QuickChecking Static Analysis Properties

Jan Midtgaard and Anders Møller

Abstract Interpretation Winter School

From forthcoming ICST'15 paper

DTU

Who checks the checker?

DTU

Many static analyses build on...

Theorem (Tarski 1955). Given

- a complete lattice $\langle L; \sqsubseteq \rangle$ and
- a monotone (order-preserving) function $f : L \to L$,

then the fixed points of f form a complete lattice.

In particular f has a least and a greatest fixed point.

Many static analyses build on. . .

Theorem (Tarski 1955). Given

- a complete lattice $\langle L; \sqsubseteq \rangle$ and
- a monotone (order-preserving) function $f : L \to L$,

then the fixed points of f form a complete lattice.

In particular f has a least and a greatest fixed point.

Software tools depend on these premises.

DTU

Many static analyses build on...

Theorem (Tarski 1955). Given

- □ a complete lattice $\langle L; \sqsubseteq \rangle$ and
- □ a monotone (order-preserving) function $f : L \to L$,

then the fixed points of f form a complete lattice.

In particular f has a least and a greatest fixed point.

Software tools depend on these premises.

How do we ensure them?

DTU

A range of approaches…

for ensuring static analysis properties:

- basic testing
- informal monotonicity arguments
- pen-and-paper monotonicity/soundness proofs
- …
- static monotonicity analysis (Murawski-Yi:VMCAI02)
- automated fixed point checking (Blazy-Laporte-Maroneze-Pichardie:SAS13)
- machine-formalized proofs (Cachera-Jensen-Pichardie-Rusu:TCS05)

DTU

This work (read: cat out of the bag)

Check essential properties (lattice, monotonicity, . . .)

- □ with QuickCheck (Claessen-Hughes:ICFP’00) and
- □ an embedded DSL

for increased confidence in your static analysis code.

DTU

This work (read: cat out of the bag)

Check essential properties (lattice, monotonicity, . . .)

- with QuickCheck (Claessen-Hughes:ICFP'00) and
- an embedded DSL

for increased confidence in your static analysis code.

Pro: lightweight,
effective in terms of coverage

DTU

This work (read: cat out of the bag)

Check essential properties (lattice, monotonicity, . . .)

- with QuickCheck (Claessen-Hughes:ICFP'00) and
- an embedded DSL

for increased confidence in your static analysis code.

Pro: lightweight,
effective in terms of coverage

Con: ***confidence***,
not proof

DTU

A simple example (1/3)

A two-element lattice expressed as an OCaml module:

```
module L = struct
  let name = "example lattice"
  type elem = Top | Bot
  let leq a b = match a,b with
    | Bot, _ -> true
    | _, Top -> true
    | _, _   -> false
  let join e e' = if e = Bot then e' else Top
  let meet e e' = if e = Bot then Bot else e'
  (* ... *)

  let to_string e = if e = Bot then "Bot" else "Top"
end
```

DTU

A simple example (1/3)

A two-element lattice expressed as an OCaml module:

```
module L = struct
  let name = "example lattice"
  type elem = Top | Bot
  let leq a b = match a,b with
    | Bot, _ -> true
    | _, Top -> true
    | _, _   -> false
  let join e e' = if e = Bot then e' else Top
  let meet e e' = if e = Bot then Bot else e'
  (* ... *)
  let arb_elem = Arbitrary.among [Bot; Top]
  let to_string e = if e = Bot then "Bot" else "Top"
end
```

We extend the lattice with a **generator**
of arbitrary lattice elements.

DTU

A simple example (2/3)

Using a reusable functor of lattice property tests

```
# let module LTests = GenericTests(L) in
  run_tests LTests.suite;;
```

DTU

A simple example (2/3)

Using a reusable functor of lattice property tests

```
# let module LTests = GenericTests(L) in
  run_tests LTests.suite;;
  check 19 properties...
testing property leq reflexive in example lattice...
  [✓] passed 1000 tests (0 preconditions failed)
testing property leq transitive in example lattice...
  [✓] passed 1000 tests (0 preconditions failed)
testing property leq anti symmetric in example lattice...
  [✓] passed 1000 tests (0 preconditions failed)
testing property bot is lower bound in example lattice...
  [✓] passed 1000 tests (0 preconditions failed)
testing property join commutative in example lattice...
  [✓] passed 1000 tests (0 preconditions failed)

... (28 additional lines cut)...

tests run in 0.02s
[✓] Success! (passed 19 tests)
```

we can quickcheck `L` for a range of lattice properties!

DTU

A simple example (3/3)

– and there's more!

Suppose we have a lattice operation:

```
(*  flip : L -> L  *)
let flip e = if e = L.Bot then L.Top else L.Bot
```

Note: This guy is **not** monotone.

DTU

A simple example (3/3)

– and there's more!

Suppose we have a lattice operation:

```
(*  flip : L -> L  *)
let flip e = if e = L.Bot then L.Top else L.Bot
```

Note: This guy is **not** monotone.

Let's test it:

```
# let flip_desc = ("flip",flip) in
  run (testsig (module L) -<-> (module L) =:  flip_desc);;
```

A simple example (3/3)

– and there's more!

Suppose we have a lattice operation:

```
(*  flip : L -> L  *)
let flip e = if e = L.Bot then L.Top else L.Bot
```

Note: This guy is **not** monotone.

Let's test it:

```
# let flip_desc = ("flip",flip) in
  run (testsig (module L) -<-> (module L) =:  flip_desc);;
  testing property 'flip monotone in 1. argument'...
  [✗] 270 failures over 1000 (print at most 1):
  (Bot, Top)
```

using our type-safe EDSL with mathmode-inspired arrow syntax $\texttt{flip} : L \xrightarrow{\sqsubseteq} L$

DTU

The rest of this talk

- Introduction
- QuickCheck, briefly
- A more complex example: Lua type analysis
  - Lua (briefly)
  - A static analysis of Lua
  - Testing lattices
  - Testing lattice operations
  - Evaluation
- Conclusion

DTU

QuickCheck, briefly (1/2)

Throughout we use OCaml with the ’qcheck’ library

E.g., monotonicity $\forall \mathtt{a}, \mathtt{b}.\ \mathtt{a} \sqsubseteq \mathtt{b} \Rightarrow \mathtt{flip\ a} \sqsubseteq \mathtt{flip\ b}$
translates into a function modeling the implication:

```
fun (a,b) -> Prop.assume (L.leq a b); L.leq (flip a) (flip b)
```

QuickCheck, briefly (1/2)

Throughout we use OCaml with the 'qcheck' library

E.g., monotonicity $\forall \mathtt{a}, \mathtt{b}.\ \mathtt{a} \sqsubseteq \mathtt{b} \Rightarrow \mathtt{flip\ a} \sqsubseteq \mathtt{flip\ b}$
translates into a function modeling the implication:

```
fun (a,b) -> Prop.assume (L.leq a b); L.leq (flip a) (flip b)
```

Combinators for composing generators:

```
let arb_pair = Arbitrary.pair L.arb_elem L.arb_elem
```

DTU

QuickCheck, briefly (1/2)

Throughout we use OCaml with the 'qcheck' library

E.g., monotonicity $\forall a, b.\ a \sqsubseteq b \Rightarrow \texttt{flip}\ a \sqsubseteq \texttt{flip}\ b$
translates into a function modeling the implication:

```
fun (a,b) -> Prop.assume (L.leq a b); L.leq (flip a) (flip b)
```

Combinators for composing generators:

```
let arb_pair = Arbitrary.pair L.arb_elem L.arb_elem
```

A monotonicity test written out:

```
# let mon_test = mk_test arb_pair
        (fun (a,b) -> Prop.assume (L.leq a b); L.leq (flip a) (flip b));;
  val mon_test : QCheck.test = <abstr>
```

DTU

QuickCheck, briefly (1/2)

Throughout we use OCaml with the 'qcheck' library

E.g., monotonicity $\forall a, b.\ a \sqsubseteq b \Rightarrow \mathtt{flip}\ a \sqsubseteq \mathtt{flip}\ b$
translates into a function modeling the implication:

```
fun (a,b) -> Prop.assume (L.leq a b); L.leq (flip a) (flip b)
```

