Exhaustive Enumeration

April 18, 2018

1 Brute-force Algorithms

Fix a combinatorial problem as a tuple $(\mathcal{I}, \mathcal{S})$, where $\mathcal{I}$ is the set of instances of the problem, and $\mathcal{S}$ are the set of solutions. A verifier $v : \mathcal{I} \times \mathcal{S} \to \texttt{bool}$ is a function that returns true iff the given solution is valid for the given instance. A metric $m : \mathcal{S} \to \mathbb{R}$ is a measure of the optimality of a solution. An algorithm $a : \mathcal{I} \to \mathcal{S}$ is then a procedure of finding the optimal solution to each instance of a problem. A brute-force algorithm would be the following:

```
fun (i : I) =>
let candidates = filter (fun s => v (i,s)) (enumerate S)
ranked = map (fun s => (m s, s)) candidates
in
max (fun ((m1,_),(m2,_)) => m1 >= m2) candidates
```

Here, `enumerate` returns an exhaustive list of all possible elements the solution space $\mathcal{S}$. For now, we consider only finite spaces; in addition, we only consider inductively generated spaces.

2 Inductive Types

In addition to sums and products, we are interested in the crucial recursive type. In order to leverage the theory of combinatorial species, we limit to only *functorial* types that act like containers. Below is a table of the inductive types we want to enumerate.

Ind.Types	τ ::=		
	`void`	$\emptyset$	void
	`unit`	$\star$	unit
	`prod`$(\tau_1; \tau_2)$	$\tau_1 \times \tau_2$	product
	`sum`$(\tau_1; \tau_2)$	$\tau_1 + \tau_2$	sum
	`rec`$(t.\tau)$	$\mu(t.\tau)$	recursive type

3 Combinatorial Species

We base the enumeration procedure losely on the theory of combinatorial species. Informally, a species is a mapping from a set of labels to structures built from those labels. In order to account for the generality of enumeration, we consider multisorted species, where a structure can draw from multiple label sets.

$\mathbb{0}_i$	$\mathbb{0}_i(U_1, ..., U_k) = \emptyset$	Empty Species
$\mathbb{1}_i$	$\mathbb{1}_i(U_1, ..., U_k) = \begin{cases} \{\star\} & \text{if } U_i = \emptyset \\ \emptyset & \text{otherwise} \end{cases}$	Unit Species
$\mathbb{X}_i$	$\mathbb{X}(U_1, ..., U_k) = \begin{cases} \{x\} & \text{if } U_i = \{x\} \\ \emptyset & \text{otherwise} \end{cases}$	Singleton Species
$\mathbb{F} + \mathbb{G}$	$(\mathbb{F} + \mathbb{G})(U_1, ..., U_k) = \mathbb{F}(U_1, ..., U_k) \uplus \mathbb{G}(U_1, ..., U_k)$	Species Sum
$\mathbb{F} \bullet \mathbb{G}$	$(\mathbb{F} \bullet \mathbb{G})(U_1, ..., U_k) = \bigcup_{i: X_i \sqcup Y_i = U_i} \mathbb{F}(X_1, ..., X_k) \times \mathbb{G}(Y_1, ..., Y_k)$	Paritional Product
$\mu(f.\mathbb{G})$	$\mu(f.\mathbb{G})(U_1, ..., U_k) = \texttt{map fold}(\mathbb{G}[\mu(f.\mathbb{G})/f](U_1, ..., U_k))$	Recursive Species

4 Coq Representation

The above definition can be implemented in Coq as follows [2] [1]:

```
Definition rec (F : Type -> Type): Type := forall X, (F X-> X) -> X.

Definition rfold F  : forall X, (F X -> X) -> (rec F) -> X :=
fun X k t => t X k.

Class Functor (F : Type -> Type) : Type :=
{
   liftF : forall {X Y}, (X -> Y) -> (F X -> F Y)
}.

Definition rin F `{Functor (F)} : F (rec F) -> rec F :=
fun s => fun X k => k ((liftF (rfold F X k)) s).

Definition rout F `{Functor (F)} : rec F -> F (rec F) :=
rfold F (F (rec F)) (liftF (rin F )).

Section s.
Variable ts : list Type.
Inductive Species : list Type -> Type -> Type :=
| zero : forall C, Species C (Empty_set)
| one :  forall C {x} (m : member x ts), Species C ( unit)
| singleton :  forall C {x} (m : member x ts), Species C ( tget' _ ts m)
| cprod : forall C {F G}, Species C F -> Species C G -> Species C ( prod F G)
| pprod : forall C {F G}, Species C F -> Species C G -> Species C ( prod F G)
| ssum : forall C {F G}, Species C F -> Species C G -> Species C (sum F G)
| srec : forall C F `{Functor (F)}, Species ((rec F)::C) (F (rec F)) ->
  Species C (rec F)
| svar : forall C F, member F C -> Species C F.
End s.

Definition enumerate (n : nat) {ts : list Type} {t} {C} (s : Species ts C t) :
forall tls : tlist ts, hlist (Species ts C) C -> list t.
Admitted.
```

Here, `Species ts C t` is a type family parameterized by a list of sorts `ts` and indexed by a context of type declarations `C` and the return type `t`. The context is necessary in order to model type substitution for recursive

species.

The enumeration procedure pattern matches on a member `s : Species ts C t` and operates according to the table above. `tls` is a heterogeneous list (e.g. `[(nat,[1;2;3]]), (string, ["a","b","c"]),...]`) representing the different label sets. The type `hlist F l` is also heterogeneous list, but such that `F` is a family indexed by elements of `l`. This serves as the current substitution map or type environment. Lastly, the parameter `n : nat` is a place-holder until I figure out how to convince the termination checker.

Enumerating all lists on the label set $[\{1,2,3\}]$:

```
Definition enum_limit := 100.
Definition ts : list Type := [nat : Type].
Definition H : member (nat : Type) ts := HFirst.
Definition F1 := (fun X => sum unit (prod (tget' _ ts H) X)).
Instance F1F : Functor F1 :=
{
  liftF := fun X Y f x =>
  match x with
  | inl tt => inl tt
  | inr (a,x') => inr (a, f x')
  end
}.
Definition C1 := [nrec F1].
Definition s := srec ts []  F1
(ssum ts C1 (one ts C1 (H)) (pprod ts C1 (singleton ts C1 H) (svar ts C1 (nrec F1) H))).
Definition tls : tlist ts := Hcons nat [1;2;3] Hnil.
Definition C : hlist (Species ts []) [] := HNil.

Compute enumerate enum_limit s tls C
= [fun (X : Type) (x : F1 X -> X) => x (inr (1, x (inr (2, x (inr (3, x (inl tt)))))));
   fun (X : Type) (x : F1 X -> X) => x (inr (1, x (inr (3, x (inr (2, x (inl tt)))))));
   fun (X : Type) (x : F1 X -> X) => x (inr (2, x (inr (1, x (inr (3, x (inl tt)))))));
   fun (X : Type) (x : F1 X -> X) => x (inr (2, x (inr (3, x (inr (1, x (inl tt)))))));
   fun (X : Type) (x : F1 X -> X) => x (inr (3, x (inr (1, x (inr (2, x (inl tt)))))));
   fun (X : Type) (x : F1 X -> X) => x (inr (3, x (inr (2, x (inr (1, x (inl tt)))))))]
     : list (nrec F1).
```

All $(\texttt{nat} + \texttt{string}) \times \texttt{nat})$ on $[\{1;2\}, \{\text{“}a\text{”}, \text{“}b\text{”}, \text{“}c\text{”}\}]$:

```
Definition ts2 : list Type := [nat : Type; string : Type].
Definition tls2 : tlist ts2 := Hcons nat [1;2] (Hcons string ["a";"b";"c"] Hnil).
Definition s2 := pprod ts2 []
  (ssum ts2 [] (singleton ts2 [] HFirst) (singleton ts2 [] (HNext nat [string] HFirst)))
  (singleton ts2 [] HFirst).
Definition C2 : hlist (Species ts2 []) [] := HNil.
Compute enumerate enum_limit s2 tls2 C2.
 = [(inl 1, 2); (inl 2, 1); (inl 1, 2); (inl 2, 1); (inl 1, 2); (inl 2, 1); (inl 1, 2);
       (inr "a", 2); (inl 2, 1); (inr "a", 1); (inl 1, 2); (inl 2, 1); (inl 1, 2); (inr "b", 2);
       (inl 2, 1); (inr "b", 1); (inl 1, 2); (inr "c", 2); (inl 2, 1); (inr "c", 1);
       (inl 1, 2); (inl 2, 1)]
     : list (
          (tget' nat ts2 HFirst + tget' string ts2 (HNext nat [string] HFirst))
        * tget' nat ts2 HFirst)
```

5 Persistent Universes

In order to capture the fact that some enumerations reuse elements of the label set, we need to introduce *persistent* labels and modify species enumeration accordingly. Besides the singleton and product species, everything should be extended in the obvious way.

$\mathbb{X}_{i;}$ $\quad \mathbb{X}_{i;}(U_1,...,U_k;V_1,...,V_l) = \begin{cases} \{x\} & \text{if } U_i = \{x\} \\ \emptyset & \text{otherwise} \end{cases}$ Singleton Species

$\mathbb{X}_{;i}$ $\quad \mathbb{X}_{;i}(U_1,...,U_k;V_1,...,V_l) = V_i$ Singleton Species

$\mathbb{F} \bullet \mathbb{G}$ $\quad (\mathbb{F} \bullet \mathbb{G})(U_1,...,U_k;V_1,...,V_l) = \bigcup_{i:X_i \sqcup Y_i = U_i} \mathbb{F}(X_1,...,X_k;V_1,...,V_l) \times \mathbb{G}(Y_1,...,Y_k;V_1,...,V_l)$ Paritional Product

6 Size Search

One heuristic to speed up the search process is to prune classes of the solution space based on the specific problem. One simple class is the size of the solution. One way to define this is the size of the label set that generates the solution set. Conveniently, our formulation of species essentially allow us to easily generate all structures on a label set of a certain size. Then the enumeration procedure would look like:

```
fun search p k i (n,m) =
if i not in (n,m) then []
else
    let s = subset i L in
    let Ss = flatten (map (enum S p) s) in
    let fSs = filter p Ss in
    k fSs i (n,m)
```

Where `p` is a property on the target structure type and `k` is a continuation that directs the enumeration. For example, in vertex cover, we would have

```
p = is_vc
k = fun cur i (n,m) => if cur = [] then search p k (i+1) (n,m) else cur
i = 1
(n,m) = (1,|V|)
```

Symmetrically, if the problem was longest path, we would have:

```
p = is_path
k = fun cur i (n,m) => if cur = [] then search p k (i-1) (n,m) else cur
i = |V|
(n,m) = (1,|V|)
```

References

[1] Adam Chlipala. *Certified Programming with Dependent Types.*
`http://adam.chlipala.net/cpdt/cpdt.pdf`

[2] Philip Wadler. *Recursive types for free!*
`http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt`