}(F \rightarrow G) \\
& | & \text{assuming } l_F{:}F \text{ in } (p \,;\, \text{note } l_G{:}G) \\
& | & \text{cases } \vec{F} \text{ for } l{:}G \\
& | & \text{showedCase } i \text{ of } l : F_1 \vee \ldots \vee F_n \\
& | & \text{byContradiction } l{:}F \text{ in } p \\
& | & \text{contradiction } l{:}F \\
& | & \text{instantiate } l{:}\forall \vec{x}.F \text{ with } \vec{t} \\
& | & \text{witness } \vec{t} \text{ for } l{:}\exists \vec{x}.F \\
& | & \text{pickWitness } \vec{x} \text{ for } l_F{:}F \text{ in } (p \,;\, \text{note } l_G{:}G) \\
& | & \text{pickAny } \vec{x} \text{ in } (p \,;\, \text{note } l{:}F) \\
& | & \text{induct } l{:}F \text{ over } n \text{ in } p
\end{array}
$$

integrated proof commands in Jahob

Prob II: Programmable Proof

proof commands are nice, but ...

- obs: manual proofs are inevitable for complex reasoning tasks
- obs: using a fixed set of proof commands feels neither comprehensive nor extensible

$$
\begin{array}{lcl}
p & ::= & p_1 \,;\, p_2 \\
 & | & \text{assert } l\text{:}\, F \text{ from } \vec{h} \\
 & | & \text{note } l\text{:}F \text{ from } \vec{h} \\
 & | & \text{localize in } (p \,;\, \text{note } l\text{:}F) \\
 & | & \text{mp } l\text{:}(F \rightarrow G) \\
 & | & \text{assuming } l_F\text{:}F \text{ in } (p \,;\, \text{note } l_G\text{:}G) \\
 & | & \text{cases } \vec{F} \text{ for } l\text{:}G \\
 & | & \text{showedCase } i \text{ of } l : F_1 \vee \ldots \vee F_n \\
 & | & \text{byContradiction } l\text{:}F \text{ in } p \\
 & | & \text{contradiction } l\text{:}F \\
 & | & \text{instantiate } l\text{:}\forall \vec{x}.F \text{ with } \vec{t} \\
 & | & \text{witness } \vec{t} \text{ for } l\text{:}\exists \vec{x}.F \\
 & | & \text{pickWitness } \vec{x} \text{ for } l_F\text{:}F \text{ in } (p \,;\, \text{note } l_G\text{:}G) \\
 & | & \text{pickAny } \vec{x} \text{ in } (p \,;\, \text{note } l\text{:}F) \\
 & | & \text{induct } l\text{:}F \text{ over } n \text{ in } p
\end{array}
$$

integrated proof commands in Jahob

Prob II: Programmable Proof

proof commands are nice, but …

- obs: manual proofs are inevitable for complex reasoning tasks
- obs: using a fixed set of proof commands feels neither comprehensive nor extensible

principle: support using the full language to **programmatically** construct proof

$$
\begin{array}{lcl}
p & ::= & p_1 \,;\, p_2 \\
 & | & \text{assert } l\text{:}\, F \text{ from } \vec{h} \\
 & | & \text{note } l\text{:}F \text{ from } \vec{h} \\
 & | & \text{localize in } (p \,;\, \text{note } l\text{:}F) \\
 & | & \text{mp } l\text{:}(F \rightarrow G) \\
 & | & \text{assuming } l_F\text{:}F \text{ in } (p \,;\, \text{note } l_G\text{:}G) \\
 & | & \text{cases } \vec{F} \text{ for } l\text{:}G \\
 & | & \text{showedCase } i \text{ of } l : F_1 \vee \ldots \vee F_n \\
 & | & \text{byContradiction } l\text{:}F \text{ in } p \\
 & | & \text{contradiction } l\text{:}F \\
 & | & \text{instantiate } l\text{:}\forall \vec{x}.F \text{ with } \vec{t} \\
 & | & \text{witness } \vec{t} \text{ for } l\text{:}\exists \vec{x}.F \\
 & | & \text{pickWitness } \vec{x} \text{ for } l_F\text{:}F \text{ in } (p \,;\, \text{note } l_G\text{:}G) \\
 & | & \text{pickAny } \vec{x} \text{ in } (p \,;\, \text{note } l\text{:}F) \\
 & | & \text{induct } l\text{:}F \text{ over } n \text{ in } p
\end{array}
$$

integrated proof commands in Jahob

The LCF Solution

logic for computable functions

The LCF Solution

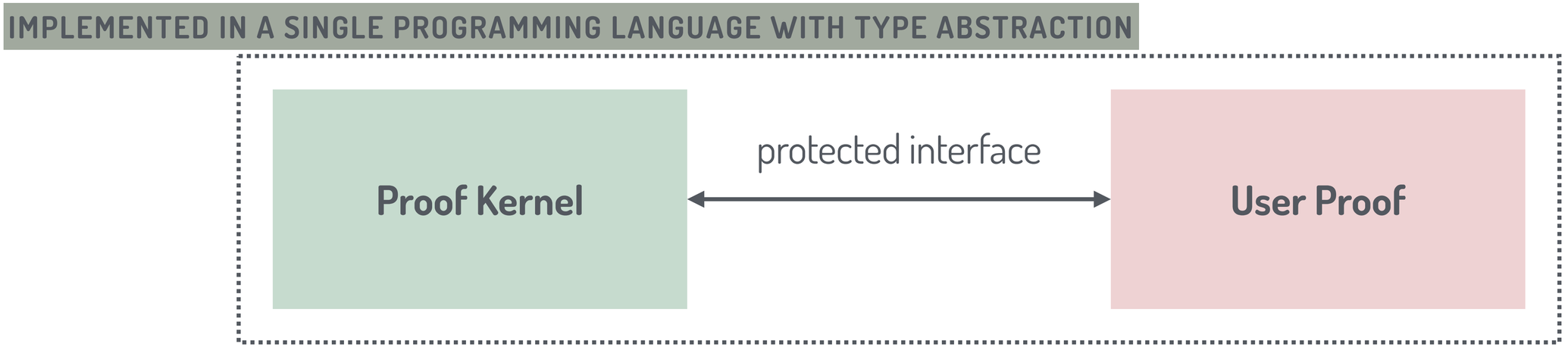
logic for computable functions

- the **LCF architecture** for theorem proving by Robin Milner in the 1970s

The LCF Solution

logic for computable functions

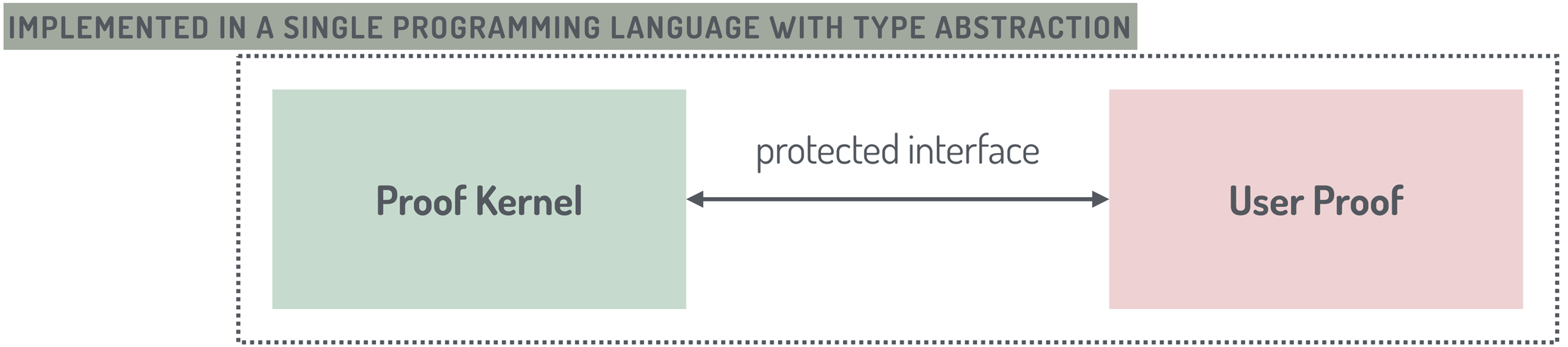
- the **LCF architecture** for theorem proving by Robin Milner in the 1970s
- programming formal proof is to **compute theorems** using a **protected interface**



The LCF Solution

logic for computable functions

- the **LCF architecture** for theorem proving by Robin Milner in the 1970s
- programming formal proof is to **compute theorems** using a **protected interface**
- **correct-by-construction** even when extended with arbitrary user proof code