Combinators for composing generators:

```
let arb_pair = Arbitrary.pair L.arb_elem L.arb_elem
```

A monotonicity test written out:

```
# let mon_test = mk_test arb_pair
        (fun (a,b) -> Prop.assume (L.leq a b); L.leq (flip a) (flip b));;
  val mon_test : QCheck.test = <abstr>
# run mon_test;;
testing property <anon prop>...
  [✗] 27 failures over 100
```

DTU

QuickCheck, briefly (2/2)

To improve usability 'qcheck' supports more arguments:

```
mk_test ~n:1000 ~pp:pp_pair ~name:"flip monotone"
  arb_pair (fun (a,b) -> Prop.assume (L.leq a b); L.leq (flip a) (flip b))
```

which will

- name the property,
- run 1000 tests instead, and
- use the pretty printer `pp_pair` defined as

  ```
  let pp_pair = PP.pair L.to_string L.to_string
  ```

  to print counter examples

QuickCheck, briefly (2/2)

To improve usability 'qcheck' supports more arguments:

```
mk_test ~n:1000 ~pp:pp_pair ~name:"flip monotone"
  arb_pair (fun (a,b) -> Prop.assume (L.leq a b); L.leq (flip a) (flip b))
```

which will

- name the property,
- run 1000 tests instead, and
- use the pretty printer `pp_pair` defined as

  ```
  let pp_pair = PP.pair L.to_string L.to_string
  ```

  to print counter examples

Compare above to

```
let flip_desc = ("flip",flip) in
testsig (module L) -<-> (module L) =: flip_desc
```

A more complex example:
Lua and a static analysis for it

Lua?

Lua, in brief

- Developed in Brazil, +20 years ago
- Lightweight language (few, well-chosen features):
  - Dynamically typed
  - First-class functions
  - Builtin tables (associative arrays)
  - ...
- Multiparadigm (FP, OO, ...)
- Lightweight, cross-platform implementation
- Standalone and easily embeddable
- ...

DTU

An example

This 5-line program:

```
1   function mktable(f)
2       return { x = f("x"), y = f("y") }
3   end
4
5   mktable(function (z) return z.." component" end)
```

illustrates Lua's records and first-class functions.

Lua type analysis keywords

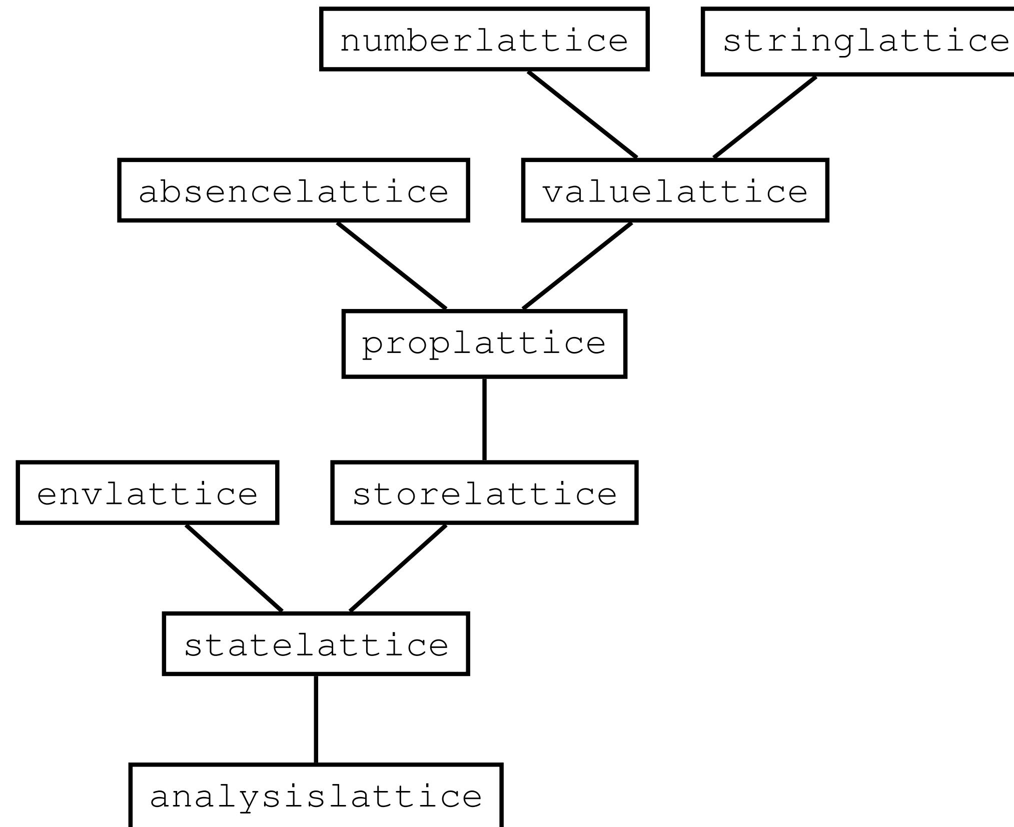
We build a

- forward,
- flow-sensitive,
- *attribute-independent*,
- monomorphic

flow analysis over a suitable lattice structure.

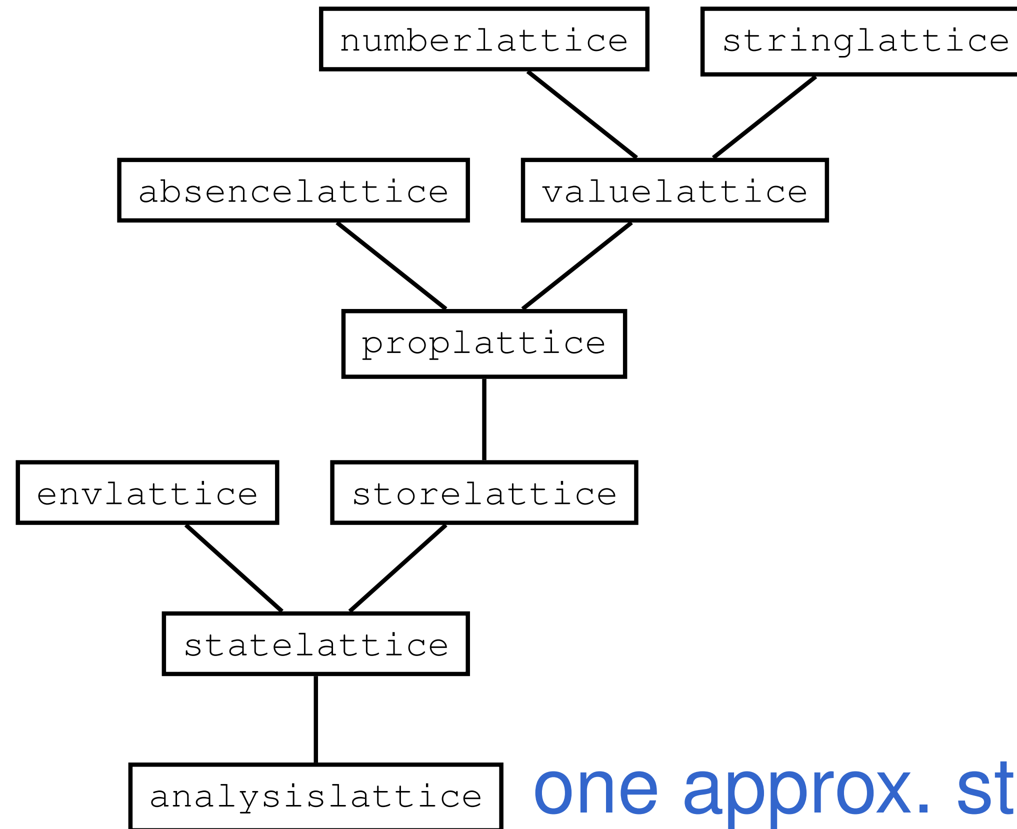
DTU

A lattice for Lua type analysis (1/2)



