Security-Typed Programming within Dependently-Typed Programming

Dan Licata
Joint work with Jamie Morgenstern

Carnegie Mellon University

Supported by NSF CCF-0702381 and CNS-0716469

Security-Typed Programming

- Access control: who gets access to what?
  read a file
  play a song
  make an FFI call

- Information flow: what can they do with it?
  post the file contents on a blog
  copy the mp3
  save the result in a database

Security-Typed Programming

- **Access control: who gets access to what?**
  read a file
  play a song
  make an FFI call

- Information flow: what can they do with it?
  post the file contents on a blog
  copy the mp3
  save the result in a database

Access Control

Alice

Read secret.txt

Server

Access control list (ACL) for secret.txt

Alice: r
Bob: rw

(slide by Kumar Avijit)

Access Control

Alice

Read secret.txt

Server

Access control list (ACL) for secret.txt

Alice: r
Bob: rw

Enforcement: Authentication + ACL lookup

(slide by Kumar Avijit)

Decentralized Access Control

Alice

Read paper.pdf

Digital library

- All students of members can read papers
- CMU is a member

CMU

- Alice is a student
- Charlie is a student
- …

(slide by Kumar Avijit)

Decentralized Access Control

Alice

Read paper.pdf

Digital library

- All students of members can read papers
- CMU is a member

CMU

- Alice is a student
- Charlie is a student
- …

Need a mechanism to specify and enforce decentralized policies...

(slide by Kumar Avijit)

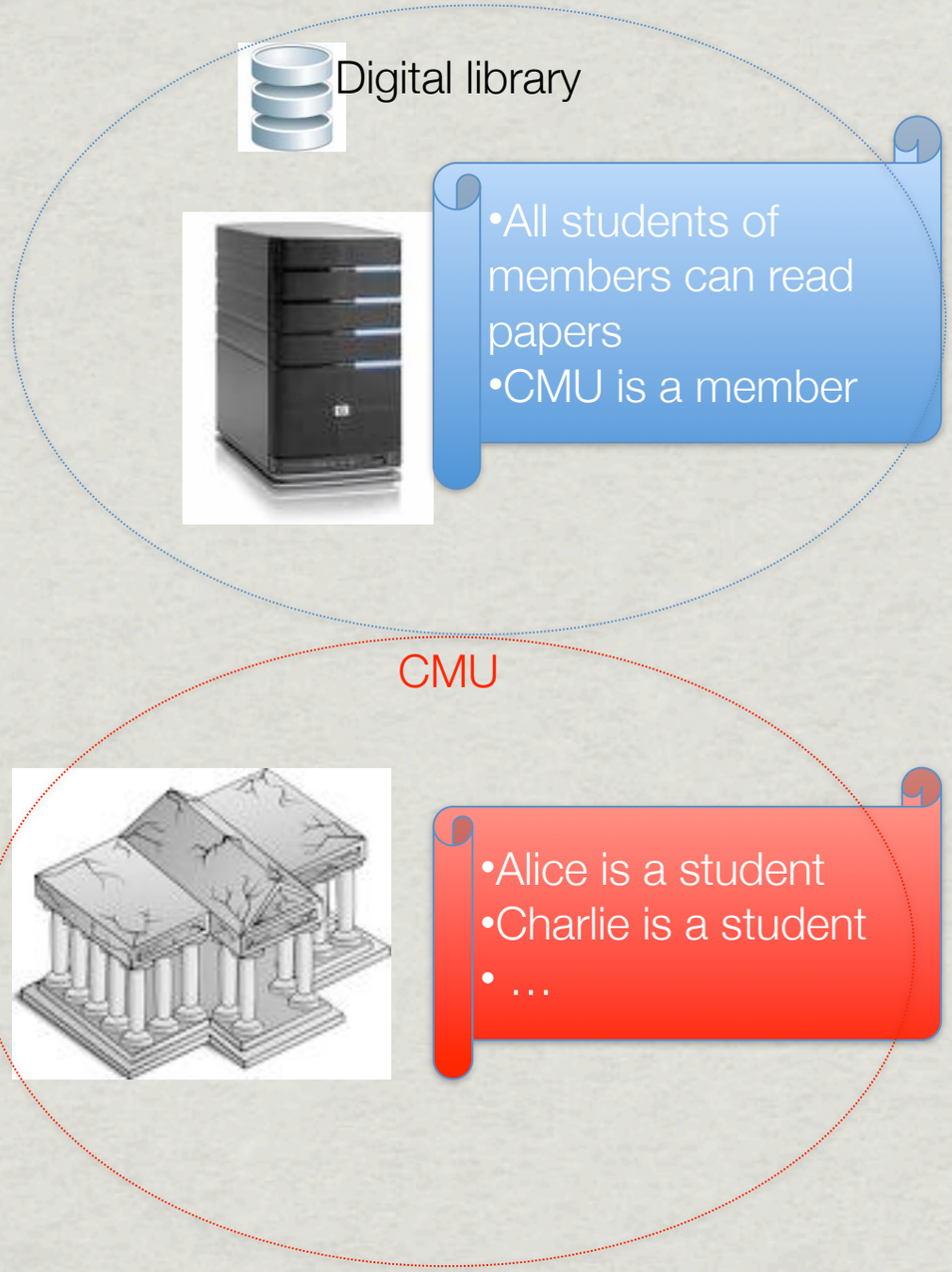
Decentralized Access Control

ACM **says** ∀ s:principal,
∀ i:principal,
∀ p:paper,
(member(i) ∧ i **says** student(s))
⊃ MayRead(s, p)

...

CMU **says** student(Alice)

...