CStar Demo: Proof Programming

a practice of LCF in C


PEKING UNIVERSITY
1898

it is now already possible to prove VCs in C

it is now already possible to prove VCs in C

but we can do better by introducing
program-proof states

CStar Demo: Program-Proof State

essentially, a symbolic representation of the program state


PEKING UNIVERSITY
1898

a VC is a prop "pre |-- post" at a program point

both are program-proof states

a VC is a prop "pre |-- post" at a program point

both are program-proof states

a VC is a prop "pre |-- post" at a program point

proving a VC is coupled with programming in CStar

both are program-proof states

a VC is a prop "pre |-- post" at a program point

proving a VC is coupled with programming in CStar
you write a proof to transform from pre to post

CStar Demo: Program Proof

unify program and proof

The Architecture of CStar

how to ensure correct-by-construction

The Architecture of CStar

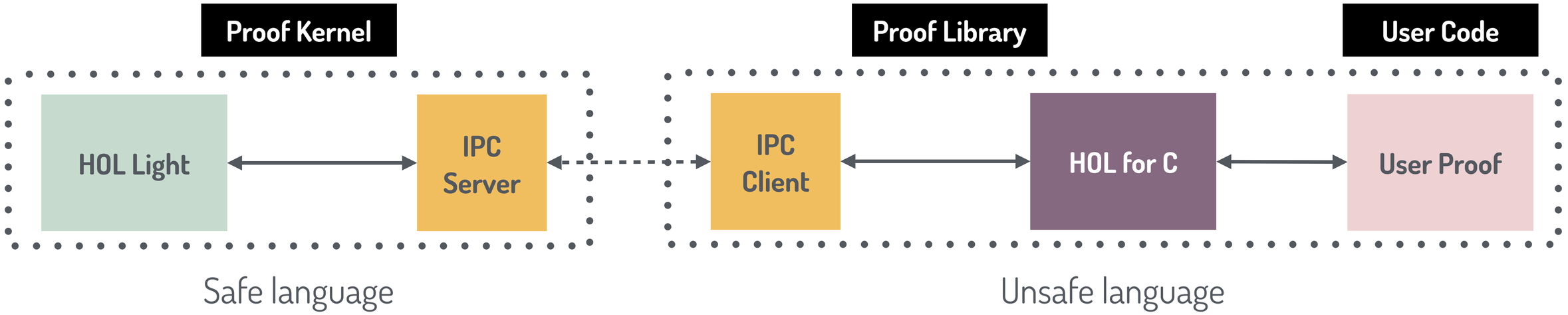
how to ensure correct-by-construction

- the original LCF design heavily relies on **type abstraction** provided by the ML language

The Architecture of CStar

how to ensure correct-by-construction

- the original LCF design heavily relies on **type abstraction** provided by the ML language
- we worked out a **language-agnostic** variant of the LCF-architecture with **process separation**



The Architecture of CStar

how to maintain the program-proof states

The Architecture of CStar

how to maintain the program-proof states

- compute symbolic heap information via **forward symbolic execution**

The Architecture of CStar

how to maintain the program-proof states

- compute symbolic heap information via **forward symbolic execution**
- CStar now employs (but is largely **agnostic** of) QCP's symbolic executor for **separation-logic** reasoning

QCP: A Practical Separation Logic-based C Program Verification Tool

Xiwei Wu[1], Yueyang Feng[1,*], Xiaoyang Lu[1,*], Tianchuan Lin[1], Kan Liu[1], Zhiyi Wang[2], Shushu Wu[1], Lihan Xie[1], Chengxi Yang[1], Hongyi Zhong[1], Naijun Zhan[2], Zhenjiang Hu[2], and Qinxiang Cao[1,†]

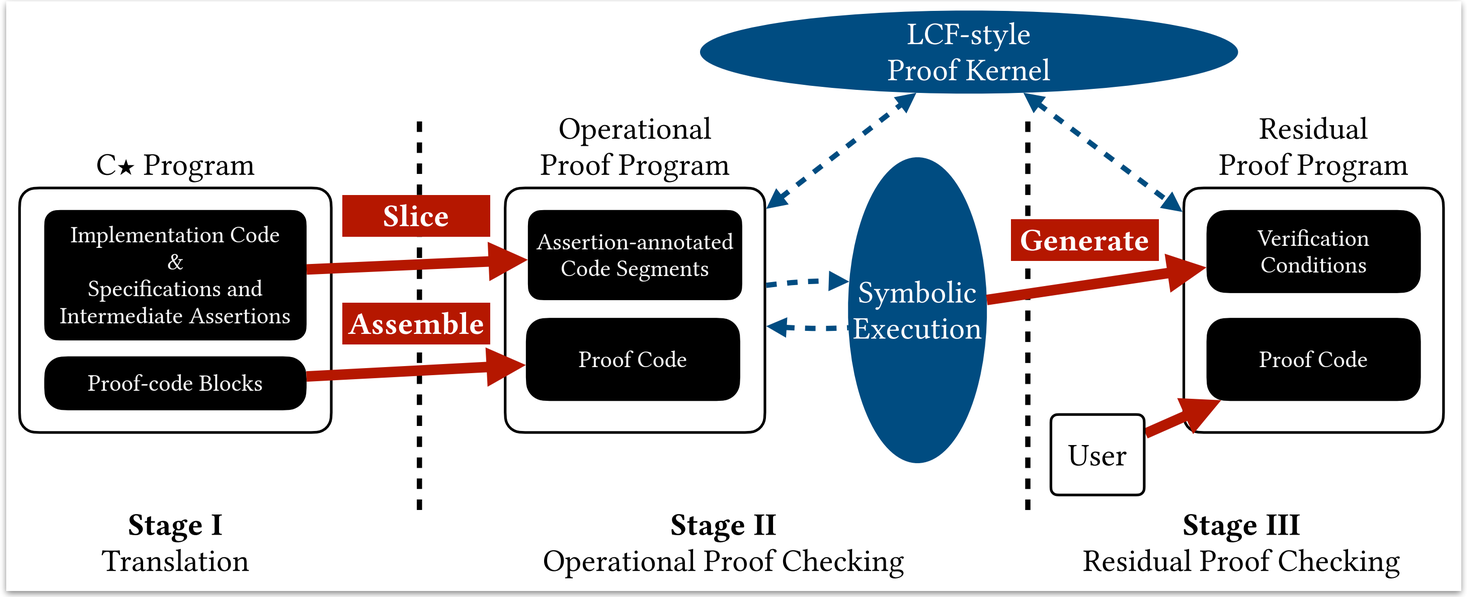
[1] Shanghai Jiao Tong University
{yashen, fyyvexoben, luxy1115, caoqinxiang}@sjtu.edu.cn
[2] Peking University
2301111964@stu.pku.edu.cn

https://arxiv.org/pdf/2505.12878

The Architecture of CStar

how to maintain the program-proof states

- compute symbolic heap information via **forward symbolic execution**
- CStar now employs (but is largely **agnostic** of) QCP's symbolic executor for **separation-logic** reasoning
- QCP evolves the program-proof states for program code; CStar evolves the program-proof states for proof code



QCP: A Practical Separation Logic-based C Program Verification Tool

Xiwei Wu[1], Yueyang Feng[1,*], Xiaoyang Lu[1,*], Tianchuan Lin[1], Kan Liu[1], Zhiyi Wang[2], Shushu Wu[1], Lihan Xie[1], Chengxi Yang[1], Hongyi Zhong[1], Naijun Zhan[2], Zhenjiang Hu[2], and Qinxiang Cao[1,†]

[1] Shanghai Jiao Tong University
{yashen, fyyvexoben, luxy1115, caoqinxiang}@sjtu.edu.cn
[2] Peking University
2301111964@stu.pku.edu.cn

https://arxiv.org/pdf/2505.12878

users can now use the full power of C
to conduct proof and verification

CStar Demo: Tactics

all implemented outside the proof kernel

CStar Demo: SMT Integration

achieve controllable automation

CStar Demo: Proof Encapsulation

one can implement program-proof libraries

CStar

Unifying Programming and Verification in C

CStar

Unifying Programming and Verification in C

prob i: proof encapsulation

CStar

Unifying Programming and Verification in C

prob i: proof encapsulation

prob ii: programmable proof

CStar

Unifying Programming and Verification in C

principle i: integrate **proof-specification-implementation** together into one language

prob i: proof encapsulation

prob ii: programmable proof

CStar

Unifying Programming and Verification in C

principle i: integrate **proof-specification-implementation** together into one language

prob i: proof encapsulation

prob ii: programmable proof

CStar

Unifying Programming and Verification in C

principle i: integrate **proof-specification-implementation** together into one language

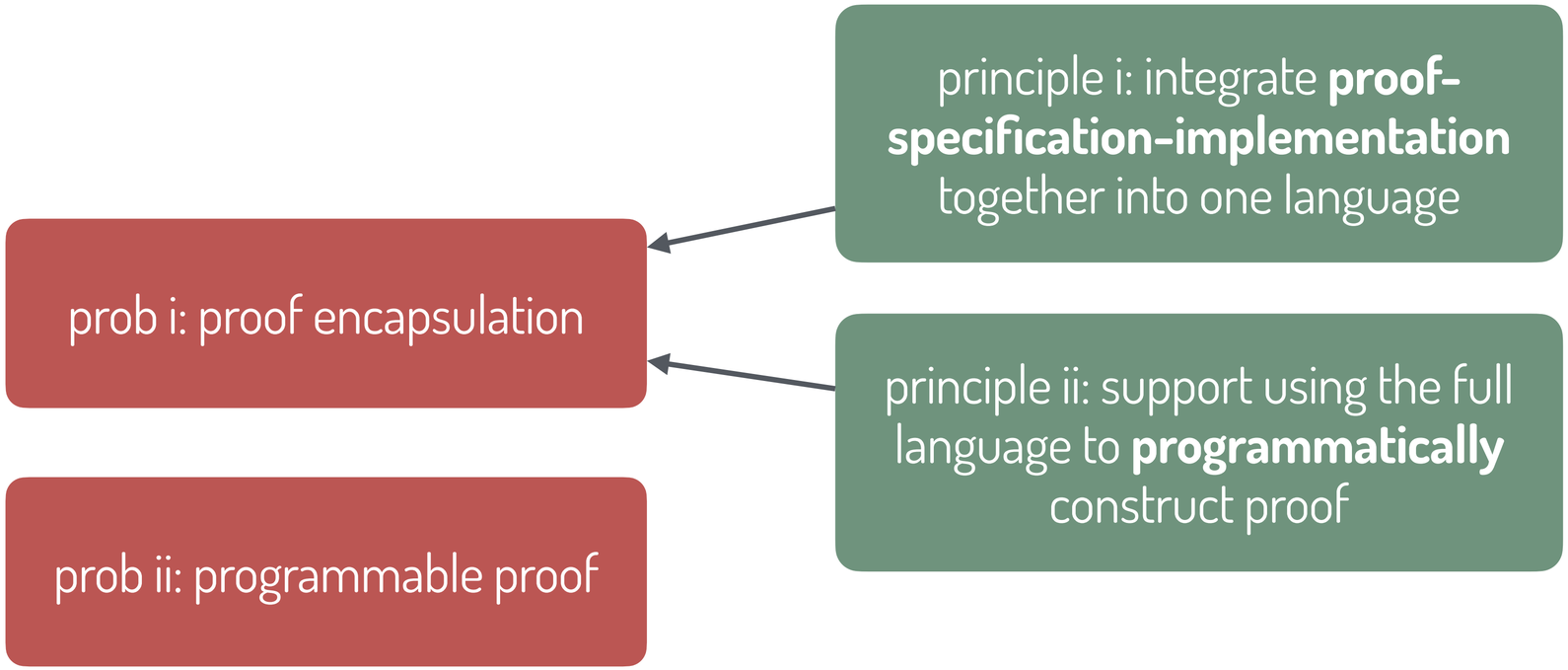
principle ii: support using the full language to **programmatically** construct proof