Akin to TAJS lattice for JavaScript (Jensen-al:SAS09)

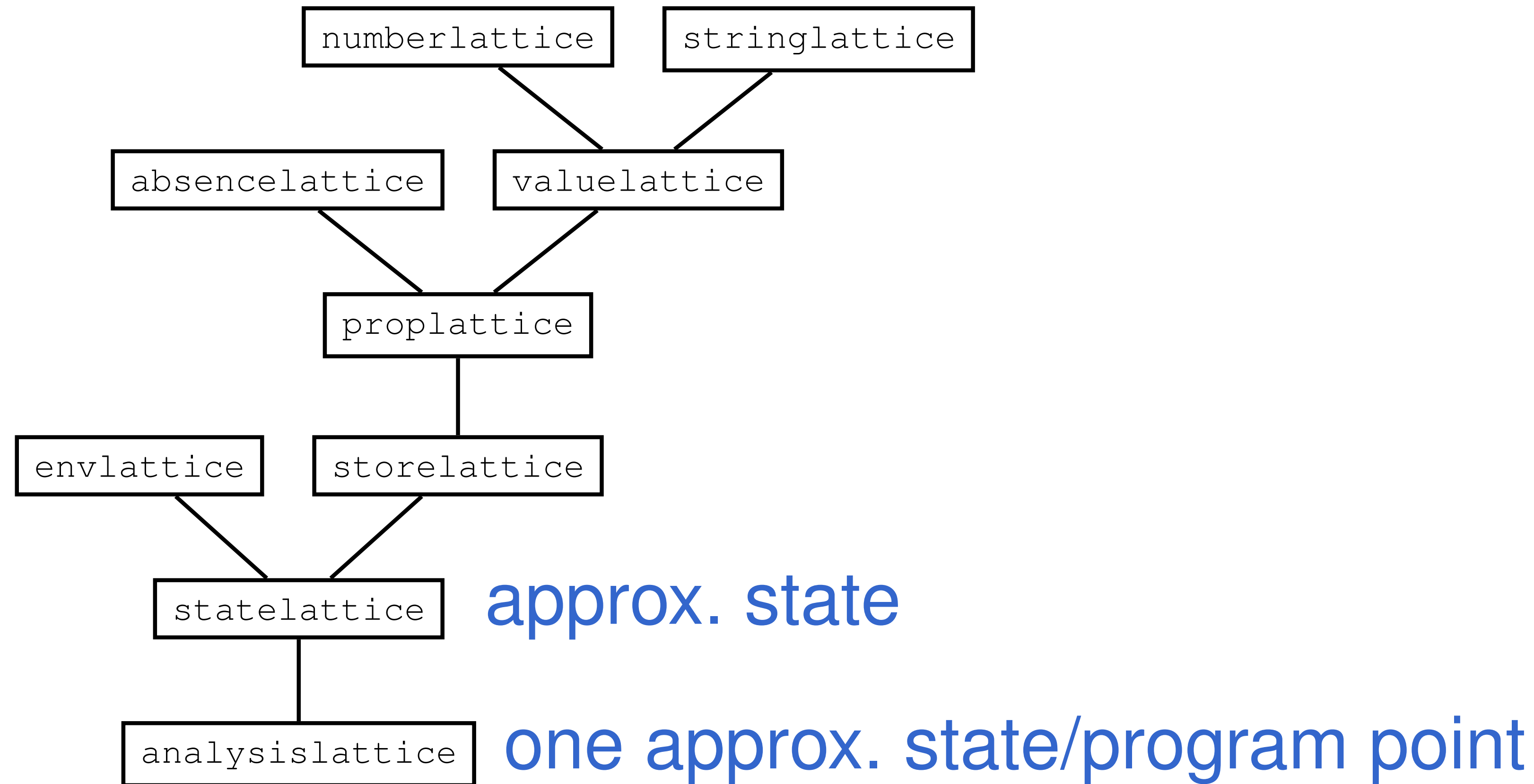
A lattice for Lua type analysis (1/2)



one approx. state/program point

Akin to TAJS lattice for JavaScript (Jensen-al:SAS09)

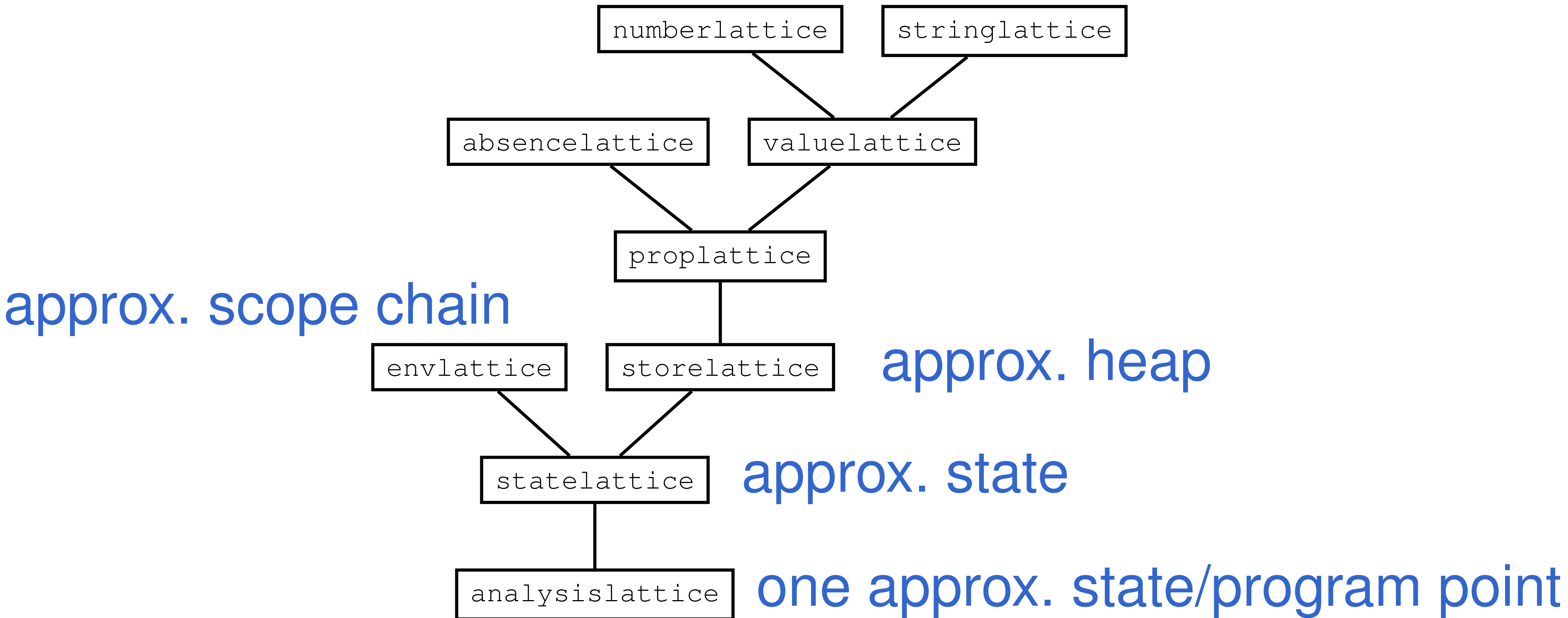
A lattice for Lua type analysis (1/2)



Akin to TAJS lattice for JavaScript (Jensen-al:SAS09)

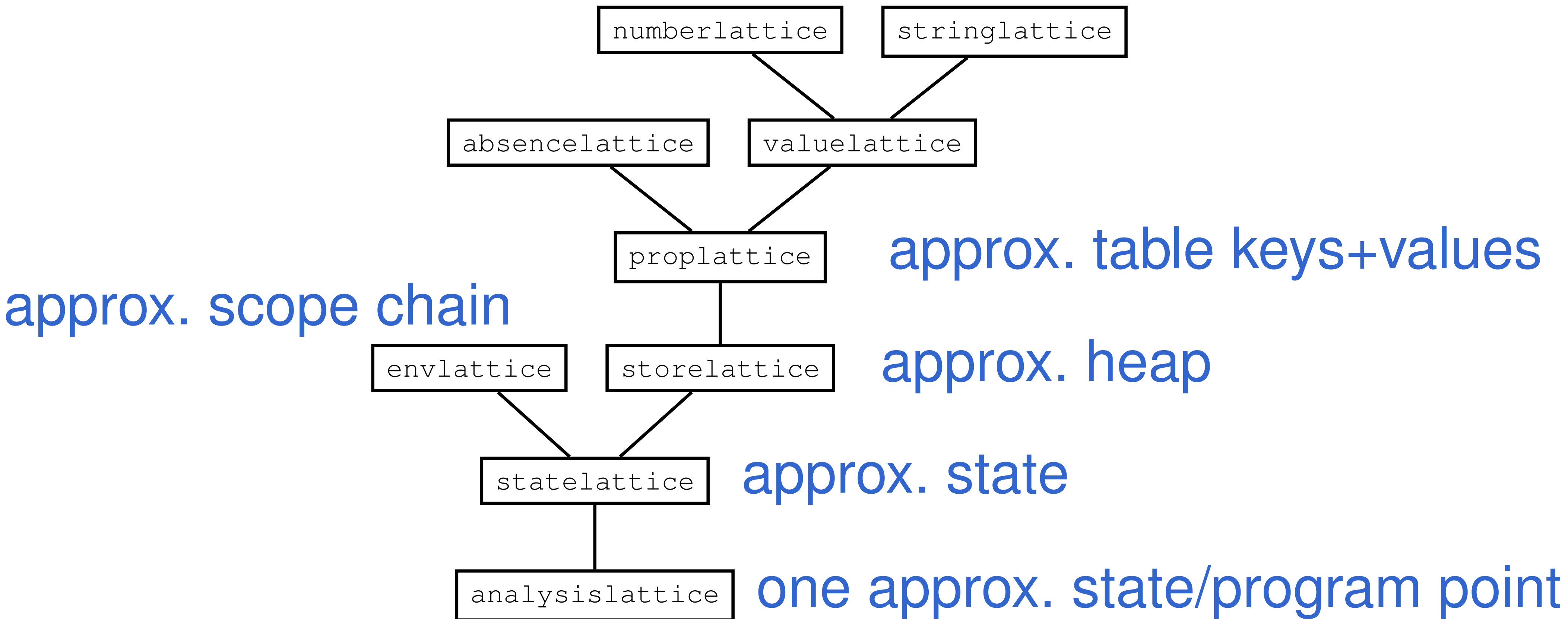
DTU

A lattice for Lua type analysis (1/2)



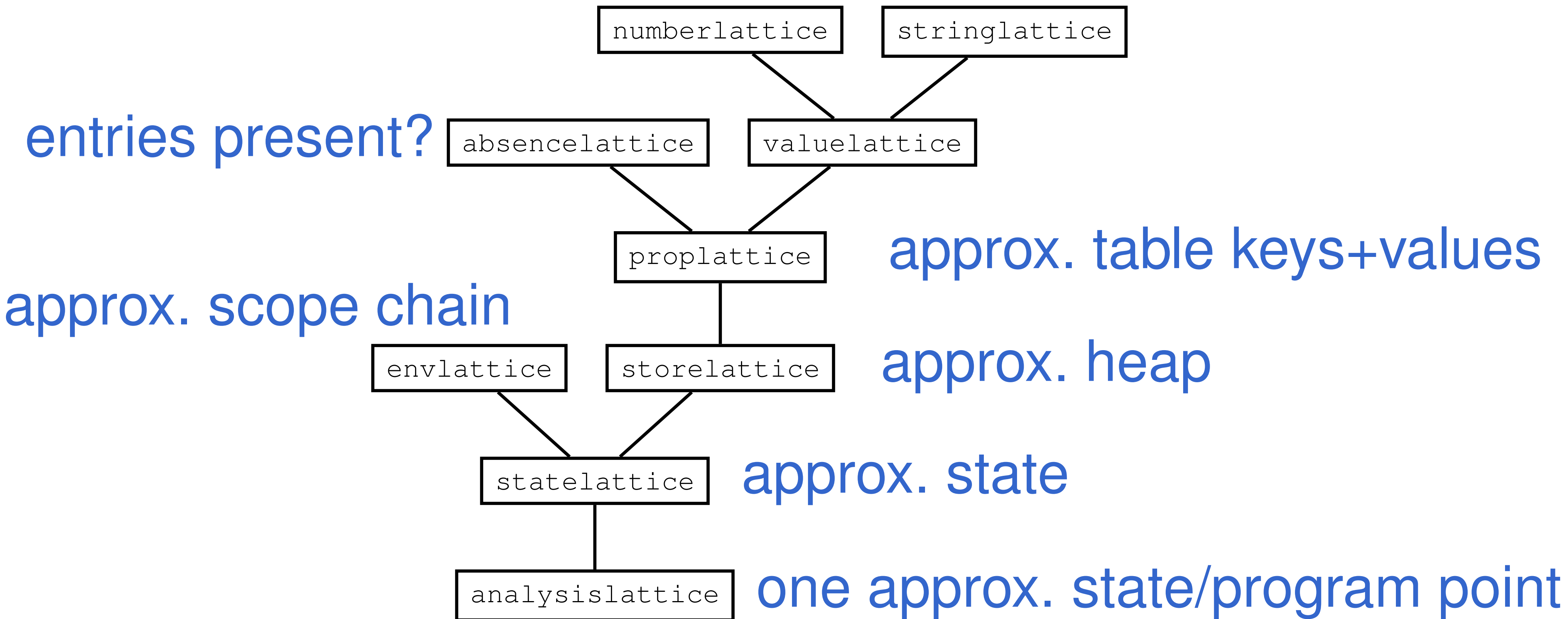
Akin to TAJS lattice for JavaScript (Jensen-al:SAS09)

A lattice for Lua type analysis (1/2)



Akin to TAJS lattice for JavaScript (Jensen-al:SAS09)

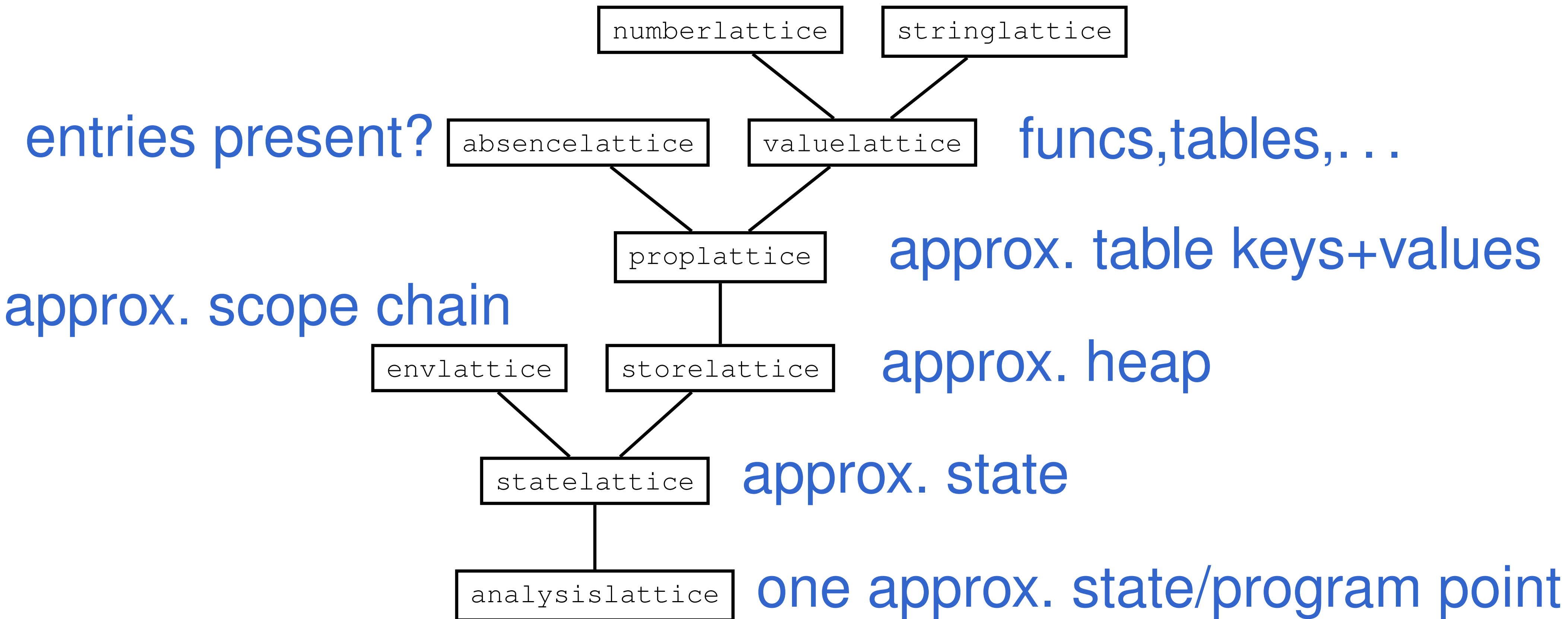
A lattice for Lua type analysis (1/2)