(slide by Kumar Avijit)

Proof Carrying Authorization

[Appel+Felten]

Alice

Read "paper.pdf"

Digital library

- All students of members can read papers
- CMU is a member

CMU

- Alice is a student
- Charlie is a student
- …

(slide by Kumar Avijit)

Proof Carrying Authorization

[Appel+Felten]

Alice

Read “paper.pdf”

p : mayread(Alice,paper.pdf)

Digital library

- All students of members can read papers
- CMU is a member

CMU

- Alice is a student
- Charlie is a student
- …

(slide by Kumar Avijit)

Proof Carrying Authorization

Alice

Read “paper.pdf”

p : mayread(Alice,paper.pdf)

Proof OK?

Access granted

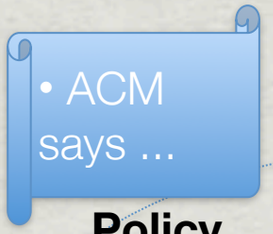
Runtime error

Proof Carrying Authorization

Alice

Read “paper.pdf”

p : mayread(Alice,paper.pdf)



Policy

Proof OK?

Access granted

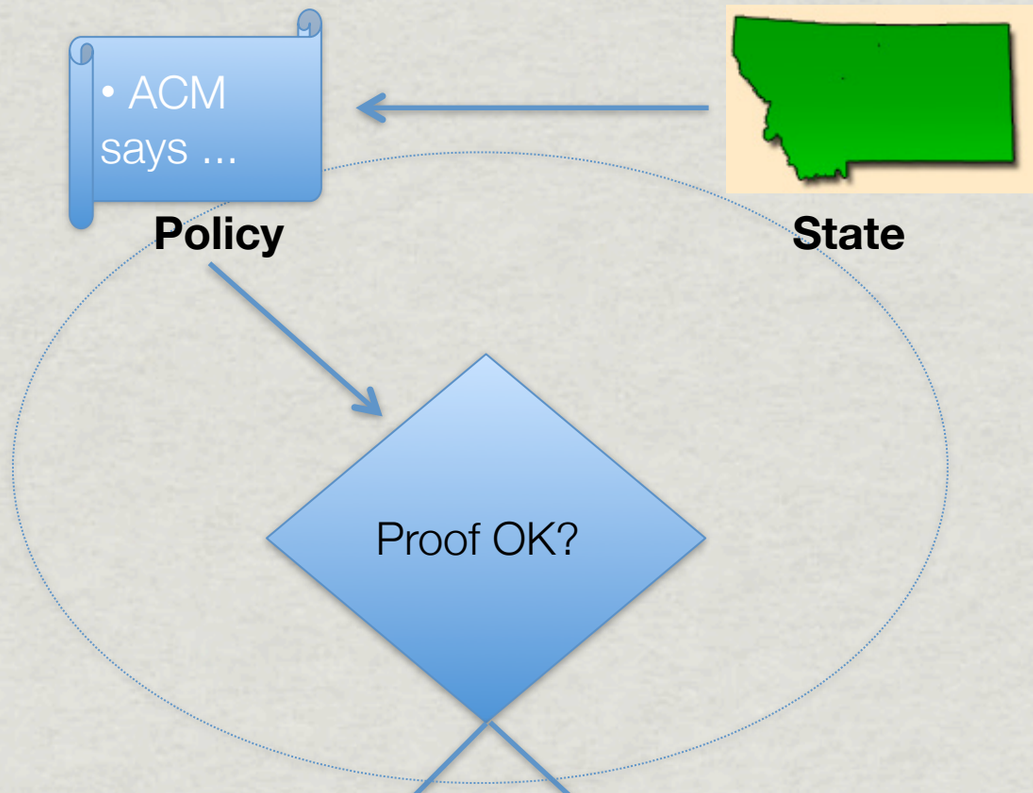
Runtime error

Proof Carrying Authorization

Alice

Read “paper.pdf”

p : mayread(Alice,paper.pdf)



Access granted

Runtime error

An API for PCA

Alice

Read “paper.pdf”

p : mayread(Alice,paper.pdf)

Digital library

An API for PCA

read : prin → file → proof → contents

Alice

Read “paper.pdf”

p : mayread(Alice,paper.pdf)

Digital library

An API for PCA

read : prin → file → proof → contents

Alice

Read “paper.pdf”

p : mayread(Alice,paper.pdf)

read(Alice,paper.pdf,p)

Digital library

An API for PCA

read : prin → file → proof → contents

e.g. read(Alice, paper.pdf,p)

Problems:

- p might not be a well-formed proof
- p might not be a proof of the right theorem!

Dependent Types!

