Equations Reloaded - Supplementary Material

ANONYMOUS AUTHOR(S)

1 SIZED-BASED MEASURE LIBRARY

We use a type class Sized for sizes on arbitrary types.

```
Class Sized (A : Type) := size : A → nat.
```

For lists, we must be careful to define the sizing function so that it takes *SizeA* as a parameter so that later size functions can use it in a nested manner and satisfy the guardness check. To do so we define it in a section.

```
Section list_size. Context {A : Type} {SizeA : Sized A}.

  Equations list_size : Sized (list A) by struct :=
  list_size nil := 0;
  list_size (cons x xs) := S (size x + list_size xs).
End list_size.
```

The following section derives a generic map_size function for Sized types.

```
Section map_size. Context {A : Type} {SizeA : Sized A}.

  Equations? map_size {B} (l : list A) (g : ∀ (x : A), size x < size l → B) : list B :=
  map_size nil _ := nil;
  map_size (cons x xs) g := cons (g x _) (map_size xs (fun x H ⇒ g x _)).
  Proof. all:cbn;lia. Qed.
```

It has the expected spec when the function does not use the size information.

```
  Lemma map_size_spec {B} (g : A → B) (l : list A) : map_size l (fun x _ ⇒ g x) = List.map g l.
```

This proves a stronger specification for map witnessing that it passes smaller arguments to its function argument *g*.

```
  Lemma map_size_transform {B} (g : A → B) (l : list A) (P : A → B → Prop) :
    (∀ a (Ha : size a < size l), P a (g a)) → Forall2 P l (map g l).
End map_size.
```

2 SIZE-BASED NESTED WELL-FOUNDED RECURSION

The term_size function uses the oveloaded *list_size* above.

```
Equations term_size {n} : Sized (term n) by struct :=
  term_size (Var v) := 1;
  term_size (Lam t) := S (size t);
  term_size (App t l) := S (size t + size l).
```

Finally subst_term can be defined using recursion on sizes.

```
Equations? subst {k l} (σ : fin k → term l) (t : term k) : term l by wf (term_size t) lt := {
  (Var v) [σ] ⇒ σ v;
  (Lam t) [σ] ⇒ Lam (t [extend_var σ]);
```

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

```
    (App t l) [σ] ⇒ App (t [σ]) (map_size l (fun t Inl ⇒ t [σ])) }
  where "t [ σ ]" := (subst σ t) : term.
  Proof. all:unfold size in *; simpl; lia. Defined.
  Hint Rewrite @map_size_spec : subst.
```

An eliminator talking only about map can also be derived if desired.

```
  Lemma subst_term'_elim_all :
    ∀ (P : ∀ k l : nat, (fin k → term l) → term k → term l → Prop),
    (∀ k l σ (f : fin k), P k l σ (Var f) (σ f)) →
    (∀ k l σ (t : term (S k)), P (S k) (S l) (extend_var σ) t (t [extend_var σ]) →
      P k l σ (Lam t) (Lam (t [extend_var σ]))) →
    (∀ k l σ (t : term k) (ts : list (term k)), P k l σ t (t [σ]) →
      Forall2 (P k l σ) ts (map (subst σ) ts) →
      P k l σ (App t ts) (App (t [σ]) (map (subst σ) ts))) →
    ∀ k l σ t, P k l σ t (t [σ]).
  Proof. intros P ????. apply (subst_elim _); intros *; auto.
         intros Ht Hl; generalize (map_size_transform _ _ _ Hl); simp subst.
  Qed.
```

3 MUTUAL WELL-FOUNDED RECURSION THROUGH GADTS

We present an instance of this pattern for the term substitution functions which is actually nested, but it can apply to arbitrary nested or mutual definitions. The subst_ty family encodes the types of our functions.

```
  Inductive subst_ty : ∀ (A : Type) (P : A → Type), Type :=
  | tysubst : subst_ty (Σ {k l} (σ : fin k → term l), term k) (fun a ⇒ term a.2.1)
  | tysubsts : subst_ty (Σ {k l} (σ : fin k → term l), list (term k)) (fun a ⇒ list (term a.2.1)).
```

We will use a simple measure, assuming an overloaded size function and definition of size for terms and lists of sized objects.

```
  Equations measure {A P} (t : subst_ty A P) (x : A) : nat :=
    measure tysubst (_, _, _, t) ⇒ size t;
    measure tysubsts (_, _, _, l) ⇒ size l.
```

We define the function by recursion on the abstract packed argument according to this measure. Using dependent pattern matching, the clauses for tysubst refine the argument and return type to match the type of *subst_term* and similarly for tysubsts, we can hence do pattern-matching as usual on the actual arguments. Termination is easily proven by reasoning on sizes.

```
  Equations? subst {A P} (t : subst_ty A P) (x : A) : P x by wf (measure t x) lt := {
    (Var v) [σ] ⇒ σ v;
    (Lam t) [σ] ⇒ Lam (t [extend_var σ]);
    (App t l) [σ] ⇒ App (t [σ]) (subst tysubsts (_, _, σ, l));
    subst t1 (k, l, σ, ts) ⇒ map_size ts (fun t Ints ⇒ t [σ] : term l) }
  where "t [ σ ]" := (subst tysubst (_, _, σ, t)) : term.
  Proof. all:repeat (unfold size; simp term_size); lia. Defined.
```

4 WELL-FOUNDED NESTED RECURSION AS MUTUAL RECURSION WITH SIZES

The following example uses just dependent elimination on a finite type (booleans) and shows that this also applies to nested recursive definitions. We use a simpler type of rose trees here.

We first define rose trees and their Sized instance.

```
Section RoseMut. Context {A : Set}.

  Inductive t : Set :=
  | leaf (a : A) : t
  | node (l : list t) : t.

  Equations t_size : Sized t by struct :=
  t_size (leaf _) := 0;
  t_size (node l) := S (size l).
```

The measure just takes the size. Here we encode the mutuality using branching on a boolean.

```
  Equations mutmeasure (b : bool) (arg : if b then t else list t) : nat :=
  mutmeasure true t := size t;
  mutmeasure false lt := size lt.
```

The argument and return type depend on the function label (true or false here) and any well-founded recursive call is allowed.

```
  Equations? elements (b : bool) (x : if b then t else list t) : if b then list A else list A
    by wf (mutmeasure b x) lt :=
  elements true (leaf a) := [a];
  elements true (node l) := elements false l;
  elements false nil := nil;
  elements false (cons t ts) := elements true t ++ elements false ts.
  Proof. all:cbn; lia. Qed.
```

Dependent return types also possible of course. elements_dep is a trivial copy of elements that additionally shows it is computing a sublist of elements. It requires a dependent return type elements_dep_type.

```
  Equations elements_dep_type (b : bool) (x : if b then t else list t) : Type :=
    elements_dep_type true t := { l' : list A | ∀ x, In x l' → In x (elements true t) };
    elements_dep_type false l := { l' : list A | ∀ x, In x l' → In x (elements false l) }.
```

We put ourselves in *Program* mode to have coercions of subset types. We reset the default obligation tactic which is otherwise applied to initial goals when using Equations?.

```
  Obligation Tactic := idtac.
  Set Program Mode.

  Equations? elements_dep (b : bool) (x : if b then t else list t) : elements_dep_type b x
    by wf (mutmeasure b x) lt :=
  elements_dep true (leaf a) := [a];
  elements_dep true (node l) := elements_dep false l;
  elements_dep false nil := nil;
  elements_dep false (cons t ts) := elements_dep true t ++ elements_dep false ts.
  Proof.
    all:(simp mutmeasure elements).
    2-3:(cbn; lia). all:destruct elements_dep; simpl. apply i.
    destruct elements_dep;simpl. intros. rewrite app_in in *. intuition.
  Qed.

End RoseMut.
```