

Model-Based Testing, Part 1: QuickChecking Patricia Trees

Jan Midtgaard

MMMI, SDU

SM2-TES, Day 4

A bug's tale...

QuickCheck, briefly (Claessen-Hughes:00)

QuickCheck \approx “Testing at a higher level of abstraction”

Tests are described by **a generator** and **a property**.

QuickCheck, briefly (Claessen-Hughes:00)

QuickCheck \approx “Testing at a higher level of abstraction”

Tests are described by **a generator** and **a property**.

Example: McCarthy’s 91 function

```
let rec mc x = if x > 100 then x - 10 else mc (mc (x + 11))
```

QuickCheck, briefly (Claessen-Hughes:00)

QuickCheck \approx “Testing at a higher level of abstraction”

Tests are described by **a generator** and **a property**.

Example: McCarthy’s 91 function

```
let rec mc x = if x > 100 then x - 10 else mc (mc (x + 11))
```

We can test it against the (buggy) specification:

```
let mc91_const =  
  Test.make ~name:"McCarthy_91_constant" ~count:1000  
    small_signed_int (fun n -> mc n = 91)
```

QuickCheck, briefly (Claessen-Hughes:00)

QuickCheck \approx “Testing at a higher level of abstraction”

Tests are described by **a generator** and **a property**.

Example: McCarthy’s 91 function

```
let rec mc x = if x > 100 then x - 10 else mc (mc (x + 11))
```

We can test it against the (buggy) specification:

```
let mc91_const =  
  Test.make ~name:"McCarthy_91_constant" ~count:1000  
    small_signed_int (fun n -> mc n = 91)
```

```
# QCheck_runner.run_tests ~verbose:true [mc91_const];;  
generated error fail pass / total      time test name  
[X]    10     0     1     9 / 1000      0.0s McCarthy 91 constant
```

```
--- Failure -----
```

```
Test McCarthy 91 constant failed (41 shrink steps):
```

QuickCheck, briefly (fixed)

QuickCheck \approx “Testing at a higher level of abstraction”

Tests are described by **a generator** and **a property**.

Example: McCarthy’s 91 function

```
let rec mc x = if x > 100 then x - 10 else mc (mc (x + 11))
```

We can then test it according to the real specification:

```
let mc91_spec =  
  Test.make ~name:"McCarthy_91_corr_spec" ~count:1000  
    small_signed_int (fun n -> if n <= 101  
                                then mc n = 91  
                                else mc n = n - 10)
```

QuickCheck, briefly (fixed)

QuickCheck \approx “Testing at a higher level of abstraction”

Tests are described by **a generator** and **a property**.

Example: McCarthy’s 91 function

```
let rec mc x = if x > 100 then x - 10 else mc (mc (x + 11))
```

We can then test it according to the real specification:

```
let mc91_spec =  
  Test.make ~name:"McCarthy_91_corr._spec" ~count:1000  
    small_signed_int (fun n -> if n <= 101  
                                then mc n = 91  
                                else mc n = n - 10)
```

```
# QCheck_runner.run_tests ~verbose:true [mc91_spec];;
```

```
generated error fail pass / total      time test name  
[✓] 1000      0      0 1000 / 1000      0.0s McCarthy 91 corr. spec
```

```
=====
```

```
success (ran 1 tests)
```

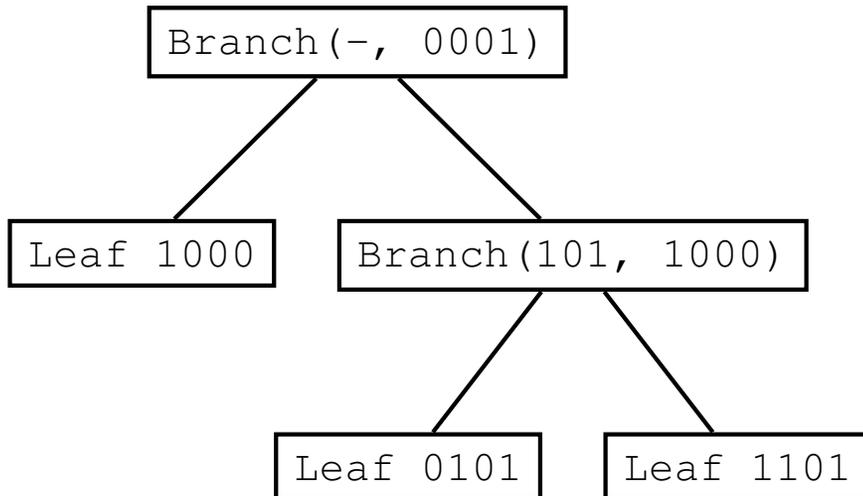
Patricia trees (Morrison:68,Okasaki-Gill:98)

