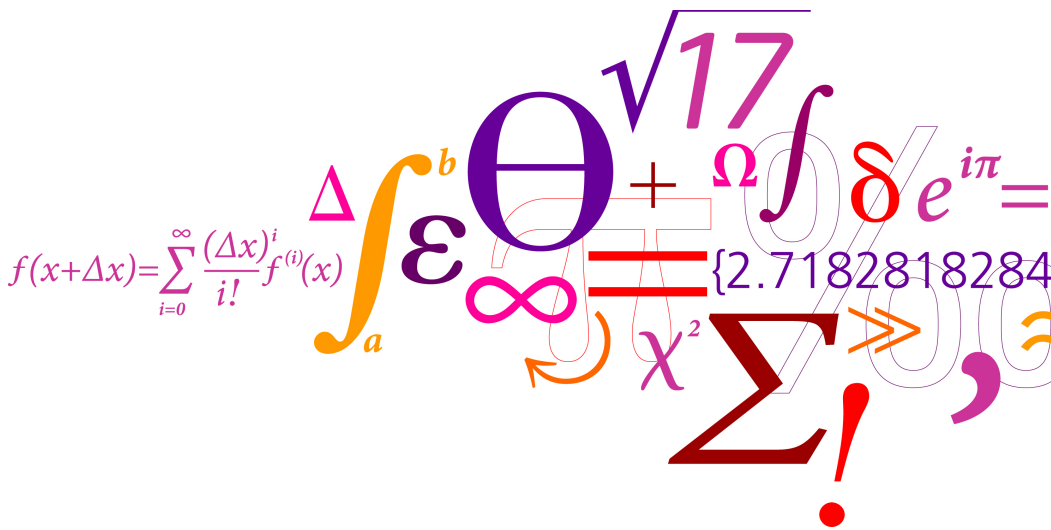
02257 Applied Functional Programming

Property-based testing: Generators for well-formed structures

Michael R. Hansen



DTU Compute
Department of Applied Mathematics and Computer Science

Well-formed programs

Suppose you are implementing a programming language:

What abstract (generic) properties are you interested in validating?

In generic terms one could, for example, validate that

- meaningful programs behave as expected

- What does meaningful mean?
- How could generators for such programs be constructed?

An example: expressions with local declarations (1)

```
type E =  | V of string  | C of int
          | Let of string * E * E
          | Add of E * E;;

type Env = Map<string,int>;;

//eval: E -> Env -> int
let rec eval e m =
         match e with
         | V x -> Map.find x m
         | C n -> n
         | Let(x,e1,e) -> let v1 = eval e1 m
                          eval e (Map.add x v1 m)
         | Add(e1,e2)  -> eval e1 m + eval e2  m;;
```

- A expression *e* is closed if it does not contain a free variable

We are interested in showing:

for all closed expressions *e*, environments *m*:

`eval` ***e m*** = `eval` ***e Map.empty***

An example: expressions with local declarations (2)

The property is easy to express

```
let rec free =
    function
    | C _           -> Set.empty
    | V x           -> Set.singleton x
    | Add(e1,e2)    -> Set.union (free e1) (free e2)
    | Let(x,e1,e2) -> Set.union (free e1)
                                (Set.remove x (free e2));;

let closed e = free e = Set.empty;;

let closedProp e m = eval e m = eval e Map.empty;;
```

Provided

- we can generate random samples of closed programs

Overview

Generators for

- Small strings
- Small environments
- “Already defined” variables
- Closed expressions

A generator for small strings

```
let charsSeqGen c1 c2 = seq { for c in c1 .. c2 do
                                yield gen { return c} }

let myCharGen =
   Gen.oneof [gen { return! Gen.oneof (charsSeqGen 'a' 'z')
                    gen { return! Gen.oneof (charsSeqGen 'A' 'Z')
```

- A sequence (computation) expression generates a sequence of generators

A generator for strings of length $<= 4$:

```
let mySmallStringGen =
   gen { let! i = Gen.choose (1, 4)
         let! cs = Gen.listOfLength i myCharGen
         let ss = List.map string cs
         return String.concat "" ss }
```

It should be possible to simplify this

A generator for small environments

```
let mySmallEnvGen  =
   gen { let! i = Gen.choose (0, 5)
         let! vs = Gen.listOfLength i mySmallStringGen
         let! ns = Gen.listOfLength i Arb.generate<int>
         return Map.ofList (List.zip vs ns)      };;
```

A computation expression, like `gen {let!···}` defines a recipe for generating environments:

- pick randomly a number from $0, 1, 2, 3, 4, 5$ call it i
- generate a random list of length i of strings call it vs
- generate a random list of length i of numbers call it ns
- return a map $\cdots$

No dish is cooked yet — computation expressions are lazy

- This is useful/necessary when declaring recursive objects that are not functions:

infinite sequences, parsers, generators, ...

A generator for variables and constants

A generator that pick between a non-empty list of variables *vs*:

```
let myVarGen vs =
   gen { let! i = Gen.choose(0, List.length vs - 1)
         return vs.[i] }
```

A leaf-generator that can generate only know variables:

```
let myCGen    = Gen.map C Arb.generate<int>;;

let myVGen vs = Gen.map V (myVarGen vs)

let myLeafGen vs =
    if vs<>[]
    then Gen.oneof [myCGen ; myVGen vs]
    else myCGen;;
```

Now we can generate closed expressions

A generator for closed expressions

- A parameter *vs* keeps track of variables for which we have bindings

The generator is now obtained by already introduced techniques:

```
let myEGen =
    let rec myE vs n =
        match n with
        | 0 -> myLeafGen vs
        | _ -> Gen.oneof [myLeafGen vs;
                          Gen.map2 (fun x y -> Add(x,y))
                                   (myE vs (n/2))
                                   (myE vs (n/2));
                          myLetGen vs n]

    and myLetGen vs n = gen { let! x  = mySmallStringGen
                              let! e1  = myE vs (n/2)
                              let! e = myE (x::vs) (n/2)
                              return Let(x,e1,e) }

    Gen.sized (myE []);;
```

“Controlled” generators for

- small samples of various kinds
- well-formed programs of certain kinds

Notice computation expressions

```
builder {let! ...}
```

are lazy