Equations for Hereditary Substitution in Leivant's Predicative System F: a case study

Cyprien Mangin
Univ Paris Diderot & École Polytechnique
Paris, France
cyprien.mangin@m4x.org

Matthieu Sozeau
Inria Paris & PPS, Univ Paris Diderot
Paris, France
matthieu.sozeau@inria.fr

This paper presents a case study of formalizing a normalization proof for Leivant's Predicative System F [6] using the EQUATIONS package. Leivant's Predicative System F is a stratified version of System F, where type quantification is annotated with kinds representing universe levels. A weaker variant of this system was studied by Stump & Eades [5, 3], employing the hereditary substitution method to show normalization. We improve on this result by showing normalization for Leivant's original system using hereditary substitutions and a novel multiset ordering on types. Our development is done in the COQ proof assistant using the EQUATIONS package, which provides an interface to define dependently-typed programs with well-founded recursion and full dependent pattern-matching. EQUATIONS allows us to define explicitly the hereditary substitution function, clarifying its algorithmic behavior in presence of term and type substitutions. From this definition, consistency can easily be derived. The algorithmic nature of our development is crucial to reflect languages with type quantification, enlarging the class of languages on which reflection methods can be used in the proof assistant.

1 Introduction

EQUATIONS [10] is a toolbox built as a plugin on top of the COQ proof assistant for writing dependently-typed programs in COQ. Given a high-level specification of a function, using dependent pattern-matching and complex recursion schemes, its purpose is to compile it to pure COQ terms. This compilation scheme builds on the work of Goguen et al [4], which explains dependent pattern-matching [2] in terms of manipulations of propositional equalities. In essence, dependent pattern-matching is compiled away using a reduction-preserving encoding with eliminators for the equality type between datatypes. In addition to this compilation scheme, EQUATIONS also automatically derives the resulting equations as propositional equalities, abstracting entirely from the encoding of pattern-matching found in the actual compiled definition, and an elimination scheme corresponding to the graph of the function. This elimination scheme can then be used to simplify proofs that directly follow the case-analysis and recursion behavior of the function without repeating it. EQUATIONS supports definitions using arbitrarily complex well-founded recursion schemes, including the nested kind of recursion found in hereditary substitution functions, and the generation of unfolding lemmas and elimination schemes for those as well. Additionally, EQUATIONS plays well with the Program extension of COQ to manipulate subset types (also known as refinement types).

The purpose of this paper is to present a case study of using EQUATIONS to show the normalization of Predicative System F, in an algorithmic way such that the normalization function can actually be run inside the proof assistant.

Predicative System F was introduced by Leivant [6] to study the logical strength of different extensions of arithmetic. Using kinds to represent levels of allowed predicative quantification, he can show

I. Cervesato and K. Chaudhuri (Eds.): Tenth International Workshop
on Logical Frameworks and Meta-Languages: Theory and Practice
EPTCS 185, 2015, pp. 71–86, doi:10.4204/EPTCS.185.5

that super-elementary functions can be represented in this system. He employs a Tait-Girard logical relation proof to argue normalization.

Stump and Eades [5] took the system and studied a normalization proof using hereditary substitution. However, they needed to define a variant of the system where the kinding rule of universal quantifications is more restrictive, which makes level $n+1$ not closed by quantifications over types in level n. They claim that the same derivations can be done in their system but give no proof of such fact. We remedy this situation by giving a simple normalization proof still based on hereditary substitution using a novel ordering on types based on multisets of kinds.

The hereditary substitution function ends up being defined as one would do in e.g. ML, but combining Program and EQUATIONS, it can be shown to inhabit a richer type, providing a proof that the function indeed computes normal forms given inputs in normal form. The pre and postconditions of the hereditary subsitution function, which will be explicited later, are actually necessary to justify the termination of this function. From this it is easy to derive normalization and show that the system is consistent (relatively to Coq's theory, *with the K axiom* currently, although we hope to have an axiom-free version working by the time of the workshop).

The paper is organized as follows: in §2 we present a gentle introduction to the EQUATIONS package and its features and quickly explain the main differences with the original presentation from [10]. Then we summarize the standard definitions and metatheoretical results on Predicative System F that we proved and highlight the main differences with the presentation of [5]. We provide the COQ development of our proof supplemented with some commentary to help follow along. First in §3 we present the language definition, with its typing and reduction rules. Then we show in §4 some metatheoretical properties on this language, such as substitution lemmas and regularity. Section 5 is dedicated to showing strong normalization of the calculus, which includes defining a well-founded ordering to justify the termination of the hereditary substitution function, and its definition itself using the EQUATIONS package. We also provide as an appendix the code which is produced from this definition by COQ's extraction mechanism. Finally, we compare with related work and conclude in §6.

2 Equations

EQUATIONS allows one to write recursive functions by specifying a list of clauses with a pattern on the left and a term on the right, à la Agda [9] and Epigram [8]. Here is an example recursive definition on lists, where wildcards corresponds to arbitrary fresh variables in patterns:

```
Equations length {A} (l : list A) : nat :=
length _ [] ⇒ 0;
length _ (cons _ t) ⇒ S (length t).
```

The package starts by building a splitting tree for the definition and then compiles it to a pure COQ term. From the splitting tree, it also derives the equations as propositional equalities, which can be more robust to use than reduction when writing proofs about the constant, although in this particular case the compiled definition is the same as the one from the standard library. Here we have two leaves in the computation tree hence two equations:

```
Check length_equation_1 : ∀ A : Type, length [] = 0.
Check length_equation_2 : ∀ (A : Type) (a : A) (l : list A), length (cons a l) = S (length l).
```

These two equations are automatically added to a rewrite hint database named length and can be used during proofs using the *simp* length tactic. In addition, an elimination principle for length is derived. Note

that length_comp is just a definition of the return type of length in terms of its arguments, i.e. it is $\lambda\ A\ l$, nat here:

```
Check length_elim : ∀ P : ∀ (A : Type) (l : list A), length_comp l → Prop,
    (∀ A : Type, P A [] 0) →
    (∀ (A : Type) (a : A) (l : list A), P A l (length l) → P A (cons a l) (S (length l))) →
    ∀ (A : Type) (l : list A), P A l (length l).
```

This elimination principle can be used in proofs to eliminate *calls* to length and refine at the same time the arguments and results of the call in the goal. For example, to prove the following lemma, one can apply the functional elimination principle using the *funelim* tactic to eliminate the length l call:

```
Lemma length_rev {A} (l : list A) : length (rev l) = length l.
Proof.
  funelim (length l).
```

We get two subgoals, easily solved by simplification and arithmetic.

```
  A : Type
  ============================
   length (rev []) = 0

  A : Type
  a : A
  l : list A
  H : length (rev l) = length l
  ============================
   length (rev (cons a l)) = S (length l)

  - simp length.
  - simpl; rewrite length_app, H; simp length; omega.
Qed.
```

2.1 Dependent Pattern-Matching

EQUATIONS handles not only simple pattern-matching on inductive types, but also dependent pattern-matching on inductive *families*. With respect to the standard COQ match construct, it eases the definition of complex pattern-matchings by compiling in the proof term all the inversion and unification steps that must be witnessed. Here is an example with the le relation on natural numbers.

```
Inductive le : nat → nat → Set :=
| lz : ∀ {n}, 0 ≤ n
| ls : ∀ {m n}, m ≤ n → (S m) ≤ (S n) where "m ≤ n" := (le m n).
```

Proving antisymmetry of this relation requires only two cases, because pattern matching on the first argument determines the endpoints of the second argument:

```
Equations antisym {m n : nat} (x : m ≤ n) (y : n ≤ m) : m = n :=
antisym _ _ x y by rec x ⇒
antisym _ _ lz lz ⇒ eq_refl;
```

```
antisym _ _ (ls x) (ls y) ⇒ f_equal S (antisym x y).
```

More precisely, in the x = lz case, it is possible during the translation to deduce that m must be 0, which implies that y cannot be some application of ls. These deductions are done automatically by EQUATIONS, which allows to reduce this proof to its simplest form. Writing it explicitely in pure COQ would be actually annoying and require explicit mention of impossible cases and surgical rewritings with equalities.

2.2 Recursion

Note that we use a clause by *rec x* ⇒ here in addition to the pattern-matching. This is a different kind of right-hand-side, that allows to specify the recursion scheme of the function. We are using well-founded recursion on the $m \leq n$ hypothesis here. The implicit ordering used is actually automatically derived using a *Derive Subterm* `for` le command, and corresponds to the transitive closure of the direct subterm relation, i.e. the deep structural recursion ordering.

The compiled definition cannot be checked using the built-in structural guardness check of COQ, because the equality manipulations appearing in the term go outside of the subset of recursion schemes recognized by it. It would have to handle commutative cuts and specific constructs on the equality type. Also, the syntactic check can be very slow on medium-sized terms.

The solution here, using the logic to justify the recursive calls, means that we are freed from any syntactic restriction, and any logical justification for termination is allowed. At each recursive call, we must simply provide a proof that the given argument is strictly smaller than the initial one in the subterm relation. An automatic proof search using the constructors of the subterm relation for le solves these subgoals for us here, otherwise they are given as obligations for the user to prove.

As in the case of length, we provide equations and an elimination principle for the definition. In case well-founded recursion is used, we first prove an unfolding lemma for the definition which allows us to remove any reasoning on the termination conditions after the definition. The equations are as expected:

```
Check antisym_equation_1 : antisym lz lz = eq_refl.
Check antisym_equation_2 : ∀ (n1 m0 : nat) (l : n1 ≤ m0) (l0 : m0 ≤ n1),
    antisym (ls l) (ls l0) = f_equal S (antisym l l0).
```

And the elimination principle, with the correct inductive hypothesis in the recursive case:

```
Check antisym_elim : ∀ P : ∀ (m n : nat) (x : m ≤ n) (y : n ≤ m), antisym_comp x y → Prop,
    P 0 0 lz lz eq_refl
    → (∀ (n1 m0 : nat) (l : n1 ≤ m0) (l0 : m0 ≤ n1), P n1 m0 l l0 (antisym l l0) →
       P (S n1) (S m0) (ls l) (ls l0) (f_equal S (antisym l l0))) →
  ∀ (m n : nat) (x : m ≤ n) (y : n ≤ m), P m n x y (antisym x y).
```

The last feature of EQUATIONS necessary to write real definitions is the `with` construct. This construct allows to do pattern-matching on intermediary results in a definition. A typical example is the filter function on lists, which selects all elements of the original list respecting some boolean predicate:

```
Context {A} (p : A → bool).

Equations filter (l : list A) : list A :=
filter [] := [] ;
filter (cons a l) ⇐ p a ⇒ { | true := cons a (filter l) ; | false := filter l }.
```

The ⇐ *p a* ⇒ right-hand side adds a new pattern to the left-hand side of its subprogram, for an object of type bool here. The subprogram is actually defined as another proxy constant, which takes as

arguments the variables a, p and a new variable of type bool. The clauses of the subprogram can shortcut the filter (cons a l) part of the pattern which is automatically inferred from the enclosing left-hand side.

The generated equations for such definitions go through the proxy constant, hence we have two equations for filter and two for *filter_helper_1*, which is the name of the proxy constant. To generate the elimination principle, a mutually inductive graph is generated, and the predicate applying to the subprogram is defined in terms of the original one, adding an equality between the new variable and the exact term it is applied to in the enclosing program. This way, we cannot forget during proofs that the true or false cases are actually results of a call to p a. Note that there are three leaves in the original program (and splitting tree) hence three cases to consider here.

```
Check ⋆filter_elim : ∀ (A : Type) (p : A → bool) (P : list A → list A → Prop)
 (P0:=λ (a : A) (refine : bool) (l H : list A) , p a = refine → P (cons a l) H) ,
  P [] [] → (∀ (a : A) (l : list A) , P l (filter p l) → P0 a true l (cons a (filter p l))) →
  (∀ (a : A) (l : list A) , P l (filter p l) → P0 a false l (filter p l)) →
  ∀ l : list A , P l (filter p l).
```

In general, the term used as a new discriminee is abstracted from the context and return type at this point of the program before checking the subprogram. In that case the eliminator predicate for the subprogram has a dependent binding for the t = refine hypothesis that is used to rewrite in the type of hypotheses and results. This is examplified in the following classical example:

```
Inductive incl {A} : list A → list A → Prop :=
  stop : incl nil nil
| keep {x : A} {xs ys : list A} : incl xs ys → incl (cons x xs) (cons x ys)
| skip {x : A} {xs ys : list A} : incl xs ys → incl (xs) (cons x ys).
```

We define list inclusion inductively and show that filtering out some elements from a list xs results in a included list.

```
Equations(nocomp) sublist {A} (p : A → bool) (xs : list A) : incl (filter p xs) xs :=
sublist A p nil := stop ;
sublist A p (cons x xs) with p x := {
  | true := keep (sublist p xs) ; | false := skip (sublist p xs) }.
```

Here at the with node, the return type is incl (if p x then cons x (filter p xs) else filter p xs) (cons x xs). We abstract p x from the return type and check the new subprogram in context A P x xs (refine : bool) with return type: incl (if refine then cons x (filter p) xs else filter p xs) (cons x xs).

Each of the patterns instantiates refine to a constructor, so the return type reduces to the two expected cases matching with conclusions of the incl relation. The (*nocomp*) option indicates that we do not want the return type to be defined using a *_comp* constant. Indeed, the term keep (sublist p xs) would not be well-typed, as it is expected to have type *sublist_comp* p (x :: xs) which is incl (filter p (x :: xs)) (x :: xs), but has type incl (x :: filter p xs) (x :: xs). This is just a technical limitation we hope to remove in the future.

```
Check ⋆sublist_elim : ∀ (P : ∀ (A : Type) (p : A → bool) (xs : list A) , incl (filter p xs) xs → Prop)
(P0:= λ (A : Type) (p : A → bool) (a : A) (refine : bool) (l : list A)
 (H : incl (filter_obligation_2 (filter p) a refine l) (cons a l)) ,
 ∀ Heq : p a = refine , P A p (cons a l)
    (eq_rect_r (λ r : bool , incl (filter_obligation_2 (filter p) a r l) (cons a l)) H Heq)) ,
  (∀ (A : Type) (p : A → bool) , P A p [] stop) →
  (∀ (A : Type) (p : A → bool) (a : A) (l : list A) ,
```

```
    P A p l (sublist p l) → P0 A p a true l (keep (sublist p l))) →
  (∀ (A : Type) (p : A → bool) (a : A) (l : list A),
    P A p l (sublist p l) → P0 A p a false l (skip (sublist p l))) →
  ∀ (A : Type) (p : A → bool) (xs : list A), P A p xs (sublist p xs).
```

The resulting elimination principle, while maybe not so useful in that case as this program constructs a *proof*, shows the explicit rewriting needed in the definition of the subpredicate *P0*.

This concludes our exposition of EQUATIONS and we now turn to the formalization of Predicative System F.

3 Typing and reduction

3.1 Definition of terms

Recall that Predicative System F is a typed lambda calculus with type abstractions and applications. Our type structure is very simple here, with just the function space and universal quantification on kinded type variables. We use an absolutely standard de Bruijn encoding for type and term variables. The kinds (a.k.a universe levels) are represented using natural numbers. Our development is based on Jérôme Vouillon's solution to the POPLmark challenge for System F^{sub} [11].

```
Definition kind := nat.

Inductive typ : Set :=
  | tvar : nat → typ | arrow : typ → typ → typ | all : kind → typ → typ.
```

We will write $\forall\ X :*\ k.\ T$ for the quantification over types of kind k.

```
Inductive term : Set :=
  | var : nat → term
  | abs : typ → term → term
  | app : term → term → term
  | tabs : kind → term → term
  | tapp : term → typ → term.
```

Our raw terms are simply the abstract syntax trees.