read : prin → file → proof
→ contents

read : (k : prin) (f : file) (p : proof(mayread(k,f))
→ contents

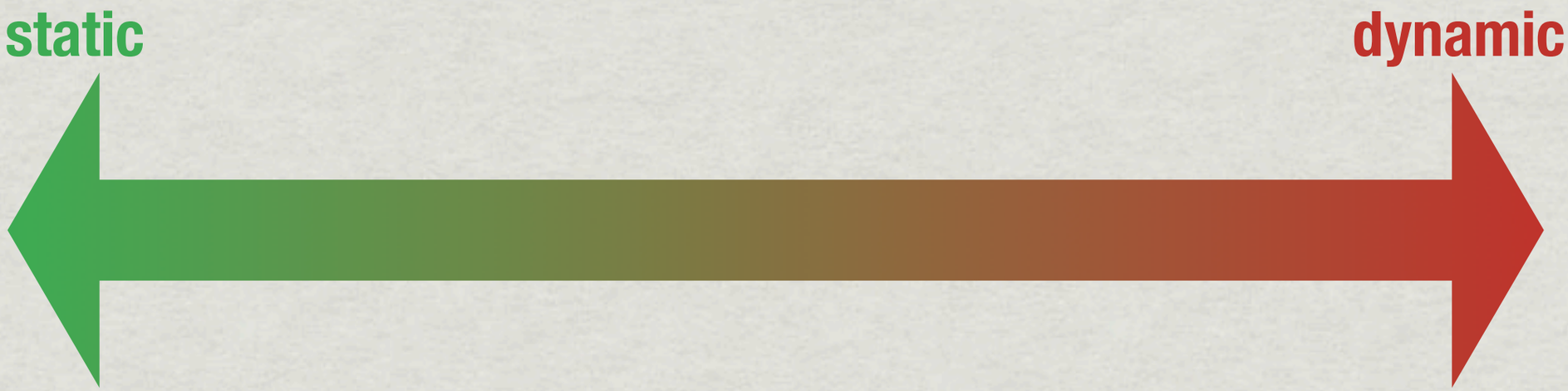
Dependent Types!

read : prin → file → proof
→ contents

read : (k : prin) (f : file) (p : proof(mayread(k,f))
→ contents

- typing ensures p is a well-formed proof
- theorem is explicit in p's type

Verification Spectrum



- Predict the policy
- Prove consequences statically
- Failures only if prediction was wrong

- Do all proving at run-time

Verification Spectrum

- Predict the policy
- Prove consequences statically
- Failures only if prediction was wrong

- Do all proving at run-time

Dependent PCA

Several new languages:

- PCML5 [Avijit,Datta,Harper, TLDI'10]
- Aura [Jia,Vaughan,Zdancewic,*et al.,* ICFP'08]
- Fine [Swamy,Chen,Chugh, ESOP'10]
- F7 [Gordon,Bengston,Bhargavan,Fournet,Maffeis, CSF'08]
- …

This paper:

We can do security-typed programming within an existing dependently-typed language

Our library

Supports programming as in

- **PCML5 [Avijit,Datta,Harper, TLDI'10]**
- **Aura [Jia,Vaughan,Zdancewic,*et al.,* ICFP'08]**
- **Fine [Swamy,Chen,Chugh, ESOP'10]**
- F7 [Gordon,Bengston,Bhargavan,Fournet,Maffeis, CSF'08]
- …

Aglet: Security-typed Programming in Agda

1. Representing an authorization logic

2. Compile-time and run-time theorem proving

3. Stateful and dynamic policies

Aglet: Security-typed Programming in Agda

1. **Representing an authorization logic**

2. Compile-time and run-time theorem proving

3. Stateful and dynamic policies

Dependent Types!

read : file → prin → proof
→ contents

read : (k : prin) (f : file) (p : proof(mayread(k,f))
→ contents

- typing ensures p is a well-formed proof
- theorem is explicit in p's type

Representing BL_0 [Garg+Pfenning]

CMU **says** student(Alice) → **says**(Prin CMU, student(Prin Alice))

Representing BL_0 [Garg+Pfenning]

CMU **says** student(Alice) → **says**(Prin CMU, student(Prin Alice))

```
data Propo where
  says : Principal → Propo → Propo
  ...
```

Representing BL_0 [Garg+Pfenning]

Sequent as indexed inductive definition:

$\Gamma \vdash A$  data _⊢_ : Ctx → Propo → Type

Representing BL_0 [Garg+Pfenning]

Sequent as indexed inductive definition:

$\Gamma \vdash A$ → `data _⊢_ : Ctx → Propo → Type`

Classifying *only* well-formed derivations:

$\mathcal{D}$ / $\Gamma \vdash A$ ⟷ $\mathcal{D} : \Gamma \vdash A$

Representing BL_0 [Garg+Pfenning]

Sequent as indexed inductive definition:

$\Gamma \vdash A$ → `data _⊢_ : Ctx → Propo → Type`

Classifying *only* well-formed derivations:

$$\overset{\mathcal{D}}{\Gamma \vdash A} \quad \leftrightarrow \quad \mathcal{D} : \Gamma \vdash A$$

Inference rules as datatype constructors:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B} \quad \rightarrow$$

```
⊃R : ∀ {Γ A B}
   → (A :: Γ) ⊢ B
   → Γ ⊢ (A ⊃ B)
```

Representing BL_0 [Garg+Pfenning]

Sequent as indexed inductive definition:

$\Gamma \vdash A$ → data _⊢_ : Ctx → Propo → Type

Classifying *only* well-formed derivations:

$$\mathcal{D} \quad \Gamma \vdash A \qquad \leftrightarrow \qquad \mathcal{D} : \Gamma \vdash A$$

Inference rules as datatype constructors:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B}$$

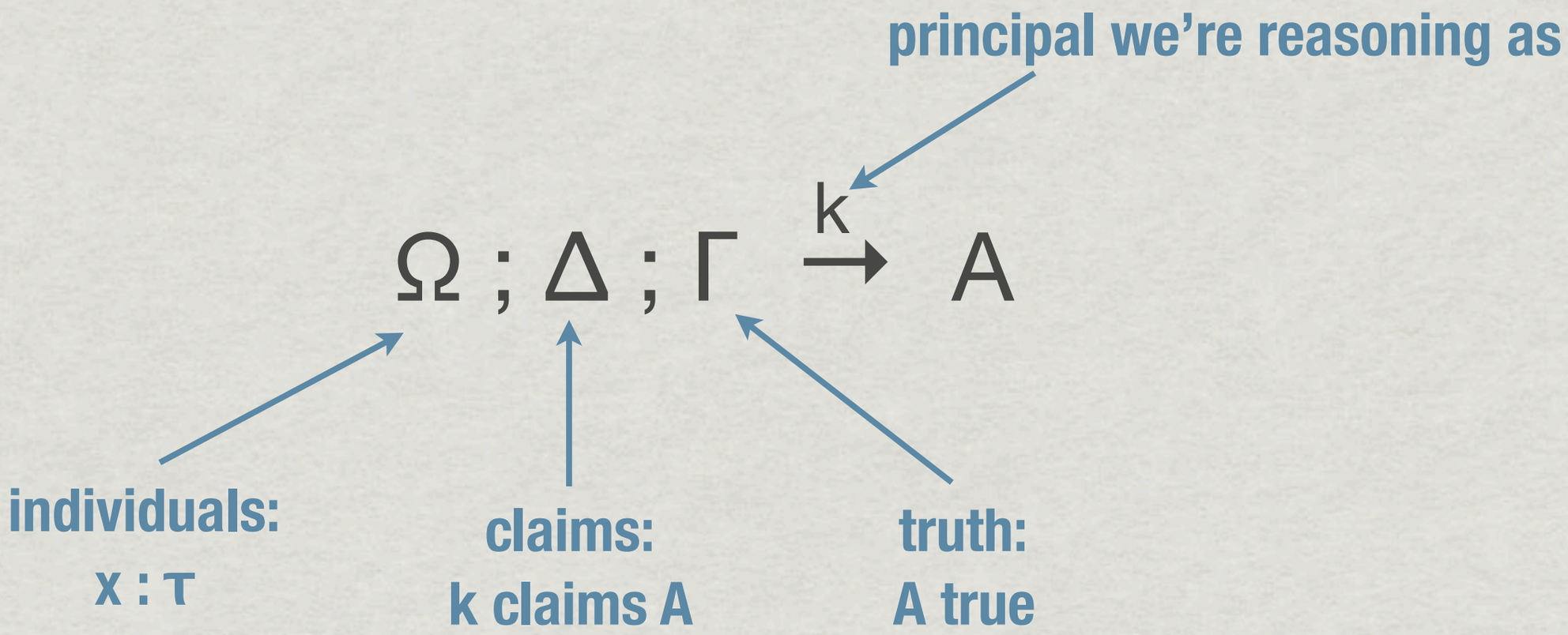
→

⊃R : ∀ {Γ A B}
→ (A :: Γ) ⊢ B
→ Γ ⊢ (A ⊃ B)

dependent de Bruijn indices

BL0 [Garg+Pfenning]

Logic with **says** modality: CMU **says** student(Alice)



```
⌜j = ⌜all as "Jamie"

jread : ◯ ⌜j String (λ    → ⌜j)
jread = read (‣ Prin "Jamie") (‣ File "secret·txt") ?

jreadprint : ◯ ⌜j Unit (λ _ → ⌜j)
jreadprint = jread >>= λ x →
             print ("the secret is: " ^ x)

drdprnt : ◯ (⌜all as "Dan") Unit (λ _ → ⌜all as "Dan")
drdprnt = sudo (‣ Prin "Dan" ) (‣ Prin "Jamie")
               (solve proveReplace)
               (λ _ → solve proveReplace)
               (solve (prove 15))
               jreadprint
```

```
⌜j = ⌜all as "Jamie"

jread : ◯ ⌜j String (λ    → ⌜j)
jread = read (‣ Prin "Jamie") (‣ File "secret·txt") ?

jreadprint : ◯ ⌜j Unit (λ _ → ⌜j)
jreadprint = jread >>= λ x →
             print ("the secret is: " ^ x)

drdprnt : ◯ (⌜all as "Dan") Unit (λ _ → ⌜all as "Dan")
drdprnt = sudo (‣ Prin "Dan" ) (‣ Prin "Jamie")
               (solve proveReplace)
               (λ _ → solve proveReplace)
               (solve (prove 15))
               jreadprint
```

Outline

1. Representing an authorization logic

2. Compile-time and run-time theorem proving

3. Stateful and dynamic policies

Theorem Prover

We implemented a *certified* theorem prover:

$$\text{prove} : (\Theta : \text{Ctx})\ (A : \text{Propo}) \to \text{Maybe}\ (\Theta \vdash A)$$

Theorem Prover

We implemented a *certified* theorem prover:

prove : **(n : nat)** (Θ : Ctx) (A : Propo) → Maybe (Θ ⊢ A)

search depth

Theorem Prover

We implemented a *certified* theorem prover:

prove : **(n : nat)** (Θ : Ctx) (A : Propo) → Maybe (Θ ⊢ A)

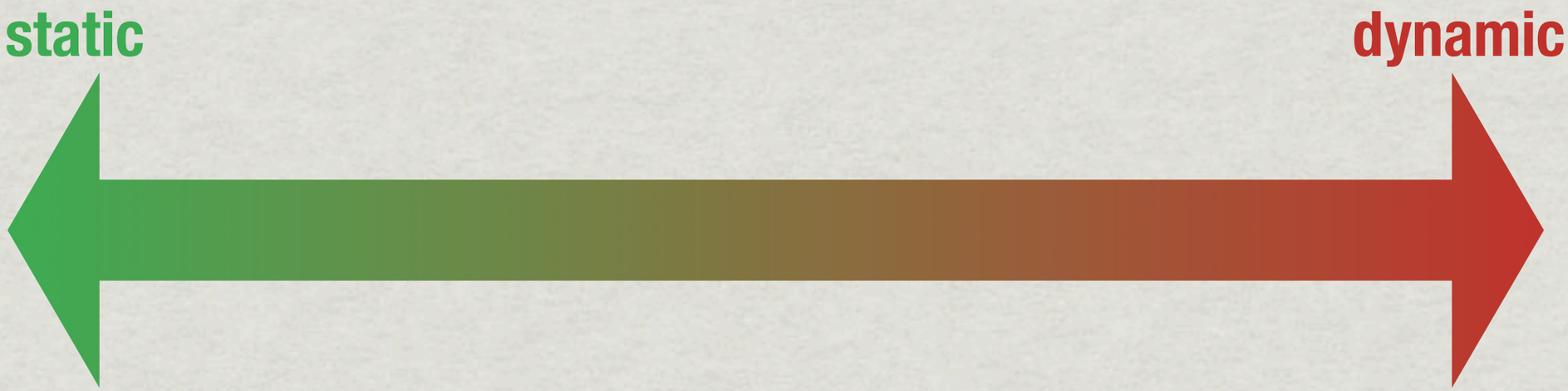
↑

search depth

Important that Propos are *inductive*!

```
data Propo where
  says : Principal → Propo → Propo
  ...
```

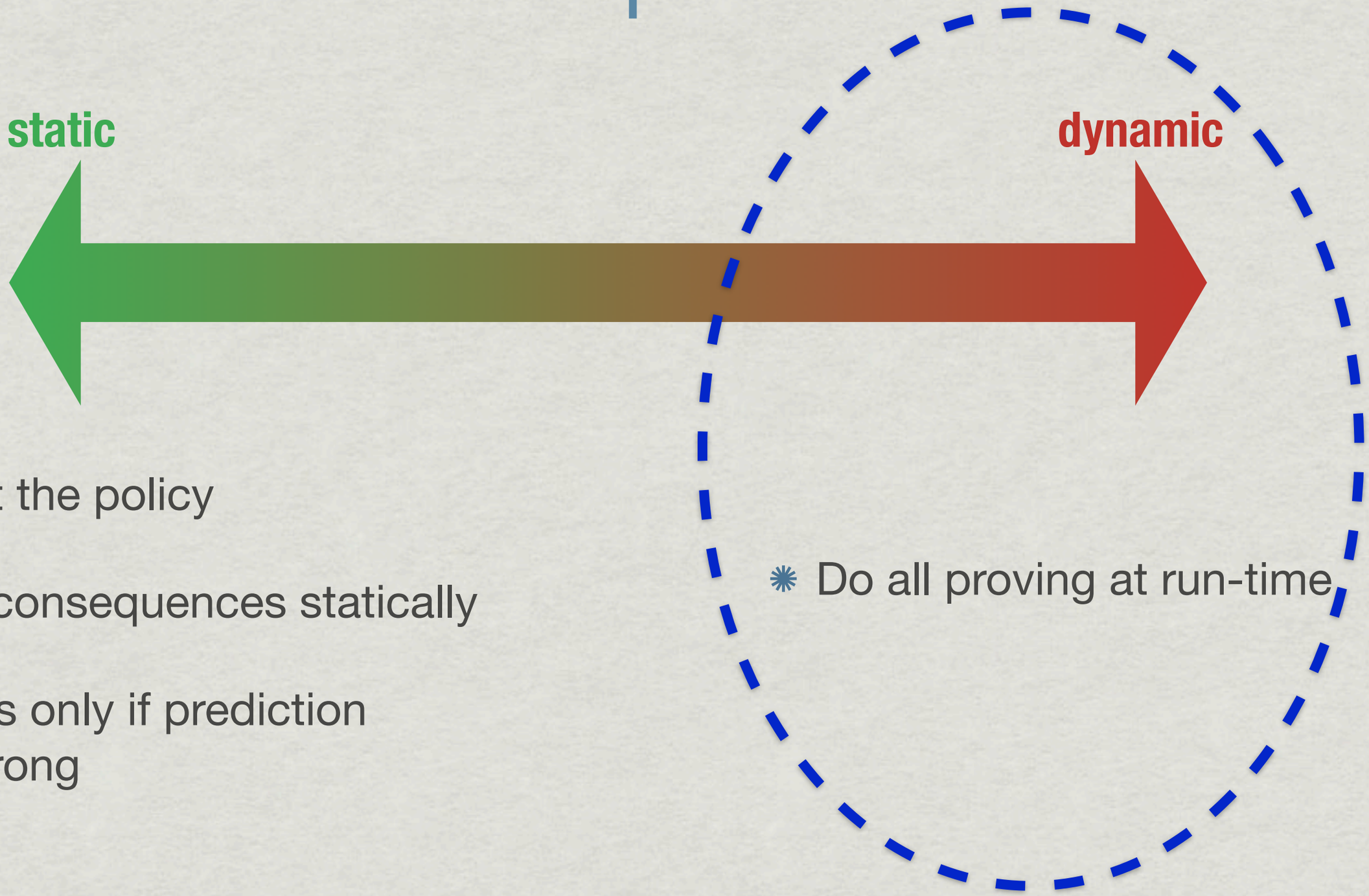
Verification Spectrum



- Predict the policy
- Prove consequences statically
- Failures only if prediction was wrong

- Do all proving at run-time

Verification Spectrum



- Predict the policy
- Prove consequences statically
- Failures only if prediction was wrong

- Do all proving at run-time

Run-time Proving

prove : (n:nat) (Θ : Ctx) (A : Prop) → Maybe (Θ ⊢ A)

```
tryRead : Ctx → Prin → File → Maybe(String)
tryRead Γ k f = case (prove 15 Γ Mayread(f,p)) of
                  None → None
                  Some proof → Some (read k f proof)
```

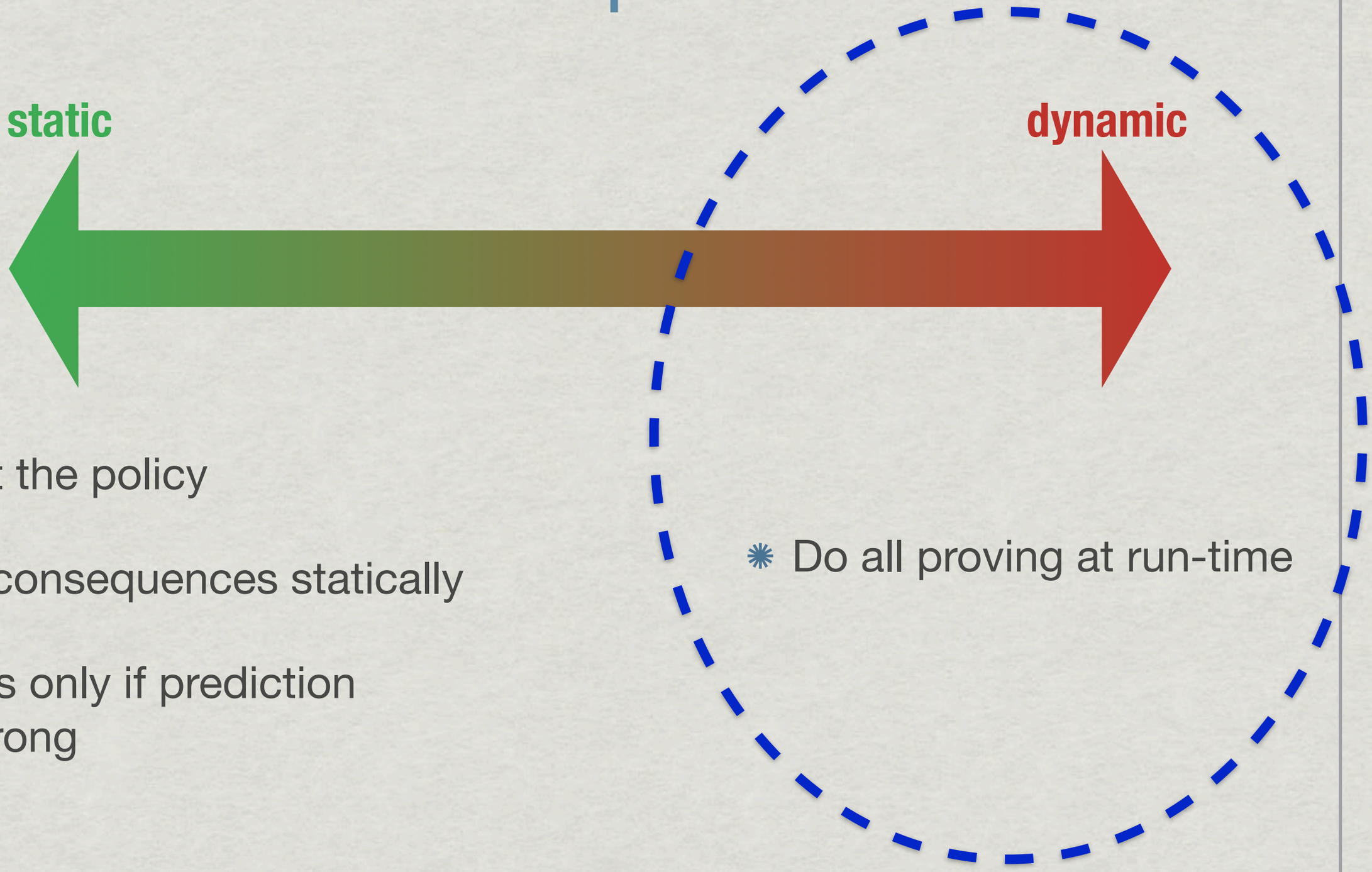
Run-time Proving

prove : (n:nat) (Θ : Ctx) (A : Prop) → Maybe (Θ ⊢ A)

```
tryRead : Ctx → Prin → File → Maybe(String)
tryRead Γ k f = case (prove 15 Γ Mayread(f,p)) of
                  None → None
                  Some proof → Some (read k f proof)
```

use prove like “look up in ACL”

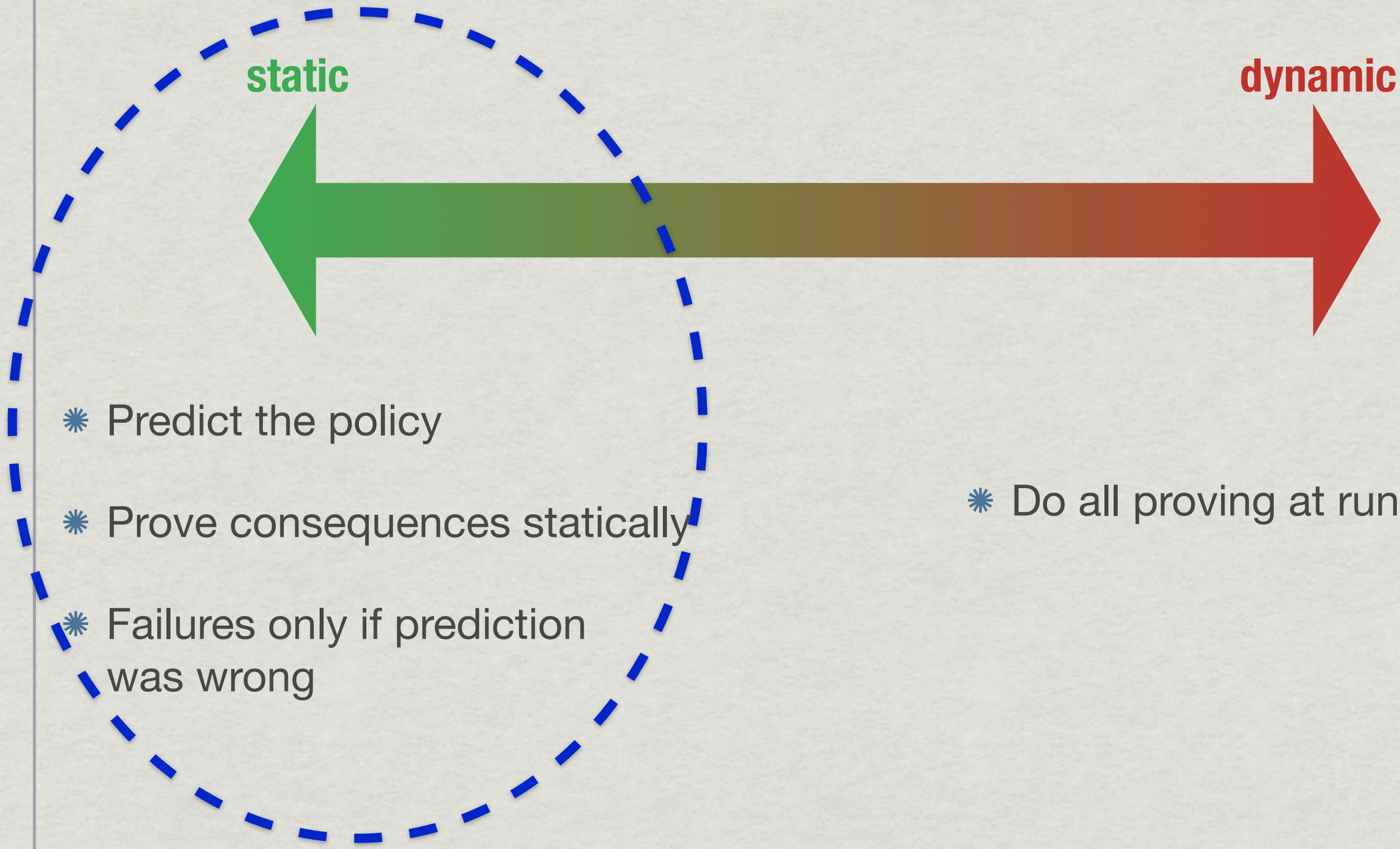
Verification Spectrum



- Predict the policy
- Prove consequences statically
- Failures only if prediction was wrong

- Do all proving at run-time

Verification Spectrum



- Predict the policy
- Prove consequences statically
- Failures only if prediction was wrong

- Do all proving at run-time

Compile-time Proving

For Γpol a static (known at compile-time) policy:

Γpol = CMU says student(Alice) ::
ACM says A :: ...

For a call

read(Alice, paper.pdf, ?)

can verify *at compile-time* that ? can be filled in

Compile-time Proving

Γpol = 

proof? : Maybe (Γpol ⊢ Mayread(Alice, paper.pdf))

proof? = prove 15 Γpol (Mayread(Alice, paper.pdf))

Compile-time Proving

Γpol = 

proof? : Maybe (Γpol ⊢ Mayread(Alice, paper.pdf))
proof? = prove 15 Γpol (Mayread(Alice, paper.pdf))

- proof? computes to either None or Some(pf)

Compile-time Proving

Γpol = 

proof? : Maybe (Γpol ⊢ Mayread(Alice, paper.pdf))
proof? = prove 15 Γpol (Mayread(Alice, paper.pdf))

- proof? computes to either None or Some(pf)
- run *at compile-time* and get value out

Compile-time Proving

ML/Haskell:

valOf : Maybe A → A

run-time error if it's None

Compile-time Proving

ML/Haskell:

valOf : Maybe A → A

run-time error if it's None

Agda:

valOf : (s : Maybe A) → ? → A

only well-typed if s is equal to Some(pf)

Compile-time Proving

```
IsSome : ∀ {A} → Maybe A → Type
IsSome (Some _) = Unit
IsSome None     = Void
```

Compile-time Proving

```
IsSome : ∀ {A} → Maybe A → Type
IsSome (Some _) = Unit
IsSome None     = Void
```

```
valOf : ∀ {A} → (s : Maybe A) → IsSome s → A
valOf (Some x) _ = x
valOf None     (v                      ) = ?
```

Compile-time Proving

```
IsSome : ∀ {A} → Maybe A → Type
IsSome (Some _) = Unit
IsSome None     = Void
```

```
valOf : ∀ {A} → (s : Maybe A) → IsSome s → A
valOf (Some x) _ = x
valOf None     (v : IsSome None) = ?
```

Compile-time Proving

```
IsSome : ∀ {A} → Maybe A → Type
IsSome (Some _) = Unit
IsSome None     = Void
```

```
valOf : ∀ {A} → (s : Maybe A) → IsSome s → A
valOf (Some x) _ = x
valOf None     (v :      Void      ) = ?
```

Compile-time Proving

```
IsSome : ∀ {A} → Maybe A → Type
IsSome (Some _) = Unit
IsSome None     = Void
```

```
valOf : ∀ {A} → (s : Maybe A) → IsSome s → A
valOf (Some x) _ = x
valOf None     (v :      Void      ) = impossibe v
```

Compile-time Proving

theProof : Γpol ⊢ Mayread(Alice, paper.pdf)
theProof = valOf proof? <>

Given

valOf : ∀ {A} → (s : Maybe A) → IsSome s → A

proof? : Maybe (Γpol ⊢ Mayread(Alice, paper.pdf))

Compile-time Proving

theProof : Γpol ⊢ Mayread(Alice, paper.pdf)
theProof = valOf proof? <>

Agda type error if theorem prover fails

Given

valOf : ∀ {A} → (s : Maybe A) → IsSome s → A

proof? : Maybe (Γpol ⊢ Mayread(Alice, paper.pdf))

Outline

1. Representing an authorization logic

2. Compile-time and run-time theorem proving

3. Stateful and dynamic policies

Read with policy

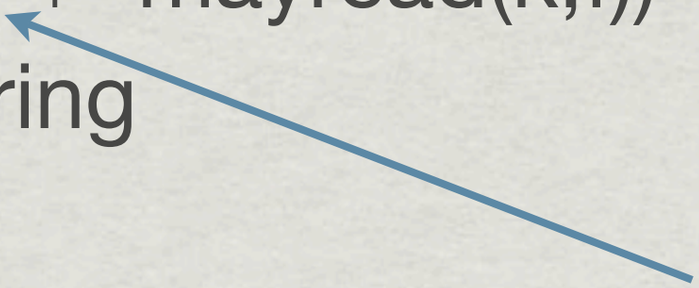
read : (f : file) (k : prin)
(p : Γ ⊢ mayread(k,f))
→ string

Read with policy

read : (f : file) (k : prin)
(p : Γ ⊢ mayread(k,f))
→ ◯ string

Read with policy

read : (f : file) (k : prin)
(p : Γ ⊢ mayread(k,f))
→ ◯ string



represents the policy; where does it come from?

Read with policy

read : (f : file) (k : prin)
(p : Γ ⊢ mayread(k,f))
→ ◯ string

represents the policy; where does it come from?

Want policies to be:

- dynamic: not known until run-time
- stateful: can change during execution (chown)

Indexed Monad

Represent computations with a type

[cf. HTT]

Indexed Monad

Represent computations with a type

[cf. HTT]

read : (f : file) (k : prin) (p : Γ ⊢ mayread(k,f))→ ◯ Γ string Γ

Indexed Monad

Represent computations with a type

[cf. HTT]

read : (f : file) (k : prin) (p : Γ ⊢ mayread(k,f)) → ◯ Γ string Γ

chown : (f : file) (k1 k2 : prin)
(p : (Γ,owns(k1,f)) ⊢ maychown(k1,f))
→ ◯ (Γ,owns(k1,f)) string (Γ,owns(k2,f))

Indexed Monad

$$\begin{aligned} \mathsf{read} : {} & (f : \mathsf{file})\ (k : \mathsf{prin}) \\ & (p : \Gamma \vdash \mathsf{mayread}(k, f)) \\ \rightarrow {} & \bigcirc\ \Gamma\ \mathsf{string}\ \Gamma \end{aligned}$$

Indexed Monad

read : (f : file) (k : prin)
(p : Γ ⊢ mayread(k,f))
→ ◯ Γ string Γ

need to track who you're running as [AH07]

Indexed Monad

running as k

```
read : (f : file) (k : prin)
       (p : Γ ⊢ mayread(k,f) & as(k))
  →  ◯ Γ string Γ
```

Indexed Monad

running as k

```
read : (f : file) (k : prin)
       (p : Γ ⊢ mayread(k,f) & as(k))
  →  ◯ Γ string Γ
```

```
sudo : (f : file) (k1 k2 : prin)
     →  Γ,as(k1) ⊢ maysu(k1,k2)
     →  ◯ (Γ,as(k2)) C (Γ',as(k2))
     →  ◯ (Γ,as(k1)) C (Γ',as(k1))
```

More examples

- file access control (more details)
- located computation
- combination with information flow
- conference management server with several phases (submission, reviewing, notification, …)

Summary

Can do security-typed programming within DTP

- Indexed inductive definition to represent proofs
- Theorem prover to discharge proof obligations, run at compile-time and run-time
- Indexed monad to manage stateful+dynamic policies

Feature Requests

How could a DTPL better support this application?

- Speed or interface to theorem provers

- Reflection (prover works well at extremes but not in the middle) 

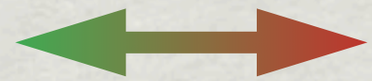
- Binding+scope (logic)

Feature Requests

How could a DTPL better support this application?

- Speed or interface to theorem provers

  term repr. / [Brady et al.] **[Kariso]**

- Reflection (prover works well at extremes but not in the middle) 

- Binding+scope (logic)

Feature Requests

How could a DTPL better support this application?

- Speed or interface to theorem provers

  term repr. / [Brady et al.] **[Kariso]**

- Reflection (prover works well at extremes but not in the middle)

  quoteGoal

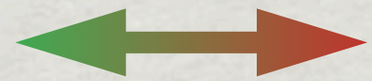
- Binding+scope (logic)

Feature Requests

How could a DTPL better support this application?

- Speed or interface to theorem provers

  term repr. / [Brady et al.] **[Kariso]**

- Reflection (prover works well at extremes  but not in the middle)

  quoteGoal

- Binding+scope (logic)

  Dan's thesis, coming next month

Thanks for listening!

code at http://www.cs.cmu.edu/~drl

Summary

Can do security-typed programming within DTP

- Indexed inductive definition to represent proofs
- Theorem prover to discharge proof obligations, run at compile-time and run-time
- Indexed monad to manage stateful+dynamic policies