02913 Advanced Analysis Techniques

QuickCheck, Day 6

Jan Midtgaard

DTU Compute

DTU

Outline

- Patrick & Mathias' 2016 Project Presentation
- Exercises from Friday
- Project/exam formalities
- Case study:
  QuickChecking Static Analysis Properties

2016 Project Presentation

DTU

Patrick Kasting / Mathias Justesen: Testing OCaml's Compiler Backends by Generating Random Lambda Terms

DTU

Exercises

Friday's exercises

Project/exam formalities

DTU

Project/exam formalities

- Tuesday, Jan 10, is the last day of lectures
  - Afterwards you start project work
  - You can do a group project (1-3 members)
  - I'd like to talk briefly to each of you about your project topic
- Thursday morning, Jan 19, you hand in a project report (and code)
- Friday, Jan 20, you present your projects to the class and receive + answer questions
  - You receive a combined grade for the project report + presentation + answers
  - Measure of success: ability to apply QuickCheck (to a project of your choice)

QuickChecking Static Analysis Properties

DTU

This work

Based on a research paper:

QuickChecking Static Analysis Properties

Jan Midtgaard and Anders Møller

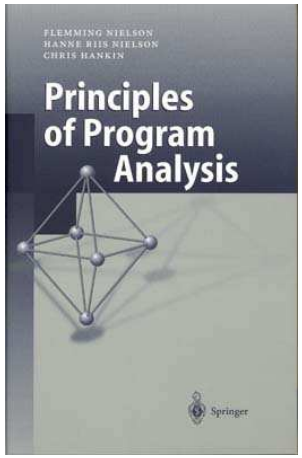
8th IEEE International Conference on Software Testing, Verification and Validation (ICST'15)

`http://janmidtgaard.dk/papers/Midtgaard-Moeller%3aICST15.pdf`

Static Analysis?

Static Analysis?

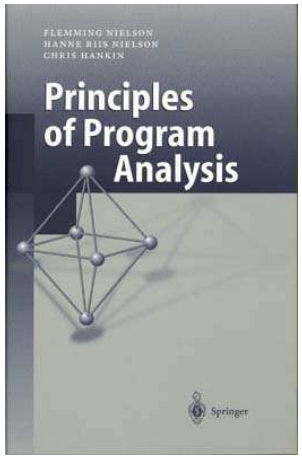
A *static analysis* predicts properties about a program without running it.



Applications:

Static Analysis?

A *static analysis* predicts properties about a program without running it.

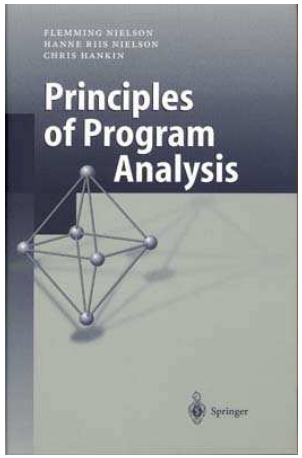


Applications: Lots

detect run-time errors (null-pointer dereference, array out of bounds,. . .), drive optimization, IDE type feedback, drive refactoring, verify software contracts, . . .

Static Analysis?

A *static analysis* predicts properties about a program without running it.



Applications: Lots

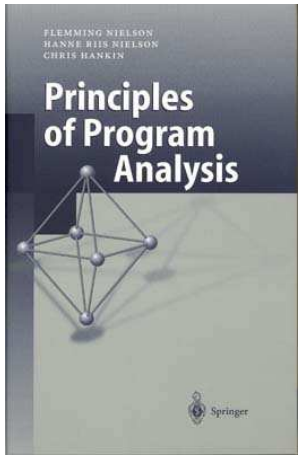
detect run-time errors (null-pointer dereference, array out of bounds,. . .), drive optimization, IDE type feedback, drive refactoring, verify software contracts, . . .

But there's a catch:

- uncomputable in general
- so we settle for (sound) approximations that account for all possible executions

Static Analysis? Take 02242!

A *static analysis* predicts properties about a program without running it.



Applications: Lots

detect run-time errors (null-pointer dereference, array out of bounds,. . .), drive optimization, IDE type feedback, drive refactoring, verify software contracts, . . .

But there's a catch:

- uncomputable in general
- so we settle for (sound) approximations that account for all possible executions

Who tests the tester?

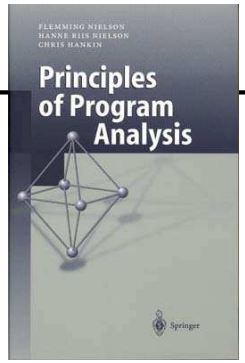
Or rather:

How do we test the tester?

Or rather:

What do we test it for?

Fact: **Many** static analyses build on. . .



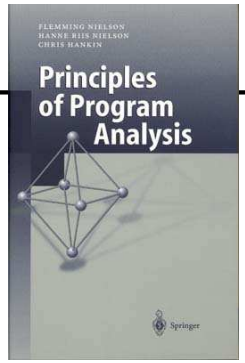
Theorem (Tarski 1955). Given

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

then f's fixed points ($f(x) = x$) form a complete lattice.

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

Fact: **Many** static analyses build on. . .



Theorem (Tarski 1955). Given

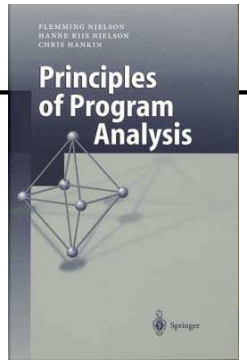
- a complete lattice $\langle L; \sqsubseteq \rangle$ and
- a monotone (order-preserving) function $f : L \to L$,

then f's fixed points ($f(x) = x$) form a complete lattice.

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

Software tools depend on these premises.

Fact: **Many** static analyses build on. . .



Theorem (Tarski 1955). Given

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

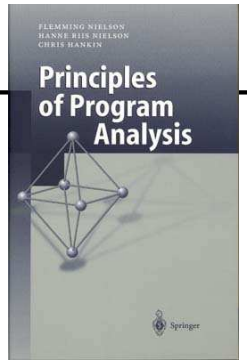
then f's fixed points ($f(x) = x$) form a complete lattice.

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

Software tools depend on these premises.

The properties embody the programmer's intention.

Fact: **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 f's fixed points ($f(x) = x$) form a complete lattice.

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

Software tools depend on these premises.

The properties embody the programmer's intention.

How do we ensure them?

A range of approaches. . .

for ensuring such *static analysis properties*:

- basic testing (unit test lattices, blackbox test analysis)
- informal monotonicity arguments
- pen-and-paper monotonicity/soundness proofs
- . . .
- machine-formalized proofs (Cachera-al:TCS05)
- automated fixed point checking (Blazy-al:SAS13)
- static monotonicity analysis (Murawski-Yi:VMCAI02)

DTU

‘QuickChecking Static Analysis Properties’

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

- with randomized property-based testing
  aka QuickCheck (Claessen-Hughes:ICFP’00) and
- an embedded domain specific language (EDSL)

for increased confidence in your static analysis code.

‘QuickChecking Static Analysis Properties’

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

- with randomized property-based testing
  aka QuickCheck (Claessen-Hughes:ICFP’00) and
- an embedded domain specific language (EDSL)

for increased confidence in your static analysis code.

Pro: lightweight,
type safe,
effective in terms of coverage

‘QuickChecking Static Analysis Properties’

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

- with randomized property-based testing
  aka QuickCheck (Claessen-Hughes:ICFP’00) and
- an embedded domain specific language (EDSL)

for increased confidence in your static analysis code.

Pro: lightweight,
type safe,
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
    | Top, Bot -> 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
```

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
    | Top, Bot -> 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.

A simple example (2/3)

Using our reusable functor of lattice property tests

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

A simple example (2/3)

Using our 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!

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.

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

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

```
# 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)
```

The rest of this talk

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

DTU

QuickCheck, briefly (1/2)

Throughout we use OCaml with the 'qcheck' library v.0.3

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

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

QuickCheck, briefly (1/2)

Throughout we use OCaml with the 'qcheck' library v.0.3

E.g., monotonicity $\forall \texttt{a}, \texttt{b}.\ \texttt{a} \sqsubseteq \texttt{b} \Rightarrow \texttt{flip a} \sqsubseteq \texttt{flip b}$ 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
```

QuickCheck, briefly (1/2)

Throughout we use OCaml with the 'qcheck' library v.0.3

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>
```

QuickCheck, briefly (1/2)

Throughout we use OCaml with the 'qcheck' library v.0.3

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>
# run mon_test;;
testing property <anon prop>...
  [✗] 27 failures over 100
```

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

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

Lua?

Adobe Lightroom 4

CISCO

Samsung SMART TV

Logitech

WIKIPEDIA

DTU

Lua, briefly

Lightweight language (few, well-chosen features):

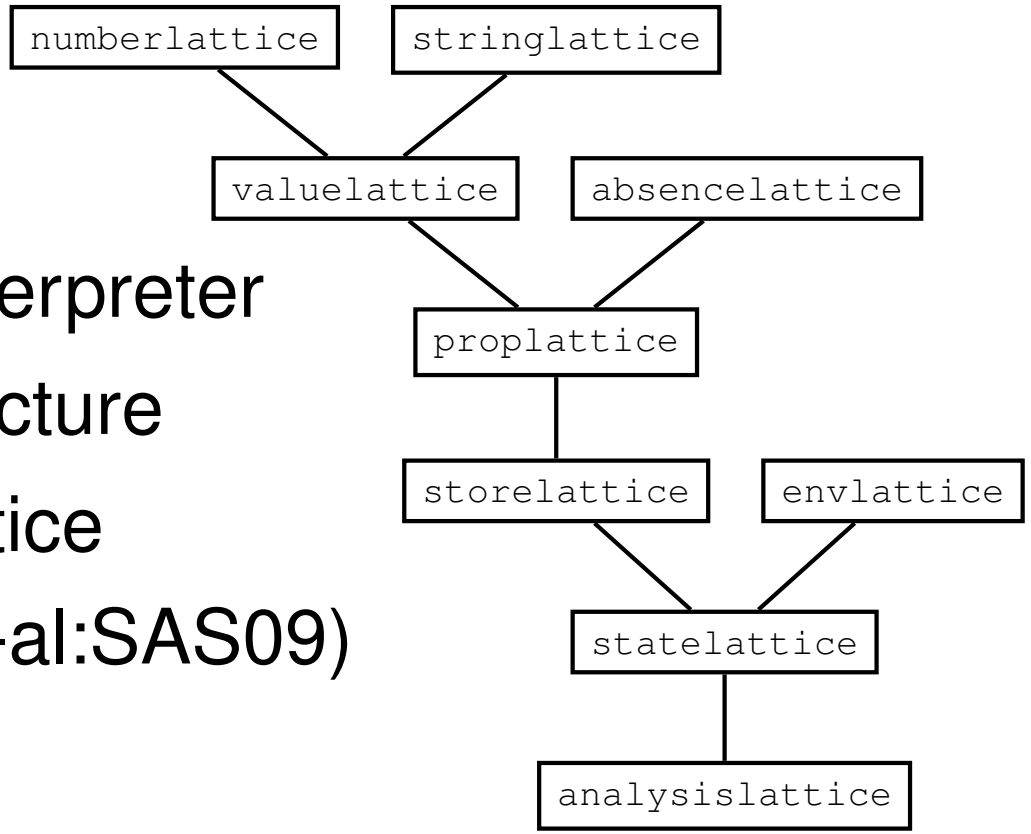
- Dynamically typed
- First-class functions
- Builtin tables (associative arrays)
- ...

Example:

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

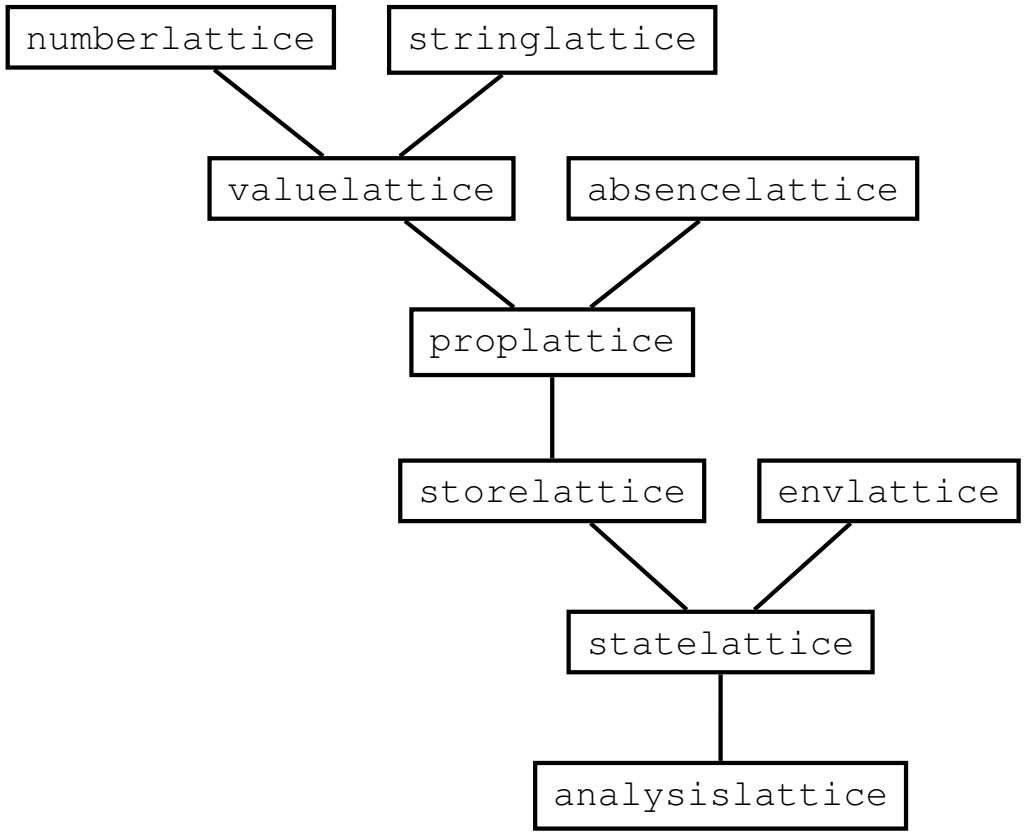
Lua type analysis by example

The static analysis is an approximate (abstract) interpreter over a suitable lattice structure (akin to the JavaScript lattice of Jensen-al:SAS09)




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

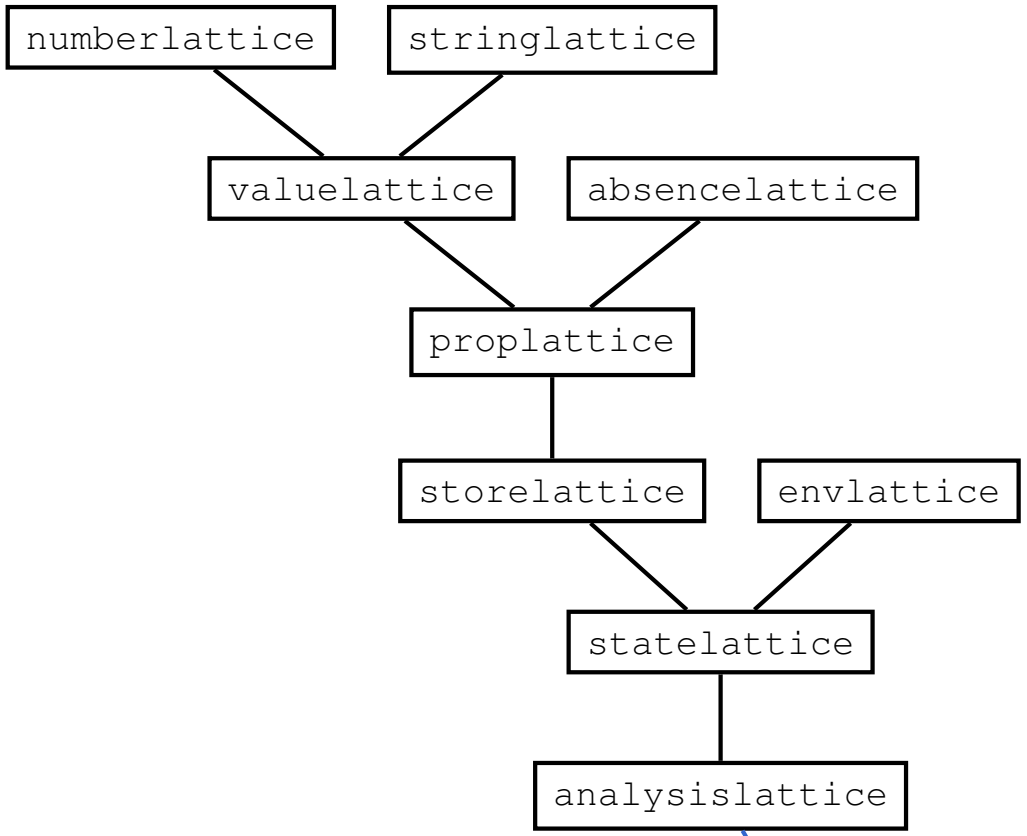
Lua type analysis by example




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

Lua type analysis by example

After this call



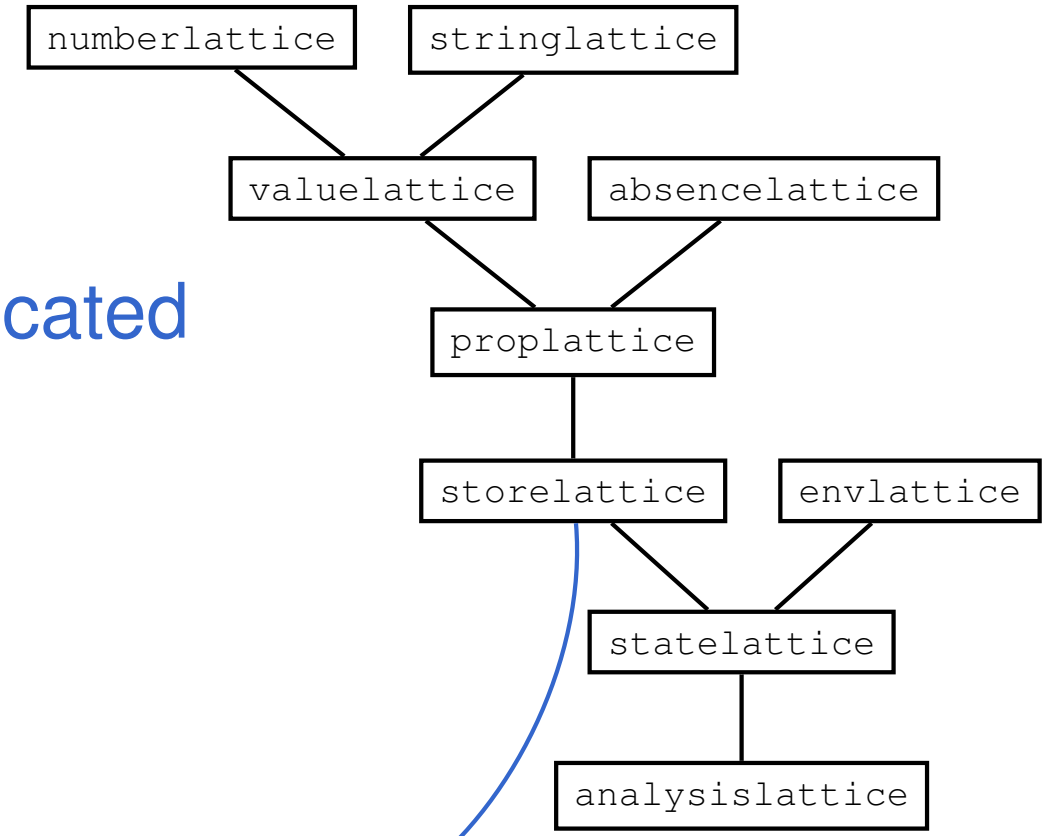

```
1  function mktable(f)
2      return { x = f("x"), y = f("y") }
3  end
4
5  mktable(function (z) return z.." component" end)
```