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

Patricia trees (Morrison:68,Okasaki-Gill:98)

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

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



Traverse bits from LSB to MSB

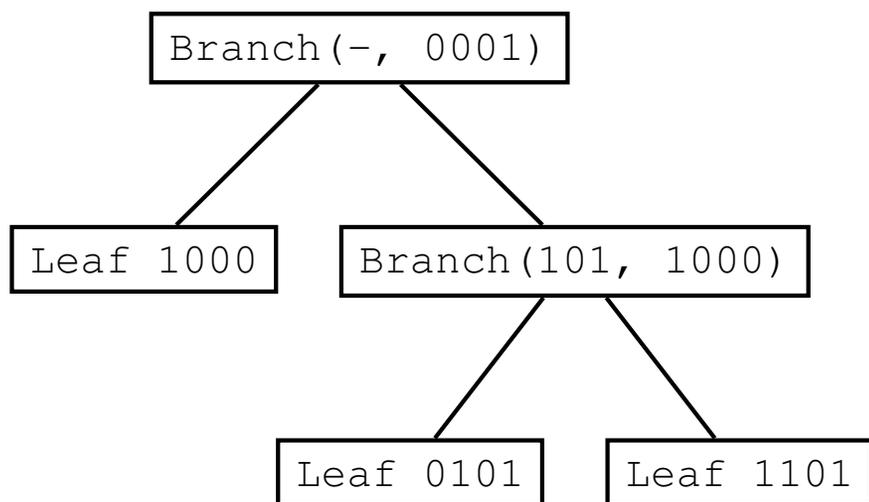
Branch(pre, brbit) nodes:

- pre is a shared prefix
- brbit is a branching bit

Patricia trees (Morrison:68, Okasaki-Gill:98)

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

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



Traverse bits from LSB to MSB

Branch(`pre`, `brbit`) nodes:

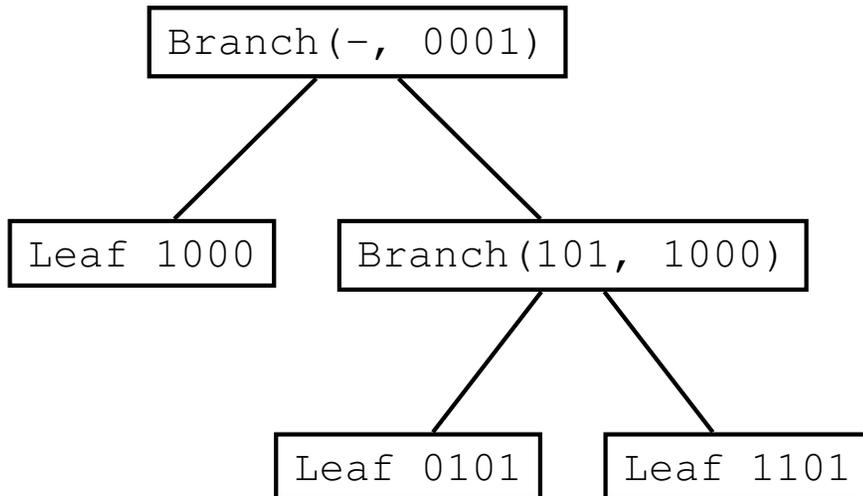
- `pre` is a shared prefix
- `brbit` is a branching bit

`ptrees/ptset` is a popular OCaml implementation,

Patricia trees (Morrison:68,Okasaki-Gill:98)

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

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



Traverse bits from LSB to MSB

Branch(pre, brbit) nodes:

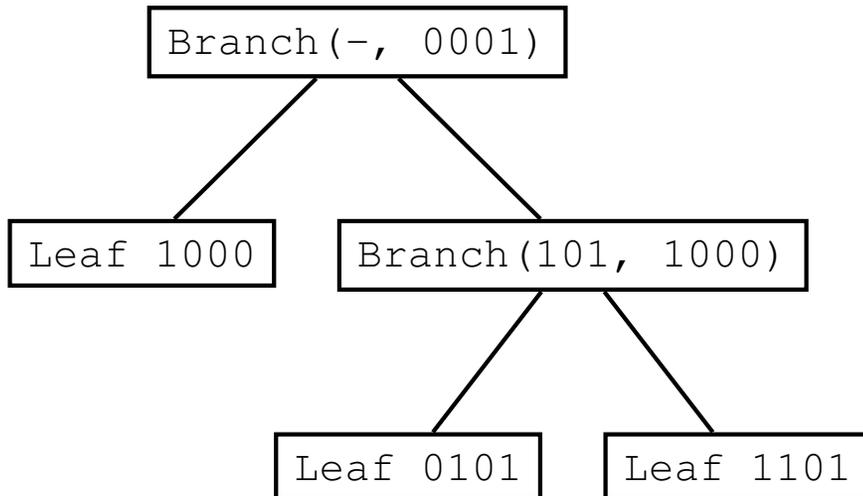
- pre is a shared prefix
- brbit is a branching bit