prob i: proof encapsulation

prob ii: programmable proof

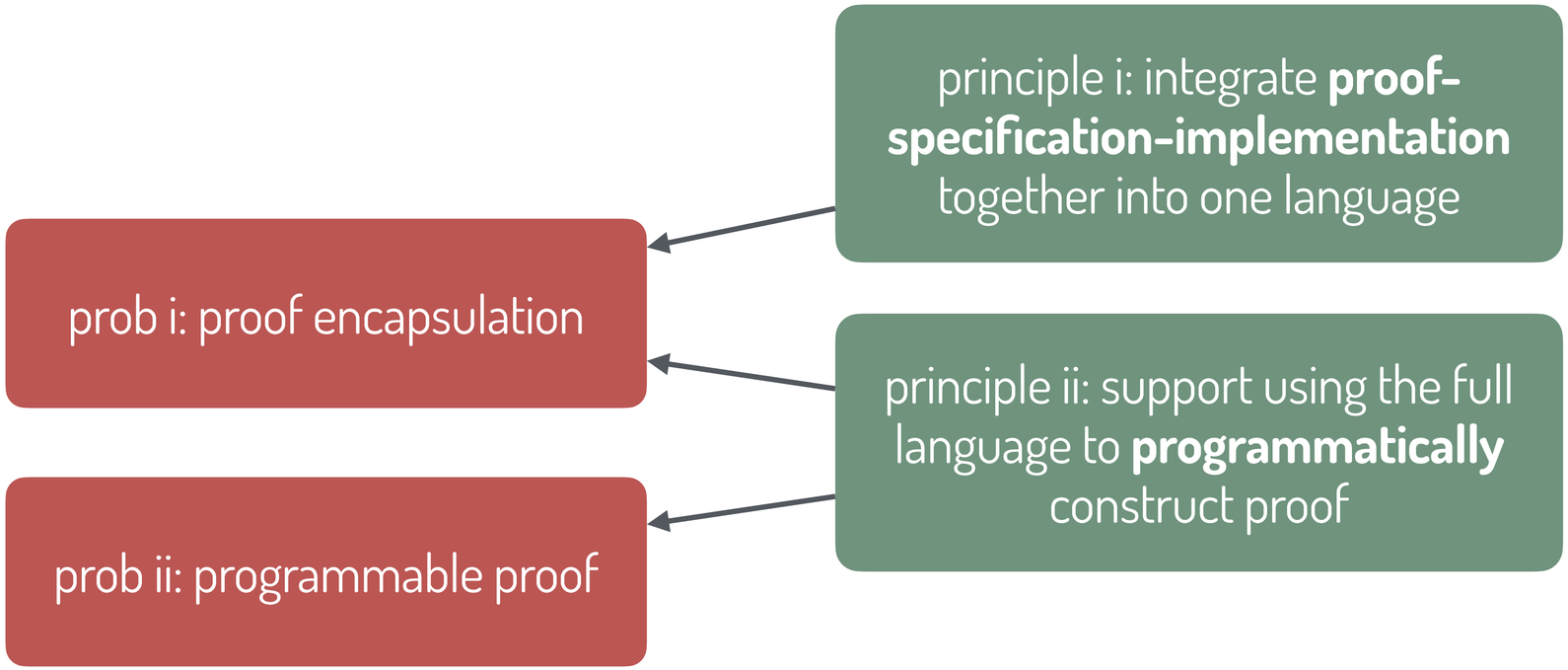
CStar

Unifying Programming and Verification in C



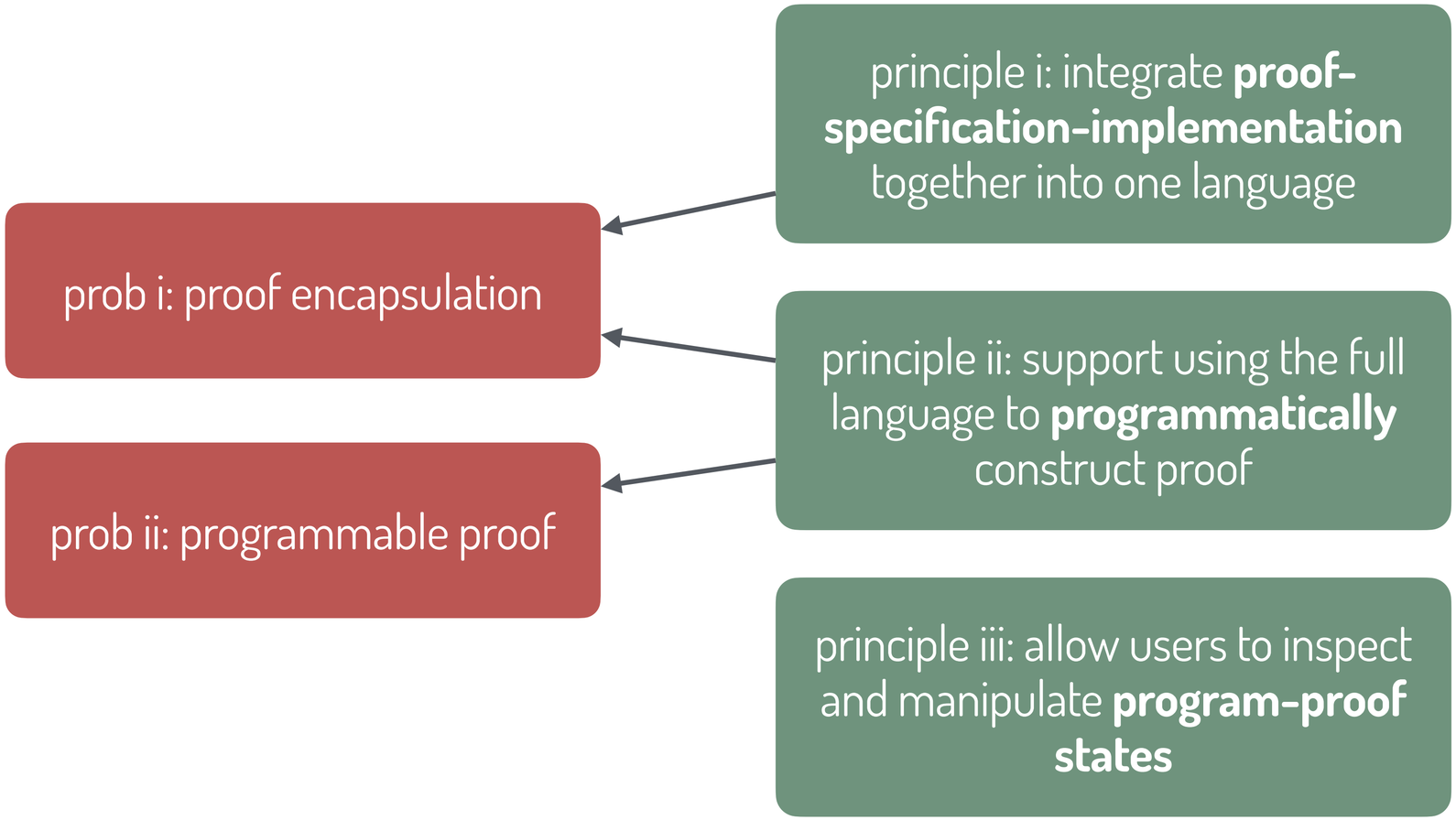
CStar

Unifying Programming and Verification in C



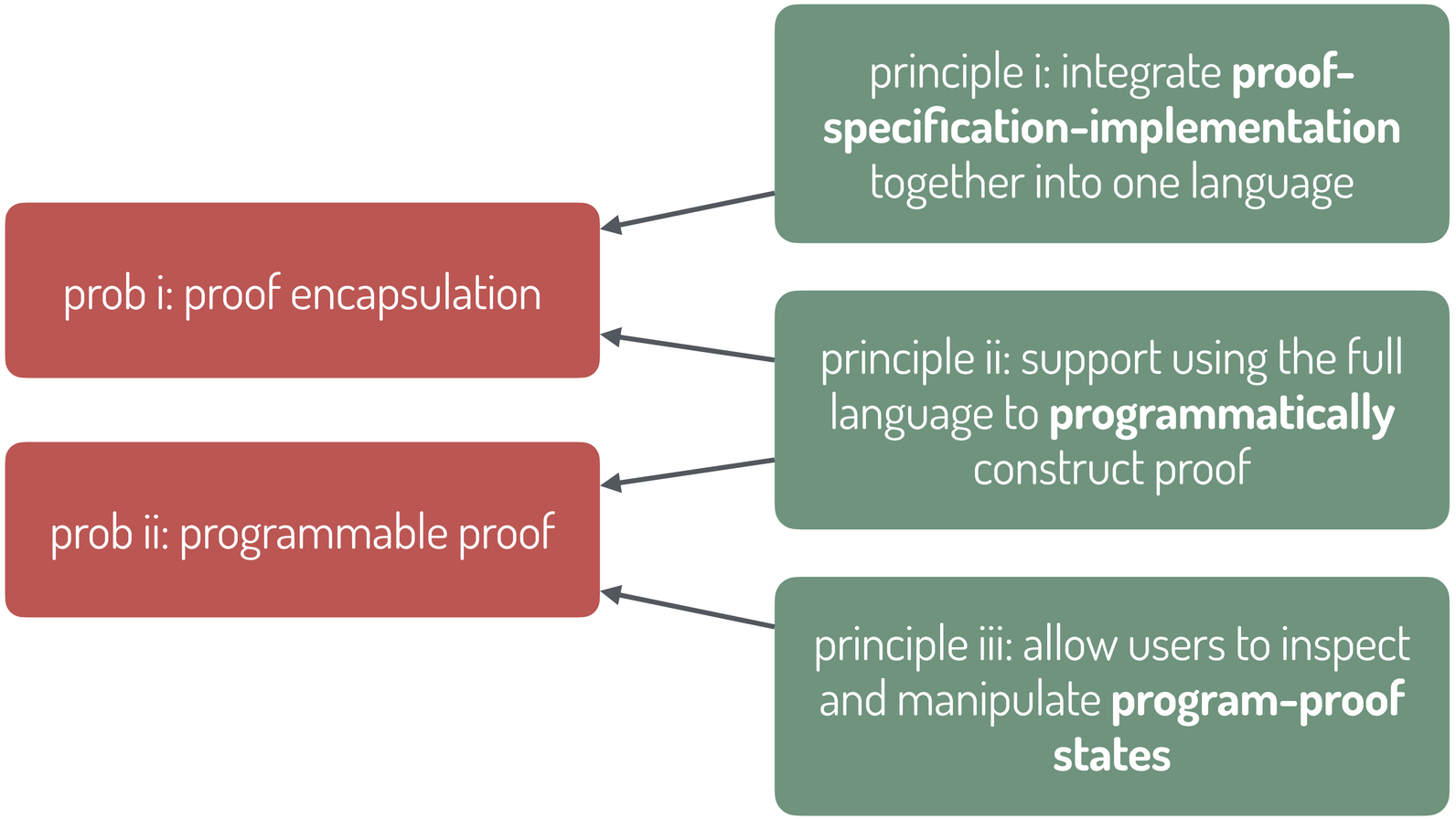
CStar

Unifying Programming and Verification in C



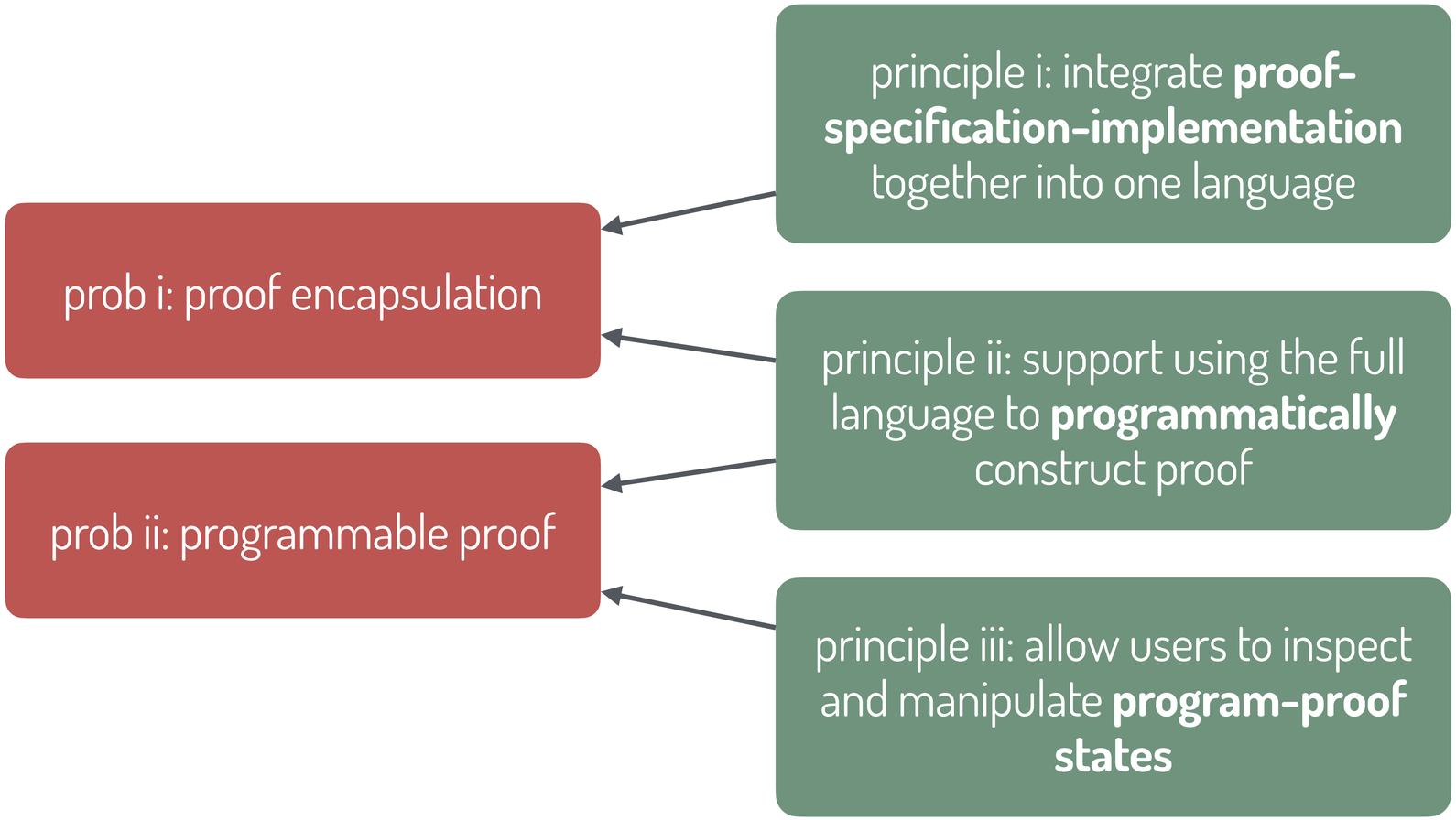
CStar

Unifying Programming and Verification in C



