Random Generators for Dependent Types

Peter Dybjer, Qiao Haiyan, and Makoto Takeyama

Cover meeting
17 March, 2005

Random Generators in Agda/Alfa

A random generator for a type D is a function

$$f :: \texttt{Rand} \rightarrow D$$

where `Rand` is the type of random seeds.

A random generator for an indexed family of types $P\ i$ for $i :: I$ is a function

$$f :: \texttt{Rand} \rightarrow \texttt{sig}\ \{i :: I;\ p :: P\ i\}$$

Remark: $P\ i$ can be empty.

We focus on inductively defined dependent types (inductive families)

Binary trees as random seeds

`Rand` is implemented as the set of binary trees of natural numbers:

```
Rand :: Set = data Leaf (k :: Nat)                    :: Rand
                 | Node (k :: Nat) (l, r :: Rand) :: Rand
```

A generator for lists

```
List(A::Set) :: Set = data nil :: List A
                         | cons (a::A) (as::List A) :: List A

genList :: (A :: Set) -> (Rand -> A) -> Rand -> List A
genList A g (Leaf _)      = nil
genList A g (Node _ l r) = cons (g l) (genList A g r)
```

This is an instance of a generic strategy for parameterized term algebras ("algebraic data types"): randomly choose a constructor and generate its arguments by using either parameter generators, or by the generators for previously defined simple sets, or by recursive calls, all using sub-seeds of the given seed. When the seed is not large enough, it terminates by choosing a non-recursive constructor.

Inductive families

General form of formation rule:

$$
\begin{aligned}
P :: \quad & (A_1 :: \sigma_1) \to \cdots \to (A_N :: \sigma_N) \to \\
& (a_1 :: \alpha_1) \to \cdots \to (a_M :: \alpha_M) \to \\
& \texttt{Set}
\end{aligned}
$$

General form of introduction rule (ordinary, finitary inductive definitions)

$$
\begin{aligned}
intro :: \quad & (A_1 :: \sigma_1) \to \cdots \to (A_N :: \sigma_N) \to \\
& (b_1 :: \beta_1) \to \cdots \to (b_K :: \beta_K) \to \\
& (u_1 :: P\ q_{11}\ \ldots\ q_{1M}) \to \\
& \ldots \\
& (u_L :: P\ q_{L1}\ \ldots\ q_{LM}) \to \\
& P\ p_1\ \ldots\ p_M
\end{aligned}
\qquad (P\text{-Intro}_{intro})
$$

The inductive family of finite sets

The indexed family $\texttt{Fin}\ n$ ($n :: \texttt{Nat}$) of sets with n elements:

```
Fin :: Nat -> Set
  = data C0 (n :: Nat)                    :: Fin (succ n)
       | C1 (n :: Nat) (i :: Fin  n) :: Fin (succ n)
```

Rules

– formation $\texttt{Fin} :: \texttt{Nat} \to \texttt{Set}$ $(N = 0,\ M = 1)$

– introduction $\mathrm{C}_0 :: (n :: \texttt{Nat}) \to \texttt{Fin}\,(\texttt{succ}\,n)$ $(K = 1,\ L = 0)$

$\mathrm{C}_1 :: (n :: \texttt{Nat}) \to \texttt{Fin}\ n \to \texttt{Fin}\,(\texttt{succ}\,n)$ $(K = 1,\ L = 1)$

The inductive family of untyped lambda terms

`Term` n (n :: `Nat`) represents the set of lambda terms with at most n free variables (using de Bruijn indices).

```
Term :: Nat -> Set
 = data var (n :: Nat) (i :: Fin  (succ n)) :: Term (succ n)
      | abs (n :: Nat) (t :: Term (succ n)) :: Term n
      | app (n :: Nat) (t1, t2 :: Term n)   :: Term n
```

The inductive family of vectors

An example with one parameter type A is the `Nat`-indexed family `Vec` where elements of `Vec` n are length-n vectors.

```
Vec (A :: Set) :: Nat -> Set
  = data  nil' :: Vec A  zero
        | cons' (n :: Nat) (a :: A) (as :: Vec A n)
                :: Vec A (succ n)
```

A generator for the inductive family of vectors

```
genVec :: (A :: Set) -> (Rand -> A) ->
          Rand -> sig { ind :: Nat; obj :: Vec A ind }

genVec A g (Leaf _ )    = struct ind = zero; obj = nil'
genVec A g (Node _ l r) = let { as = genVec A g r } in
                           struct ind = succ  as.ind
                                  obj = cons' as.ind (g l) as.obj
```

The generator maps the parameter generator g to the given tree seen as a (right-spine) list of (left) subtrees.

The general form of a generator for parameterized inductive families

A generator for the family

$$\begin{aligned} P :: \quad & (A_1 :: \texttt{Set}) \to \cdots \to (A_N :: \texttt{Set}) \to \\ & (a_1 :: \alpha_1) \to \cdots \to (a_M :: \alpha_M) \to \\ & \texttt{Set} \end{aligned}$$

is a function

$$\begin{aligned} genP :: \quad & (A_1 :: \texttt{Set}) \to \cdots \to (A_N :: \texttt{Set}) \to \\ & (g_1 :: \texttt{Rand} \to A_1) \to \cdots \to (g_N :: \texttt{Rand} \to A_N) \to \\ & \texttt{Rand} \to \texttt{sig}\ \{a_1 :: \alpha_1;\ \cdots;\ a_M :: \alpha_M;\ p :: P\ a_1\ \ldots\ a_M\} \end{aligned}$$

where A_i are parameters and g_i are *parameter generators*.

Generators for Inhabited Inductive Families

If $P\ i$ is inhabited for all $i :: I$, then a surjective generator

$$genP :: \texttt{Rand} \to \texttt{sig}\ \{ind :: I;\ obj :: P\ ind\}$$

can be defined from a surjective generator $genP'\ i$ for each $P\ i$. It first generates an index using $genI$, then an element of $P\ i$ using $genP'\ i$.

A generator for finite sets

Fin $(\texttt{succ}\, n)$ is inhabited for all $n :: \texttt{Nat}$. A surjective generator for this family can be defined by using a generator for Nat to generate the index n and use it as input for the following generator for the family:

```
genFin' :: (n :: Nat) -> Rand -> Fin (succ n)
genFin'  zero      _             = C0  zero
genFin' (succ m) (Leaf _)      = C0 (succ m)
genFin' (succ m) (Node _ l r) = C1 (succ m) (genFin' m l)
```

The inductive family of balanced binary trees

```
Bal :: (n :: Nat) -> Set = data
   Empty                                   :: Bal zero
  | C00 (t1, t2 :: Bal n)                  :: Bal (succ n)
  | C01 (t1 :: Bal n) (t2 :: Bal (succ n)) :: Bal (succ (succ n))
  | C10 (t1 :: Bal (succ n)) (t2 :: Bal n) :: Bal (succ (succ n))
```

`Bal` n is inhabited for all n. So we can first generate an n and then an element of `Bal` n using the generator `genBal'` on the next page.

A generator for balanced binary trees

```
genBal' :: (n :: Nat) -> Rand -> Bal n
genBal'  zero              _              = Empty
genBal' (succ zero)        _              = C00 Empty Empty
genBal' (succ (succ n)) (Leaf k)          =
   let t  = genBal' (succ n) (Leaf k) in  C00 t t
genBal' (succ (succ n)) (Node k l r) =
   let b1 = genBal' (succ n) l
       b2 = genBal' (succ n) r
       b3 = genBal'       n  r
   in  choice3 k (C00 b1 b2) (C01 b3 b1) (C10 b1 b3)
```

where `choice3` $k\ a_0\ a_1\ a_2 = a_{(k \bmod 3)}$

A generator for lambda terms

`Term` n is also inhabited for each n. So again, a generator can be written by first generating an n and then using a generator:

```
genTerm' :: (n :: Nat) -> Rand -> Term n
```

```
genTerm’ zero     (Leaf _)       = abs zero (var zero (C0 zero))
genTerm’ zero     (Node k l r) =
    let t1 :: Term (succ zero) = genTerm’ (succ zero) l
        t2 :: Term       zero  = genTerm’       zero  l
        t3 :: Term       zero  = genTerm’       zero  r
    in  choice2 k (abs zero t1) (app zero t2 t3)
genTerm’ (succ m) (Leaf k)     = var m (genFin’ m (Leaf k))
genTerm’ (succ m) (Node k l r) =
    let t1 :: Term (succ (succ m)) = genTerm’ (succ (succ m)) l
        t2 :: Term       (succ m)  = genTerm’       (succ m)  l
        t3 :: Term       (succ m)  = genTerm’       (succ m)  r
    in  choice2 k (abs (succ m) t1) (app (succ m) t2 t3)
```

Simple inductive families

- The formation rule $P :: I \to \texttt{Set}$ has no parameter, and the single index set I is simple.

- Each introduction rule has the form

$$\begin{aligned} intro :: \quad & (i_1 :: I) \to \cdots \to (i_K :: I) \to \\ & (u_1 :: P\ i_1) \to \cdots (u_K :: P\ i_K) \to \\ & P\ p \end{aligned}$$

- P is not empty; there must be a constructor without arguments.

- But $P\,i$ can be empty for some i.

