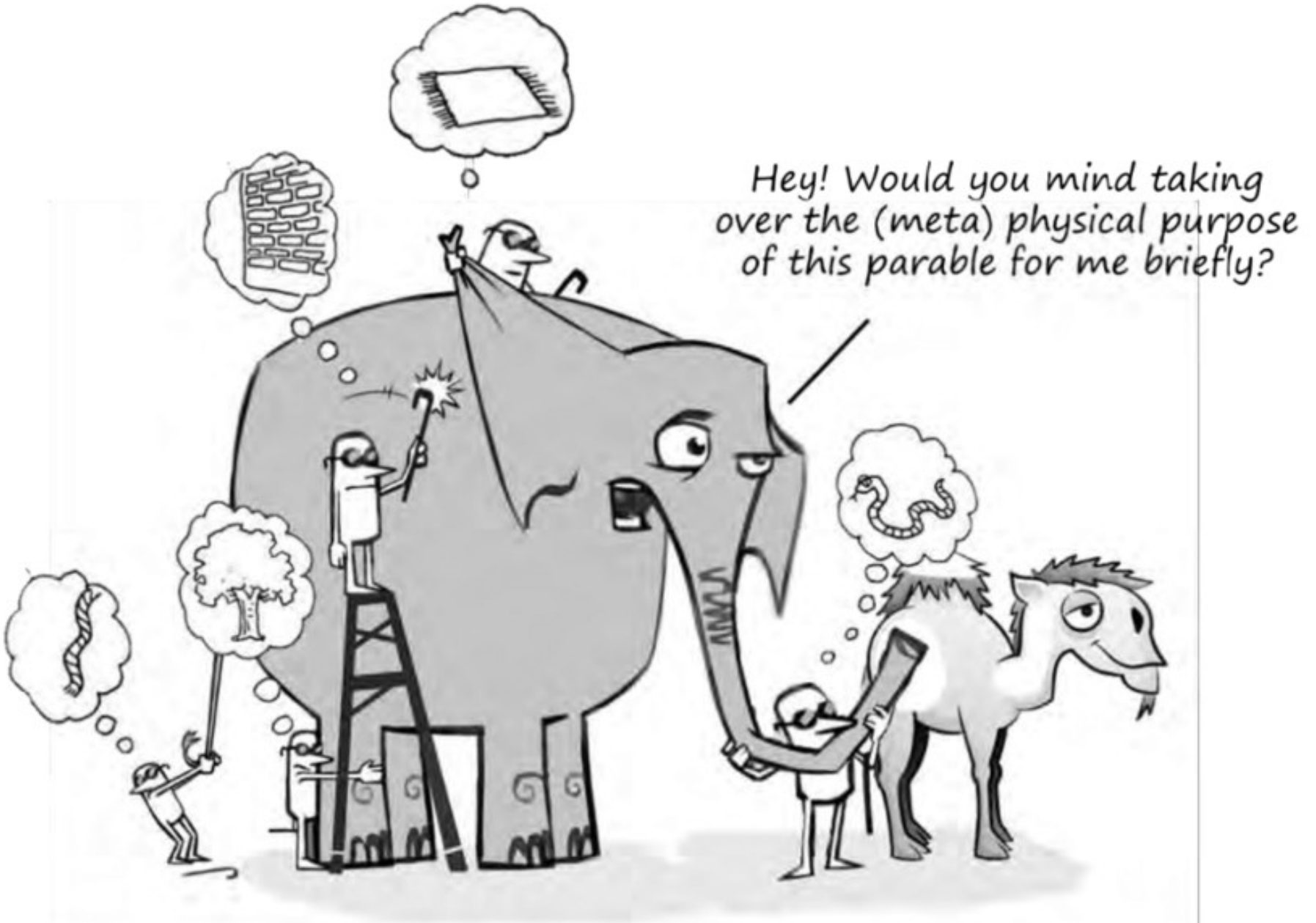
Towards Automatic Structured Inference of Module Abstraction

Festkolloquium on the occasion of
the 65th birthday of Prof. Dr. Wolfgang Reif
June 6 -- Augsburg University

Gidon Ernst
Marian Lingsch-Rosenfeld

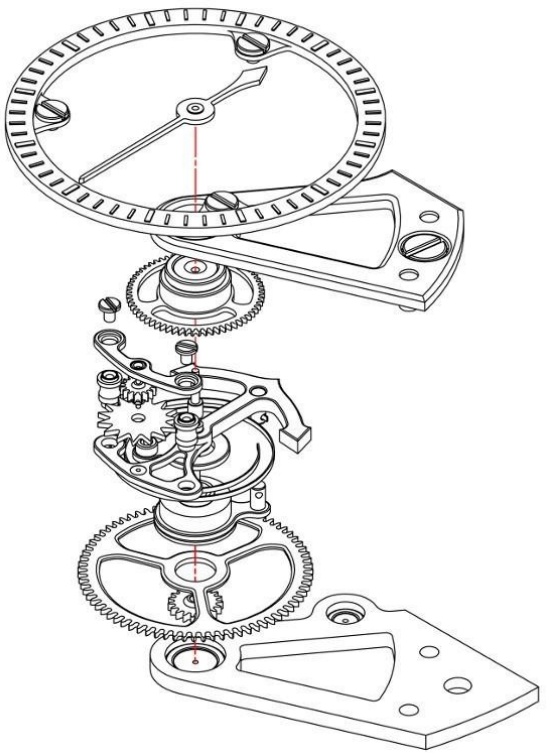
LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

SoSy-Lab Software Systems


Hey! Would you mind taking over the (meta) physical purpose of this parable for me briefly?

Software Verification (Analogy)

technical system



laws of physics

specification

$$\frac{d\ hour}{d\ minute} = \frac{1}{60}$$

Software Verification

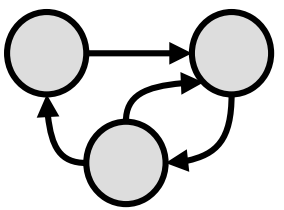
software system

```
 = p->pSrc;
t = &pSrc->a[0];
nt = &pLeft[1];
i=0; i<pSrc->nSrc-1; i++, pRight++, pLeft++){
ole *pRightTab = pRight->pTab;
t isOuter;

( NEVER(pLeft->pTab==0 || pRightTab==0) ) con
Outer = (pRight->fg.jointype & JT_OUTER)!=0;
```

laws of programming

specification

`read(write(x)) = x`



$O(n)$

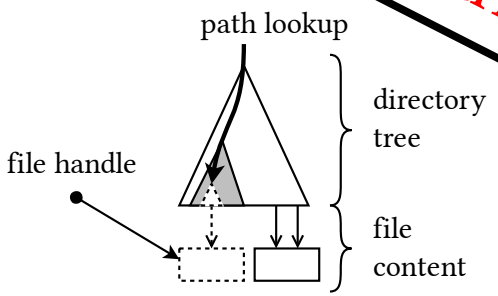
abstraction = trust

- easy to understand
- precise, unambiguous
- application-specific

Example: Flashix [Reif+, 2009…]

challenges: abstraction and modularity

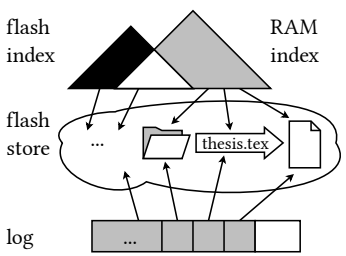
Dynamic Logic

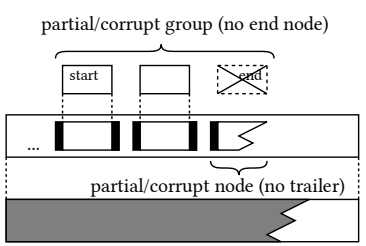


Theory: step-wise refinement
with power-cuts and concurrency

Practical: >15KLoC formal models and proofs
follow-up: trustworthy code generator

3 PhDs defended, ~20 papers published





Example: Array List in Dafny

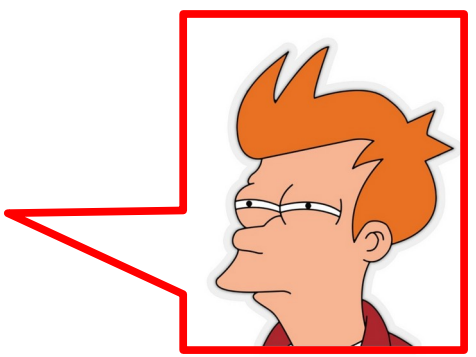

```
class ArrayList<T> {
  var data: array<T>
  var length: int
  ghost var content: seq<T>

  method add(last: T)
    ensures  content == old(content) + [last]
    requires invariant()
    ensures  invariant()


}
```

- behavior: easy & intuitive
- implementation as well as abstraction mechanism is private to the class

Example: Array List in Dafny


```
class ArrayList<T> {
  var data: array<T>
  var length: int
  ghost var content: seq<T>

  method add(last: T)
    ensures  content == old(content) + [last]
    requires invariant()
    ensures  invariant()

  predicate invariant() { ... }
}
```

- behavior: easy & intuitive
- implementation as well as abstraction mechanism is private to the class



Actually...