3.2 Shiftings and substitutions

We define the different operations of shifting and substitutions with the EQUATIONS package, we only show the substitution function here which uses a `with` right-hand side. All the development can be downloaded or browsed at `http://equations-fpred.gforge.inria.fr`.

```
Check shift_typ : ∀ (X : nat) (t : term), term.
Check tsubst : typ → nat → typ → typ.

Equations subst (t : term) (x : nat) (t' : term) : term :=
subst (var y) x t' ⇐ lt_eq_lt_dec y x ⇒ {
  | inleft (left _) ⇒ var y;
  | inleft (right _) ⇒ t';
  | inright _ ⇒ var (y - 1) };
subst (abs T1 t2) x t' ⇒ abs T1 (subst t2 (1 + x) (shift 0 t'));
```

```
subst (app t1 t2) x t' ⇒ app (subst t1 x t') (subst t2 x t');
subst (tabs k t2) x t' ⇒ tabs k (subst t2 x (shift_typ 0 t'));
subst (tapp t1 T2) x t' ⇒ tapp (subst t1 x t') T2.

Check subst_typ : term → nat → typ → term.
```

3.3 Contexts

We define the contexts env and the two functions get_kind and get_var which access the context. A context is an interleaving of types and terms contexts. Vouillon's great idea is to have parallel de Bruijn indexings for type and term variables, which means separate indices for type and term variables. That way, shifting and substitution of one kind does not influence the other, making weakening and substitution lemmas much simpler, we follow this idea here.

```
Inductive env : Set :=
  | empty : env | evar : env → typ → env | etvar : env → kind → env.
```

Note that EQUATIONS allows wildcards and overlapping clauses with a first match semantics, as usual.

```
Equations(nocomp) get_kind (e : env) (X : nat) : option kind :=
get_kind empty _ ⇒ None;
get_kind (evar e _) X ⇒ get_kind e X;
get_kind (etvar _ T) O ⇒ Some T;
get_kind (etvar e _) (S X) ⇒ get_kind e X.
```

We need the functorial map on option types to ease writing these partial lookup functions.

```
Equations opt_map (A B : Set) (f : A → B) (x : option A) : option B :=
opt_map _ _ f (Some x) ⇒ Some (f x);
opt_map _ _ _ None ⇒ None.

Equations(nocomp) get_var (e : env) (x : nat) : option typ :=
get_var empty _ ⇒ None;
get_var (etvar e _) x ⇒ opt_map (tshift 0) (get_var e x);
get_var (evar _ T) O ⇒ Some T;
get_var (evar e _) (S x) ⇒ get_var e x.
```

3.4 Well-formedness conditions

We also define some well-formedness conditions for types, terms and contexts. Namely, in a type (resp. in a term), the variables must all be kinded (resp. typed). We just show the wf_typ definition here, those follow Stump and Haye's work.

```
Equations wf_typ (e : env) (T : typ) : Prop :=
wf_typ e (tvar X) ⇒ get_kind e X ≠ None;
wf_typ e (arrow T1 T2) ⇒ wf_typ e T1 ∧ wf_typ e T2;
wf_typ e (all k T2) ⇒ wf_typ (etvar e k) T2.
```

3.5 Kinding and typing rules

The kinding rules are the main difference between Leivant's and Stump's presentations. We refer to these works for pen and paper presentations of these systems, due to lack of space, we cannot include them here. The case for universal quantification sets the level of a universal type at 1 + max *k k'*, where *k* and *k'* are respectively the domain and codomain kinds, in Stump's case, which allows for a straightforward order on types based on levels, but this means that each level is not closed under products from lower levels anymore. In other words, multiple quantifications at the same level raise the overall level. For example (∀ *X* :* 0. *X*) :* 1 as expected but ∀ *X* :* 0. ∀ *Y* :* 0. *X* :* (1 + max 0 (1 + max 0 0)) = 2. This is a very strange behavior.

We use the standard predicative product rule which sets the product level to max (*k*+1) *k'*, which directly corresponds to Martin-Löf's Predicative Type Theory. Note that the system includes cumulativity through the Var rule which allows to lift a type variable declared at level *k* into any higher level *k'*.

```
Inductive kinding : env → typ → kind → Prop :=
| T_TVar : ∀ (e : env) (X : nat) (k k' : kind), wf_env e →
    get_kind e X = Some k → k ⇐ k' → kinding e (tvar X) k'
| T_Arrow : ∀ e T U k k', kinding e T k → kinding e U k' → kinding e (arrow T U) (max k k')
| T_All : ∀ e T k k', kinding (etvar e k) T k' → kinding e (all k T) (max (k+1) k').
```

The typing relation is straightforward. Just note that we check for well-formedness of environments at the variable case, so typing derivations are always well-formed.

```
Inductive typing : env → term → typ → Prop :=
  | T_Var (e : env) (x : nat) (T : typ) : wf_env e → get_var e x = Some T → typing e (var x) T
  | T_Abs (e : env) (t : term) (T1 T2 : typ) :
      typing (evar e T1) t T2 → typing e (abs T1 t) (arrow T1 T2)
  | T_App (e : env) (t1 t2 : term) (T11 T12 : typ) :
      typing e t1 (arrow T11 T12) → typing e t2 T11 → typing e (app t1 t2) T12
  | T_Tabs (e : env) (t : term) (k : kind) (T : typ) :
      typing (etvar e k) t T → typing e (tabs k t) (all k T)
  | T_Tapp (e : env) (t : term) k (T1 T2 : typ) :
      typing e t (all k T1) → kinding e T2 k → typing e (tapp t T2) (tsubst T1 0 T2).
```

3.6 Reduction rules

To define normalization we must formalize the reduction relation of the calculus. Beta-redexes for this calculus are the application of an abstraction to a term and the application of a type abstraction to a type.

```
Inductive red : term → term → Prop :=
  | E_AppAbs (T : typ) (t1 t2 : term) : red (app (abs T t1) t2) (subst t1 0 t2)
  | E_TappTabs k (T : typ) (t : term) : red (tapp (tabs k t) T) (subst_typ t 0 T).
```

We define the transitive closure of reduction on terms by closing `red` by context.

```
Inductive sred : term → term → Prop :=
  | Red_sred t t' : red t t' → sred t t'
  | sred_trans t1 t2 t3 : sred t1 t2 → sred t2 t3 → sred t1 t3
  | Par_app_left t1 t1' t2 : sred t1 t1' → sred (app t1 t2) (app t1' t2)
  | Par_app_right t1 t2 t2' : sred t2 t2' → sred (app t1 t2) (app t1 t2')
  | Par_abs T t t' : sred t t' → sred (abs T t) (abs T t')
```

```
| Par_tapp t t' T : sred t t' → sred (tapp t T) (tapp t' T)
| Par_tabs k t t' : sred t t' → sred (tabs k t) (tabs k t').

Definition reds t n := clos_refl sred t n.
```

We can prove usual congruence lemmas on reds showing that it indeed formalizes parallel reduction.

4 Metatheory

The metatheory of the system is pretty straightforward and follows the one of F^{sub} closely. We only mention the main idea for type substitution and the statements of the main metatheoretical lemmas.

To formalize type substitution, we use a proposition env_subst that corresponds to the environment operation: $E, X :: k, E' \Rightarrow E, (X \mapsto T')\ E'$ assuming $E \vdash T' :: k$. In other words, env_subst $X\ T\ e\ e'$ holds whenever we can find environments E, E' and a kind k such that $E \vdash T' :: k$ and $e = E, X :: k, E'$ and $e' = E, (X \mapsto T')\ E'$.

```
Inductive env_subst : nat → typ → env → env → Prop :=
| es_here (e : env) (T : typ) (k : kind) : kinding e T k → env_subst 0 T (etvar e k) e
| es_var (X : nat) (T T' : typ) (e e' : env) :
    env_subst X T' e e' → env_subst X T' (evar e T) (evar e' (tsubst T X T'))
| es_kind (X : nat) k (T' : typ) (e e' : env) :
    env_subst X T' e e' → env_subst (1 + X) (tshift 0 T') (etvar e k) (etvar e' k).
```

4.1 Typing and well-formedness

Actually, both kinding and typing imply well-formedness. In other words, it is possible to kind a type T in an environment e only if both the type and the environment are well-formed.

```
Lemma kinding_wf (e : env) (T : typ) (k : kind) : kinding e T k → wf_env e ∧ wf_typ e T.
```

4.2 Weakening

We only show the main weakening lemma for typing: if e' results from e by inserting a type variable at position X with any kind, the term and types of a typing derivation can be shifted accordingly to give a new typing derivation in the extended environment.

```
Lemma typing_weakening_kind_ind (e e' : env) (X : nat) (t : term) (U : typ) :
  insert_kind X e e' → typing e t U → typing e' (shift_typ X t) (tshift X U).
```

Weakening by a term variable preserves typing as well.

```
Lemma typing_weakening_var (e : env) (t : term) (U V : typ) :
  wf_typ e V → typing e t U → typing (evar e V) (shift 0 t) U.
```

4.3 Narrowing

As the system includes a kind of subtyping relation due to level cumulativity, we can prove a narrowing property for derivations. Again we define a judgment formalizing that a context e' is a narrowing of a context e if they are identical but for one type variable binding $(T : k')$ in e' and $(T : k)$ in e with $k' < k$.

```
Inductive narrow : nat → env → env → Set :=
    narrow_0 (e : env) (k k' : kind) : k' < k → narrow 0 (etvar e k) (etvar e k')
  | narrow_extend_kind (e e' : env) (k : kind) (X : nat) :
      narrow X e e' → narrow (1 + X) (etvar e k) (etvar e' k)
  | narrow_extend_var (e e' : env) (T : typ) (X : nat) :
      wf_typ e' T → narrow X e e' → narrow X (evar e T) (evar e' T).
```

Before we can show narrowing, we have to show that kinding respects cumulativity: If it is provable that a type T has kind k in the context e, then we can also prove that it has any kind k' for $k \leq k'$.

```
Lemma kinding_transitive e T k k' : kinding e T k → k ≤ k' → kinding e T k'.
```

Narrowing is a strong property, in the sense that a type T can have in a narrowing of a context e any kind that it can have in e itself.

```
Lemma typing_narrowing_ind (e e' : env) (X : nat) (t : term) (U : typ) : narrow X e e' → typing e t U
→ typing e' t U.
```

4.4 Substitution

Now, substitution lemmas can be proven for the various substitution functions.

```
Lemma subst_preserves_typing (e : env) (x : nat) (t u : term) (V W : typ) :
  typing e t V → typing (remove_var e x) u W → get_var e x = Some W → typing (remove_var e x)
(subst t x u) V.

Lemma subst_typ_preserves_typing (e : env) (t : term) (U P : typ) k :
  typing (etvar e k) t U → kinding e P k → typing e (subst_typ t 0 P) (tsubst U 0 P).
```

Finally, we prove regularity, which is to say that the type of any well-typed term is kinded. This is a consequence of the fact that any well-formed type is kindable. All these results correspond directly to the paper proofs of Stump and Hayes.

```
Theorem regularity (e : env) (t : term) (U : typ) : typing e t U → ∃ k : kind, kinding e U k.
```

5 Normalization

To show that hereditary substitution is well-defined, we must provide an order of termination. In our case, we will have a lexicographic combination of a multiset ordering on kinds. To formalize this, we reuse CoLoR's [1] library of multisets and definition of the multiset order. Those are multisets on ordered types, here natural numbers with the usual ordering, which is well-founded.

```
Notation "X <_m Y" := (MultisetLt gt X Y) (at level 70).

Definition wf_multiset_order : well_founded (MultisetLt gt).
```

The kinds_of function computes the multiset of kinds appearing in a type, which reduces to the bounds of universal quantifications.

```
Equations kinds_of (t : typ) : Multiset :=
kinds_of (tvar _) ⇒ empty; kinds_of (arrow T U) ⇒ union (kinds_of T) (kinds_of U);
kinds_of (all k T) ⇒ union {k} (kinds_of T).
```

Clearly, the singleton multiset built from any valid kind for T bounds the bag of kinds appearing in T, according to the kinding rules. This is proved by induction on the kinding derivation:

```
Lemma kinds_of_kinded e T k : kinding e T k → kinds_of T <mul { k }.
```

Kinds in a type are invariant by shifting or lifting. This is a simple example of a proof by functional elimination. The T argument and result of kinds_of T get refined and we just need to simplify the right hand sides according to the definitions of kinds_of and tshift, using rewriting not computation, and finish by rewriting with the induction hypotheses.

```
Lemma kinds_of_tshift X T : kinds_of (tshift X T) = kinds_of T.
Proof.
  funelim (kinds_of T); simp kinds_of tshift; now rewrite H, ?H0.
Qed.
```

For type substitution of T in U however, an exact arithmetic relation holds. We know that the multiset of kinds of the substituted type can appear a finite number of times in the resulting type, along with the original kinds of U.

```
Lemma kinds_of_tsubst e e' X T U k : env_subst X T e e' → kinding e U k →
  ∃ n : nat, kinds_of (tsubst U X T) =mul= kinds_of U + mul_sum n (kinds_of T).
```

This allows us to derive a general result about kindings of universal types: any well-kinded instance substitution produces a type with a strictly smaller bag of kinds. This is the central result needed to show termination. In Stump's work, the measure considered was solely the depth of types, and only through the stricter kinding invariant could the order be shown well-founded.

```
Lemma kinds_of_tsubst_all e U k k' T : kinding e (all k U) k' →
  kinding e T k → kinds_of (tsubst U 0 T) <mul kinds_of (all k U).
```

5.1 Definition of the measure.

We first define the depth of a type as being the number of universal quantifications and type variables in that type.

```
Equations depth (t : typ) : nat :=
depth (tvar _) ⇒ 1; depth (arrow T U) ⇒ (depth T + depth U)%nat;
depth (all k U) ⇒ S (depth U).
```

Of course, it cannot be zero, which is useful since it allows to have depth T < depth (arrow T U), which will be needed to prove the well-foundedness of the hereditary substitution.

```
Lemma depth_nz t : 0 < depth t.
```

The order that we will use on types is a lexicographical order on the multiset of kinds and the depth. As the first part of the lexicographic product is a multiset, and as those should not be compared with the Leibniz equality but rather a specific setoid equality, we defined a generalized notion of lexicographic product up-to an equivalence relation on the first component, here meq which represents multiset equality.

```
Definition relmd : relation (Multiset × nat) := lexprod (MultisetLt gt) meq lt.
```

It is well-founded, relying ultimately on the well-foundedness of lt.

```
Definition wf_relmd : well_founded relmd.
```

The actual order is relmd *on* the kinds_of and depth measures on types.

```
Definition order (x y : typ) := relmd (kinds_of x, depth x) (kinds_of y, depth y).
```

```
Definition wf_order : well_founded order.
```

It is well-founded and clearly transitive. `Lemma order_trans t u v : order t u → order u v → order t v.`

As we expected, we can compare a type with an arrow on that type, on the left and on the right.

```
Lemma order_arrow_l : ∀ A B, order A (arrow A B).
```

```
Lemma order_arrow_r : ∀ A B, order B (arrow A B).
```

We also define the reflexive closure of this order. It will be useful to express the postcondition of the hereditary substitution function, as we will explain below.

```
Definition ordtyp := clos_refl order.
```

Finally, we define the size of a term as usual.

```
Equations(nocomp) term_size (t : term) : nat :=
term_size (var _) ⇒ 0; term_size (abs T t) ⇒ S (term_size t);
term_size (app t u) ⇒ S (term_size t + term_size u);
term_size (tabs k t) ⇒ S (term_size t); term_size (tapp t U) ⇒ S (term_size t).
```

```
Definition wf_term_size : well_founded (MR term_size lt) := wf_inverse_image lt term_size lt_wf.
```

The hereditary substitution order is a lexicographic combination of the order on the multisets of kinds in the substituted term's type, the number of universal quantifiers and type variables in the substituted term's type, and the term size of the substituend. In other words, with U the type of the substituted term and t the substituend, we first compare the multiset of kinds in U, then the depth of U, and ultimately the size of t.

```
Definition her_order : relation (typ × term) :=
  lexprod order (fun x y ⇒ kinds_of x = kinds_of y ∧ depth x = depth y) (MR term_size lt).
```

```
Instance WF_her_order : WellFounded her_order.
```

5.2 The model

We now turn to the interpretation proper. We characterize the normal forms as a subset of the terms using mutually-inductive normal and neutral judgments. The plan is to show that the hereditary substitution function, when given two terms in normal form will produce terms in normal form. We can already expect some complications as normal terms also include neutral ones...

```
Inductive normal : term → Prop :=
| normal_abs T t : normal t → normal (abs T t)
| normal_tabs k t : normal t → normal (tabs k t)
| normal_neutral r : neutral r → normal r
with neutral : term → Prop :=
| neutral_var i : neutral (var i)
| neutral_app t n : neutral t → normal n → neutral (app t n)
| neutral_tapp t T : neutral t → neutral (tapp t T).
```

A term t is said to be a canonical inhabitant of a type T in environment e if $e \vdash t : T$ and t is in normal form. Our goal will be to show that every typeable term can be normalized to a canonical one.

```
Definition canonical e t T := typing e t T ∧ normal t.
```

We define a relation expressing that n is the interpretation of some arbitrary term t of type T and in environment e. `Definition interp e t T n := reds t n ∧ canonical e n T.`

5.3 Hereditary substitution

As we said, hereditary substitution takes two terms t and u in normal form and returns a term which is the result of substituting u in t at some index. From a purely algorithmic point of view, we only need t, u and the index X to compute the result of this function. However, we need more to prove its correctness.

First of all, the well-founded order that we use to justify its termination is an order on the type of the substituted and on the substituend, which is why the function hsubst also takes as an argument the type of the substituted term.

We then need a typing environment for t and u, which is not useful from a computational point of view but will serve to prove the termination and the correctness of hsubst. To this effect, we will decorate the function with a precondition and a postcondition. We define those in the universe of propositions to underline the fact that they are not useful in a computational way.

```
Definition pre (t : typ × term) (u : term) (X : nat) (p : env × typ) : Prop :=
    (get_var (fst p) X = Some (fst t) ∧ canonical (fst p) (snd t) (snd p)
     ∧ canonical (remove_var (fst p) X) u (fst t)).
```

There is one subtle point in the formulation of the postcondition. When we substitute in an application app *t1 t2*, it may be that the result of substituting in *t1* is an abstraction abs *T t*. If that's the case, to preserve the invariant that the result of hsubst is in normal form, we have to call again hsubst to perform the beta-reduction. However to do this, we need to know that the type of *t1* is smaller than the type of the substituted term. Note that we need to add a side-condition to this property, which is not always true: if the substituted variable did not appear at all, then there is no reason to have any relation between those types. There is a relation only if the original term was not an abstraction but the substituted term is.

```
Equations is_abs (t : term) : Prop :=
is_abs (abs _ _) ⇒ True; is_abs (tabs _ _) ⇒ True; is_abs _ ⇒ False.

Definition post (t : typ × term) (u : term) (X : nat) (r : term) (p : env × typ) : Prop :=
  interp (remove_var (fst p) X) (subst (snd t) X u) (snd p) r ∧
  (¬ is_abs (snd t) → is_abs r → ordtyp (snd p) (fst t)).
```

The Program mode proposed by Coq interacts nicely with Equations, in that it allows us to just return a term, and provide later the postcondition, thanks to subtyping of subset types. In the same way, we can treat a value returned by a call as if it was just the term. We use a standard encoding for the ghost p : env × typ variable. The (noind) option disables the generation of the graph and elimination principle for the function, its type and computational behavior is all we need here.

```
Equations(noind) hsubst (t : typ × term) (u : term) (X : nat) (P : ∃ (p : env × typ), pre t u X p) :
  {r : term | ∀ (p : env × typ), pre t u X p → post t u X r p} :=
hsubst t u X P by rec t her_order ⇒
hsubst (pair U t) u X P ⇐ t ⇒ {
```

```
| var i ⇐ lt_eq_lt_dec i X ⇒ {
  | inleft (right p) ⇒ u; | inleft (left p) ⇒ var i;
  | inright p ⇒ var (pred i) };
| abs T t ⇒ abs T (hsubst (U, t) (shift 0 u) (S X) _) ;
| tabs k t ⇒ tabs k (hsubst (tshift 0 U, t) (shift_typ 0 u) X _) ;
| tapp t T ⇐ hsubst (U, t) u X _ ⇒ {
  | exist (tabs k t') P ⇒ subst_typ t' 0 T;
  | exist r P ⇒ tapp r T };
| app t1 t2 ⇐ hsubst (U, t2) u X _ ⇒ {
  | exist r2 P2 ⇐ hsubst (U, t1) u X _ ⇒ {
    | exist (abs T' t') P1 ⇒ hsubst (T', t') r2 0 _;
    | exist r1 P1 ⇒ app r1 r2 } } }.
```

With hsubst defined, it is now easy to implement a normalize function which takes a term and returns its normal form. As for hsubst, we add a precondition and a postcondition which allow to show correctness by construction.

```
Definition pre' (t : term) (p : env × typ) : Prop :=
  typing (fst p) t (snd p).
Definition post' (t : term) (n : term) (p : env × typ) : Prop :=
  interp (fst p) t (snd p) n.

Equations(noind) normalize (t : term) (P : ∃ (p : env × typ), pre' t p) :
  {n : term | ∀ (p : env × typ), pre' t p → post' t n p} :=
normalize (var i) P ⇒ var i;
normalize (abs T1 t) P ⇒ abs T1 (normalize t _) ;
normalize (app t1 t2) P ⇐ normalize t2 _ ⇒ {
  | exist t2' P2' ⇐ normalize t1 _ ⇒ {
    | exist (abs T t) P1' ⇒ hsubst (T, t) t2' 0 _ ;
    | exist t1' P1' ⇒ app t1' t2' } };
normalize (tabs k t) P ⇒ tabs k (normalize t _) ;
normalize (tapp t T) P ⇐ normalize t _ ⇒ {
  | exist (tabs k t') P' ⇒ subst_typ t' 0 T ;
  | exist t' P' ⇒ tapp t' T }.
```

The existence of the normalize function is in itself a proof of the strong normalization of Leivant's Predicative System F.

```
Theorem normalization e t T : typing e t T → ∃ n, reds t n ∧ typing e n T ∧ normal n.
```

5.4 Consistency

It is easy to show consistency based on the normalization function. We just need lemmas showing that neutral terms cannot inhabit any type in an environment with only a type variable, by inversion on the neutrality derivation.

```
Lemma neutral_tvar t k T : neutral t → typing (etvar empty_env k) t T → False.
```

Consistency is then proved using case analysis on an assumed typing derivation of falsehood at any universe level k. Informally, it is showing that $\forall\, X :^* k$, X is not inhabited for any k.

```
Corollary consistency k : ¬ ∃ t, typing empty_env t (all k (tvar 0)).
```

6 Related Work and Conclusion

There are many formalizations of similar calculi, and we do not claim any originality there. However, to our knowledge, the multiset ordering used to show normalization is original. The point of this paper is more to show that the EQUATIONS plugin is ready to handle more consequent developments and showcase its features. Is has similar expressivity w.r.t. Agda and Idris, but derives more principles, and everything is compiled down to vanilla COQ terms, so it does not change the trusted code base except for the use of K, which we are hopeful we can get rid of by the time of the workshop.

It would be interesting to study extensions of the language with type recursion. As shown by Leivant, this would allow to type terms that are not typeable in second-order lambda calculus. We will also need to extend the language with existentials, pairs and a minimal notion of inductive types to be able to handle a larger class of programs. One of the possible venues for generalization is to extend the work of Malecha et al [7] to reflect a larger fragment of GALLINA, the language of COQ.

References

[1] Frédéric Blanqui: *CoLoR, a Coq Library on Rewriting and Termination*. Available at http://color.inria.fr.

[2] Thierry Coquand (1992): *Pattern Matching with Dependent Types*. Available at http://www.cs.chalmers.se/~coquand/pattern.ps. Proceedings of the Workshop on Logical Frameworks.

[3] Harley D Eades III (2014): *The semantic analysis of advanced programming languages*. Ph.D. thesis, The University of Iowa. Available at http://metatheorem.org/wp-content/papers/thesis.pdf.

[4] Conor McBride Healfdene Goguen & James McKinna (2006): *Eliminating Dependent Pattern Matching*. Available at http://cs.ru.nl/~james/RESEARCH/goguen2006.pdf.

[5] Harley D. Eades III & Aaron Stump (2010): *Hereditary Substitution for Stratified System F*. In: *International Workshop on Proof-Search in Type Theories*, A FLoC workshop, Edinburgh, Scotland. Available at http://homepage.divms.uiowa.edu/~astump/papers/pstt-2010.pdf.

[6] Daniel Leivant (1990): *Finitely stratified polymorphism*. Technical Report, Carnegie Mellon University. Available at http://repository.cmu.edu/cgi/viewcontent.cgi?article=2961&context=compsci.

[7] Gregory Malecha, Adam Chlipala & Thomas Braibant (2014): *Compositional Computational Reflection*. In Gerwin Klein & Ruben Gamboa, editors: *ITP'14*, *Lecture Notes in Computer Science* 8558, Springer, pp. 374–389, doi:10.1007/978-3-319-08970-6_24. Available at http://dx.doi.org/10.1007/978-3-319-08970-6.

[8] Conor McBride (2005): *Epigram: Practical Programming with Dependent Types*. *Advanced Functional Programming*, pp. 130–170, doi:10.1007/11546382_3.

[9] Ulf Norell (2007): *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden. Available at http://www.cs.chalmers.se/~ulfn/papers/thesis.html.

[10] Matthieu Sozeau (2010): *Equations: A Dependent Pattern-Matching Compiler*. In: *First International Conference on Interactive Theorem Proving*, Springer, doi:10.1007/978-3-642-14052-5_29.

[11] Jérôme Vouillon: *POPLmark challenge solution*. Available at http://www.seas.upenn.edu/~plclub/poplmark/vouillon.html.

A Extracted code

```
let hereditary_subst t u x =
  let rec fix_F x0 =
    let h = fun y -> fix_F y in
    (fun u0 x1 _ ->
    let Pair (t0, t1) = x0 in
    (match t1 with
     | Var n ->
       (match lt_eq_lt_dec n x1 with
        | Inleft s ->
          (match s with
           | Left -> Var n
           | Right -> u0)
        | Inright -> Var (pred n))
     | Abs (t2, refine) ->
       Abs (t2, (h (Pair (t0, refine)) (shift O u0) (S x1) __))
     | App (refine1, refine2) ->
       (match h (Pair (t0, refine1)) u0 x1 __ with
        | Abs (t2, x2) ->
          h (Pair (t2, x2)) (h (Pair (t0, refine2)) u0 x1 __) O __
        | x2 -> App (x2, (h (Pair (t0, refine2)) u0 x1 __)))
     | Tabs (k, refine) ->
       Tabs (k, (h (Pair ((tshift O t0), refine)) (shift_typ O u0) x1 __))
     | Tapp (refine, t2) ->
       (match h (Pair (t0, refine)) u0 x1 __ with
        | Tabs (k, x2) -> subst_typ x2 O t2
        | x2 -> Tapp (x2, t2))))
  in fix_F t u x __

type normalize_comp = term

(** val normalize : term -> normalize_comp **)

let rec normalize = function
| Var n -> Var n
| Abs (t0, t1) -> Abs (t0, (normalize t1))
| App (t1, t2) ->
  (match normalize t1 with
   | Abs (t0, x0) -> hereditary_subst (Pair (t0, x0)) (normalize t2) O
   | x -> App (x, (normalize t2)))
| Tabs (k, t0) -> Tabs (k, (normalize t0))
| Tapp (t0, t1) ->
  (match normalize t0 with
   | Tabs (k, x) -> subst_typ x O t1
   | x -> Tapp (x, t1))
```