The inductive family (predicate) of even numbers

```
Even :: Nat -> Set
= data C0                               :: Even zero
     | C1 (n :: Nat) (p :: Even n) :: Even (succ (succ n))
```

A generator of even numbers and proof objects for evenness:

```
genEven :: Rand -> sig { ind :: Nat; obj :: Even ind }
genEven (Leaf k) = struct ind = zero; obj = C0
genEven (Node k l r) = let g1 = genEven l
       in struct ind = succ (succ g1.ind)
                 obj = C1 g1.ind g1.obj
```

Another generator of elements of finite sets

The inductive family of finite sets is a simple inductive family so we can write a generator using the same technique. In this case, the generator has the type:

```
genFin :: Rand -> sig ind :: Nat; obj :: Fin ind
```

and is defined as follows:

```
genFin (Leaf k)     = struct ind = genNat (Leaf k); obj = C0 ind
genFin (Node k l r) = let
           g1 :: GFin = genFin r
        in  struct ind = succ g1.ind; obj = C1 g1.ind g1.obj
```

Inductive Definitions and Logic Programs

- The motivation for considering simple inductive families is to have as few constraints as possible between indices and elements, in order to facilitate random generation.

- However, representing intricate constraints is often the very purpose of defining an indexed family.

- To cover some of those cases, we introduce unification and backtracking in a generation algorithm.

- The idea is based on the relationship between inductive families and logic programs (Hagiya and Sakurai 1984).

Horn clauses for theorems

Horn clauses corresponding to to the axioms and inference rules of a system due to Lukasiewicz:

```
thm((P => Q) => ((Q => R) => (P => R))).
thm((~P => P) => P).
thm(P => (~P => Q)).
thm(Q) :- thm(P), thm(P => Q).
```

Running the query `thm(X)` on a Prolog implementation, we can obtain theorems (schemas) as solutions for `X`; for example

```
X = (((_A => _B) => (_C => _B)) => _D) => ((_C => A) => _D)
```

Type theory and logic programs

Type theory	Logic programming
Family of sets $P :: D \rightarrow \texttt{Set}$	Predicate P
an introduction rule	a Horn clause
inductive definition of P	logic program defining P

We call an inductive family arising from a logic program a Horn inductive family. This is a subset of the general class of inductive families considered in type theory.

An inductive family of theorems

Formula is an inductively defined set of formulas.

```
Thm :: Formula -> Set = data
   ax1 (p, q, r :: Formula)
   :: Thm ((p => q) => ((q => r) => (p => r)))
 | ax2 (p       :: Formula)
   :: Thm ((-p => p) => p)
 | ax3 (p, q    :: Formula)
   :: Thm (p => (-p => q))
 | mp  (p, q    :: Formula) (x :: Thm p) (y :: Thm (p => q))
   :: Thm q
```

Another connection between inductive families and logic programs

```
nat(zero).
nat(succ(X)) :-  nat(X).
formula(var(P)) :- nat(P).
formula(~P) :- formula(P).
formula(P => Q) :- formula(P), formula(Q).

thm1((P => Q) => ((Q => R) => (P => R)), ax1(P,Q,R))
                   :- formula(P), formula(Q), formula(R).
thm1((~P => P) => P, ax2(P)) :- formula(P).
thm1(P => (~P => Q), ax3(P,Q)) :- formula(P), formula(Q).
thm1(Q, mp(P,Q,X,Y)) :- thm1(P, X), thm1(P => Q, Y).
```

Generating theorems and derivations

We can obtain a theorem and its derivation as solutions for `X` and `Y` in the query `thm1(X, Y)`. For example,

```
X = (var(zero) => var(zero)) =>
     ((var(zero) => var(zero)) => (var(zero) => var(zero)))
Y = ax1(var(zero), var(zero), var(zero))
```

So the problem of generating a pair `(X :: Formula, Y :: Thm X)` in dependent type theory corresponds to the task of solving a query `thm1(X, Y)`. In this way, we can use a Prolog interpreter to generate elements patterns of Horn inductive families. If we randomise the Prolog interpreter and randomly instantiate the patterns, then we get a random generator for Horn inductive families.

A generator for theorems

It is based on a more general generator for theorem *patterns*, that is, formula patterns whose ground instantiations are all theorems.

$$\texttt{genTP :: Rand -> (t :: Pat) -> Maybe (}\sigma\texttt{ :: Subst, ThmPat t[}\sigma\texttt{])}$$

generates theorem patterns which fit into a given formula pattern $t :: \texttt{Pat}$. With a seed s, $\texttt{genTP}\, s\, t$ either *succeeds* and returns some $\texttt{Just}\,(\sigma, d)$, or *fails* and returns `Nothing`. In case of success, we have a theorem pattern $t[\sigma]$ with derivation $d :: \texttt{ThmPat}\, t[\sigma]$.

The type of formula patterns `Pat` is a simple set with four constructors. We have the same three constructors as `Formula` but also a fourth constructor $\texttt{X} :: \texttt{Nat} \to \texttt{Pat}$ for *pattern variables* (logical variables denoting indeterminate formulas).

Concluding remarks

- When a set or a family is (Horn) inductively generated we can also randomly generate or recursively enumerate its elements.

- This is a generic technique. A generator can be written for the whole class of Horn inductive families. (Efficiency is not guaranteed, just like in Prolog.)

- The technique does not only apply to dependent type theory. A variant can be used in predicate logic with inductively defined predicates.