Software Engineers:
"formal = difficult & strange"

```
class ArrayList<T> {
  var data: array<T>
  var length: int
  ❊❊❑▲▼ ❖❀❒ ❋❑■▼❊■▼✚ ▲❊❑✣✶✝

  method add(last: T)
    ❊■▲◆❒❊▲ ❋❑■▼❊■▼ †† ❑●❊✈❋❑■▼❊■▼✉ ☞ ✻●❀▲▼✻
    ❒❊❑◆❉❒❊▲ ❉■❖❀❒❉❀■▼✈✉
    ❊■▲◆❒❊▲ ❉■❖❀❒❉❀■▼✈✉

  ❒❒❊❊❉❋❀▼❊ ❉■❖❀❒❉❀■▼✈✉ ❛ ✎✎✎ ❝
}
```

Example: Specification Inference


```
method Abs(x: int)
    returns (y: int)




{
    if 0 ⩽ x then {
        y := x;
    } else {
        y := -x;
    }
}
```

Example: Specification Inference


```
method Abs(x: int)
    returns (y: int)
    requires true
    ensures 0 ≤ x ⟹ y == x
    ensures x < 0 ⟹ y == -x
{
    if 0 ≤ x then {
        y := x;
    } else {
        y := -x;
    }
}
```

Contribution: An Approach for the Inference of Module Abstractions

behavioral (= executable) implementation

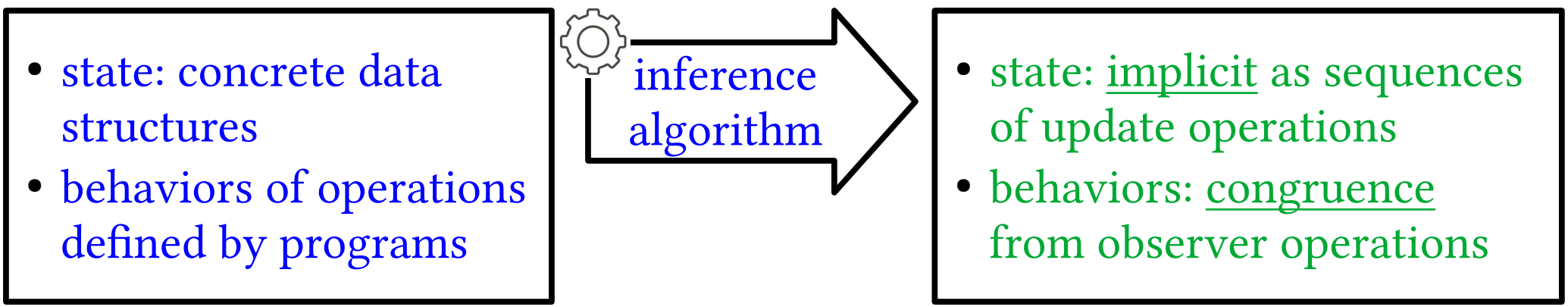
- state: concrete data structures
- behaviors of operations defined by programs

Framework: Structured Algebraic Specifications [Reif+ 98, ...]

Contribution: An Approach for the Inference of Module Abstractions

behavioral (= executable) implementation

axiomatic (= abstract) characterization



Framework: Structured Algebraic Specifications [Reif+ 98, ...]

✎ Behavioral Equivalence

“Observable” module (🐘) [Goguen 99]

- `type St` state representation
- `obs: In x St → Out` observer operations
- `upd: In x St → St` update operations

} behavioral signature

✎ Behavioral Equivalence

"Observable" module (🐘) [Goguen 99]

- `type St` — state representation
- `obs: In x St → Out` — observer operations
- `upd: In x St → St` — update operations

} behavioral signature

Theorem [Bidoit & Hennicker 99]: Unique model class when

states are identified by the observations one can make

$$st_1 = st_2 \iff \forall\ in.\ obs(in, st_1) = obs(in, st_2)$$

observations can be explained from prior states via lemmas

$$obs(in, upd(in', st)) = \chi(in, in', obs(\_, st))$$

Example: Stacks

behavioral (= executable) implementation

```
type St⟨α⟩ := List⟨α⟩

empty()       := []
push(x, xs)   := x :: xs
pop([])       := []
pop(x :: xs)  := xs
top([])       := None
top(x :: xs)  := Some(x)
```

