Random Generators for Dependent Types

Peter Dybjer[1], Qiao Haiyan[1] and Makoto Takeyama[2]

[1] Department of Computing Science,
Chalmers University of Technology,
412 96 Göteborg, Sweden
{peterd,qiao}@cs.chalmers.se
[2] Research Centre for Verification and Semantics
National Institute of Advanced Industrial Science and Technology
Nakoji 3-11-46, Amagasaki Hyogo, 661-097 Japan
makoto.takeyama@aist.go.jp

Abstract. We show how to write surjective random generators for several different classes of inductively defined types in dependent type theory. We discuss both non-indexed (simple) types and indexed families of types. In particular we show how to use the relationship between indexed inductive definitions and logic programs: the indexed inductive definition of a type family corresponds to a logic program, and generating an object of a type in the family corresponds to solving a query for the logic program. As an example, we show how to write a surjective random generator for theorems in propositional logic by randomising the Prolog search algorithm.

1 Introduction

Random testing is a quick way to find bugs both in programs and their specifications [4]. It also facilitates proof development in type theory [8, 9]. When doing random testing in type theory, we need to write random generators for types. A random generator for a type D is a function that has random seeds as inputs and objects of D as outputs. When D is a simple datatype, the type of the generator is $\texttt{Rand} \to D$ [8], where `Rand` is the type of random seeds. In the case of a dependent type (an indexed family of types) $P\ i$ for $i :: I$ (we write $i :: I$ to indicate that i is an object of type I), we wish to generate a pair (i, p) of indices $i :: I$ and objects $p :: P\ i$. That is, the type of the generators for the dependent type P is $\texttt{Rand} \to \texttt{sig}\ \{i :: I,\ p :: P\ i\}$, where $\texttt{sig}\ \{i :: I,\ p :: P\ i\}$ denotes a dependent record type: the first field has type I and the second field has a type $P\ i$ that depends on the value i of the first field. However, since $P\ i$ can be empty, we need to decide how to generate an index i so that this is not the case. In this paper, we discuss some difficulties that arise when writing generators for dependent types and present some solutions for several classes of inductive definitions (see Section 4–7). In particular, we get a very general class of generators by using the fact that generating objects of inductively defined indexed families is similar to solving queries in logic programs. This is because an ordinary inductive definition of an indexed family of types (a predicate under the Curry-Howard correspondence) can be seen as a logic program and vice versa [10]. We also discuss how to use logic programming techniques for writing generators.

Examples are implemented in Agda/Alfa [5, 11], an interactive proof editor based on Martin-Löf type theory. We slightly modify its concrete syntax to make it easier to follow the examples. The formal proofs which are omitted in the paper can be found at `http://www.cs.chalmers.se/~qiao/papers/`.

Acknowledgement. This research is partly supported by the Cover project funded by SSF (the Swedish Foundation for Strategic Research). The aim of the Cover project is to build tools where random testing and proving (automatic and interactive) can be combined, see `http://coverproject.org/`. In particular we develop tools based on dependent type theory, and we therefore need to develop random generators for dependent types.

2 Inductive Families

In this section, we briefly describe the scheme for introducing new set formers in Martin-Löf's dependent type theory given by Dybjer [6]. We follow the usual terminology where a "set" is a small type. Sets are either inductively defined or formed from previously defined sets by dependent function set formation and dependent record set formation. In this article we restrict ourself to ordinary (or finitary) inductive definitions. See [6] for a discussion about ordinary vs. generalised (or infinitary) inductive definitions. See also [7] for a discussion of further generalising the notion of an inductive definition in dependent type theory.

We will only show the formation rule and the introduction rules, and omit the elimination rules and equality rules. The reader is referred to [6] for details.

The dependent type theory here is based on the logical framework for Martin-Löf type theory [12] and has four forms of judgements: $\sigma :: \texttt{Type}$, $p :: \sigma$, $\sigma = \tau$ and $p = q :: \sigma$.