CStar

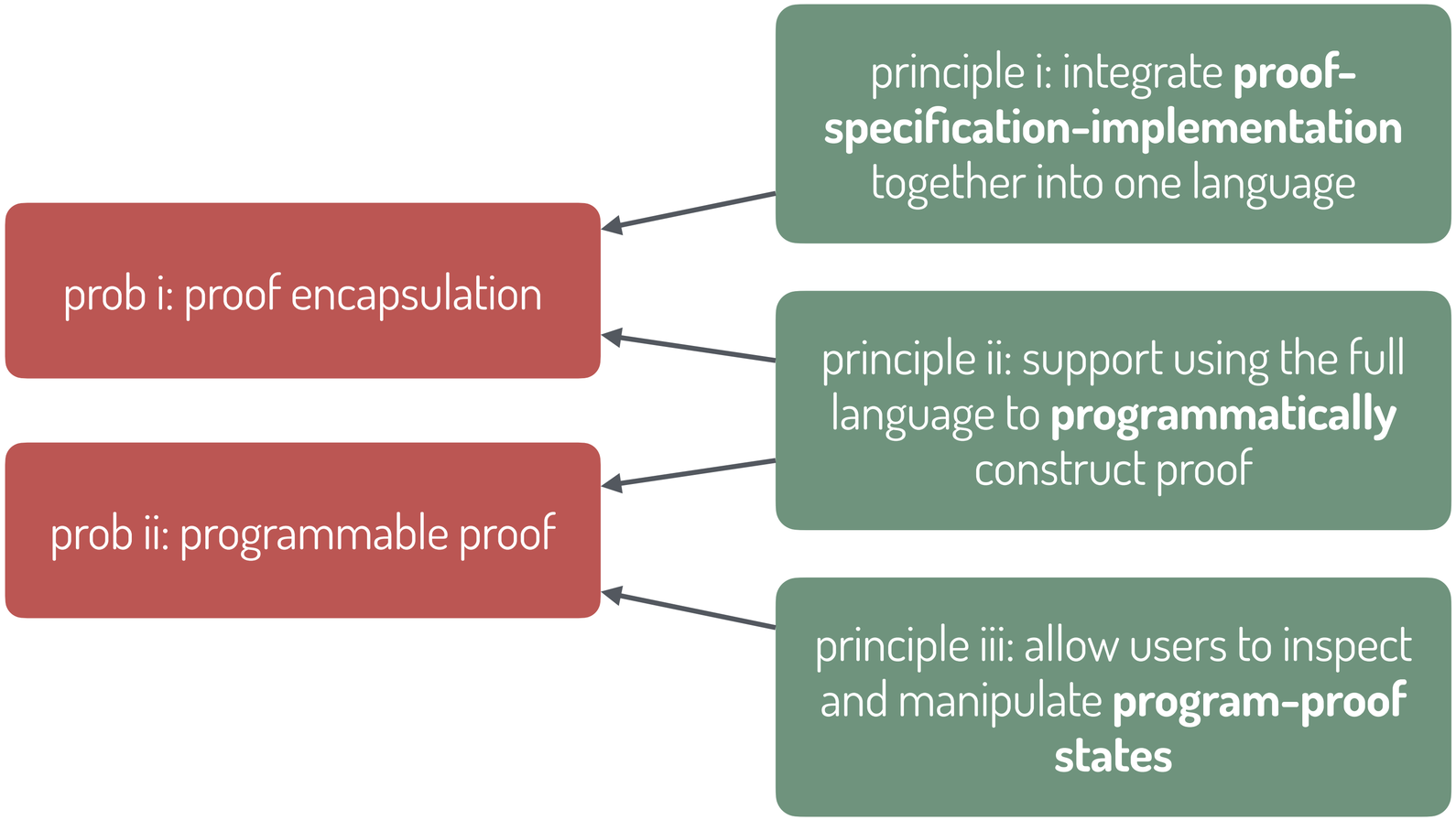
Unifying Programming and Verification in C



- **CStar reuses clang's toolchain**
  - use clang for compilation
  - use clangd for language server

CStar

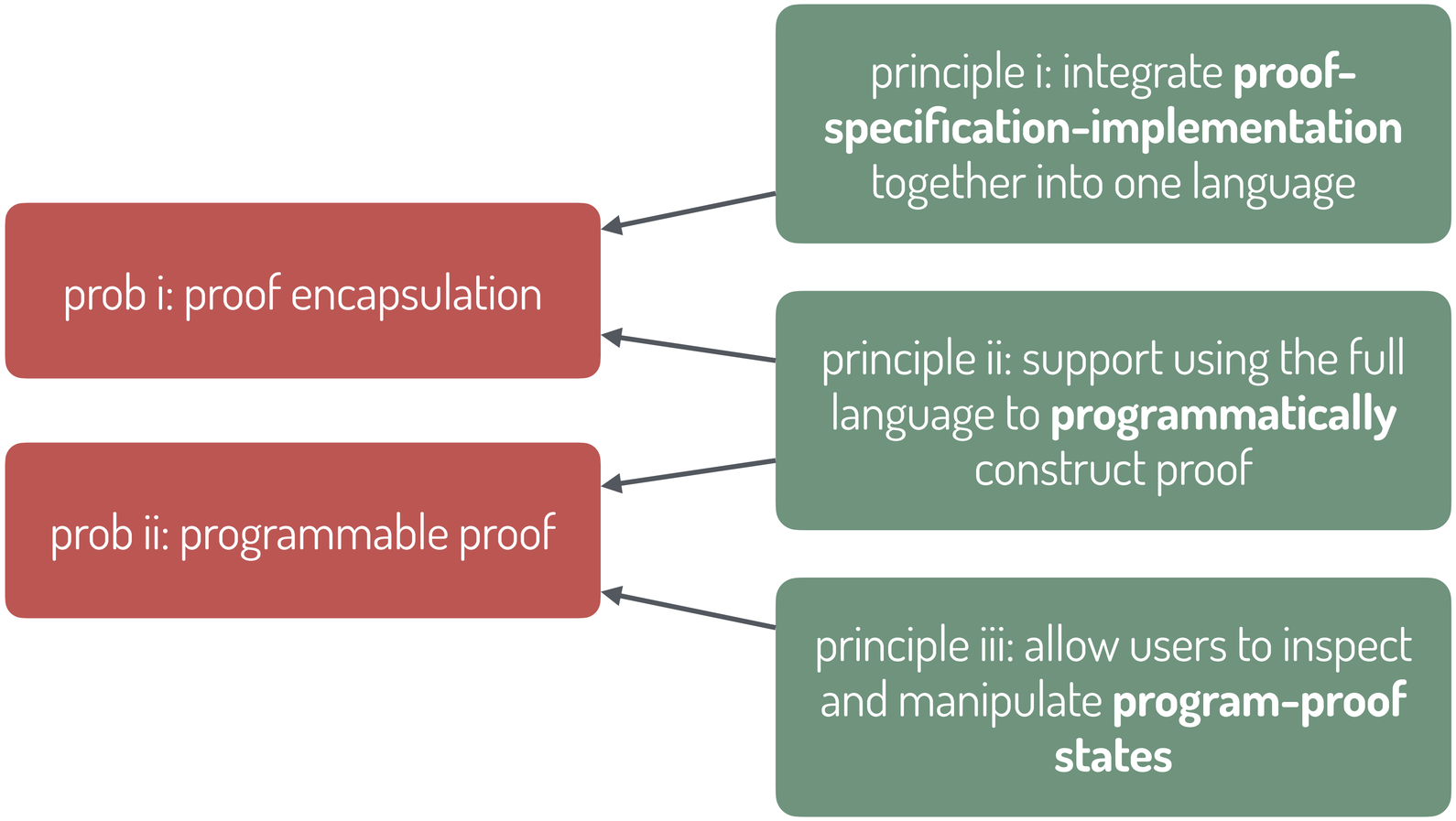
Unifying Programming and Verification in C



- **CStar reuses clang's toolchain**
  - use clang for compilation
  - use clangd for language server
- **its design is language-agnostic**