Example: Stacks

behavioral (= executable) implementation

```
type St⟨α⟩ := List⟨α⟩

empty()       := []
push(x, xs)   := x :: xs
pop([])       := []
pop(x :: xs)  := xs
top([])       := None
top(x :: xs)  := Some(x)
```

axiomatic (= abstract) characterization

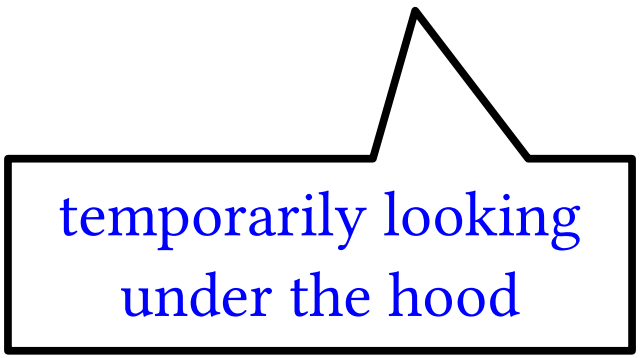
```
type St⟨α⟩ generated by
empty, push, pop

top(empty())      == ???
top(push(x, st)) == ???
top(pop(st))      == ???

st₁ == st₂
  ⟺ top(st₁) == top(st₂)
```

Developing the Axiomatization

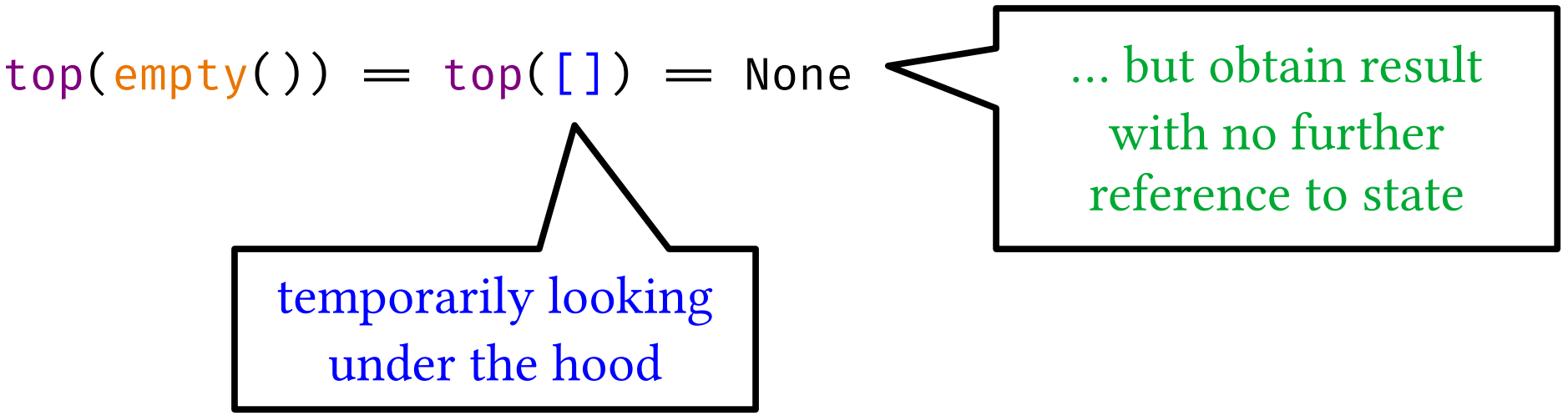

```
top(empty()) == ???
```

Developing the Axiomatization


```
top(empty()) == top([]) == ???
```



Developing the Axiomatization



Developing the Axiomatization


```
top(empty()) == top([]) == None

top(push(x, xs)) == top(x :: xs) == Some(x)

top(pop(xs))
  == ???
```

Developing the Axiomatization


```
top(empty()) = top([]) = None

top(push(x, xs)) = top(x :: xs) = Some(x)

top(pop(xs))
  = match xs
      case []      ⇒ None
      case y :: ys ⇒ top(ys)
```

discerns cases
to apply Def. pop

Developing the Axiomatization


```
top(empty()) == top([]) == None

top(push(x, xs)) == top(x :: xs) == Some(x)

top(pop(xs))
  == match xs
     case []      ⇒ None
     case y :: ys ⇒ top(ys)
```

discerns cases to apply Def. pop

not an acceptable lemma

- top is too weak observer
- results in bad equivalence

Inference of Module Abstractions

Given: module implementation (🐘)

Goal: find nice characterization of possible behaviors

Contribution

✎	Structured:	theoretical foundations
⚙	Automatic:	provide an algorithm
🚧	“Towards”:	lemma inference as key building block

✎ Adding Expressiveness

Idea 1: "reify" problematic updates as data

```
data Γ := □ | Pop(Γ)
```

Idea 2: introduce generalized observers

```
top*(□, st)       ⇜⇝ top(st)
top*(Pop(c), st) ⇜⇝ top*(c, pop(st))
```

↝ defines `top*` during lemma discovery

↜ defines `top` in the axiomatic spec.