The rules of type formation are the following:

- $\texttt{Set} :: \texttt{Type}$,
- if $\alpha :: \texttt{Set}$, then $\alpha :: \texttt{Type}$,
- if $\sigma :: \texttt{Type}$ and $\tau[A] :: \texttt{Type}$ under the assumption $A :: \sigma$, then $(A :: \sigma) \to \tau[A] :: \texttt{Type}$ (dependent function type) and $\texttt{sig}\ \{A :: \sigma;\ B :: \tau[A]\} :: \texttt{Type}$ (dependent record type, also called *signature*).

Notation:

- We mostly use letters $\sigma, \tau, \cdots$ for types; $\alpha, \beta, \cdots$ for sets (observe that sets are special types); $p, q, \cdots$ for elements of a set; $A, B, \cdots$ for variables of a type; and $a, b, u \cdots$ for variables of a set.

– We write $\tau[A]$ when we emphasise that τ may depend on variable A (that is, A may occur free in τ). This however is optional: τ may depend on any variable in scope regardless of the notation. The result of substituting the object s for A in τ is written $\tau[s/A]$.
– The general form of a signature is $\texttt{sig}\ \{A_1 :: \sigma_1;\ \cdots;\ A_N :: \sigma_N\}$. It has as objects *records* (also called *structures*) $\texttt{struct}\{A_1 = s_1;\ \cdots;\ A_N = s_N\}$ where $s_i :: \sigma_i[s_1/A_1, \cdots, s_{i-1}/A_{i-1}]$. A structure is a labelled tuple of objects of appropriate types. The dot operation $(-).A_i$ selects its A_i component; writing r for the structure above, we have that $r.A_i = s_i$.
– A nondependent function type, written $\sigma \to \tau$, is the special case of $(A :: \sigma) \to \tau[A]$ where A does not occur in τ.

2.1 Formation Rule

For each set former P, there is one formation rule that has the form

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

where σ_i are types and α_i are sets. We call A_i *parameters* and a_i *indices.* For readability, we omit the parameters and write $P\ a_1\ \ldots\ a_M$ instead of $P\ A_1\ \ldots\ A_N\ a_1\ \ldots\ a_M$.

2.2 Introduction Rules

There are finitely many introduction rules for each set former. Each introduction rule for the set former P above has the form

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

where β_i are sets, $p_j :: \alpha_j[p_1/a_1, \cdots, p_{j-1}/a_{j-1}]$ $(1 \leq j \leq M)$, and similarly for q_{ij} for each i. We call b_i *non-recursive* and u_i *recursive* arguments of the constructor *intro*.

2.3 Examples

We show some instances of the general schema [6] and how they are written in Agda/Alfa [5, 11].

Example 1 (Natural numbers). The set Nat of natural numbers has no parameters and indices. The rules are

– formation $\quad$ Nat :: Set $\quad$ ($N = M = 0$; in Nat-Formation)
– introduction $\quad$ zero :: Nat $\quad$ ($K = L = 0$; in Nat-Intro$_{\texttt{zero}}$)
$\quad$ succ :: Nat → Nat $\quad$ ($K = 0,\ L = 1$; in Nat-Intro$_{\texttt{succ}}$)

The concrete syntax in Agda/Alfa is

```
Nat :: Set = data zero              :: Nat
                | succ (n :: Nat) :: Nat
```

Example 2 (Finite sets). The indexed family Fin n (n :: Nat) of sets with just n elements has the following rules:

– formation $\quad$ Fin :: Nat → Set $\quad$ ($N = 0,\ M = 1$)
– introduction $\quad$ C_0 :: (n :: Nat) → Fin(succ n) $\quad$ ($K = 1,\ L = 0$)
$\quad$ C_1 :: (n :: Nat) → Fin n → Fin(succ n) $\quad$ ($K = 1,\ L = 1$)

The Agda/Alfa syntax is

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

Example 3 (Untyped λ-terms). The set Term n (n :: Nat) of λ-terms whose free variables are among $\{\mathrm{var}_0, \cdots, \mathrm{var}_{n-1}\}$ (using de Bruijn indices), is a member of the Nat-indexed family Term defined as follows.

```
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
```

Example 4 (Vectors of specified length). An example with one parameter A_1 (σ_1 = Set) 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)
```

In Agda/Alfa, constructors are polymorphic with respect to the parameters and need not be explicitly applied to them.

3 Generators

For the rest of the paper, we restrict σ_i in the schema in Section 2 to be the type Set.

3.1 Definition of Generators

A generator for the family P in Section 2.1 is a function

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

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

We have chosen to implement a seed in Rand as a binary tree of natural numbers [8]. The definition in Agda/Alfa is

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

Example 5. The following function is a generator for Vec.

```
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 idea behind this generator is to map the parameter generator g to the given tree seen as a (right-spine) list of (left) subtrees. (We omitted some braces and semicolons using the so called layout rule of the Agda/Alfa syntax.)

3.2 Surjective Generators

A generator (with instantiation of parameters and parameter generators) is *surjective* if it can generate, given a suitable seed, any element of any member set of the target family. A reason for writing generators in Agda/Alfa is that it becomes possible to formally prove this fundamental correctness property of generators.

For example, we can prove by induction that genVec A g is surjective whenever the parameter generator g is surjective. In Agda/Alfa we formally define

```
Surj :: (A :: Set) -> (Rand -> A) -> Set
Surj A g = (x :: A) -> sig rand :: Rand; prf :: Id A (g rand) x
  --  (In predicate logic, it reads ∀x :: A. ∃rand :: Rand. g rand = x.)

surj_genVec::(A :: Set) -> (g :: Rand -> A) -> Surj A g ->
             Surj sig{ind :: Nat; obj :: Vec A ind} (genVec A g)
surj_genVec A g p = ⟨ ··· the proof omitted ··· ⟩
```

4 Generators for Simple Sets

A *simple* set, possibly parameterised, is an inductive family with the following restriction (using the notation from Section 2):

- Its formation rule has only parameters and no indices ($M = 0$).
- For each introduction rule, the type β_i of each non-recursive argument is either a parameter A_j or a previously defined simple set.
- It is inhabited (non-empty); that is, at least one introduction rule has no recursive arguments.

A generator for simple P is easy to write: it randomly chooses a constructor and generates its arguments by parameter generators, 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. As each seed is finite, the problem of non-termination discussed in [4] does not arise here.

Example 6 (Lists). The set `List A` of lists with elements in the set `A` is parameterised in `A`. A generator for it can be defined as follows:

```
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 indeed a simplified version of `genVec` and easily seen to preserve surjectivity of the parameter generator g.

5 Generators for Inhabited Indexed Sets

An *inhabited indexed set* is an inductively defined indexed family with the following restrictions:

- Its formation rule $P :: I \to \texttt{Set}$ has no parameters, and the single index set I is a simple set with a surjective generator $genI :: \texttt{Rand} \to I$.
- For all $i :: I$, the set $P\,i$ is inhabited.

The extension to families with parameters and several indices is straightforward.

For such a family P, a surjective generator $genP :: \texttt{Rand} \to genPsig$, where $genPsig = \texttt{sig}\ \{ind :: I;\ obj :: P\,ind\}$, 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 $genP'\,i$.