CStar

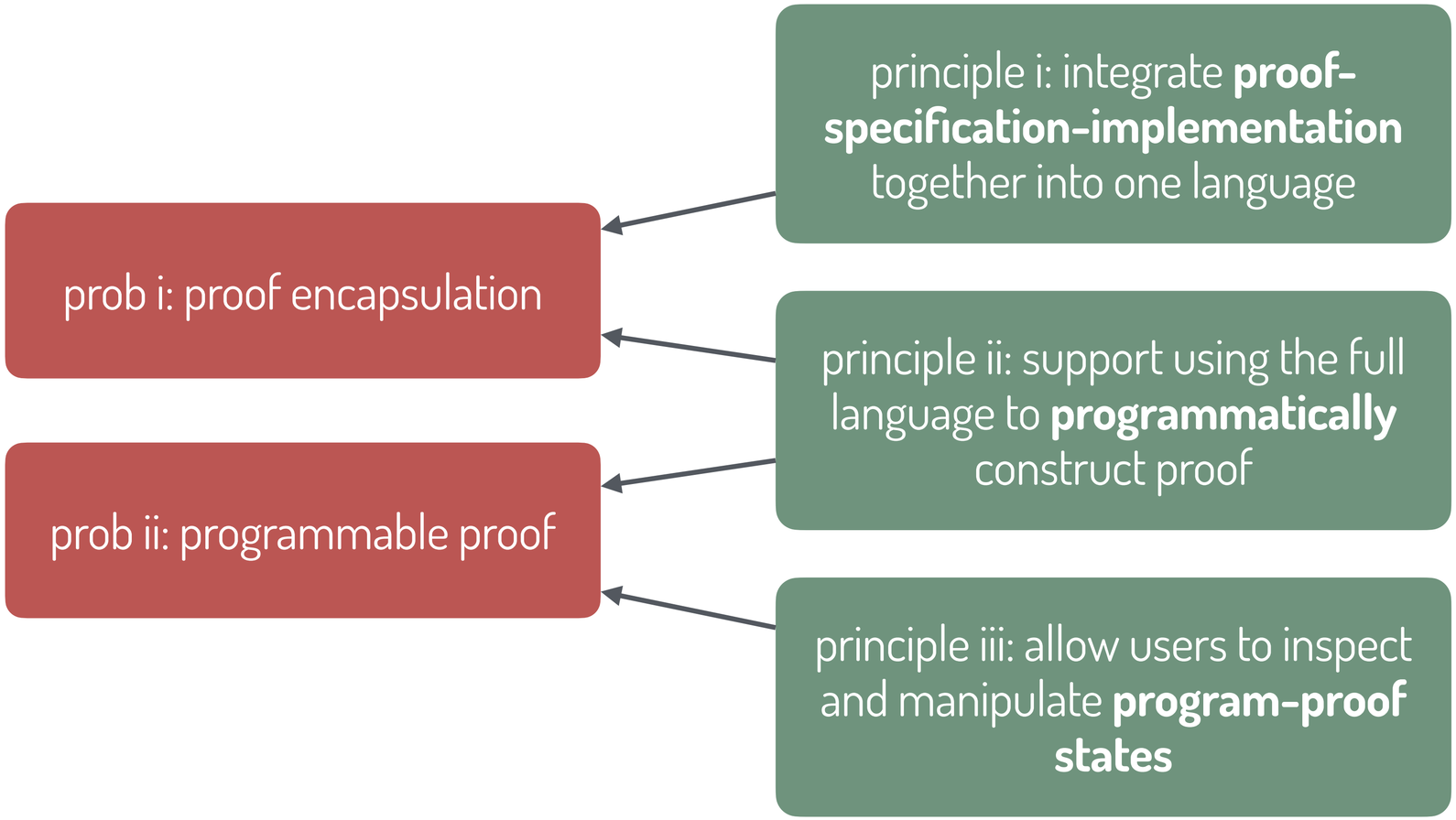
Unifying Programming and Verification in C



- **CStar reuses clang's toolchain**
  - use clang for compilation
  - use clangd for language server
- **its design is language-agnostic**
  - RustStar?

CStar

Unifying Programming and Verification in C



- **CStar reuses clang's toolchain**
  - use clang for compilation
  - use clangd for language server
- **its design is language-agnostic**
  - RustStar?
  - you own language-Star?

What are Still Missing

there are two looming "clouds"

What are Still Missing

there are two looming "clouds"

- minor one: the logical layer is functional, which is not C-style

What are Still Missing

there are two looming "clouds"

- minor one: the logical layer is functional, which is not C-style
  - the logical core is simply-typed lambda calculus, which is in many ways simpler than C

What are Still Missing

there are two looming "clouds"

- minor one: the logical layer is functional, which is not C-style
  - the logical core is simply-typed lambda calculus, which is in many ways simpler than C
- **major one: program-proof states are non-local**

What are Still Missing

there are two looming "clouds"

- minor one: the logical layer is functional, which is not C-style
  - the logical core is simply-typed lambda calculus, which is in many ways simpler than C
- **major one: program-proof states are non-local**
  - at every program point, there is **a single global state**

What are Still Missing

there are two looming "clouds"

- minor one: the logical layer is functional, which is not C-style
  - the logical core is simply-typed lambda calculus, which is in many ways simpler than C
- **major one: program-proof states are non-local**
  - at every program point, there is **a single global state**

```
for (; (order + 1) < pool->max_order; order++)
/*@ inv let p_i2 = (((integer) p) - __hyp_vmemmap) / 32 @*/
/*@ inv let Z = ZeroPage((pointer) ((p_i2 * 4096) - hyp_physvirt_offset), 1, order) @*/
/*@ inv let OP = Owned(pool) @*/
/*@ inv let hyp_vmemmap = (pointer) __hyp_vmemmap @*/
/*@ inv let start_i2 = (*pool).range_start / 4096 @*/
/*@ inv let end_i2 = (*pool).range_end / 4096 @*/
/*@ inv let off_i = hyp_physvirt_offset / 4096 @*/
/*@ inv let V2 = each (integer i; start_i2 <= i && i < end_i2)
                    {Owned<struct hyp_page>(hyp_vmemmap+(i*32)) } @*/
/*@ inv let p_page = V2.value[p_i2] @*/
/*@ inv let p_page_tweaked2 = (p_page){.order = order} @*/
/*@ inv each(integer i; start_i2 <= i && i < end_i2)
          {vmemmap_b_wf(i, hyp_vmemmap, V2.value[p_i2 = p_page_tweaked2], pool, *pool)} @*/
/*@ inv each(integer i; 0 <= i && i < ((*pool).max_order))
          {freeArea_cell_wf(i, hyp_vmemmap, V2.value, pool, (*pool))} @*/
/*@ inv hyp_pool_wf(pool, *pool, hyp_vmemmap, hyp_physvirt_offset) @*/
/*@ inv let R = each(integer i; start_i2 <= (i + off_i) && (i + off_i) < end_i2
                    && (V2.value[i + off_i]).refcount == 0
                    && (V2.value[i + off_i]).order != (hyp_no_order ()))
                  { ZeroPage(((pointer) 0) + (i*4096), 1, (V2.value[i+off_i]).order) } @*/
/*@ inv 0 <= order; order+1 <= (*pool).max_order @*/
/*@ inv cellPointer(hyp_vmemmap,32,start_i2,end_i2,p) @*/
/*@ inv (p_page.refcount) == 0; (p_page.order) == (hyp_no_order ()); (p_page.pool) == pool @*/
/*@ inv (p_page.node.next) == &(p->node); (p_page.node.prev) == &(p->node) @*/
/*@ inv order_aligned(p_i2,order) @*/
/*@ inv (p_i2 * 4096) + (page_size_of_order(order)) <= (*pool).range_end @*/
/*@ inv each(integer i; {p_i}@start < i && i < end_i2)
          ((({V.value[i]}@start).refcount == 0) || ((V2.value[i]) == {V.value[i]}@start)} @*/
/*@ inv {__hyp_vmemmap} unchanged; {hyp_physvirt_offset} unchanged; {pool} unchanged @*/
/*@ inv ({*pool}@start){.free_area = (*pool).free_area} == *pool @*/
{
    buddy = __find_buddy_avail(pool, p, order);
    if (!buddy)
            break;
    /*CN*/instantiate vmemmap_b_wf, hyp_page_to_pfn(buddy);
    /*CN*/unpack ZeroPage (hyp_page_to_virt(p), 1, order);
    /*CN*/unpack ZeroPage (hyp_page_to_virt(buddy), 1, order);
    /*CN*/lemma_attach_inc_loop(*pool, p, order);
    /*CN*/lemma2(hyp_page_to_pfn(p), order);
    /*CN*/lemma_page_size_of_order_inc(order);
    /*CN*/if ((buddy->node).next != &pool->free_area[order])
    /*CN*/  instantiate vmemmap_b_wf,
                        hyp_page_to_pfn(container_of((buddy->node).next, struct hyp_page, node));
    /*CN*/if ((buddy->node).prev != &pool->free_area[order])
    /*CN*/  instantiate vmemmap_b_wf,
                        hyp_page_to_pfn(container_of((buddy->node).prev, struct hyp_page, node));
    /*CN*/if ((buddy->node).prev != (buddy->node).next);
    list_del_init(&buddy->node);
    buddy->order = HYP_NO_ORDER;
    p = min(p, buddy);
    /*CN*/pack ZeroPage (hyp_page_to_virt(p), 1, order + 1);
}
```

excerpts from the verified `attach_page` function in CN

What are Still Missing

there are two looming "clouds"

- minor one: the logical layer is functional, which is not C-style
  - the logical core is simply-typed lambda calculus, which is in many ways simpler than C
- **major one: program-proof states are non-local**
  - at every program point, there is **a single global state**
  - it breaks modularity and readability of reasoning

```
for (; (order + 1) < pool->max_order; order++)
/*@ inv let p_i2 = (((integer) p) - __hyp_vmemmap) / 32 @*/
/*@ inv let Z = ZeroPage((pointer) ((p_i2 * 4096) - hyp_physvirt_offset), 1, order) @*/
/*@ inv let OP = Owned(pool) @*/
/*@ inv let hyp_vmemmap = (pointer) __hyp_vmemmap @*/
/*@ inv let start_i2 = (*pool).range_start / 4096 @*/
/*@ inv let end_i2 = (*pool).range_end / 4096 @*/
/*@ inv let off_i = hyp_physvirt_offset / 4096 @*/
/*@ inv let V2 = each (integer i; start_i2 <= i && i < end_i2)
                    {Owned<struct hyp_page>(hyp_vmemmap+(i*32)) } @*/
/*@ inv let p_page = V2.value[p_i2] @*/
/*@ inv let p_page_tweaked2 = (p_page){.order = order} @*/
/*@ inv each(integer i; start_i2 <= i && i < end_i2)
          {vmemmap_b_wf(i, hyp_vmemmap, V2.value[p_i2 = p_page_tweaked2], pool, *pool)} @*/
/*@ inv each(integer i; 0 <= i && i < ((*pool).max_order))
          {freeArea_cell_wf(i, hyp_vmemmap, V2.value, pool, (*pool))} @*/
/*@ inv hyp_pool_wf(pool, *pool, hyp_vmemmap, hyp_physvirt_offset) @*/
/*@ inv let R = each(integer i; start_i2 <= (i + off_i) && (i + off_i) < end_i2
                    && (V2.value[i + off_i]).refcount == 0
                    && (V2.value[i + off_i]).order != (hyp_no_order ()))
                  { ZeroPage(((pointer) 0) + (i*4096), 1, (V2.value[i+off_i]).order) } @*/
/*@ inv 0 <= order; order+1 <= (*pool).max_order @*/
/*@ inv cellPointer(hyp_vmemmap,32,start_i2,end_i2,p) @*/
/*@ inv (p_page.refcount) == 0; (p_page.order) == (hyp_no_order ()); (p_page.pool) == pool @*/
/*@ inv (p_page.node.next) == &(p->node); (p_page.node.prev) == &(p->node) @*/
/*@ inv order_aligned(p_i2,order) @*/
/*@ inv (p_i2 * 4096) + (page_size_of_order(order)) <= (*pool).range_end @*/
/*@ inv each(integer i; {p_i}@start < i && i < end_i2)
          ((({V.value[i]}@start).refcount == 0) || ((V2.value[i]) == {V.value[i]}@start)} @*/
/*@ inv {__hyp_vmemmap} unchanged; {hyp_physvirt_offset} unchanged; {pool} unchanged @*/
/*@ inv ({*pool}@start){.free_area = (*pool).free_area} == *pool @*/
{
    buddy = __find_buddy_avail(pool, p, order);
    if (!buddy)
            break;
    /*CN*/instantiate vmemmap_b_wf, hyp_page_to_pfn(buddy);
    /*CN*/unpack ZeroPage (hyp_page_to_virt(p), 1, order);
    /*CN*/unpack ZeroPage (hyp_page_to_virt(buddy), 1, order);
    /*CN*/lemma_attach_inc_loop(*pool, p, order);
    /*CN*/lemma2(hyp_page_to_pfn(p), order);
    /*CN*/lemma_page_size_of_order_inc(order);
    /*CN*/if ((buddy->node).next != &pool->free_area[order])
    /*CN*/  instantiate vmemmap_b_wf,
                        hyp_page_to_pfn(container_of((buddy->node).next, struct hyp_page, node));
    /*CN*/if ((buddy->node).prev != &pool->free_area[order])
    /*CN*/  instantiate vmemmap_b_wf,
                        hyp_page_to_pfn(container_of((buddy->node).prev, struct hyp_page, node));
    /*CN*/if ((buddy->node).prev != (buddy->node).next);
    list_del_init(&buddy->node);
    buddy->order = HYP_NO_ORDER;
    p = min(p, buddy);
    /*CN*/pack ZeroPage (hyp_page_to_virt(p), 1, order + 1);
}
```