Expectation: lemmas for such observers "more feasible"

✎ Adding Expressiveness

Idea 1: "reify" problematic updates as data

```
data Γ := □ | Pop(Γ)
```

Idea 2: introduce generalized observers

```
top*(□, st)       ⇜⇝ top(st)
top*(Pop(c), st) ⇜⇝ top*(c, pop(st))
```

Expectation: lemmas for such observers "more feasible"

Consequence: behavioral equivalence now strong enough

```
st₁ = st₂
  ⟺ ∀ c. top*(c, st₁) = top*(c, st₂)
```

Proposition: suffices to work with generalized observers only

Completing the Axiomatization


```
top*(Pop(c), push(x, st))
  == top*(c, st)
```

via the implementation:
`pop(push(x, xs)) == xs`

```
top*(c, pop(st))
  == top*(Pop(c), st)
```

by construction

```
top*(c, empty())
  == None

top*(□, push(x, st))
  == Some(x)
```

"obvious"

Inference of Module Abstractions

Given: module implementation ()

Goal: find nice characterization of possible behaviors

Contribution

 Structured: theoretical foundations

 Automatic: provide an algorithm

 “Towards”: lemma inference as key building block

Inference Algorithm

Input System as Behavioral Algebraic Specification

Output Axiomatic Characterization + Signature Extension

State in *i*-th refinement loop

- reified operations `data Γ_i := □ | ...`
- unsolved equations `obs*(in, c, upd(in', st)) == ???`
- discovered lemmas `... == χ(in, in', obs*(_, st))`

Oracles

- discovery mechanism for lemmas
- calculation of global well-founded order

Inference of Module Abstractions

Given: module implementation (🐘)

Goal: find nice characterization of possible behaviors

Contribution

- ✎ Structured: theoretical foundations
- ⚙ Automatic: provide an algorithm
- 🚧 "Towards": lemma inference as key building block

🚧 Lemma Inference as Key Building Block 30/35

Task: Given `lhs(x) == ???` find `rhs` so that `lhs(x) == rhs(x)`

Avoid: Overly-complex and redundant `rhs`

Classic approach: term enumeration [Buchberger, Claessen, ...]

- *huge* combinatorial search space `rhs ∈` Terms(vars=x,size=k)

🚧 Lemma Inference as Key Building Block 31/35

Task: Given `lhs(x) == ???` find `rhs` so that `lhs(x) == rhs(x)`

Avoid: Overly-complex and redundant `rhs`

Classic approach: term enumeration [Buchberger, Claessen, ...]

- *huge* combinatorial search space `rhs ∈ Terms(vars=x,size=`k`)`

Idea [Ernst+]: calculate with functions [Burstall, Darlington 77])

- need new methods for comparison, e.g., recursive unification
- Insights (not here): much more efficient, lemmas of "good" shape

Example: Lemmas from Fusion [Wadler 88] 32/35

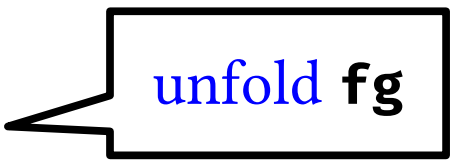
Let

```
fg(c) := top*(c, empty())
```

Example: Lemmas from Fusion [Wadler 88] 33/35

Let `fg(c) := top*(c, empty())`

```
fg(c) == top*(c, empty())
      == match c
           case □          ⇒ top*(□, empty()) == None
           case Pop(c')    ⇒  ...
```



Example: Lemmas from Fusion [Wadler 88] 34/35

Let `fg(c) := top*(c, empty())`

```
fg(c) = top*(c, empty())
      = match c
          case □          ⇒ top*(□, empty()) = None
          case Pop(c')    ⇒ top*(c', pop(empty()) = fg(c')
```

unfold **fg**

fold **fg**

Now easy to recognize that **fg** is constant `None`

Summary

Contribution:

Approach to infer axiomatic abstractions from implementations

Outlook

- implementation and experiments
- demonstrate lemma discovery as a key enabler for novel reasoning

Big thanks to you, Prof. Reif, for your inspiration and guidance
during the Elite-SE master and during the PhD.

We wish you a happy birthday and lots of interesting research yet to come!