Lua type analysis by example

After this call

a record originating here

has been allocated




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

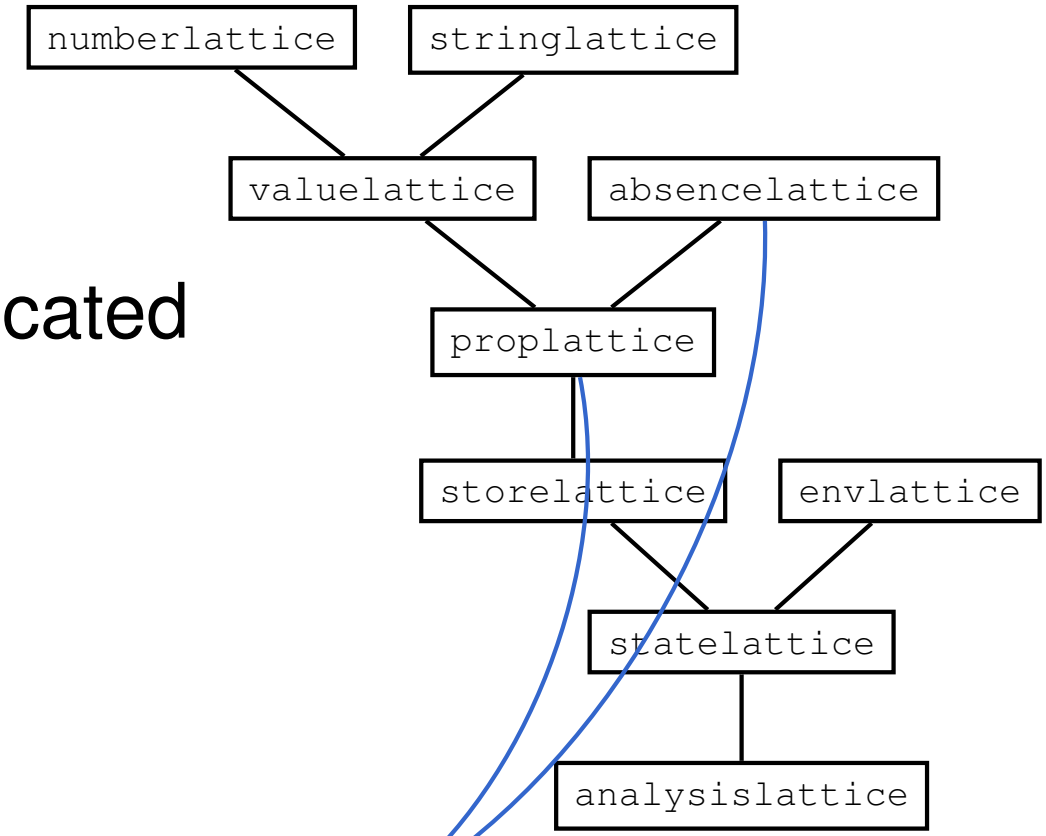
Lua type analysis by example

After this call

a record originating here

has been allocated

which definitely has




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

Lua type analysis by example

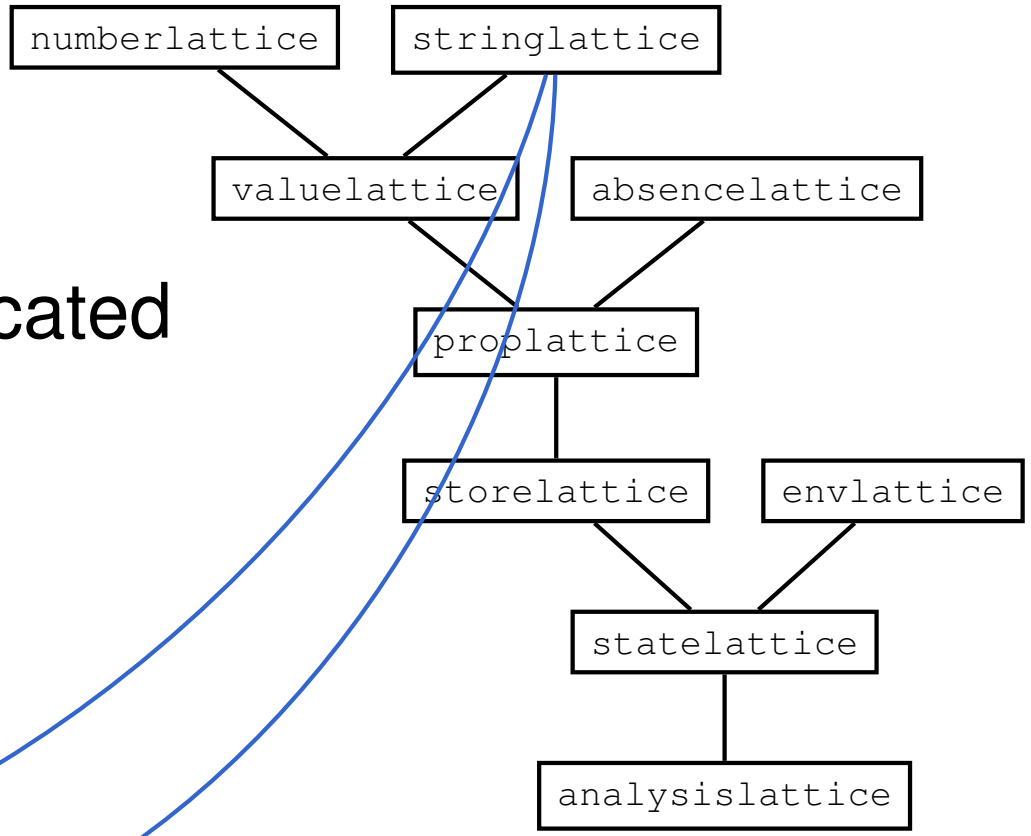
After this call

a record originating here

has been allocated

which definitely has

entries `x` and `y`




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

Lua type analysis by example

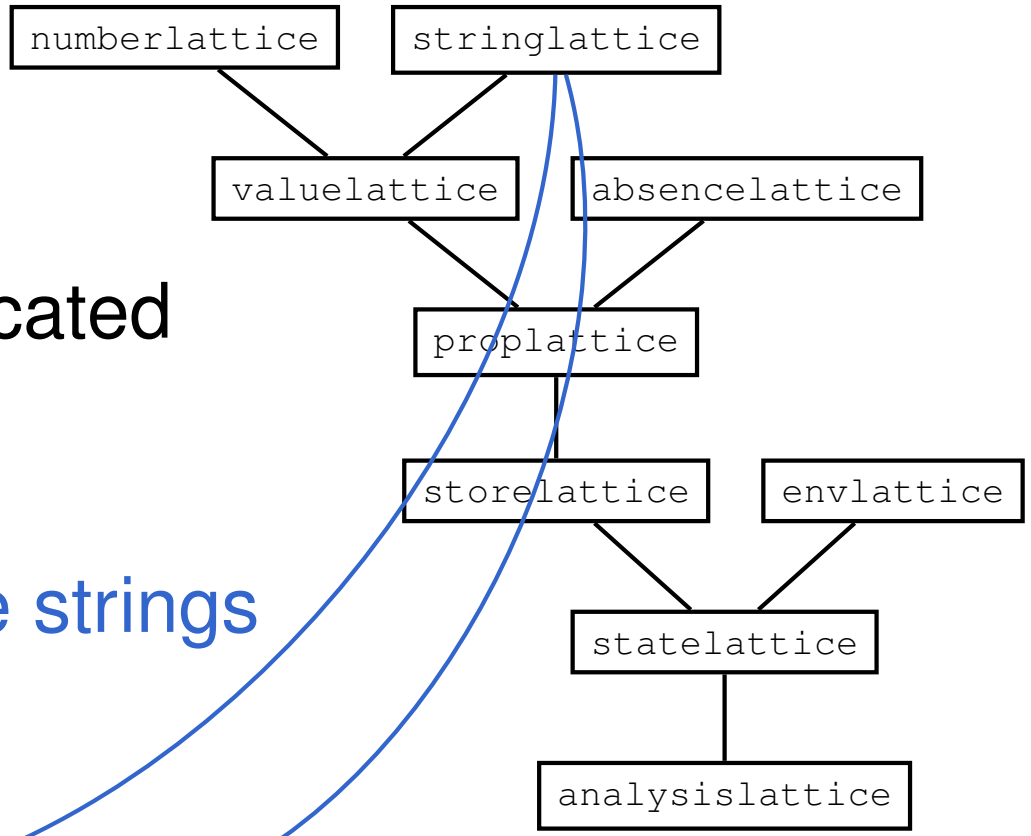
After this call

a record originating here

has been allocated

which definitely has

entries `x` and `y` which are strings




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

Lattices for Lua type analysis

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

Testing the lattices

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:

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

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

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

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

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

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

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:

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

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

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

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

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

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

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:

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

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

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

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

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

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

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

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

Generating arbitrary lattice elements

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

absencelattice: (two-elem lattice)

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

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`

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: By generating an arbitrary association list:

```
let arb_entries = Arb.(list (pair arb_label pl_arb_elem))
```

and passing the result to a generic helper function:

```
let build_map mt add ls =
   List.fold_right (fun (k,v) acctbl -> add k v acctbl) ls mt
```

Improving generators

`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

`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

- These are not much harder to write.

Testing the lattice operations

Testing the lattice operations (1/3)

We suggest testing lattice operations for three (five) properties:

- Monotonicity $\forall a, b.\ a \sqsubseteq b \implies f(a) \sqsubseteq f(b)$

  Crucial to ensure termination of the static analysis

Testing the lattice operations (1/3)

We suggest testing lattice operations for three (five) properties:

- Monotonicity $\forall a, b.\ a \sqsubseteq b \implies f(a) \sqsubseteq f(b)$

  Crucial to ensure termination of the static analysis

- Strictness $f(\bot) = \bot$

  May suggest approximation improvements

Testing the lattice operations (1/3)

We suggest testing lattice operations for three (five) properties:

- Monotonicity $\forall a, b.\ a \sqsubseteq b \implies f(a) \sqsubseteq f(b)$

  Crucial to ensure termination of the static analysis

- Strictness $f(\bot) = \bot$

  May suggest approximation improvements

- Invariance $\forall a, b.\ a = b \implies f(a) = f(b)$

  Can help catch "equality related" errors

  (cf. Holdermans:PPDP13)

- Extensiveness $\forall a.\ a \sqsubseteq f(a)$ added in 9 LOC
- Distributivity $\forall a, b.\ f(a \sqcup b) = f(a) \sqcup f(b)$ in 10 LOC

Testing the lattice operations (2/3)

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'')))
```

Clearly this will not scale to bigger projects

Testing the lattice operations (2/3)

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'')))
```

Clearly this will not scale to bigger projects

Props ∼ functions, so let's build them with combinators!

Testing the lattice operations (3/3)

For this purpose we offer a combinator-based EDSL

We provide base combinators + 2 for adding arguments:

```
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; ]
```

Testing the lattice operations (3/3)

For this purpose we offer a combinator-based EDSL

We provide base combinators + 2 for adding arguments:

```
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; ]
```

From combinators to infix 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; ]
```

Testing the lattice operations (3/3)

For this purpose we offer a combinator-based EDSL

We provide base combinators + 2 for adding arguments:

```
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; ]
```

From combinators to infix 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; ]
```

Much more compact

Testing the lattice operations (3/3)

For this purpose we offer a combinator-based EDSL

We provide base combinators + 2 for adding arguments:

```
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; ]
```

From combinators to infix 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; ]
```

Much more compact and type safe

Type safe?

The EDSL is embedded type-safely in OCaml:

```
# (testsig (module Str) ---> (module Str) -<-> (module Str)) for_op;;
```

Type safe?

The EDSL is embedded type-safely in OCaml:

```
# (testsig (module Str) ---> (module Str) -<-> (module Str)) for_op;;
- : string * (Str.elem -> Str.elem -> Str.elem) -> QCheck.test
  = <fun>
```

Type safe?

The EDSL is embedded type-safely in OCaml:

```
# (testsig (module Str) ---> (module Str) -<-> (module Str)) for_op;;
- : string * (Str.elem -> Str.elem -> Str.elem) -> QCheck.test
  = <fun>
```

Hence it will catch type errors in the QuickCheck code:

```
# let str_concat = ("Str.concat",Str.concat) in
  testsig (module Str) --> (module Str) -<-> (module Bool) =: str_concat;;
```

DTU

Type safe?

The EDSL is embedded type-safely in OCaml:

```
# (testsig (module Str) ---> (module Str) -<-> (module Str)) for_op;;
- : string * (Str.elem -> Str.elem -> Str.elem) -> QCheck.test
  = <fun>
```

Hence it will catch type errors in the QuickCheck code:

```
# let str_concat = ("Str.concat",Str.concat) in
  testsig (module Str) --> (module Str) -<-> (module Bool) =: str_concat;;
Error: This expression has type
         string * (Str.elem -> Str.elem -> Str.elem)
       but an expression was expected of type
         string * (Str.elem -> Str.elem -> Bool.elem)
       Type Str.elem = Stringlattice.elem
       is not compatible with type Bool.elem = bool
```

Type safe?

The EDSL is embedded type-safely in OCaml:

```
# (testsig (module Str) ---> (module Str) -<-> (module Str)) for_op;;
- : string * (Str.elem -> Str.elem -> Str.elem) -> QCheck.test
  = <fun>
```

Hence it will catch type errors in the QuickCheck code:

```
# let str_concat = ("Str.concat",Str.concat) in
  testsig (module Str) --> (module Str) -<-> (module Bool) =: str_concat;;
Error: This expression has type
         string * (Str.elem -> Str.elem -> Str.elem)
       but an expression was expected of type
         string * (Str.elem -> Str.elem -> Bool.elem)
       Type Str.elem = Stringlattice.elem
       is not compatible with type Bool.elem = bool
```

How?

Type safe?

The EDSL is embedded type-safely in OCaml:

```
# (testsig (module Str) ---> (module Str) -<-> (module Str)) for_op;;
- : string * (Str.elem -> Str.elem -> Str.elem) -> QCheck.test
  = <fun>
```

Hence it will catch type errors in the QuickCheck code:

```
# let str_concat = ("Str.concat",Str.concat) in
  testsig (module Str) --> (module Str) -<-> (module Bool) =: str_concat;;
Error: This expression has type
         string * (Str.elem -> Str.elem -> Str.elem)
       but an expression was expected of type
         string * (Str.elem -> Str.elem -> Bool.elem)
       Type Str.elem = Stringlattice.elem
       is not compatible with type Bool.elem = bool
```

How? By (mis)using continuation-passing style

Analysis prototype

Functional, written in OCaml ($\sim$ 4700 LOC)

- One module per lattice (with associated operations)
- AST walker, written monadically

DTU

Analysis prototype

Functional, written in OCaml ($\sim$ 4700 LOC)

- One module per lattice (with associated operations)
- AST walker, written monadically

Reusable LCheck module: 391 LOC (functor,EDSL,. . .)

Lua case study: 1213 LOC on top.

Analysis prototype

Functional, written in OCaml ($\sim$ 4700 LOC)

- One module per lattice (with associated operations)
- AST walker, written monadically

Reusable LCheck module: 391 LOC (functor,EDSL,. . .)

Lua case study: 1213 LOC on top.

Checks 871 props (290 lattice + 581 operation props)

Hence approx. 2 LOC / checked property

(Each property is tested on 1000 random inputs)

DTU

Analysis prototype

Functional, written in OCaml ($\sim$ 4700 LOC)

- One module per lattice (with associated operations)
- AST walker, written monadically

Reusable LCheck module: 391 LOC (functor,EDSL,. . .)

Lua case study: 1213 LOC on top.

Checks 871 props (290 lattice + 581 operation props)

Hence approx. 2 LOC / checked property

(Each property is tested on 1000 random inputs)

So far: no quickchecking of tree walker

DTU

Initial Findings

- Copy-paste error in meet of `absencelattice`
- Several non-strict lattice operations (fix improves precision):

```
let unop op lat = match op with
  | Ast.Uminus ->
    let lat' = coerce_tonum lat in
    if may_be_number lat' (* unary minus of number (or better) is number *)
    then number
    else bot
  | Ast.Length ->
    if may_be_strings lat || may_be_table lat
    then number
    else bot    (* length of everything but strings and tables is number *)
  | Ast.Not    ->
    bool   (* negation of anything is bool *)
```

- Not all operations should be strict, e.g., `PL.add`

Quickchecking is a reason to (re)consider properties

A `proplattice` war story (1/3)

Consider `proplattice` that represents Lua's tables.