excerpts from the verified `attach_page` function in CN

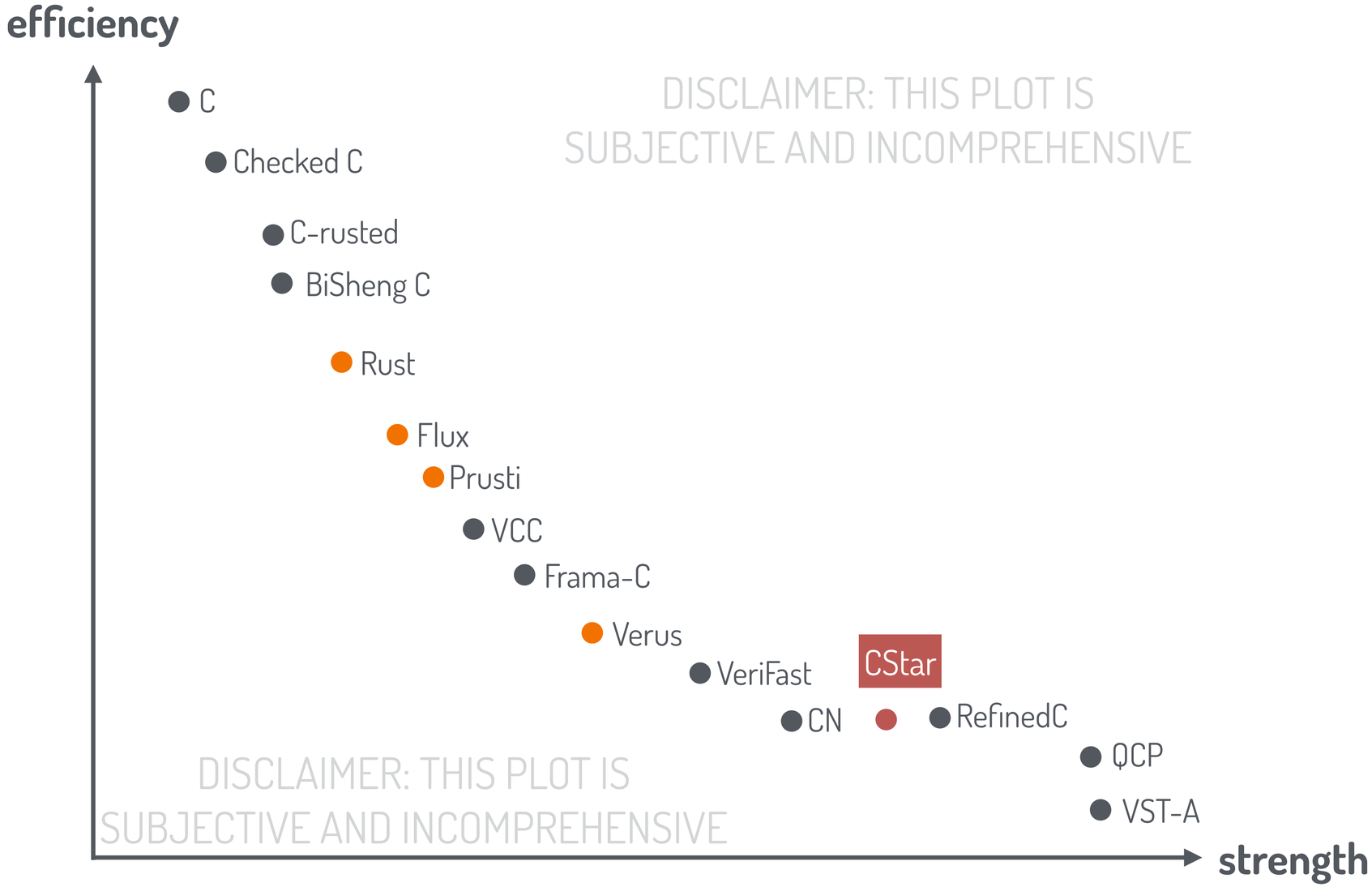
What are Still Missing

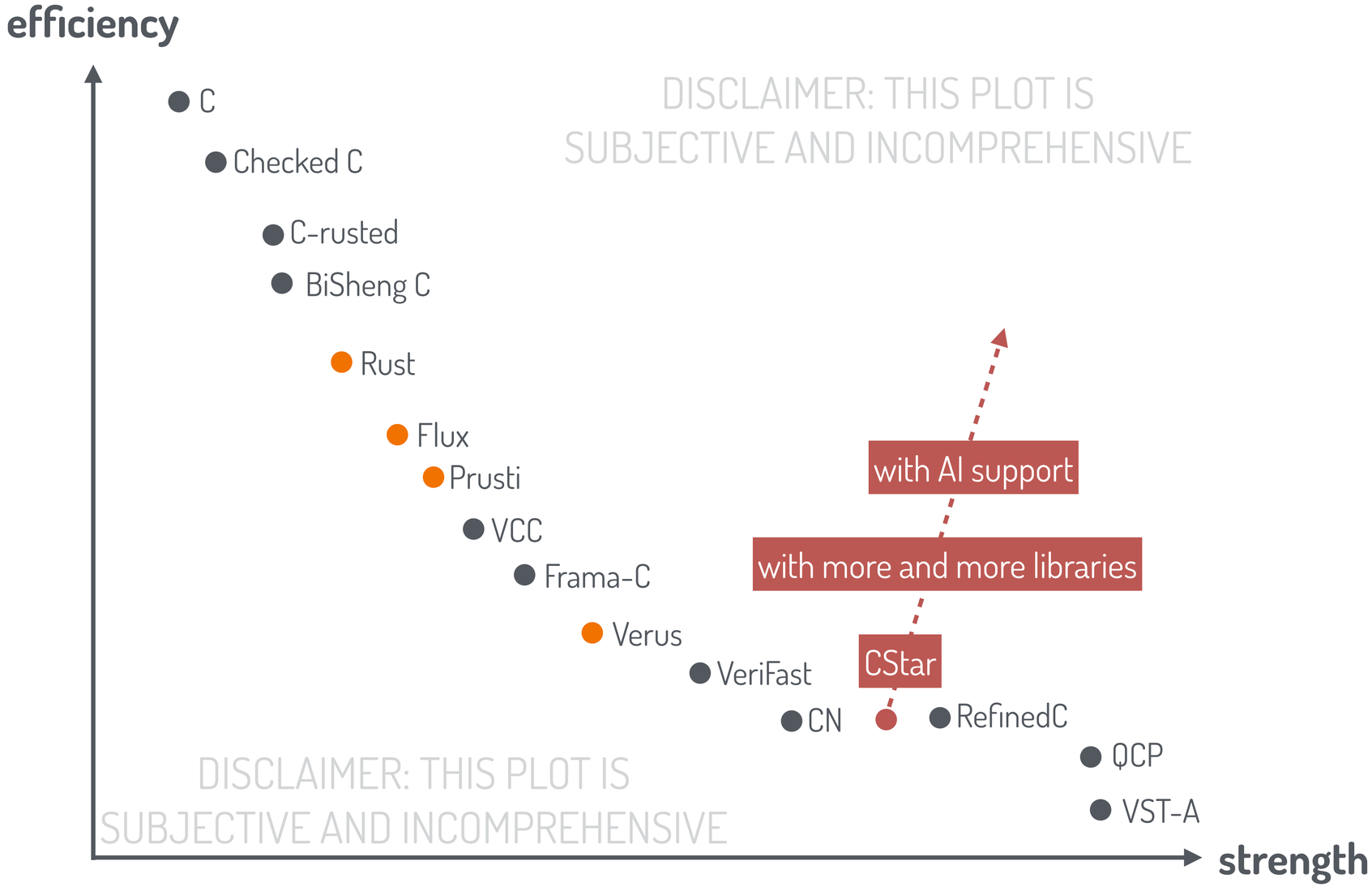
there are two looming "clouds"

- minor one: the logical layer is functional, which is not C-style
  - the logical core is simply-typed lambda calculus, which is in many ways simpler than C
- **major one: program-proof states are non-local**
  - at every program point, there is **a single global state**
  - it breaks modularity and readability of reasoning
  - ongoing work: program-proof states reified as **local capabilities**

```
for (; (order + 1) < pool->max_order; order++)
/*@ inv let p_i2 = (((integer) p) - __hyp_vmemmap) / 32 @*/
/*@ inv let Z = ZeroPage((pointer) ((p_i2 * 4096) - hyp_physvirt_offset), 1, order) @*/
/*@ inv let OP = Owned(pool) @*/
/*@ inv let hyp_vmemmap = (pointer) __hyp_vmemmap @*/
/*@ inv let start_i2 = (*pool).range_start / 4096 @*/
/*@ inv let end_i2 = (*pool).range_end / 4096 @*/
/*@ inv let off_i = hyp_physvirt_offset / 4096 @*/
/*@ inv let V2 = each (integer i; start_i2 <= i && i < end_i2)
                    {Owned<struct hyp_page>(hyp_vmemmap+(i*32)) } @*/
/*@ inv let p_page = V2.value[p_i2] @*/
/*@ inv let p_page_tweaked2 = (p_page){.order = order} @*/
/*@ inv each(integer i; start_i2 <= i && i < end_i2)
          {vmemmap_b_wf(i, hyp_vmemmap, V2.value[p_i2 = p_page_tweaked2], pool, *pool)} @*/
/*@ inv each(integer i; 0 <= i && i < ((*pool).max_order))
          {freeArea_cell_wf(i, hyp_vmemmap, V2.value, pool, (*pool))} @*/
/*@ inv hyp_pool_wf(pool, *pool, hyp_vmemmap, hyp_physvirt_offset) @*/
/*@ inv let R = each(integer i; start_i2 <= (i + off_i) && (i + off_i) < end_i2
                     && (V2.value[i + off_i]).refcount == 0
                     && (V2.value[i + off_i]).order != (hyp_no_order ()))
                   { ZeroPage(((pointer) 0) + (i*4096), 1, (V2.value[i+off_i]).order) } @*/
/*@ inv 0 <= order; order+1 <= (*pool).max_order @*/
/*@ inv cellPointer(hyp_vmemmap,32,start_i2,end_i2,p) @*/
/*@ inv (p_page.refcount) == 0; (p_page.order) == (hyp_no_order ()); (p_page.pool) == pool @*/
/*@ inv (p_page.node.next) == &(p->node); (p_page.node.prev) == &(p->node) @*/
/*@ inv order_aligned(p_i2,order) @*/
/*@ inv (p_i2 * 4096) + (page_size_of_order(order)) <= (*pool).range_end @*/
/*@ inv each(integer i; {p_i}@start < i && i < end_i2)
          ((({V.value[i]}@start).refcount == 0) || ((V2.value[i]) == {V.value[i]}@start)} @*/
/*@ inv {__hyp_vmemmap} unchanged; {hyp_physvirt_offset} unchanged; {pool} unchanged @*/
/*@ inv ({*pool}@start){.free_area = (*pool).free_area} == *pool @*/
{
    buddy = __find_buddy_avail(pool, p, order);
    if (!buddy)
            break;
    /*CN*/instantiate vmemmap_b_wf, hyp_page_to_pfn(buddy);
    /*CN*/unpack ZeroPage (hyp_page_to_virt(p), 1, order);
    /*CN*/unpack ZeroPage (hyp_page_to_virt(buddy), 1, order);
    /*CN*/lemma_attach_inc_loop(*pool, p, order);
    /*CN*/lemma2(hyp_page_to_pfn(p), order);
    /*CN*/lemma_page_size_of_order_inc(order);
    /*CN*/if ((buddy->node).next != &pool->free_area[order])
    /*CN*/  instantiate vmemmap_b_wf,
                        hyp_page_to_pfn(container_of((buddy->node).next, struct hyp_page, node));
    /*CN*/if ((buddy->node).prev != &pool->free_area[order])
    /*CN*/  instantiate vmemmap_b_wf,
                        hyp_page_to_pfn(container_of((buddy->node).prev, struct hyp_page, node));
    /*CN*/if ((buddy->node).prev != (buddy->node).next);
    list_del_init(&buddy->node);
    buddy->order = HYP_NO_ORDER;
    p = min(p, buddy);
    /*CN*/pack ZeroPage (hyp_page_to_virt(p), 1, order + 1);
}
```

excerpts from the verified `attach_page` function in CN


efficiency
DISCLAIMER: THIS PLOT IS
SUBJECTIVE AND INCOMPREHENSIVE
C
Checked C
C-rusted
BiSheng C
Rust
Flux
Prusti
VCC
Frama-C
Verus
VeriFast
CStar
CN
RefinedC
QCP
VST-A
DISCLAIMER: THIS PLOT IS
SUBJECTIVE AND INCOMPREHENSIVE
strength


efficiency
DISCLAIMER: THIS PLOT IS
SUBJECTIVE AND INCOMPREHENSIVE
C
Checked C
C-rusted
BiSheng C
Rust
Flux
Prusti
VCC
Frama-C
Verus
VeriFast
CN
CStar
RefinedC
QCP
VST-A
with AI support
with more and more libraries
DISCLAIMER: THIS PLOT IS
SUBJECTIVE AND INCOMPREHENSIVE
strength