```
genP' :: (i :: I) -> Rand -> P i    -- assumed given.
```

```
genP :: Rand -> genPsig
genP (Node _ l r) = struct ind = genI l; obj = genP' ind r
genP s            = struct ind = genI s; obj = genP' ind s
```

In fact, one can formally prove that

```
surj_genP :: Surj I genI -> ((i :: I)-> Surj (P i) (genP' i))->
             Surj genPsig genP
surj_genP p q = ···
```

Examples of defining $genP'$ for various P follow.

Example 7. Fin(succ n) is inhabited for all n :: Nat. A surjective generator for the family λn :: Nat. Fin(succ n) can be defined as follows:

```
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)
```

Example 8. A binary tree is balanced if, at each node, the height difference between its left and right subtrees is at most 1. One formulation of the set Bal n of balanced binary trees of height n, and its surjective generator genBal′ n are

```
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))

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 $a_2 = a_{(k \bmod 3)}$. Note that no part of a (non-leaf) seed contributes to the result twice; this is necessary for surjectivity, and keeps disjoint parts of the result independent of each other.

Example 9. The set Term n is nonempty for any n :: Nat, and a surjective generator can be given as follows:

```
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  choice3 k (var m (genFin' m l))
                  (abs (succ m) t1)
                  (app (succ m) t2 t3)
```

6 Generators for Simple Inductive Families

We now consider a family whose member sets are not necessarily inhabited. First, we adopt the method in Section 4 for simple sets to a restricted class of families; for these, surjective generators can be defined without backtracking.

An inductive family is *simple* if the following conditions hold:

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

$$\begin{aligned} \mathit{intro} :: {} & (x_1 :: I) \to \cdots \to (x_K :: I) \to \\ & (u_1 :: P\ x_1) \to \cdots (u_K :: P\ x_K) \to \\ & P\ p \end{aligned}$$

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

The type of a generator for P is the same as in Section 5: $\mathit{genP} :: \texttt{Rand} \to \mathit{genPsig}$. However, the choice of constructor controls the generation process, as in Section 4. First, genP randomly chooses a constructor. Then it generates the constructor arguments $i_1, \cdots, i_K, o_1, \cdots, o_K$ for $x_1, \cdots, x_K, u_1, \cdots, u_K$. Note that each of the pairs $(i_1, o_1), \cdots, (i_K, o_K)$ can be chosen as an arbitrary object of the type $\mathit{genPsig}$, and thus K recursive calls suffices for that. The result is the pair

$$(p[i_1/x_1, \cdots, i_K/x_K,\ o_1/u_1\ \cdots,\ o_K/u_K],\ \mathit{intro}\ i_1 \ldots i_K\ o_1 \ldots o_K) :: \mathit{genPsig}$$

As in Section 4 the process terminates since the sizes of seeds decrease.

It is easy to see that this method gives a surjective generator as long as we use independent random seeds in different recursive calls.

Example 10. A surjective generator for the family $\texttt{Even}\ n\ \ (n :: \texttt{Nat})$ (of sets of proofs that n is even) can be defined as follows.

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

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
```

The method can be extended to include parameters, several indices, non-recursive arguments of simple types, etc, under suitable restrictions.

7 Inductive Definitions and Logic Programs

The motivation for considering various restrictions on 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 in the next section. This section explains its basis, the relationship between indexed inductive definitions and logic programs [10].

A *Horn* inductive family is one satisfying the conditions:

- The index sets in its formation rule, and the types (sets) of non-recursive arguments in its introduction rules, all belong to previously defined Horn inductive families.
- In each introduction rule, indices appearing in types of recursive arguments and in the target type (q_{ij}, p_i) are all of constructor expressions; that is, built up from variables in scope by constructors only.

This covers a large part of ordinary inductive families, including all classes we have considered so far.

Our main example here is the family of sets of derivations in propositional calculus, indexed by their conclusions (theorems). It has no parameters and only one index. We do not explain our method for Horn families in general, but generalising the discussion from our specific example should be routine.

Let us take Łukasiewicz's system for propositional calculus. The set `Formula` of formulas is a simple set with constructors

`var`	::	`Nat` → `Formula`
`~`(−)	::	`Formula` → `Formula`
(−) `=>` (−)	::	`Formula` → `Formula` → `Formula`

where `Nat` is used to name propositional variables. The axiom schemata are:

$$\texttt{Ax1}\,p\,q\,r = (p \texttt{=>} q) \texttt{=>} ((q \texttt{=>} r) \texttt{=>} (p \texttt{=>} r))$$
$$\texttt{Ax2}\,p = (\texttt{\~{}}p \texttt{=>} p) \texttt{=>} p$$
$$\texttt{Ax3}\,p\,q = p \texttt{=>} (\texttt{\~{}}p \texttt{=>} q)$$

The only inference rule is Modus Ponens. Thus the family `Thm` p (p :: `Formula`) below defines the set of derivations of a theorem p.

```
Thm :: Formula -> Set = data
      ax1 (p, q, r :: Formula) :: Thm (Ax1 p q r)
    | ax2 (p       :: Formula) :: Thm (Ax2 p)
    | ax3 (p, q    :: Formula) :: Thm (Ax3 p q)
    | mp  (p, q    :: Formula) (x :: Thm p) (y :: Thm (p => q))
                               :: Thm q
```

This family does not fit in the simple schema of Section 6 because of `mp` (`y`'s type is indexed by the non-variable `p => q`). Suppose we try to generate arguments for `mp`, first generating a derivation $d_{\mathtt{x}}$ :: `Thm` $t_{\mathtt{p}}$ for arguments `x` and `p`. While any $t_{\mathtt{p}}$ will do here, we then must find, for `y` and `q`, a derivation $d_{\mathtt{y}}$ :: `Thm` $t_{\mathtt{q}}$ where $t_{\mathtt{q}}$ matches the specific pattern ($t_{\mathtt{p}}$ `=> _`). Although we can find such a derivation in this particular case, for other definitions there may not be such a $t_{\mathtt{q}}$. If so, we need to backtrack, generate another pair $(d'_{\mathtt{x}}, t'_{\mathtt{p}})$, and try again.

This is similar to searching for a solution of a query in logic programming. In Prolog, we can define a predicate `thm` so that `thm` p holds if [3] and only if there exists a derivation d :: `Thm` p.

```
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 as solutions for `X`; for example

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

More precisely, this is a theorem pattern (schema) with variables `_A`, $\cdots$, `_D`. We can generate a theorem by instantiating them with any elements in `Formula`.

In general, there is a correspondence between Horn inductive definitions in dependent type theory and Prolog programs under the propositions-as-sets correspondence:

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

For example, a clause in Prolog

$$P(\boldsymbol{t}) \text{ :- } P_1(\boldsymbol{t}_1), \cdots, P_K(\boldsymbol{t}_K)$$

becomes an introduction rule in type theory:

$$\mathit{intro} :: (x_1, \ldots, x_N :: D) \to P_1\ \boldsymbol{t}_1 \to \cdots \to P_K\ \boldsymbol{t}_K \to P\ \boldsymbol{t}$$

[3] 'If' direction needs some tampering with the default search order.

where D is the set inductively generated by the function symbols of the logic program (the term algebra or the Herbrand universe), and $\boldsymbol{t}_i$, $\boldsymbol{t}$ are sequences of terms in D with variables $x_1, \ldots, x_N$.

The above correspondence does not account for derivations (proof objects) d :: `Thm` p, nor for typing of objects in general. We now extend the correspondence for these.

The idea is to regard sets in type theory as unary predicates (on untyped terms) characterising their elements. For `Nat` and `Formula`, the corresponding predicates are defined by

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

A family with M indices becomes $(M+1)$-place predicates relating indices with elements of the member set at the indices. Corresponding to `Thm`, the predicate `thm1` relates a theorem with its derivation.

```
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).
```

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 directly use a Prolog interpreter to generate some elements of dependent types. If we randomise the Prolog interpreter, then we get a random generator for dependent types.

In general, a typing $b :: P\ \boldsymbol{a}$ can be represented by a predicate $P'\ (\boldsymbol{a}, b)$ in Prolog. For example, the following introduction rule for an inductive family P

$$intro :: (x_1 :: D_1) \to \cdots \to (x_N :: D_N) \to\ P_1\ \boldsymbol{t}_1 \to \cdots P_K\ \boldsymbol{t}_K \to P\ \boldsymbol{t}$$

becomes a clause of the following form:

$$P'(\boldsymbol{t}, intro(X_1, \ldots, X_N, U_1, \ldots, U_K))\ \texttt{:-}$$
$$D'_1(X_1), \cdots, D'_N(X_1, \cdots, X_{N-1}, X_N),\ P'_1(\boldsymbol{t}_1, U_1), \cdots, P'_K(\boldsymbol{t}_K, U_K).$$

where D'_i is the predicate corresponding to the set $D_i[x_1, \cdots, x_{i-1}]$.

The idea is applied to test data generation as follows. A testing form [8] below requires that $Q[\boldsymbol{d}/\boldsymbol{x}]$ to be true (inhabited) for any $\boldsymbol{d} = (d_1, \cdots, d_N)$ that satisfies the preconditions $P_i[\boldsymbol{d}/\boldsymbol{x}]$.

$$\begin{array}{l}(x_1 :: D_1) \to \cdots \to (x_N :: D_N[x_1, \cdots, x_{N-1}]) \to \\ P_1[x_1, \cdots, x_N] \to \ \cdots \ \to P_K[x_1, \cdots, x_N] \to \\ Q[x_1, \cdots, x_N]\end{array}$$

Test data $\boldsymbol{d}$ for this can be generated by searching for solutions to the query

$$\begin{array}{l}\texttt{:-}\ \ D'_1(X_1), \cdots, D'_N(X_1, \ldots, X_{N-1}, X_N), \\ \quad\ P'_1(X_1, \cdots, X_N, \_), \cdots, P'_K(X_1, \cdots, X_N, \_).\end{array}$$

In the next section, we show a generator example for theorems by randomising the Prolog search algorithm: instead of always choosing the first clause unifiable with a goal, we choose one according to random seeds.

8 A Generator for Theorems

In this section, we describe a generator for the family `Thm` in Section 7. It is based on another, more general generator `ThmPat` for theorem *patterns*, that is, formula patterns whose ground instantiations are all theorems.

The type of formula patterns, `Pat`, is a simple set with the same[4] constructors as `Formula` together with a new one `X` :: `Nat` → `Pat` for *pattern variables* (logical variables) $\mathtt{X}_0, \mathtt{X}_1, \cdots$, which stand for indeterminate formulas. Examples are $\mathtt{X}_0$ `=>` $\mathtt{X}_1$ and ($\mathtt{var}_0$ `=>` $\mathtt{var}_1$) `=>` $\mathtt{X}_1$. We choose to distinguish propositional- and pattern- variables, so that the method applies to indexing types without a `var` like constructor (for example, that of formulas on a fixed finite set of atomic propositions).

A theorem pattern is a t :: `Pat` that becomes a theorem when each of its pattern variables is instantiated by any formula; for example `Ax2` $\mathtt{X}_0$, since `ax2` p :: `Thm`((`Ax2` $\mathtt{X}_0$)$[p/\mathtt{X}_0]$) with any p :: `Formula`. They are precisely those t :: `Pat` with some derivation d :: `ThmPat` t, where `ThmPat` :: `Pat` → `Set` is defined just the same as `Thm`, but with `Pat` replacing `Formula` everywhere.

In what follows, letters $X, Y, \cdots$ range over pattern variables. Our Agda code uses a standard technique to have access to 'totally fresh' pattern variables at any point, though we omit details. Substitutions $\sigma = [t_1/X_1, \cdots, t_N/X_N]$ are represented by lists of pairs, and `Subst` is their type. The composite $\sigma_1 \triangleright \sigma_2$ of two substitutions are defined so that $t[\sigma_1 \triangleright \sigma_2] = (t[\sigma_1])[\sigma_2]$.

A pattern t *matches* an introduction rule `ax`i of `ThmPat` if it can be unified with `Ax`i $\boldsymbol{X}$, where $\boldsymbol{X}$ is appropriate number of fresh pattern variables. When this is the case, writing σ for the most general unifier of t and `Ax`i $\boldsymbol{X}$, we call the pair (σ, `ax`i $\boldsymbol{X}[\sigma]$) the *match*. For example, a match of $\mathtt{X}_0$ `=>` $\mathtt{X}_1$ with `ax2` is ([`~`X `=>` $X/\mathtt{X}_0, X/\mathtt{X}_1$], `ax2` X) with a fresh X.

We now describe a theorem pattern generator `genTP`, whose purpose is to generate not an arbitrary theorem pattern but one that fits into a given t :: `Pat`.

[4] Constructors are polymorphic in the language of Agda/Alfa.

```
genTP :: Rand -> (t :: Pat) -> Maybe (σ :: Subst, ThmPat t[σ])
```

With a seed s, genTP $s\,t$ either *succeeds* and returns some Just (σ, d), or *fails* and returns Nothing. In case of success, we have a theorem pattern $t[\sigma]$ with derivation d :: ThmPat $t[\sigma]$.

The procedure applied to pattern t is as follows: it randomly chooses an introduction rule that matches the pattern t. If the axioms are chosen, then the result is either a success with a match, or failure if there is none. If the rule mp is chosen, then we first apply genTP to a fresh variable X to obtain $(\sigma_l, d_l$ :: ThmPat $X[\sigma_l])$ Then we apply genTP to the pattern $(X$ => $t)[\sigma_l]$ and obtain $(\sigma_r, d_r$ :: ThmPat $(X$ => $t)[\sigma_l][\sigma_r])$. The final result is the composite $\sigma_l \triangleright \sigma_r$, together with the derivations $d_l[\sigma_r]$ and d_r combined by mp.

Recursive calls are made with sub-seeds of the random seed given as argument, hence genTP always terminates. A failed recursive call is dealt with by back tracking (retry loops), so long as random seeds are not exhausted.

The pseudo-code for genTP is given below. In the description, *toList* s turns a tree (seed) s into the list of left-subtrees (cf. Example 5).

```
genTP (Leaf k) t =  do
    if (t matches some of ax1, ax2, ax3)     // mp excluded.
        choose a match (σ, d) according to k, and return Just (σ, d);
    else return Nothing;

genTP (Node k l r) t =  do
    choose one of ax1, ax2, ax3, mp, according to k;

    case the choice is axi :
        if (t matches axi with match (σ, d)) return Just (σ, d);
        else return Nothing;

    case the choice is mp :
        for s_l in (toList l) {
            if (genTP s_l X  (X fresh) has the form Just (σ_l, d_l)) {
                for s_r in (toList r) {
                    if (genTP s_r ((X => t)[σ_l]) has the form Just (σ_r, d_r)) {
                        // generation succeeded.
                        σ := σ_l ▷ σ_r;
                        return Just (σ,  mp  X[σ] t[σ] d_l[σ_r] d_r);
                    }
                }
            }
        }
        return Nothing;     // seed exhausted.
```

We can prove that this is a surjective generator: for any theorem pattern t, there exists a seed s and fresh X such that genTP $s\,X$ is Just (σ, d) with $X[\sigma] = t$ and d :: ThmPat t. The Agda/Alfa code and the surjectivity proof for a slightly different version can be found in Qiao [14].

We now use genTP to define a generator genThm for Thm.

```
genThm :: Rand -> sig { ind :: Formula; obj :: Thm ind }
```

genThm $s =$ *do*
  if (genTP $s\,X$ (X fresh) has the form Just (σ, d)) {
    $\tau :=$ substitution of all pattern variables
      by arbitrary elements of Formula;
    return $(X[\sigma][\tau], d[\tau])$;
  } *else* {
    choose an ax$i$, and generate arbitrary formulas $\boldsymbol{p}$ for its arguments;
    return (Ax$i\,\boldsymbol{p}$, ax$i\,\boldsymbol{p}$);
  }

This generator can be used to test, for example, properties in [9] (where BoolExpr is used for the type of formulas).

9 Discussions and Future Work

We have identified several restricted classes of indexed families of sets for which writing surjective generators is simple. For a class of ordinary inductive definitions, generating elements of the family of sets is equivalent to solving a query in a corresponding logic program. Therefore, proof search techniques in logic programming can be used for writing generators. As an example, we implemented a surjective generator for theorems by randomising the proof search algorithm that is used in Prolog implementations. However, it is of course inconvenient to ask the user to implement the search algorithm for each new family of sets. One solution is to embed the search algorithm in the proof assistant externally or internally. Such a system would be a bit like a randomised version of Twelf [13], a logical framework where a given type family is interpreted as a logical program.

In Section 6, we described a simple schema for inductive definitions for which we can write surjective generators. It is interesting to extend the schema for which we can still write surjective generators without much difficulty. For example, we may add side conditions or allow general terms (and not only variables) as indices in the induction hypotheses. Consider, for example, the set of reachable states of a transition system. This can be defined in the following way:

```
R :: S -> Set = data
       init (s :: S)(p :: P s) :: R s
     | step (s, s' :: S)(q :: Tran s s')(p :: R s) :: R s'
```

where there are side conditions in the introduction rules: P characterise the initial states and Tran is the transition relation. One sufficient condition to have a surjective generator is: there is a surjective generator for P, and for any s0 :: S, we have a surjective generator for the family Tran s0, because we can then generate all possible next states for a given reachable state.

Recent work on generic programming [1–3] allows us to write a generic function for a class of data types. It will be interesting to see if we can use generic programming to generate surjective generators for a class of data types.

Another interesting topic is writing surjective generators for function types. Claessen and Hughes [4] show some examples of such generators. For some simple cases, we have proved that the generators are surjective (see part I in Qiao [14]).

References

1. Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming — an introduction. In *LNCS*, volume 1608, pages 28–115. Springer-Verlag, 1999. Revised version of lecture notes for AFP'98.
2. Marcin Benke. Towards generic programming in type theory. Presentation at Annual ESPRIT BRA TYPES Meeting, Berg en Dal. Submitted for publication, available from `http://www.cs.chalmers.se/~marcin/Papers/Notes/nijmegen.ps.gz`, April 2002.
3. Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10:265 – 289, 2003.
4. Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP-00)*, volume 35.9 of *ACM Sigplan Notices*, pages 268–279. ACM Press, New York, September 2000.
5. Catarina Coquand. The Agda homepage. `http://www.cs.chalmers.se/~catarina/agda`.
6. Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, 1994.
7. Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65, June 2000.
8. Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Combining testing and proving in dependent type theory. In David Basin and Burkhart Wolff, editors, *Proceedings of Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*, pages 188–203. Springer-Verlag, 2003.
9. Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Verifying Haskell programs by combining testing and proving. In *Proceedings of Third International Conference on Quality Software*, pages 272–279. IEEE Press, 2003.
10. Masami Hagiya and Takafumi Sakurai. Foundation of logic programming based on inductive definition. *New Generation Comput. (JAPAN) ISSN: 0288-3635*, 2(1):59–77, 1984. QA 76 N 48.
11. Thomas Hallgren. The Alfa homepage. `http://www.cs.chalmers.se/~hallgren/Alfa`.
12. Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf type theory: an introduction.* Oxford University Press, 1990.
13. Frank Pfenning and Carsten Schürmann. The Twelf homepage. `http://www-2.cs.cmu.edu/~twelf/`.
14. Qiao Haiyan. *Testing and Proving in Dependent Type Theory.* PhD thesis, Chalmers University of Technology, 2003.