`ptrees/ptset` is a popular OCaml implementation, which is used by JavaLib/SawJa,

Patricia trees (Morrison:68, Okasaki-Gill:98)

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

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



Traverse bits from LSB to MSB

Branch(`pre`, `brbit`) nodes:

- `pre` is a shared prefix
- `brbit` is a branching bit

`ptrees/ptset` is a popular OCaml implementation, which is used by JavaLib/SawJa, which is again used by Facebook's Infer analyzer.

What could possibly go wrong?

Non-trivial case analysis and bit fiddling in `ptset`:

```
let rec merge = function
| t1,t2 when t1==t2 -> t1
| Empty, t   -> t
| t, Empty   -> t
| Leaf k, t  -> add k t
| t, Leaf k  -> add k t
| (Branch (p,m,s0,s1) as s), (Branch (q,n,t0,t1) as t) ->
    if m == n && match_prefix q p m then
        (* The trees have the same prefix. Merge the subtrees. *)
        Branch (p, m, merge (s0,t0), merge (s1,t1))
    else if m < n && match_prefix q p m then
        (* [q] contains [p]. Merge [t] with a subtree of [s]. *)
        if zero_bit q m then
            Branch (p, m, merge (s0,t), s1)
        else
            Branch (p, m, s0, merge (s1,t))
    else if m > n && match_prefix p q n then
        (* [p] contains [q]. Merge [s] with a subtree of [t]. *)
        if zero_bit p n then
            Branch (q, n, merge (s,t0), t1)
        else
            Branch (q, n, t0, merge (s,t1))
    else
        (* The prefixes disagree. *)
        join (p, s, q, t)
```

Spilling the Patricia tree beans ...

There was a **bug**:

- in Filliâtre's `ptset` module code and
- in the Okasaki-Gill:ML98 paper

which went **unnoticed for ~19 years**.

I found it using **QuickCheck**, aka. property-based testing.

In the rest of this talk I'll (attempt to) explain you how...

For more details see:

J. Midtgaard

QuickChecking Patricia Trees

TFP'17

Outline

- QuickCheck, briefly
- Patricia trees
- Spilling the beans
- The basic idea
- Building a model
(datatype, interpreter, generator, shrinker, ...)
- Revisiting generation
- The bug and a fix
- Model-based quickchecking
- Conclusion

The `ptset` API

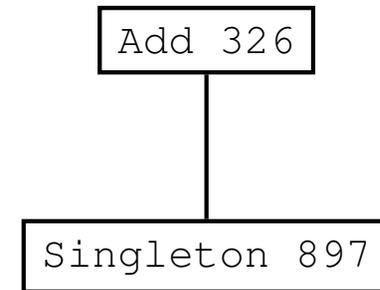
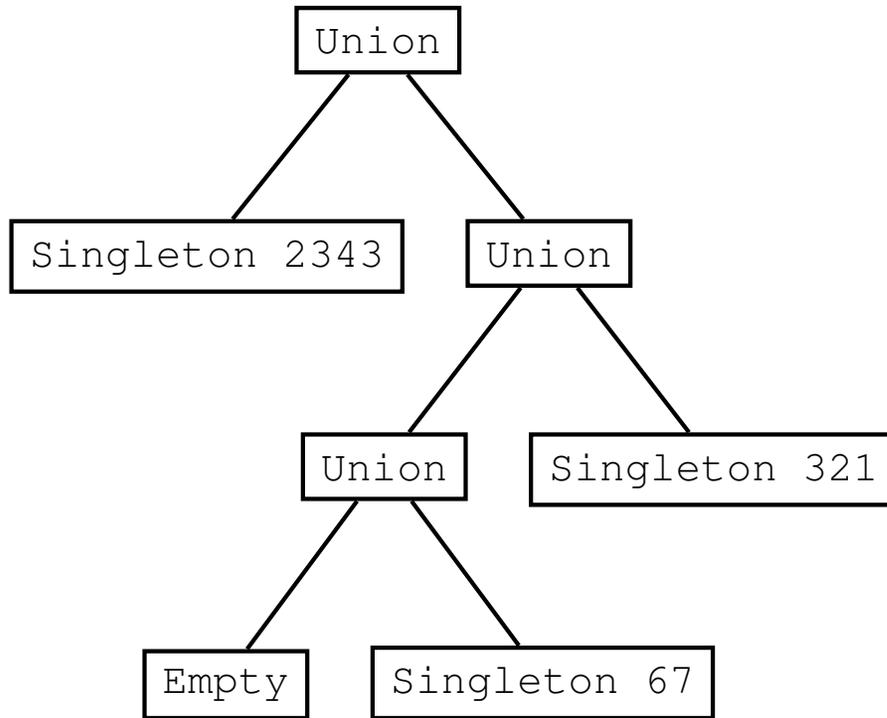
We consider the following subset of API operations:

```
val empty      : Ptset.t
val singleton  : int -> Ptset.t
val mem        : int -> Ptset.t -> bool
val add        : int -> Ptset.t -> Ptset.t
val remove     : int -> Ptset.t -> Ptset.t
val union      : Ptset.t -> Ptset.t -> Ptset.t
val inter      : Ptset.t -> Ptset.t -> Ptset.t
```

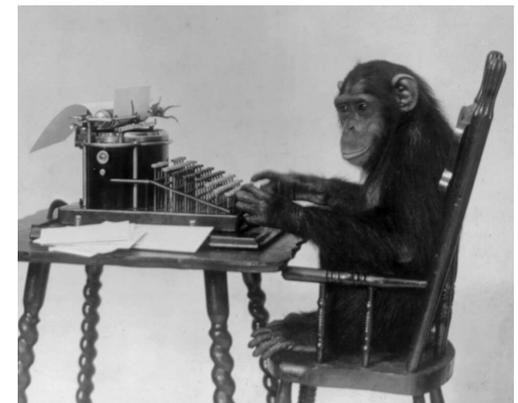
Each of them should be self-explanatory as set operations

The basic idea ... (1/2)

The basic idea is to generate arbitrary trees:



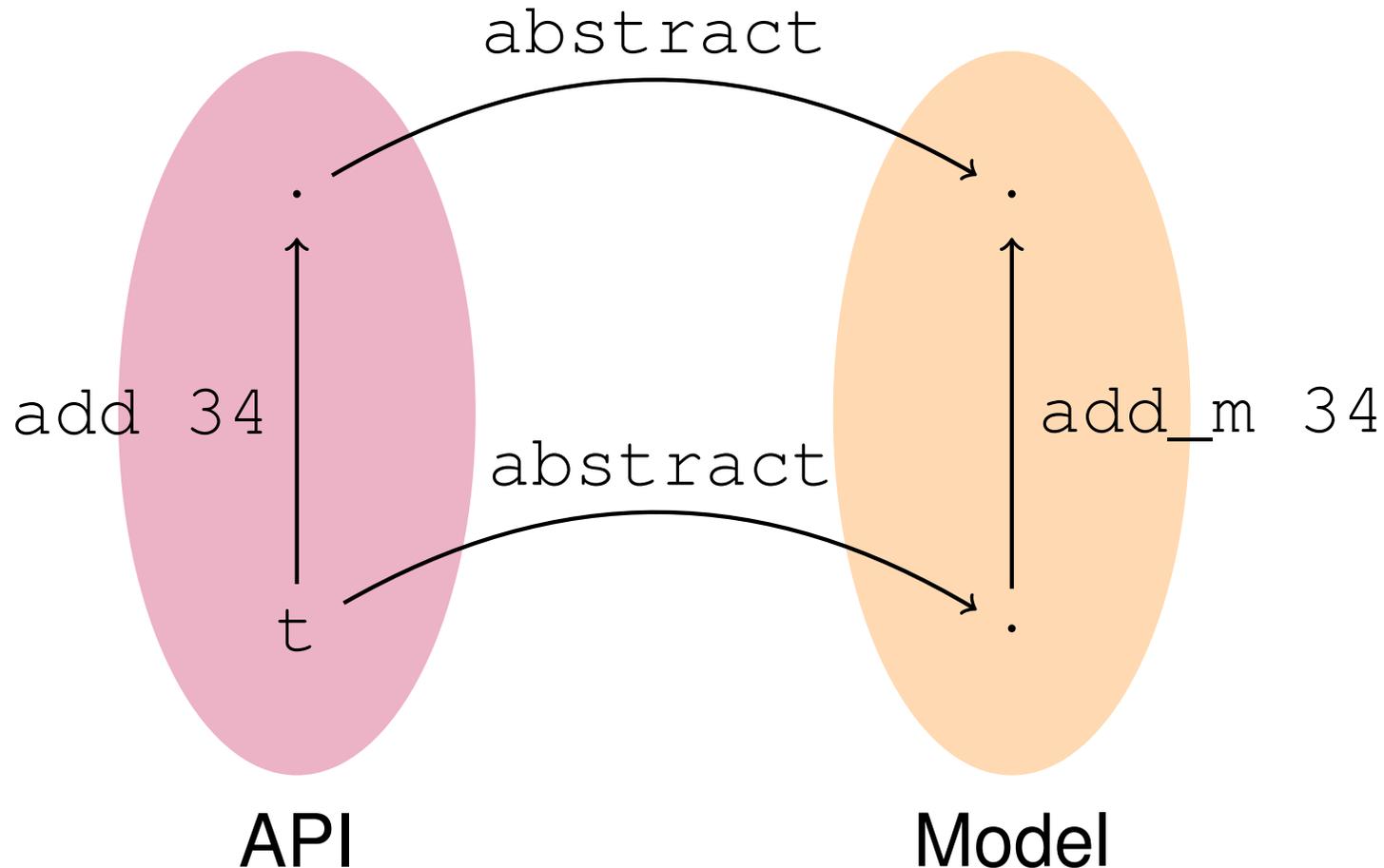
Empty



(apologies if they do not look sufficiently random)

The basic idea ... (2/2)

and ensure a commuting diagram for each such tree t :



for a suitable model and `abstract` operation, and

SDU suitably generalized beyond the `add` operation.

A datatype for set operations

We declare a datatype to represent set operations symbolically:

```
type instr_tree =  
  | Empty  
  | Singleton of int  
  | Add of int * instr_tree  
  | Remove of int * instr_tree  
  | Union of instr_tree * instr_tree  
  | Inter of instr_tree * instr_tree
```

We exclude a symbolic `Mem` constructor as a query will not give rise to a new set.

Advantage: easy to shrink + print counterexamples

API interpreter

It is straightforward to interpret trees over the API:

```
(* interpret : instr_tree -> Ptset.t *)
let rec interpret t = match t with
  | Empty          -> Ptset.empty
  | Singleton n    -> Ptset.singleton n
  | Add (n,t)      -> Ptset.add n (interpret t)
  | Remove (n,t)  -> Ptset.remove n (interpret t)
  | Union (t,t')  ->
    let s = interpret t in
    let s' = interpret t' in
    Ptset.union s s'
  | Inter (t,t')  ->
    let s = interpret t in
    let s' = interpret t' in
    Ptset.inter s s'
```

by mapping each symbolic node to the corresponding API operation (and suitable recursing).

A tree printer

It is straightforward to write a printer for the trees as a recursive descent:

```
(* to_string : instr_tree -> string *)
let rec to_string a = match a with
| Empty          -> "Empty"
| Singleton n    -> "Singleton_" ^ (string_of_int n)
| Add (n,t)      -> "Add_" ^ (string_of_int n) ^ ",_"
                    ^ (to_string t) ^ ")"
| Remove (n,t)   -> "Remove_" ^ (string_of_int n) ^ ",_"
                    ^ (to_string t) ^ ")"
| Union (t,t')   -> "Union_" ^ (to_string t) ^ ",_"
                    ^ (to_string t') ^ ")"
| Inter (t,t')   -> "Inter_" ^ (to_string t) ^ ",_"
                    ^ (to_string t') ^ ")"
```

This boilerplate could also have been auto-generated from the datatype definition.

A generator of arbitrary sets

```
(* tree_gen : int Gen.t -> instr_tree Gen.t *)
let tree_gen int_gen =
  Gen.sized (Gen.fix (fun rgen n -> match n with
    | 0 -> Gen.oneof [Gen.return Empty;
                     Gen.map (fun i -> Singleton i) int_gen]
    | _ ->
      Gen.frequency
        [(1, Gen.return Empty);
         (1, Gen.map (fun i -> Singleton i) int_gen);
         (2, Gen.map2 (fun i t -> Add (i,t)) int_gen (rgen (n-1)));
         (2, Gen.map2 (fun i t -> Remove (i,t)) int_gen (rgen (n-1)));
         (2, Gen.map2 (fun l r -> Union (l,r)) (rgen (n/2)) (rgen (n/2)));
         (2, Gen.map2 (fun l r -> Inter (l,r)) (rgen (n/2)) (rgen (n/2)));
        ]))
```

This recursive generator is

- **fueled** (with an integer n) to guarantee termination
- **weighted**, to increase chance of certain nodes
- **parameterized** over the integer generator `int_gen`

A tree shrinker (based on iterators)

```
(* tshrink : instr_tree -> instr_tree Iter.t *)  
let rec tshrink t = match t with  
  | Empty -> Iter.empty
```