Akin to TAJS lattice for JavaScript (Jensen-al:SAS09)

DTU

A lattice for Lua type analysis (1/2)



Akin to TAJS lattice for JavaScript (Jensen-al:SAS09)

A lattice for Lua type analysis (2/2)

The lattice modules have a consistent signature:

```
module type LATTICE_TOPLESS =
sig
  type elem
  val leq       : elem -> elem -> bool
  val bot       : elem
(* val top      : elem *)
  val join      : elem -> elem -> elem
  val meet      : elem -> elem -> elem
  val to_string : elem -> string
end
```

but not all have an explicit top element

DTU

Testing the lattices

DTU

Lattice properties

Lattices are partial orders:

$$\forall a \in L.\ a \sqsubseteq a \qquad \text{(reflexivity)}$$

$$\forall a, b, c \in L.\ a \sqsubseteq b \ \wedge\ b \sqsubseteq c \Rightarrow a \sqsubseteq c \qquad \text{(transitivity)}$$

$$\forall a, b \in L.\ a \sqsubseteq b \ \wedge\ b \sqsubseteq a \Rightarrow a = b \qquad \text{(anti-symmetry)}$$

with additional algebraic properties:

$$\forall a \in L.\ \bot \sqsubseteq a \ \wedge\ a \sqsubseteq \top \qquad \text{($\bot$ is lower bound, $\top$ is upper bound)}$$

$$\forall a, b \in L.\ a \sqcup b = b \sqcup a \ \wedge\ a \sqcap b = b \sqcap a \qquad \text{($\sqcup$,$\sqcap$ commutative)}$$

$$\forall a, b, c \in L.\ (a \sqcup b) \sqcup c = a \sqcup (b \sqcup c) \ \wedge\ (a \sqcap b) \sqcap c = a \sqcap (b \sqcap c) \qquad \text{($\sqcup$,$\sqcap$ associative)}$$

$$\forall a \in L.\ a \sqcup a = a \ \wedge\ a \sqcap a = a \qquad \text{($\sqcup$,$\sqcap$ idempotent)}$$

$$\forall a, b \in L.\ a \sqcup (a \sqcap b) = a \ \wedge\ a \sqcap (a \sqcup b) = a \qquad \text{($\sqcup$-$\sqcap$,$\sqcap$-$\sqcup$ absorption)}$$

$$\forall a, b \in L.\ a \sqsubseteq b \Leftrightarrow a \sqcup b = b \Leftrightarrow a \sqcap b = a \qquad \text{($\sqsubseteq - \sqcup - \sqcap$ compatible)}$$

DTU

Lattice properties

Lattices are partial orders:

$$\forall a \in L.\ a \sqsubseteq a \qquad \text{(reflexivity)}$$
$$\forall a, b, c \in L.\ a \sqsubseteq b \ \wedge\ b \sqsubseteq c \Rightarrow a \sqsubseteq c \qquad \text{(transitivity)}$$
$$\forall a, b \in L.\ a \sqsubseteq b \ \wedge\ b \sqsubseteq a \Rightarrow a = b \qquad \text{(anti-symmetry)}$$

with additional algebraic properties:

$$\forall a \in L.\ \bot \sqsubseteq a \ \wedge\ a \sqsubseteq \top \qquad \text{(}\bot\text{ is lower bound, }\top\text{ is upper bound)}$$
$$\forall a, b \in L.\ a \sqcup b = b \sqcup a \ \wedge\ a \sqcap b = b \sqcap a \qquad \text{(}\sqcup\text{,}\sqcap\text{ commutative)}$$
$$\forall a, b, c \in L.\ (a \sqcup b) \sqcup c = a \sqcup (b \sqcup c) \ \wedge\ (a \sqcap b) \sqcap c = a \sqcap (b \sqcap c) \qquad \text{(}\sqcup\text{,}\sqcap\text{ associative)}$$
$$\forall a \in L.\ a \sqcup a = a \ \wedge\ a \sqcap a = a \qquad \text{(}\sqcup\text{,}\sqcap\text{ idempotent)}$$
$$\forall a, b \in L.\ a \sqcup (a \sqcap b) = a \ \wedge\ a \sqcap (a \sqcup b) = a \qquad \text{(}\sqcup\text{-}\sqcap\text{,}\sqcap\text{-}\sqcup\text{ absorption)}$$
$$\forall a, b \in L.\ a \sqsubseteq b \Leftrightarrow a \sqcup b = b \Leftrightarrow a \sqcap b = a \qquad \text{(}\sqsubseteq - \sqcup - \sqcap\text{ compatible)}$$

So we need `arb_elem`

Lattice properties

Lattices are partial orders:

$$\forall a \in L.\ a \sqsubseteq a \qquad \text{(reflexivity)}$$

$$\forall a,b,c \in L.\ a \sqsubseteq b \ \wedge\ b \sqsubseteq c \Rightarrow a \sqsubseteq c \qquad \text{(transitivity)}$$

$$\forall a,b \in L.\ a \sqsubseteq b \ \wedge\ b \sqsubseteq a \Rightarrow a = b \qquad \text{(anti-symmetry)}$$

with additional algebraic properties:

$$\forall a \in L.\ \bot \sqsubseteq a \ \wedge\ a \sqsubseteq \top \qquad \text{($\bot$ is lower bound, $\top$ is upper bound)}$$

$$\forall a,b \in L.\ a \sqcup b = b \sqcup a \ \wedge\ a \sqcap b = b \sqcap a \qquad \text{($\sqcup$,$\sqcap$ commutative)}$$

$$\forall a,b,c \in L.\ (a \sqcup b) \sqcup c = a \sqcup (b \sqcup c) \ \wedge\ (a \sqcap b) \sqcap c = a \sqcap (b \sqcap c) \qquad \text{($\sqcup$,$\sqcap$ associative)}$$

$$\forall a \in L.\ a \sqcup a = a \ \wedge\ a \sqcap a = a \qquad \text{($\sqcup$,$\sqcap$ idempotent)}$$

$$\forall a,b \in L.\ a \sqcup (a \sqcap b) = a \ \wedge\ a \sqcap (a \sqcup b) = a \qquad \text{($\sqcup$-$\sqcap$,$\sqcap$-$\sqcup$ absorption)}$$

$$\forall a,b \in L.\ a \sqsubseteq b \Leftrightarrow a \sqcup b = b \Leftrightarrow a \sqcap b = a \qquad \text{($\sqsubseteq - \sqcup - \sqcap$ compatible)}$$

So we need `arb_elem` and equality operation in signature:

```
val arb_elem : elem Arbitrary.t
val eq       : elem -> elem -> bool
```

DTU

Generating arbitrary lattice elements

Simple generators: (writing `Arb` for `Arbitrary`)

absencelattice: (two-elem lattice)

```
Arb.among [Bot; Top]
```

DTU

Generating arbitrary lattice elements

Simple generators: (writing `Arb` for `Arbitrary`)

absencelattice: (two-elem lattice)

```
Arb.among [Bot; Top]
```

stringlattice: (three-level const-prop lattice)

```
Arb.(choose [return bot; lift const string; return top])
```

Generating arbitrary lattice elements

Simple generators: (writing `Arb` for `Arbitrary`)

absencelattice: (two-elem lattice)

```
Arb.among [Bot; Top]
```

stringlattice: (three-level const-prop lattice)

```
Arb.(choose [return bot; lift const string; return top])
```

labelsets: (set-based lattice) Using qcheck's fix combinator

```
Arb.(fix ~base:(return LabelSet.empty) (lift2 LabelSet.add arb_label))
```

Generating arbitrary lattice elements

Simple generators: (writing `Arb` for `Arbitrary`)

absencelattice: (two-elem lattice)

```
Arb.among [Bot; Top]
```

stringlattice: (three-level const-prop lattice)

```
Arb.(choose [return bot; lift const string; return top])
```

labelsets: (set-based lattice) Using qcheck's fix combinator

```
Arb.(fix ~base:(return LabelSet.empty) (lift2 LabelSet.add arb_label))
```

Composite generators:

pair lattices: `Arb.pair A.arb_elem B.arb_elem` for lattices `A` and `B`

DTU

Generating arbitrary lattice elements

Simple generators: (writing `Arb` for `Arbitrary`)

absencelattice: (two-elem lattice)

```
Arb.among [Bot; Top]
```

stringlattice: (three-level const-prop lattice)

```
Arb.(choose [return bot; lift const string; return top])
```

labelsets: (set-based lattice) Using qcheck's fix combinator

```
Arb.(fix ~base:(return LabelSet.empty) (lift2 LabelSet.add arb_label))
```

Composite generators:

pair lattices: `Arb.pair A.arb_elem B.arb_elem` for lattices `A` and `B`

map lattices: Using a helper function:

```
let build_map mt add ls =
  let rec build ls = match ls with
    | [] ->         Arb.return mt
    | (k,v)::ls -> Arb.(build ls >>= fun tbl -> return (add k v tbl)) in
  build ls
```

DTU

Improving generators (1/2)

`arb_elem` isn't always sufficient:

```
let leq_trans = (* forall a,b,c. a <= b /\ b <= c  =>  a <= c *)
  mk_test ~n:1000 ~pp:pp_triple ~name:("leq transitive in " ^ L.name)
    arb_triple (fun (a,b,c) -> Prop.assume (L.leq a b);
                               Prop.assume (L.leq b c);
                               L.leq a c)
```

as it doesn't lead to very much checking:

```
testing property leq transitive in value lattice...
  [✓] passed 1000 tests (1000 preconditions failed)
```

Improving generators (1/2)

`arb_elem` isn't always sufficient:

```
let leq_trans = (* forall a,b,c. a <= b /\ b <= c  =>  a <= c *)
  mk_test ~n:1000 ~pp:pp_triple ~name:("leq transitive in " ^ L.name)
    arb_triple (fun (a,b,c) -> Prop.assume (L.leq a b);
                               Prop.assume (L.leq b c);
                               L.leq a c)
```

as it doesn't lead to very much checking:

```
testing property leq transitive in value lattice...
  [✓] passed 1000 tests (1000 preconditions failed)
```

Solution: extend signature further:

```
val arb_elem_le : elem -> elem Arbitrary.t
```

to generate elements less than a given argument

DTU

Improving generators (2/2)

These are not much harder to write.

For example,

absencelattice:

```
let arb_elem_le e = if e = Top then arb_elem else Arb.return Bot
```

stringlattice:

```
let arb_elem_le e = match e with
  | Bot     -> Arb.return Bot
  | Const s -> Arb.among [Bot; Const s]
  | Top     -> arb_elem
```

DTU

Improving generators (2/2)

These are not much harder to write.

For example,

absencelattice:

```
let arb_elem_le e = if e = Top then arb_elem else Arb.return Bot
```

stringlattice:

```
let arb_elem_le e = match e with
  | Bot     -> Arb.return Bot
  | Const s -> Arb.among [Bot; Const s]
  | Top     -> arb_elem
```

We can use `arb_elem_le` to generate ordered pairs (and triples):

```
let ord_pair =
  Arb.(L.arb_elem >>= fun e -> pair (L.arb_elem_le e) (return e))
```

e.g., to satisfy transitivity's precondition.

Testing the lattice operations

Operation tests under the hood (1/3)

Writing things out gets loooong, e.g., for `Str.concat`:

```
(* forall s. bot = s ^ bot *)
let concat_strict_snd =
  mk_test ~n:1000 ~pp:Str.to_string ~name:("Str.concat strict in 2. arg")
    Str.arb_elem (fun s -> Str.(eq bot (concat s bot)))

(* forall s,s',s''. s' <= s''  =>  (s ^ s') <= (s ^ s'')  *)
let concat_monotone_snd =
  mk_test ~n:1000 ~pp:(PP.pair Str.to_string pp_pair) ~name:("Str.concat monotone in 2. arg")
    (Arbitrary.pair Str.arb_elem ord_pair)
    (fun (s,(s',s'')) -> Prop.assume (Str.leq s' s''); Str.(leq (concat s s') (concat s s'')))

(* forall s,s',s''. s' ~ s''  =>  (s ^ s') ~ (s ^ s'') *)
let concat_invariant_snd =
  mk_test ~n:1000 ~pp:(PP.pair Str.to_string pp_pair) ~name:("Str.concat invariant in 2. arg")
    (Arbitrary.pair Str.arb_elem Str.equiv_pair)
    (fun (s,(s',s'')) -> Prop.assume (Str.eq s' s''); Str.(eq (concat s s') (concat s s'')))
```

Props $\sim$ functions, so let's build them with combinators!

DTU

Operation tests under the hood (2/3)

We provide 3 combinators for typical properties:
`op_strict`, `op_monotone`, `op_invariant` (cf. Holdermans:PPDP13)

+ 2 for adding arguments: `pw_left`, `pw_right`

```
let str_concat = ("Str.concat",Str.concat) in
[ pw_left (module Str) op_strict    (module Str) (module Str) =:: str_concat;
  pw_left (module Str) op_monotone  (module Str) (module Str) =:: str_concat;
  pw_left (module Str) op_invariant (module Str) (module Str) =:: str_concat; ]
```

How?

DTU

Operation tests under the hood (2/3)

We provide 3 combinators for typical properties:
`op_strict`, `op_monotone`, `op_invariant` (cf. Holdermans:PPDP13)

+ 2 for adding arguments: `pw_left`, `pw_right`

```
let str_concat = ("Str.concat",Str.concat) in
[ pw_left (module Str) op_strict    (module Str) (module Str) =:: str_concat;
  pw_left (module Str) op_monotone  (module Str) (module Str) =:: str_concat;
  pw_left (module Str) op_invariant (module Str) (module Str) =:: str_concat; ]
```

How? `=::` is shorthand, infix syntax for `finalize`:

```
let finalize opsig (opname,op) =
  opsig (fun (pp,gen,prop,pname,leftargs) ->
          mk_test ~n:1000 ~pp:pp (* ... *)
            ~name:(Printf.sprintf "'%s %s in %i. argument'" opname pname leftargs)
            gen (prop op))
```

using CPS, with `finalize` supplying the initial continuation

DTU

Operation tests under the hood (3/3)

From combinators to arrow syntax:

```
let str_concat = ("Str.concat",Str.concat) in
[ testsig (module Str) ---> (module Str) -$-> (module Str) =: str_concat;
  testsig (module Str) ---> (module Str) -<-> (module Str) =: str_concat;
  testsig (module Str) ---> (module Str) -~-> (module Str) =: str_concat;
  (* ... and similar for the first argument ... *) ]
```

How?

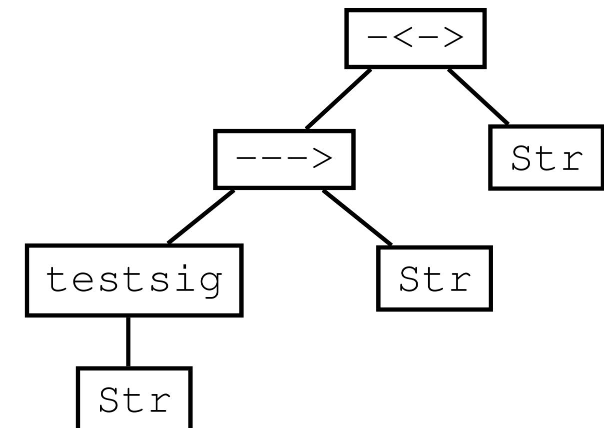
DTU

Operation tests under the hood (3/3)

From combinators to arrow syntax:

```
let str_concat = ("Str.concat",Str.concat) in
[ testsig (module Str) ---> (module Str) -$-> (module Str) =: str_concat;
  testsig (module Str) ---> (module Str) -<-> (module Str) =: str_concat;
  testsig (module Str) ---> (module Str) -~-> (module Str) =: str_concat;
  (* ... and similar for the first argument ... *) ]
```

How? (Again arrows and `=:` are infix notation)



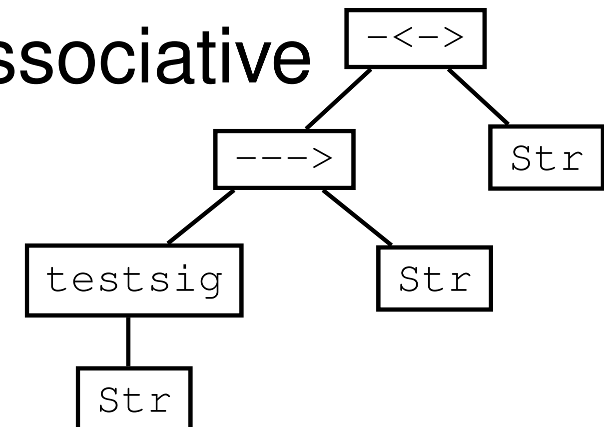
Operation tests under the hood (3/3)

From combinators to arrow syntax:

```
let str_concat = ("Str.concat",Str.concat) in
[ testsig (module Str) ---> (module Str) -$-> (module Str) =: str_concat;
  testsig (module Str) ---> (module Str) -<-> (module Str) =: str_concat;
  testsig (module Str) ---> (module Str) -~-> (module Str) =: str_concat;
  (* ... and similar for the first argument ... *) ]
```

How? (Again arrows and `=:` are infix notation)

Function app. and infix arrows are left associative



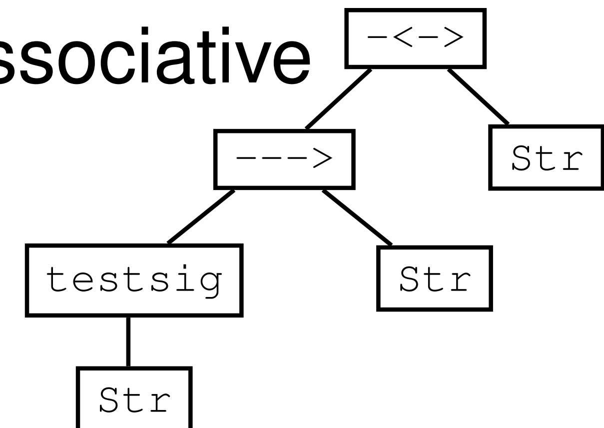
Operation tests under the hood (3/3)

From combinators to arrow syntax:

```
let str_concat = ("Str.concat",Str.concat) in
[ testsig (module Str) ---> (module Str) -$-> (module Str) =: str_concat;
  testsig (module Str) ---> (module Str) -<-> (module Str) =: str_concat;
  testsig (module Str) ---> (module Str) -~-> (module Str) =: str_concat;
  (* ... and similar for the first argument ... *) ]
```

How? (Again arrows and `=:` are infix notation)

Function app. and infix arrows are left associative



We program a DFS traversal through their definitions

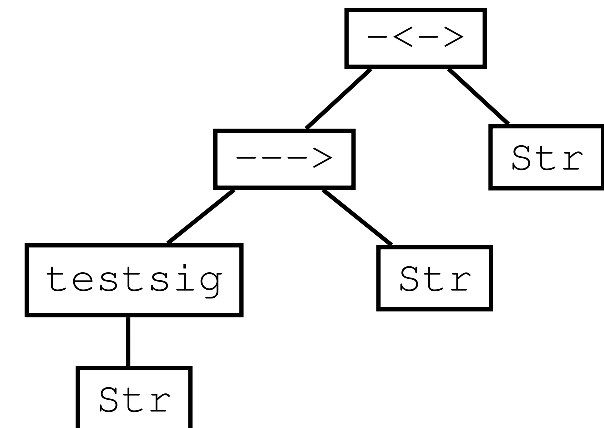
Operation tests under the hood (3/3)

From combinators to arrow syntax:

```
let str_concat = ("Str.concat",Str.concat) in
[ testsig (module Str) ---> (module Str) -$-> (module Str) =: str_concat;
  testsig (module Str) ---> (module Str) -<-> (module Str) =: str_concat;
  testsig (module Str) ---> (module Str) -~-> (module Str) =: str_concat;
  (* ... and similar for the first argument ... *) ]
```

How? (Again arrows and `=:` are infix notation)

Statically typed embedding?



DTU

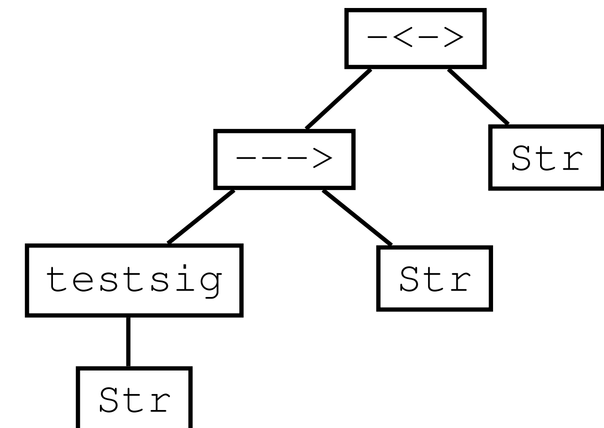
Operation tests under the hood (3/3)

From combinators to arrow syntax:

```
let str_concat = ("Str.concat",Str.concat) in
[ testsig (module Str) ---> (module Str) -$-> (module Str) =: str_concat;
  testsig (module Str) ---> (module Str) -<-> (module Str) =: str_concat;
  testsig (module Str) ---> (module Str) -~-> (module Str) =: str_concat;
  (* ... and similar for the first argument ... *) ]
```

How? (Again arrows and `=:` are infix notation)

Statically typed embedding?
Utilize CPS's result-type polymorphism
(a'la Danvy:JFP98's printf)



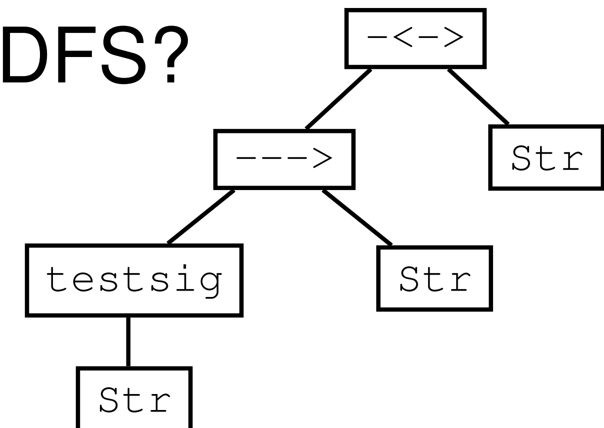
Operation tests under the hood (3/3)

From combinators to arrow syntax:

```
let str_concat = ("Str.concat",Str.concat) in
[ testsig (module Str) ---> (module Str) -$-> (module Str) =: str_concat;
  testsig (module Str) ---> (module Str) -<-> (module Str) =: str_concat;
  testsig (module Str) ---> (module Str) -~-> (module Str) =: str_concat;
  (* ... and similar for the first argument ... *) ]
```

How? (Again arrows and `=:` are infix notation)

Q: Add args with `pw_left` or `pw_right` in the DFS?



Operation tests under the hood (3/3)

From combinators to arrow syntax:

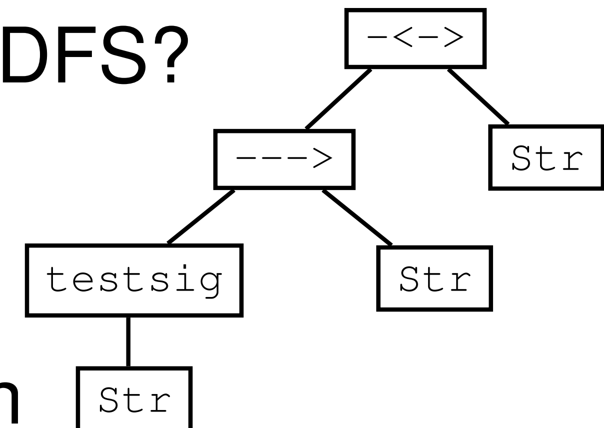
```
let str_concat = ("Str.concat",Str.concat) in
[ testsig (module Str) ---> (module Str) -$-> (module Str) =: str_concat;
  testsig (module Str) ---> (module Str) -<-> (module Str) =: str_concat;
  testsig (module Str) ---> (module Str) -~-> (module Str) =: str_concat;
  (* ... and similar for the first argument ... *) ]
```

How? (Again arrows and `=:` are infix notation)

Q: Add args with `pw_left` or `pw_right` in the DFS?

A: a DFS state machine

- one register stores latest argument
- two other registers accumulates both as argument-adding transformers!
- a `-<->` node switches their roles



[Demo]

DTU

Analysis prototype

We've built an analysis prototype

- Functional, written in OCaml
- One module per lattice (with associated operations)
- AST walker, written monadically
- In total: $\sim$ 5000 LOC

DTU

Quickcheck implementation effort

Quickcheck impl. is a 372 line module:

- functor w/19 reusable lattice property tests
- EDSL code
- reusable lattices (Booleans, pairs, . . .)

Case study: 1037 LOC on top.

DTU

Quickcheck implementation effort

Quickcheck impl. is a 372 line module:

- functor w/19 reusable lattice property tests
- EDSL code
- reusable lattices (Booleans, pairs, . . .)

Case study: 1037 LOC on top.

Checks 781 props (270 lattice + 511 operation props)

Hence approx. 2 LOC / checked property

DTU

Quickcheck implementation effort

Quickcheck impl. is a 372 line module:

- functor w/19 reusable lattice property tests
- EDSL code
- reusable lattices (Booleans, pairs, . . .)

Case study: 1037 LOC on top.

Checks 781 props (270 lattice + 511 operation props)

Hence approx. 2 LOC / checked property

So far: no quickchecking of tree walker

DTU

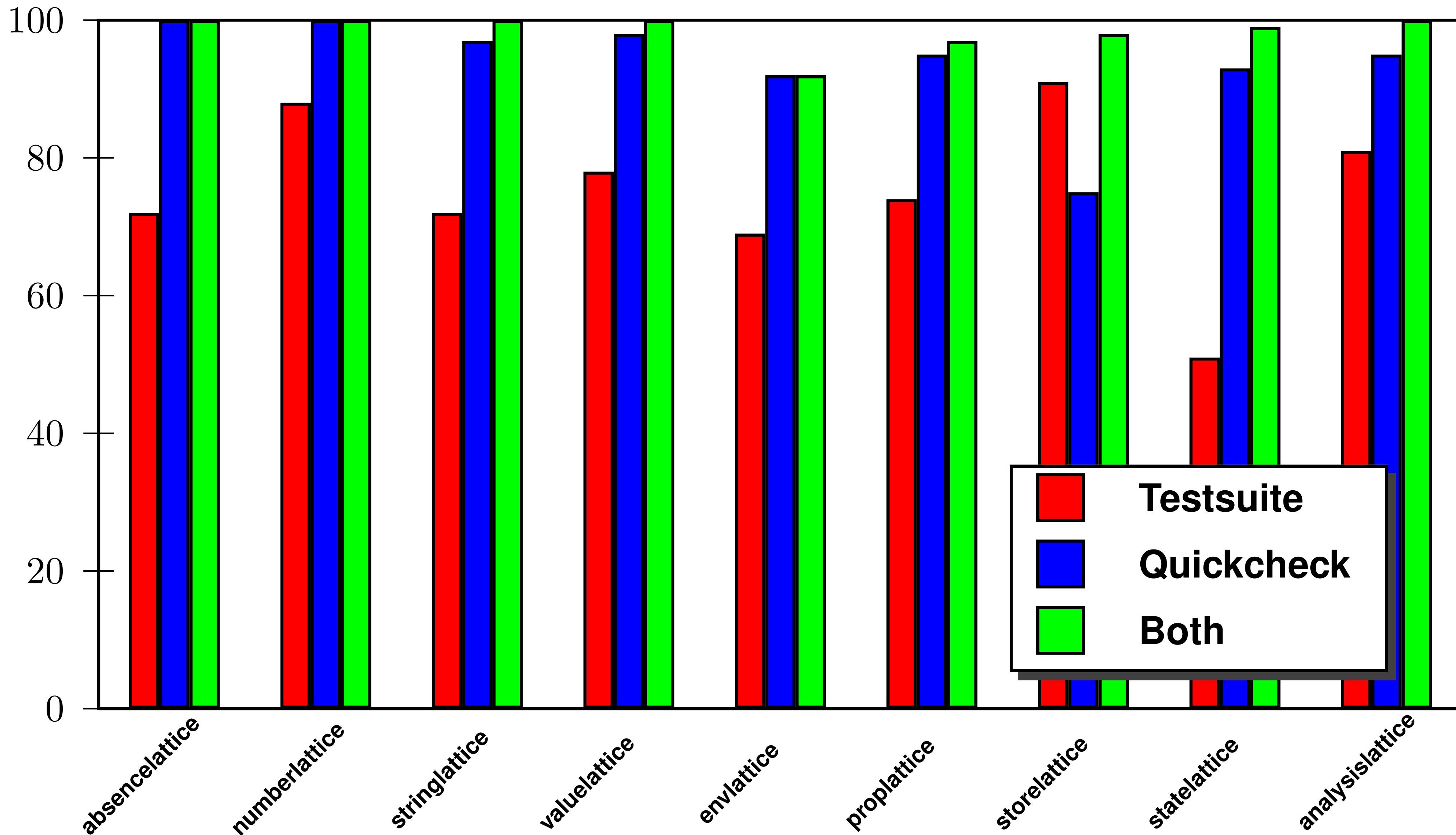
Findings

- Copy-paste error in meet of `absencelattice`
- Two non-monotone operations found and fixed (labels in `valuelattice` didn't exist in `storelattice`, lookup failure)
- Several non-strict lattice operations (fix improves precision)
- Not all operations should be strict, e.g., `PL.add`

Quickchecking is a reason to (re)consider properties

DTU

Experiment: coverage of lattice code (in %)



Excluding prettyprinters + tool frontend

Summary and conclusion

QuickCheck adds to the static analysis toolbox.

Using our approach we can check

essential static analysis properties

- which are beyond basic testing,
- in a lightweight, type-safe manner
- for increased confidence in an implementation.

DTU

Summary and conclusion

QuickCheck adds to the static analysis toolbox.

Using our approach we can check

essential static analysis properties

- which are beyond basic testing,
- in a lightweight, type-safe manner
- for increased confidence in an implementation.

Have fun: `https://github.com/jmid/lcheck`

DTU

Summary and conclusion

QuickCheck adds to the static analysis toolbox.

Using our approach we can check

essential static analysis properties

- which are beyond basic testing,
- in a lightweight, type-safe manner
- for increased confidence in an implementation.

Have fun: `https://github.com/jmid/lcheck`

DTU

Thanks