```
type elem = { table       : (VL.elem * Abs.elem) TableMap.t;
              default_key : VL.elem;   (* collective key approx.   *)
              default     : VL.elem;   (* collective value approx.  *)
              ... }
```

A $\texttt{string} \to (\texttt{VL.elem} * \texttt{Abs.elem})$ table keeps track of statically known entries (and certainty of presence)

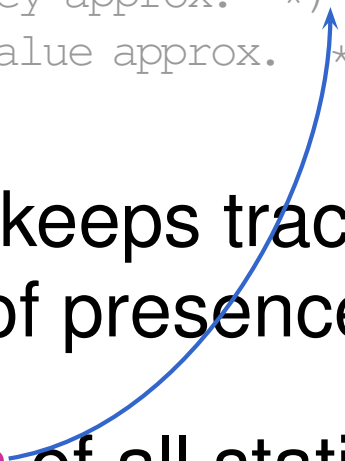
A `proplattice` war story (1/3)

Consider `proplattice` that represents Lua's tables.

```
type elem = { table       : (VL.elem * Abs.elem) TableMap.t;
              default_key : VL.elem;   (* collective key approx.   *)
              default     : VL.elem;   (* collective value approx. *)
              ... }
```

A $\texttt{string} \to (\texttt{VL.elem} * \texttt{Abs.elem})$ table keeps track of statically known entries (and certainty of presence)

One valuelattice approximates the keys of all statically unknown entries



A `proplattice` war story (1/3)

Consider `proplattice` that represents Lua's tables.

```
type elem = { table       : (VL.elem * Abs.elem) TableMap.t;
              default_key : VL.elem;  (* collective key approx.   *)
              default     : VL.elem;  (* collective value approx. *)
              ... }
```

A $\texttt{string} \to (\texttt{VL.elem} * \texttt{Abs.elem})$ table keeps track of statically known entries (and certainty of presence)

One valuelattice approximates the keys of all statically unknown entries

Another valuelattice approximates the values of all statically unknown entries

A `proplattice` war story (1/3)

Consider `proplattice` that represents Lua's tables.

```
type elem = { table       : (VL.elem * Abs.elem) TableMap.t;
              default_key : VL.elem;  (* collective key approx.   *)
              default     : VL.elem;  (* collective value approx. *)
              ... }
```

A $\texttt{string} \to (\texttt{VL.elem} * \texttt{Abs.elem})$ table keeps track of statically known entries (and certainty of presence)

One valuelattice approximates the keys of all statically unknown entries

Another valuelattice approximates the values of all statically unknown entries

With our initial generators this design passed all tests

A `proplattice` war story (2/3)

Consider `proplattice` that represents Lua's tables.

```
type elem = { table       : (VL.elem * Abs.elem) TableMap.t;
              default_key : VL.elem;  (* collective key approx.   *)
              default     : VL.elem;  (* collective value approx. *)
              ... }
```

Generating the same string twice can be useful
(e.g., for testing lookup)

Uniform generators are unlikely to do so

After revising our generators

A `proplattice` war story (2/3)

Consider `proplattice` that represents Lua's tables.

```
type elem = { table       : (VL.elem * Abs.elem) TableMap.t;
              default_key : VL.elem;  (* collective key approx.   *)
              default     : VL.elem;  (* collective value approx. *)
              ... }
```

Generating the same string twice can be useful
(e.g., for testing lookup)

Uniform generators are unlikely to do so

After revising our generators . . . tests started failing again

A `proplattice` war story (2/3)

Consider `proplattice` that represents Lua's tables.

```
type elem = { table       : (VL.elem * Abs.elem) TableMap.t;
              default_key : VL.elem;  (* collective key approx.   *)
              default     : VL.elem;  (* collective value approx. *)
              ... }
```

Generating the same string twice can be useful
(e.g., for testing lookup)

Uniform generators are unlikely to do so

After revising our generators . . . tests started failing again

`lookup_prop` failed monotonicity
$\Longrightarrow$ revise `arb_elem_le` to actually return smaller elem

A `proplattice` war story (2/3)

Consider `proplattice` that represents Lua's tables.

```
type elem = { table       : (VL.elem * Abs.elem) TableMap.t;
              default_key : VL.elem;  (* collective key approx.   *)
              default     : VL.elem;  (* collective value approx. *)
              ... }
```

Generating the same string twice can be useful
(e.g., for testing lookup)

Uniform generators are unlikely to do so

After revising our generators . . . tests started failing again

`lookup_prop` failed monotonicity
$\Longrightarrow$ revise `arb_elem_le` to actually return smaller elem

This led us to reconsider+revise the domain ordering

A `proplattice` war story (3/3)

The resulting `proplattice` design reads:

```
type elem =
  | Bot
  | Table of table
and table = { table       : (VL.elem * Abs.elem) TableMap.t;
              default_str : VL.elem;  (* fallback value for string keys *)
              default_key : VL.elem;  (* key approx. of non-string keys *)
              default     : VL.elem;  (* value approx. of non-string keys *)
              ... }
```

Failing $\sqcup$-associativity made us split the collective value over-approximation of strings and non-strings.

A `proplattice` war story (3/3)

The resulting `proplattice` design reads:

```
type elem =
  | Bot
  | Table of table
and table = { table       : (VL.elem * Abs.elem) TableMap.t;
              default_str : VL.elem;  (* fallback value for string keys *)
              default_key : VL.elem;  (* key approx. of non-string keys *)
              default     : VL.elem;  (* value approx. of non-string keys *)
              ... }
```

Failing $\sqcup$-associativity made us split the collective value over-approximation of strings and non-strings.

$\sqcup$-$\sqcap$ absorbtion ($p \sqcup (p \sqcap p') = p$) fail on uncertain entry

A `proplattice` war story (3/3)

The resulting `proplattice` design reads:

```
type elem =
  | Bot
  | Table of table
and table = { table       : (VL.elem * Abs.elem) TableMap.t;
              default_str : VL.elem;  (* fallback value for string keys *)
              default_key : VL.elem;  (* key approx. of non-string keys *)
              default     : VL.elem;  (* value approx. of non-string keys *)
              ... }
```

Failing $\sqcup$-associativity made us split the collective value over-approximation of strings and non-strings.

$\sqcup$-$\sqcap$ absorbtion ($p \sqcup (p \sqcap p') = p$) fail on uncertain entry

Fix: 'maybe absent' $\bot$-entry expresses certain absence!

A `proplattice` war story (3/3)

The resulting `proplattice` design reads:

```
type elem =
  | Bot
  | Table of table
and table = { table       : (VL.elem * Abs.elem) TableMap.t;
              default_str : VL.elem;   (* fallback value for string keys *)
              default_key : VL.elem;   (* key approx. of non-string keys *)
              default     : VL.elem;   (* value approx. of non-string keys *)
              ... }
```

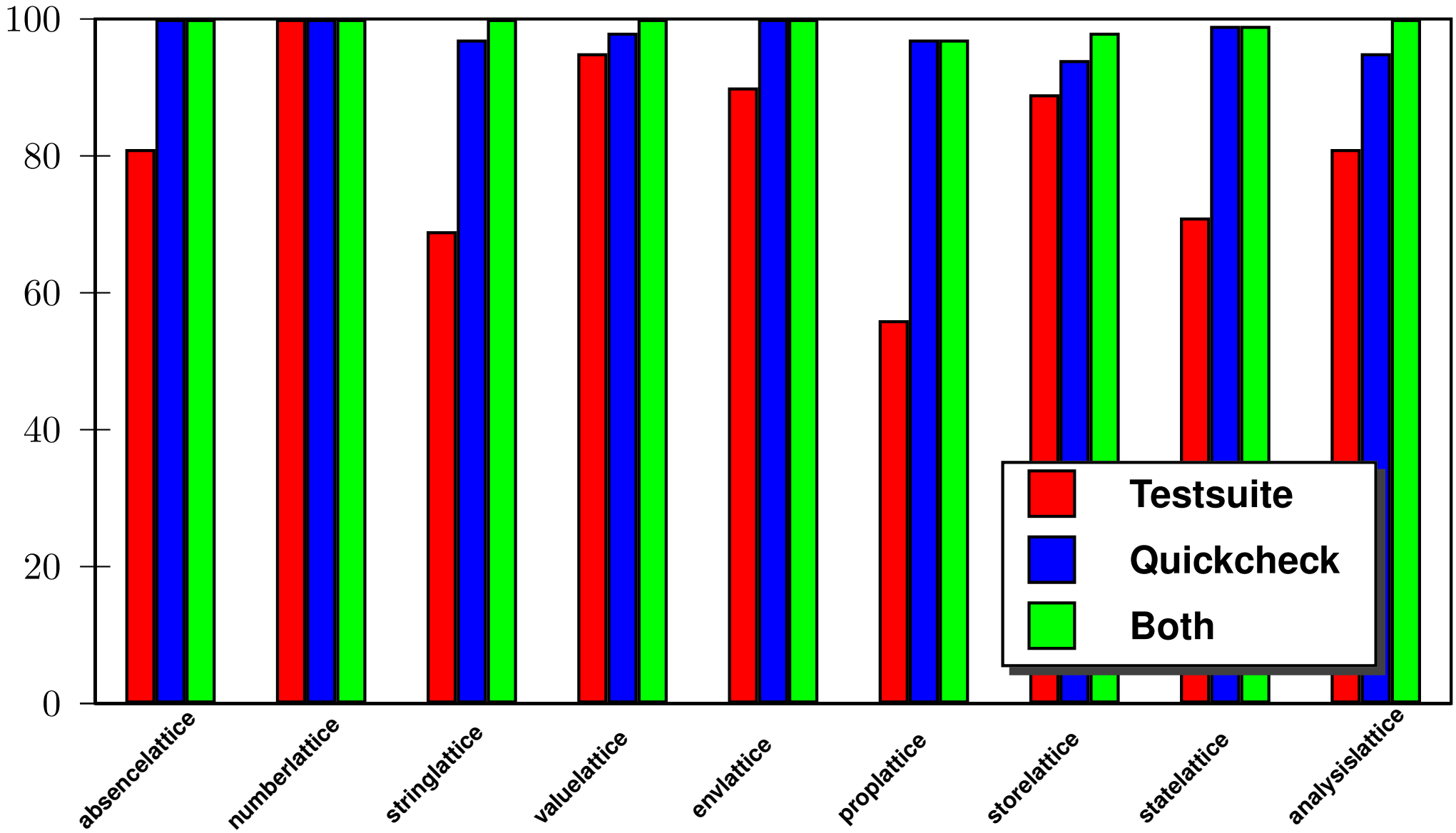
Failing $\sqcup$-associativity made us split the collective value over-approximation of strings and non-strings.

$\sqcup$-$\sqcap$ absorbtion ($p \sqcup (p \sqcap p') = p$) fail on uncertain entry

Fix: 'maybe absent' $\bot$-entry expresses certain absence!

This revision added an explicit bottom element, so, e.g., `PL.find` is also strict (uniformity)

Experiment: coverage of lattice code (in %)



Excluding prettyprinters + tool frontend

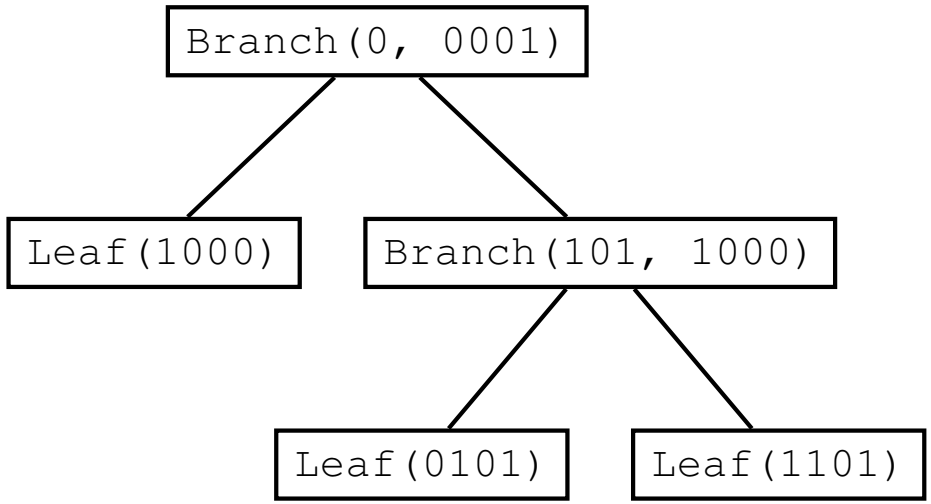
QuickChecking Patricia Trees

DTU

Patricia trees (1/5)

A Patricia tree is a data structure for representing integer sets (+ maps) compactly and functionally.

For example, we represent $\{5, 8, 13\}$ as follows:



`Branch(pre,brbit)` nodes:

- LSB distinction
- `pre` is shared prefix
- `brbit` is branching bit

The `ptrees` library is a popular OCaml implementation.

Since powersets form a lattice, let's quickcheck `ptrees`!

Testing Patricia trees (2/5)

We can cook up an interface to LCheck easily:

```
module Ptlat = struct
  let name = "Patricia Tree Intset"
  type elem = Ptset.t
  let leq  = Ptset.subset
  let join = Ptset.union
  let meet = Ptset.inter
  let bot  = Ptset.empty
  let eq   = Ptset.equal
  (* ... *)
  let arb_elem =
    Arbitrary.(fix ~max:8 ~base:(return bot) (lift2 Ptset.add arb_int))
  (* ... *)
end
```

It turns out that the choice of `arb_int` is crucial...

Testing Patricia trees (3/5)

With `arb_int` implemented as a uniform generator:

```
$ ./test.byte
check 19 properties...
testing property leq reflexive in Patricia Tree Intset...
  [✓] passed 1000 tests (0 preconditions failed)
testing property leq transitive in Patricia Tree Intset...
  [✓] passed 1000 tests (0 preconditions failed)
testing property leq anti symmetric in Patricia Tree Intset...
  [✓] passed 1000 tests (0 preconditions failed)
testing property bot is lower bound in Patricia Tree Intset...
  [✓] passed 1000 tests (0 preconditions failed)
testing property join commutative in Patricia Tree Intset...
  [✓] passed 1000 tests (0 preconditions failed)
testing property join associative in Patricia Tree Intset...
  [✓] passed 1000 tests (0 preconditions failed)

[...]

tests run in 0.04s
[✓] Success! (passed 19 tests)
```

The data structure seems to work fine. . .

Testing Patricia trees (4/5)

“Test corner cases” — 50 years of software eng.

Testing Patricia trees (4/5)

“Test corner cases” — 50 years of software eng.

Hmmm, suppose we either generate one of $\{\texttt{min\_int}, 0, \texttt{max\_int}\}$ or choose uniformly. . .

Testing Patricia trees (4/5)

"Test corner cases" — 50 years of software eng.

Hmmm, suppose we either generate one of {`min_int`, `0`, `max_int`} or choose uniformly...

```
$ ./test.byte
testing property leq reflexive in Patricia Tree Intset...
  [✓] passed 1000 tests (0 preconditions failed)
testing property leq transitive in Patricia Tree Intset...
  [✓] passed 1000 tests (14 preconditions failed)
testing property leq anti symmetric in Patricia Tree Intset...
  [✓] passed 1000 tests (0 preconditions failed)
testing property bot is lower bound in Patricia Tree Intset...
  [✓] passed 1000 tests (0 preconditions failed)
testing property join commutative in Patricia Tree Intset...
  [✓] passed 1000 tests (0 preconditions failed)
testing property join associative in Patricia Tree Intset...
  [✗] 52 failures over 1000 (print at most 1):
  ({-4611686018427387904, 0}, {0}, {4611686018427387903})
```

...

Testing Patricia trees (5/5)

What's wrong?

A helper function recursively merges two Patricia trees by testing for the rightmost branching bit.

An integer comparison is insufficient for this test because the branching bit and sign bit can coincide.

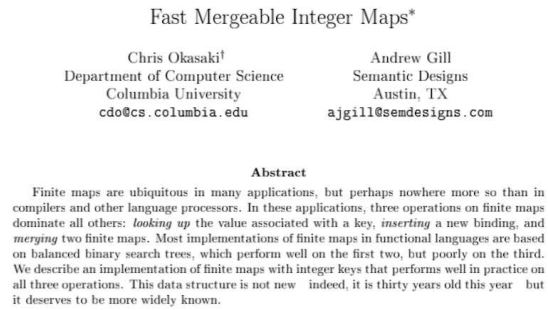
Testing Patricia trees (5/5)

What's wrong?

A helper function recursively merges two Patricia trees by testing for the rightmost branching bit.

An integer comparison is insufficient for this test because the branching bit and sign bit can coincide.

The bug has lurked in `ptrees` for years which inherited it from the Okasaki-Gill:ML98 paper's code. Jean-Christophe Filliâtre quickly made an elegant fix.



Fast Mergeable Integer Maps*

Chris Okasaki†
Department of Computer Science
Columbia University
cdo@cs.columbia.edu

Andrew Gill
Semantic Designs
Austin, TX
ajgill@semdesigns.com

Abstract

Finite maps are ubiquitous in many applications, but perhaps nowhere more so than in compilers and other language processors. In these applications, three operations on finite maps dominate all others: *looking up* the value associated with a key, *inserting* a new binding, and *merging* two finite maps. Most implementations of finite maps in functional languages are based on balanced binary search trees, which perform well on the first two, but poorly on the third. We describe an implementation of finite maps with integer keys that performs well in practice on all three operations. This data structure is not new indeed, it is thirty years old this year but it deserves to be more widely known.

Testing Patricia trees (5/5)

What's wrong?

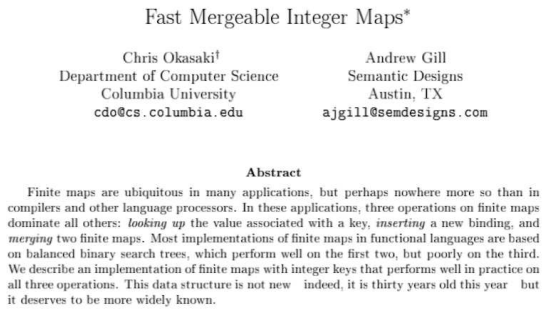
A helper function recursively merges two Patricia trees by testing for the rightmost branching bit.

An integer comparison is insufficient for this test because the branching bit and sign bit can coincide.

The bug has lurked in `ptrees` for years which inherited it from the Okasaki-Gill:ML98 paper's code. Jean-Christophe Filliâtre quickly made an elegant fix.

Significance?

`ptrees` is used by Javalib library,
which is used by the Sawja library,
which is used by Facebook's Infer tool



Fast Mergeable Integer Maps*

Chris Okasaki†
Department of Computer Science
Columbia University
cdo@cs.columbia.edu

Andrew Gill
Semantic Designs
Austin, TX
ajgill@semdesigns.com

Abstract

Finite maps are ubiquitous in many applications, but perhaps nowhere more so than in compilers and other language processors. In these applications, three operations on finite maps dominate all others: *looking up* the value associated with a key, *inserting* a new binding, and *merging* two finite maps. Most implementations of finite maps in functional languages are based on balanced binary search trees, which perform well on the first two, but poorly on the third. We describe an implementation of finite maps with integer keys that performs well in practice on all three operations. This data structure is not new indeed, it is thirty years old this year but it deserves to be more widely known.

Summary and Conclusion

Lessons learned

1. QuickChecking is a good opportunity to revisit your specification
2. Your confidence should be proportional to the quality of your generators
3. Uniform distributions are useful
   — for programming non-uniform distributions
   (bugs don't hide uniformly)
4. Machine-generated examples can be HUGE
   — shrinking support is important (but rare) to get humanly understandable counterexamples
5. Statistics (coverage stats and/or QuickCheck stats) are important to understand what's going on

DTU

Summary and conclusion

Randomized property-based testing adds to the static analysis toolbox.

Using our approach we can check

essential static analysis properties

- which are beyond current practice,
- in a lightweight, type-safe manner
- for increased confidence in an implementation
- or simply to test a research idea in the lab.

Summary and conclusion

Randomized property-based testing adds to the static analysis toolbox.

Using our approach we can check

essential static analysis properties

- which are beyond current practice,
- in a lightweight, type-safe manner
- for increased confidence in an implementation
- or simply to test a research idea in the lab.

You are welcome to play with the EDSL:

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

DTU

Lattice checking in retrospect

We did API-based quickchecking (module level)

– but with whitebox generators

Blackbox (API-based) generators would be interesting

Going for a symbolic, algebraic approach instead, would let us easily manipulate lattice values:

- shrink them
- generate only known “abstract addresses”.

The suggested static analysis project of Apron/Parma Polyhedra Library should be API-based

DTU

Summary

Today we have covered

- An example project from last year
- The project/exam formalities
- A case study regarding static analysis