A tree shrinker (based on iterators)

```
(* tshrink : instr_tree -> instr_tree Iter.t *)
let rec tshrink t = match t with
  | Empty -> Iter.empty
  | Singleton i ->
    (Iter.return Empty)
  <+> (Iter.map (fun i' -> Singleton i') (Shrink.int i))
```

A tree shrinker (based on iterators)

```
(* tshrink : instr_tree -> instr_tree Iter.t *)
let rec tshrink t = match t with
  | Empty -> Iter.empty
  | Singleton i ->
    (Iter.return Empty)
    <+> (Iter.map (fun i' -> Singleton i') (Shrink.int i))
  | Add (i,t) ->
    (Iter.of_list [Empty; t; Singleton i])
    <+> (Iter.map (fun t' -> Add (i,t')) (tshrink t))
    <+> (Iter.map (fun i' -> Add (i',t)) (Shrink.int i))
```

A tree shrinker (based on iterators)

```
(* tshrink : instr_tree -> instr_tree Iter.t *)
let rec tshrink t = match t with
| Empty -> Iter.empty
| Singleton i ->
  (Iter.return Empty)
  <+> (Iter.map (fun i' -> Singleton i') (Shrink.int i))
| Add (i,t) ->
  (Iter.of_list [Empty; t; Singleton i])
  <+> (Iter.map (fun t' -> Add (i,t')) (tshrink t))
  <+> (Iter.map (fun i' -> Add (i',t)) (Shrink.int i))
| Remove (i,t) ->
  (Iter.of_list [Empty; t])
  <+> (Iter.map (fun t' -> Remove (i,t')) (tshrink t))
  <+> (Iter.map (fun i' -> Remove (i',t)) (Shrink.int i))
| Union (t0,t1) ->
  (Iter.of_list [Empty;t0;t1])
  <+> (Iter.map (fun t0' -> Union (t0',t1)) (tshrink t0))
  <+> (Iter.map (fun t1' -> Union (t0,t1')) (tshrink t1))

(* Inter case omitted *)
```

where `<+>` is an alias for `Iter.append`.

Model + model interpreter

How do we model (read: give semantics to) a set?

We could, e.g., **model them as sorted lists.**

Model + model interpreter

How do we model (read: give semantics to) a set?

We could, e.g., **model them as sorted lists**.

API operations expressed over the model (suffix: `_m`):

```
let empty_m = []
let singleton_m i = [i]
let mem_m i s = List.mem i s
let add_m i s =
  if List.mem i s then s else List.sort compare (i::s)
let rec union_m s s' = match s, s' with
| [], _ -> s'
| _, [] -> s
| i::is, j::js -> if i < j then i::(union_m is s') else
  if i > j then j::(union_m s js) else
  i::(union_m is js)

(* ... *)
```

Model + model interpreter

How do we model (read: give semantics to) a set?

We could, e.g., **model them as sorted lists**.

API operations expressed over the model (suffix: `_m`):

```
let empty_m = []
let singleton_m i = [i]
let mem_m i s = List.mem i s
let add_m i s =
  if List.mem i s then s else List.sort compare (i::s)
let rec union_m s s' = match s, s' with
| [], _ -> s'
| _, [] -> s
| i::is, j::js -> if i < j then i::(union_m is s') else
                    if i > j then j::(union_m s js) else
                    i::(union_m is js)

(* ... *)
```

SDU  Finally: `let abstract t = Ptset.elements t`

The agreement properties

We can now test agreement of each of the API entries:

```
let singleton_test =  
  Test.make ~name:"singleton_test" ~count:10000  
  arb_int  
  (fun n -> abstract (Ptset.singleton n) = singleton_m n)
```

The agreement properties

We can now test agreement of each of the API entries:

```
let singleton_test =  
  Test.make ~name:"singleton_test" ~count:10000  
    arb_int  
    (fun n -> abstract (Ptset.singleton n) = singleton_m n)
```

```
let mem_test =  
  Test.make ~name:"mem_test" ~count:10000  
    (pair arb_tree arb_int)  
    (fun (t,n) ->  
      let s = interpret t in  
      Ptset.mem n s = mem_m n (abstract s))
```

The agreement properties

We can now test agreement of each of the API entries:

```
let singleton_test =
  Test.make ~name:"singleton_test" ~count:10000
  arb_int
  (fun n -> abstract (Ptset.singleton n) = singleton_m n)

let mem_test =
  Test.make ~name:"mem_test" ~count:10000
  (pair arb_tree arb_int)
  (fun (t,n) ->
    let s = interpret t in
    Ptset.mem n s = mem_m n (abstract s))

let union_test =
  Test.make ~name:"union_test" ~count:10000
  (pair arb_tree arb_tree)
  (fun (t,t') ->
    let s = interpret t in
    let s' = interpret t' in
    abstract (Ptset.union s s')
      = union_m (abstract s) (abstract s'))
```

Finally: time for some testing!

Let's run our model-based tests:

```
random seed: 362256415
generated error  fail  pass / total      time test name
[✓]      1      0      0  1 / 1      0.0s empty
[✓] 10000      0      0 10000 / 10000  0.0s singleton test
[✓] 10000      0      0 10000 / 10000  0.1s mem test
[✓] 10000      0      0 10000 / 10000  0.1s add test
[✓] 10000      0      0 10000 / 10000  0.1s remove test
[✓] 10000      0      0 10000 / 10000  0.1s union test
[✓] 10000      0      0 10000 / 10000  0.1s inter test
```

```
=====
success (ran 7 tests)
```

This is with `int_gen` generating integers uniformly.

Finally: time for some testing!

Let's run our model-based tests:

```
random seed: 362256415
```

```
generated error  fail  pass / total      time test name
[✓]      1      0      1 /      1      0.0s empty
[✓] 10000      0      10000 / 10000      0.0s singleton test
[✓] 10000      0      10000 / 10000      0.1s mem test
[✓] 10000      0      10000 / 10000      0.1s add test
[✓] 10000      0      10000 / 10000      0.1s remove test
[✓] 10000      0      10000 / 10000      0.1s union test
[✓] 10000      0      10000 / 10000      0.1s inter test
```

```
=====
success (ran 7 tests)
```

This is with `int_gen` generating integers uniformly.

Seems to work!

Finally: time for some testing!

Let's run our model-based tests:

```
random seed: 362256415
```

```
generated error  fail  pass / total  time test name
[✓]      1      0      1 /      1  0.0s empty
[✓] 10000      0      10000 / 10000  0.0s singleton test
[✓] 10000      0      10000 / 10000  0.1s mem test
[✓] 10000      0      10000 / 10000  0.1s add test
[✓] 10000      0      10000 / 10000  0.1s remove test
[✓] 10000      0      10000 / 10000  0.1s union test
[✓] 10000      0      10000 / 10000  0.1s inter test
```

```
=====
success (ran 7 tests)
```

This is with `int_gen` generating integers uniformly.

Seems to work!

If we repeat these 60.001 tests 10 times the implementation **still seems to function** correctly.

Best practice?

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

Best practice?

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

What are the corner/boundary cases of our
bit-fiddling Patricia tree data structure?

Best practice?

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

What are the corner/boundary cases of our
bit-fiddling Patricia tree data structure?

Ideally `arb_int` should generate these.

Best practice?

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

What are the corner/boundary cases of our
bit-fiddling Patricia tree data structure?

Ideally `arb_int` should generate these.

Here’s an attempt at a weighted integer generator:

```
let arb_int =  
  frequency [(5, small_signed_int);  
            (3, int);  
            (1, oneofl [min_int; max_int])]
```

Best practice?

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

What are the corner/boundary cases of our
bit-fiddling Patricia tree data structure?

Ideally `arb_int` should generate these.

Here’s an attempt at a weighted integer generator:

```
let arb_int =  
  frequency [(5, small_signed_int);  
            (3, int);  
            (1, oneof1 [min_int; max_int])]
```

This definition also has a reasonable chance of
generating duplicate numbers.

Rerunning our tests...

Ooops:

--- Failure -----

Test union test failed (68 shrink steps):

```
(Add (-4611686018427387904, Singleton 0),  
Add (-4611686018427387904, Singleton 1))
```

We recognize `-4611686018427387904` as `min_int`.

Rerunning our tests...

Ooops:

--- Failure -----

Test union test failed (68 shrink steps):

```
(Add (-4611686018427387904, Singleton 0),  
Add (-4611686018427387904, Singleton 1))
```

We recognize `-4611686018427387904` as `min_int`.

If this doesn't yield `{min_int, 0, 1}` what does it yield?

Rerunning our tests...

Ooops:

```
--- Failure -----
```

```
Test union test failed (68 shrink steps):
```

```
(Add (-4611686018427387904, Singleton 0),  
Add (-4611686018427387904, Singleton 1))
```

We recognize `-4611686018427387904` as `min_int`.

If this doesn't yield `{min_int, 0, 1}` what does it yield?

Over `Ptset` we actually get:

$$\{\text{min_int}, 0\} \cup \{\text{min_int}, 1\} = \{\text{min_int}, 0, \text{min_int}, 1\}$$

The problem (union just calls merge)

```
let rec merge = function
| t1,t2 when t1==t2 -> t1
| Empty, t   -> t
| t, Empty   -> t
| Leaf k, t   -> add k t
| t, Leaf k   -> add k t
| (Branch (p,m,s0,s1) as s), (Branch (q,n,t0,t1) as t) ->
  if m == n && match_prefix q p m then
    (* The trees have the same prefix. Merge the subtrees. *)
    Branch (p, m, merge (s0,t0), merge (s1,t1))
  else if m < n && match_prefix q p m then
    (* [q] contains [p]. Merge [t] with a subtree of [s]. *)
    if zero_bit q m then
      Branch (p, m, merge (s0,t), s1)
    else
      Branch (p, m, s0, merge (s1,t))
  else if m > n && match_prefix p q n then
    (* [p] contains [q]. Merge [s] with a subtree of [t]. *)
    if zero_bit p n then
      Branch (q, n, merge (s,t0), t1)
    else
      Branch (q, n, t0, merge (s,t1))
  else
    (* The prefixes disagree. *)
    join (p, s, q, t)
```

The problem (union just calls merge)

```
let rec merge = function
| t1,t2 when t1==t2 -> t1
| Empty, t -> t
| t, Empty -> t
| Leaf k, t -> add k t
| t, Leaf k -> add k t
| (Branch (p,m,s0,s1) as s), (Branch (q,n,t0,t1) as t) ->
  if m == n && match_prefix q p m then
    (* The trees have the same prefix. Merge the subtrees. *)
    Branch (p, m, merge (s0,t0), merge (s1,t1))
  else if m < n && match_prefix q p m then
    (* [q] contains [p]. Merge [t] with a subtree of [s]. *)
    if zero_bit q m then
      Branch (p, m, merge (s0,t), s1)
    else
      Branch (p, m, s0, merge (s1,t))
  else if m > n && match_prefix p q n then
    (* [p] contains [q]. Merge [s] with a subtree of [t]. *)
    if zero_bit p n then
      Branch (q, n, merge (s,t0), t1)
    else
      Branch (q, n, t0, merge (s,t1))
  else
    (* The prefixes disagree. *)
    join (p, s, q, t)
```

An elegant fix

When I showed the example to Filliâtre he provided an elegant fix within hours.

For the purposes of comparing branching bits, we can instead use this function:

```
let unsigned_lt n m = n >= 0 && (m < 0 || n < m)
```

Rerunning our tests on the fixed version finds no issues:

```
random seed: 175347559
```

generated	error	fail	pass	/	total	time	test name
[✓]	1	0	0	1 /	1	0.0s	empty
[✓]	10000	0	0	10000 /	10000	0.0s	singleton test
[✓]	10000	0	0	10000 /	10000	0.1s	mem test
[✓]	10000	0	0	10000 /	10000	0.1s	add test
[✓]	10000	0	0	10000 /	10000	0.1s	remove test
[✓]	10000	0	0	10000 /	10000	0.1s	union test
[✓]	10000	0	0	10000 /	10000	0.1s	inter test

Model-based quickchecking in general

Within the Erlang community model-based testing is quite popular. **They don't write them like we did here.**

Quviq's commercial Quickcheck port for Erlang comes with a **state-machine DSL for writing model-based tests.**

Model-based quickchecking in general

Within the Erlang community model-based testing is quite popular. **They don't write them like we did here.**

Quviq's commercial Quickcheck port for Erlang comes with a **state-machine DSL for writing model-based tests.**

Their DSL design has since been adapted by two open source Erlang libraries: Proper and Triq.

Model-based quickchecking in general

Within the Erlang community model-based testing is quite popular. **They don't write them like we did here.**

Quviq's commercial Quickcheck port for Erlang comes with a **state-machine DSL for writing model-based tests.**

Their DSL design has since been adapted by two open source Erlang libraries: Proper and Triq.

A number of **state-machine frameworks** for other languages exists:

ScalaCheck (for Scala), Hypothesis (for Python), Hedgehog (for Haskell, R, ...), fast-check (for JS/TS), Lua-QuickCheck (for Lua), RapidCheck (for C++), ...

(More about state-machine testing next time...)

Summary

We've studied an application of QuickCheck with a surprising outcome: **a 19 year old bug** in Patricia trees

Rather than randomly testing a single operation, the approach **lifts randomization to test** (the interaction of) **a sequence of operations**

A symbolic representation lets us shrink such sequences

The example also underlines the **importance of generators** with QuickCheck.

Simon Cruanes has since **reused the model** to test another Patricia tree implementation (from CC).

The testing code is available online: