Eliminating Dependent Pattern-Matching in Coq

Cyprien Mangin

Internship Report – MPRI M2
Eliminating Dependent Pattern-Matching in Coq

Cyprien Mangin, supervised by Matthieu Sozeau
Team πr^2, Laboratoire PPS, Université Paris-Diderot

August 21, 2015

General context

Coq [1] is a proof assistant which relies on the Curry-Howard isomorphism to construct certified proofs. Proving a theorem is the same thing as providing a term which inhabits a type corresponding to this theorem. In order to trust Coq, it is enough to trust its kernel, which is intentionally kept small enough for a motivated reader to understand and, hopefully, trust it. This approach can have some drawbacks, as high-level constructs have to be translated down to simpler constructs, by the user or by some part of code external to the kernel.

For example, writing dependent pattern-matching in Coq can be complicated. Simpifying this task is one of the purposes of the Equations [2] plugin. Given a high-level specification of a function, which can use dependent pattern-matching and complex recursion schemes, it will compile it to pure Coq terms.

Equations is the result of the work of Matthieu Sozeau[10], largely based on the research of Goguen et al[7]. This internship revolves around Equations as a tool to benchmark, improve and adapt to new settings. This in turn involves the study of dependent pattern-matching.

Research problem

The initial goal of this internship was to rewrite a part of Equations, in order to better control the use of the axiom K during the compilation phase. This axiom states that to prove a property depending on a proof of equality, it is enough to consider the case where this proof is the reflexivity. Equivalently, it says that any proof of equality is propositionally equal to the reflexivity. While it is useful in some cases, and even provable for a lot of types, it can be harmful when working in some contexts, like Homotopy Type Theory[12] – abbreviated HoTT. Another problem is that, as an axiom, it will block any computation in Coq that involves it.

This implies to know more about the technicalities of Coq and Equations, which meant spending some time on learning to use Equations, writing examples, understanding the underlying theory.

Most of the members of the Coq development team are also part of the team πr^2. Therefore, another goal was to approach the Coq development itself.

This internship involved a considerable amount of implementation work. Notably, I took part in the first Coq coding sprint, where I dived into the source code of Coq, and managed, thanks to the precious help of other team members like Arnaud Spiwack, to fix or improve some parts of it – concretely, I implemented a first draft of *range selectors*, which allow to apply a tactic to an arbitrary subset of the current goals; I also transitioned some tactics implemented in OCaml from the old tactics engine to the new one. I also participated to EPIT 2015 and HoTT/UF, which included a talk[11] by M. Sozeau about Equations– among other things.

Contribution

The main result of this internship is the publication of an article[9] in the LFMTP workshop, which is part of the CADE-25 conference. It presents a novel proof of consistency of Leivant's Predicative System F using the facilities of Equations.

Other contributions include rewriting parts of Equations to go towards a controlled use of the axiom K, and proposing some small features for Coq.

Arguments supporting its validity

The proof of the consistency of Leivant's Predicative System F shows quite well how Equations can be used to write a function whose termination is guaranteed by a complex measure on its inputs, while preserving the readability of the function as an algorithm to execute. It also serves to show that the code generated by Equations makes sense: it is possible to extract that code to OCaml, and we could note that the extraced code is pleasantly similar to what we could have written directly. Moreover, thanks to the elimination of the axiom K in Equations, it is actually possible to compute in Coq with the normalization function being defined.

It may not be the perfect example of pattern-matching on dependent types, but it still is an argument in favour of the usability of Equations.

Summary and future work

This internship has been a way to take part in the development of Coq and Equations, but also to learn about the internals of Coq and its theory. The team πr^2 was the perfect place for this. The publication of a paper is also an instructive step in the research life.

The follow-up of this internship will be to rewrite more parts of Equations. Currently, some information is lost during the compilation of dependent pattern-matching, which can cause some problems. Thanks to the precise formulation of the simplification rules we introduced (section 2.5 on page 7), it will be possible to keep track of that information, increasing the overall robustness of the compiler.

Another, more open, direction is to take a look at higher inductive types and how they would or should behave with Equations. This would be especially useful for working with HoTT in Coq, which is currently done in a non-computational way. This will be a topic of interest in the thesis that I will start with M. Sozeau and B. Barras.

1 Organization of this report

First, we will present Equations and its different functionalities. We will also explain somewhat shallowly its internals, insisting on the parts that are relevant to this internship.

Then, we will spend some time on the LFMTP publication about a proof of consistency of Predicative System F.

Finally, we will come back to Equations itself and a change that was implemented during this internship, in order to make Equations more robust and usable, in particular in the context of HoTT.

2 Presentation of Equations

The basic usage of Equations is to specify a function just by giving a list of clauses, each clause being a pair of a pattern and a term. We can very simply write, for example:

```
Equations negb (b : bool) : bool :=
negb true ⇒ false;
negb false ⇒ true.
```

In that case, Equations will automatically generate an elimination principle for the negb function:

```
Check negb_elim : ∀ P : ∀ b : bool, negb_comp b → Prop,
        P true false → P false true → ∀ b : bool, P b (negb b).
```

...as well as rewriting equations:

```
Check negb_equation_1 : negb true = false.
Check negb_equation_2 : negb false = true.
```

We will now present some of the features of Equations, and then shallowly explain how it works.

2.1 Dependent Pattern-Matching

As we have seen, Equations handles simple pattern-matching on inductive types such as bool. But it also handles dependent pattern-matching on inductive families. Consider the following example, inspired by the chapter DataStruct of Chlipala's book [4], of which I translated to Equations the chapters DataStruct and MoreDep:

```
Variable A : Set.

Inductive ilist : nat → Set :=
| Nil : ilist O
| Cons : ∀ {n}, A → ilist n → ilist (S n).

Inductive fin : nat → Set :=
| First : ∀ n, fin (S n)
| Next : ∀ {n}, fin n → fin (S n).
```

We define a datatype of lists indexed by their length, along with the type fin, such that fin n is isomorphic to $\{m : \mathsf{nat} \mid m < n\}$. We now want to write a function get which will take a list of length n, an element i in fin n, and returns the i-th element of the list. In pure Coq, it could look like this:

```
Fixpoint get {n} (ls : ilist n) : fin n → A :=
  match ls with
  | Nil ⇒ fun idx ⇒
      match idx in fin n' return (match n' with
      | O ⇒ A
      | S _ ⇒ unit
      end) with
      | First _ ⇒ tt
      | Next _ ⇒ tt
      end
  | Cons x ls' ⇒ fun idx ⇒
      match idx in fin n' return (fin (pred n') → A) → A with
      | First _ ⇒ fun _ ⇒ x
      | Next idx' ⇒ fun get_ls' ⇒ get_ls' idx'
      end (get ls')
end.
```

This definition is quite big for what it does. Indeed, the whole Nil branch is useless, since the fact that we are provided an element of fin n proves that n cannot be 0, and in turn proves that the Nil case is impossible. With Equations, it is possible to simply write:

```
Equations get {n} (ls : ilist n) (i : fin n) : A :=
get ?(S n) (Cons n x _) (First _) ⇒ x;
get ?(S n) (Cons n _ ls) (Next _ i) ⇒ get ls i.
```

The ?(S n) pattern indicates that Equations should not try to pattern-match on this variable, since it will be deduced from the other ones. It serves to guide the term that will be produced. It is important to recall that the term built by Equations is a pure Coq term, using nothing but native pattern-matching and simple eliminators. This way, the plugin cannot endanger the robustness of Coq as a proof assistant.

2.2 Recursion

To ensure its consistency, Coq has to make sure that every recursive definition actually terminates. To do that, it uses a syntactic check which is too conservative in many cases.

Equations allows the user to provide any logical justification for the termination of such a definition. We use this feature in the proof of consistency for Predicative System F, and as such, we will delay the presentation of this feature until then.

2.3 The `with` construct

In some definitions, we want to be able to pattern-match on an intermediary result. A typical example is the filtering on lists, which we can write the following way:

```
Variable A : Type.
Variable p : A → bool.

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

In that case, the term corresponding to the cons branch will actually be another function which takes as arguments the variables introduced by the pattern, a and l, and a new variable of type bool corresponding to $p\ a$.

2.4 How does it work?

We will now explain shallowly how Equations does its elimination of dependent pattern-matching. For a more lengthy and complete explanation of the internals of Equations, see the work of M. Sozeau[10].

The problem that Equations has to solve is, given a function type $\forall \Gamma, \tau$ and a list of clauses, to build a term of the given type to these list of clauses. To achieve this, it will consider what is called a *programming problem* of the form $\Delta \vdash \vec{p} : \Gamma$ and try to match it with each clause.

A programming problem is a substitution which associates to each variable in Γ a pattern typable in Δ. Starting with the identity substitution, where each variable in Γ is associated to itself, Equations will progressively refine this problem until it can match it with a clause.

Let us follow this informally on a very simple example. Consider the previous get function:

```
Equations get {n} (ls : ilist n) (i : fin n) : A :=
get ?(S n) (Cons n x _) (First _) ⇒ x;
get ?(S n) (Cons n _ ls) (Next _ i) ⇒ get ls i.
```

In that case, Equations starts with the programming problem:
$(n : \mathsf{nat})\ (ls : \mathsf{ilist}\ n)\ (i : \mathsf{fin}\ n) \vdash n, ls, i : (n : \mathsf{nat})\ (ls : \mathsf{ilist}\ n)\ (i : \mathsf{fin}\ n)$.
From now on, we will write Γ for $(n : \mathsf{nat})\ (ls : \mathsf{ilist}\ n)\ (i : \mathsf{fin}\ n)$.

It will try to match this to the first clause with an unification algorithm, which will get stuck and indicate that some variables need refining to be able to unify successfully – in that case, all of them. Equations will then attempt to split on each variable in order until it finds one for which the compilation succeeds, or if there is none, fails.

In this example, Equations is not allowed to split on n because of the first pattern being marked as inaccessible. Therefore it will try to split on ls, producing two subproblems, for the two constructors of $\mathsf{ilist}\ n$:

- $(i : \mathsf{fin}\ \mathsf{O}) \vdash ?(\mathsf{O}), \mathsf{Nil}, i : \Gamma$
- $(n : \mathsf{nat})\ (x : A)\ (ls : \mathsf{ilist}\ n)\ (i : \mathsf{fin}\ (\mathsf{S}\ n)) \vdash ?(\mathsf{S}\ n), \mathsf{Cons}\ n\ x\ ls, i : \Gamma$

Note that it will actually refine every other variable to take into account what can be deduced from each case. For instance, in the second problem, we know that the index n was in fact of the form $\mathsf{S}\ n$, and Equations proceeds to replace it everywhere. This mechanism relies on a systematic procedure to destruct a variable, even in presence of indexed inductive types and such complications. It can also refine the goal itself, if it depended on the variables that have been refined. We will come back to this in the next part, since it is relevant to this internship.

In the first subproblem, the unification algorithm will simply fail to unify Nil with $\mathsf{Cons}\ n\ x\ \_$ or $\mathsf{Cons}\ n\ \_\ ls$. The whole compilation should then fail, but actually, Equations will try to see if there is any variable in an empty type in the current programming problem. It is of course the case, since the type $\mathsf{fin}\ \mathsf{O}$ is empty, and Equations will use that fact to produce a term in that case.

In the second subproblem, the unification algorithms gets stuck again, this time only on the vairable $i : \mathsf{fin}\ (\mathsf{S}\ n)$. A new splitting on that variable will produce two new subproblems:

- $(n : \mathsf{nat})\ (x : A)\ (ls : \mathsf{ilist}\ n) \vdash ?(\mathsf{S}\ n), \mathsf{Cons}\ n\ x\ ls, \mathsf{First}\ n : \Gamma$
- $(n : \mathsf{nat})\ (x : A)\ (ls : \mathsf{ilist}\ n)\ (i : \mathsf{fin}\ n) \vdash ?(\mathsf{S}\ n), \mathsf{Cons}\ n\ x\ ls, \mathtt{Next}\ n\ i : \Gamma$

This time, the first subproblem will match the first clause, and the second subproblem will match the second clause. All there is left to do is to type-check the right-hand-side of each clause.

The end-result is a tree of splitting nodes, and the leaves of this tree are either directly a term, like in that example, or a refining node, corresponding to a `with` construct, introducing a new problem, with its own splitting tree attached.

Finally, Equations has to compile this tree to a Coq term. For example, each splitting node is compiled to an eliminator of the type of the variable it splits on, with a few additionnal steps used in the procedure to destruct a variable.

2.5 Splitting and simplification

During the building of the splitting tree, Equations has to split on some variable x of type $I\ \vec{t}$, where I is some inductive type and $\vec{t}$ is an instantiation of its indices. For each of the constructors of this type, we want to produce a branch in which every needed unification has been made, as we showed earlier with the get example.

A way to do it is to introduce fresh variables $\vec{s}$ and $y : I\ \vec{s}$. Then we can change a goal $\Gamma, x : I\ \vec{t}, \Gamma' \vdash \tau$ into the following goal:

$$\Gamma, \vec{s}, y : I\ \vec{s}, x : \vec{t}, \Gamma' \vdash \vec{s} \simeq \vec{t} \to x \simeq y \to \tau$$

where $\simeq$ stands for the heterogeneous equality, which allows to compare two values not necessarily in the same type, as it might the case here if some types are dependent.

The notation $\vec{s} \simeq \vec{t} \to P$ is also a shortcut for $s_1 \simeq t_1 \to \ldots \to s_n \simeq t_n \to P$, where n is the number of indices of the type I.

The standard eliminator of the type I can then be applied on y with no complications, generating one branch per constructor. In each branch, the goal is prefixed with a set of – heterogeneous – equalities relating the indices of the original instance and the indices of the constructor.

We then proceed to *simplify* these equations, by applying a set of simplifications rules until none can apply. The result is a new set of refined goals, where the original instance has been completely replaced by the possible constructors. It will also eliminate any impossible case along the way.

Simplification rules

Each rule will be presented as a triple (I, J, t) where I and J are judgments of the form $\Gamma \vdash P$ corresponding to a Coq goal, and t is a term which can have a hole $\square$ such that if $I = \Gamma \vdash P$, $J = \Gamma' \vdash Q$ and $\Gamma' \vdash \square : Q$ then $\Gamma \vdash t : P$.

In other words, t is exactly the term that would appear as a Coq proof term after the simplification, assuming I is the original goal.

A triple (I, J, t) can have some side-conditions and will be written as:

$$I \rightsquigarrow J \textbf{ with } t$$

or with side-conditions:

$$I \rightsquigarrow J \textbf{ with } t \textbf{ where } \text{some condition}$$

- Deletion: removing a reflexive equality.

$$\Gamma \vdash t = t \to P \rightsquigarrow \Gamma \vdash P \textbf{ with } \texttt{fun } (\_ : t = t) \Rightarrow \square$$

  The dependent version of the deletion rule requires the K axiom. Equations currently provides it, but it should be changed in the future as to require it to be proved, automatically if possible – for instance if this type has a decidable equality – or for the user to provide it for this specific type.

$$\Gamma \vdash \texttt{forall } H : t = t,\ P\ H \rightsquigarrow \Gamma \vdash P\ \texttt{eq\_refl} \textbf{ with } \texttt{fun } (H : t = t) \Rightarrow \texttt{K}\ P\ \square\ H$$

- Solution: replacing a variable by a term.

$$\Gamma \vdash x = t \to P\ x \rightsquigarrow \Gamma \vdash P\ t \textbf{ with } \texttt{fun } (H : x = t) \Rightarrow \texttt{J}\ P\ \square\ H$$
$$\textbf{where } x \text{ is a variable, } \Gamma \text{ does not depend on } x \text{ and } x \notin t$$

  Note that we can freely reorganize the context and revert some bindings back in the goal, in order to effectively have a context which does not depend on x. The dependent version is the same, but using a dependent version of the J elimination rule.

$$\Gamma \vdash \texttt{forall}\ H : x = t,\ P\ x\ H \rightsquigarrow \Gamma \vdash P\ t\ \texttt{eq\_refl}\ \textbf{with}$$
$$\texttt{fun}\ (H : x = t) \Rightarrow \texttt{J\_dep}\ P\ \square\ H$$
$$\textbf{where}\ x \text{ is a variable, } \Gamma \text{ does not depend on } x \text{ and } x \notin t$$

Of course, we also have these rules for the symmetrical case, where the equality is $t = x$.

- No confusion: constructors are injective and disjoint.

$$\Gamma \vdash t = u \to P \rightsquigarrow \Gamma \vdash \texttt{NoConfusion}\ t\ u \to P$$
$$\textbf{with}\ \texttt{fun}\ (H : t = u) \Rightarrow (\lambda H' : \texttt{NoConfusion}\ t\ u.\ \square\ H')(\texttt{noConfusion}\ t\ u\ H)$$
$$\textbf{where}\ t \text{ and } u \text{ are constructors}$$

  The rule of no confusion expresses the fact that constructors are injective and disjoint. ($\texttt{NoConfusion}\ t\ u : \texttt{Prop}$) returns `False` when t and u are distinct constructors, and an equality between the arguments of the constructors otherwise. Given a proof of $t = u$, $\texttt{noConfusion}\ t\ u$ returns a proof of $\texttt{NoConfusion}\ t\ u$.

  The formulation of this rule here is a little bit imprecise, as in general, it will generate not just one equality, but a sequence of possibly heterogeneous equalities between the arguments of the constructors.

- Simplification of heterogeneous equalities:

$$\Gamma \vdash t \simeq u \to P \rightsquigarrow \Gamma \vdash t = u \to P\ \textbf{with}$$
$$\texttt{fun}\ (H : t = u) \Rightarrow \texttt{simplification\_heq}\ t\ u\ \square\ H$$

  The function `simplification_heq` is a straightforward application of the `JMeq_eq` property which basically says that heterogeneous equality implies homogeneous equality. Of course, this property is not provable in pure Coq, and is in fact equivalent to the axiom K. This will be addressed in the last part of this report.

3 Predicative System F

As part of this internship, we published a paper[9] presenting a proof of the consistency of Leivant's Predicative System F[8], using the hereditary substitution technique and Equations. This is inspired by Eades' and Stump's work[6], who proved the consistency of a language similar to Predicative System F with this technique. Their language differs in the kinding rule for universal quantification, which you can find in figure 5 on page 10.

First, we will present Predicative System F itself, both the original version by Leivant and the modified version by Stump. Then we will explain how hereditary substitution can be used to prove its consistency, specifically in Stump's case. Finally, we will show how we proved it for the original system, and how Equations was useful to do that.

$$
\begin{array}{rcl}
t & := & x \mid \lambda x : T.t \mid t\ t \mid \Lambda X : k.t \mid t\ [T] \\
T & := & X \mid T \to T \mid \forall X : k.T \\
k & := & *_n\ (n \in \mathbb{N})
\end{array}
$$

Figure 1: Syntax of Predicative System F

3.1 The language

Predicative System F is basically System F_ω with universes levels added. The syntax is very simple and can be found in figure 1. Kinds are assimilated to natural numbers.

To this syntax we add typing (figure 3 on the next page) and kinding (figure 4 on the following page) rules, which rely on some well-formedness rules for contexts (figure 6 on the next page). Finally, we give to the language the obvious reduction rules, which are closed under context. They can be found in figure 2.

$$
\begin{array}{rcl}
(\Lambda X : *_p.t)\ [\phi] & \rightsquigarrow & t\ [\phi/X] \\
(\lambda x : \phi.t)\ t' & \rightsquigarrow & t\ [t'/x]
\end{array}
$$

Figure 2: Reduction rules for Predicative System F

The consistency of this language was proved by Leivant when he first introduced it. He used Girard's candidates of reducibility to prove its normalization – note that if you remove the universes levels, then you just get System F_ω. Consistency follows easily from that result and subject reduction.

While proving consistency by remarking that we can embed this language in another one is fine, it is also interesting to come up with a proof which is more fine-tuned for the language. Hereditary substitution as a technique to prove the normalization of the language has this merit, along with a more constructive presentation of normalization.

Eades and Stump accomplished that for a similar language, which we will call stratified System F, by opposition to Predicative System F. The only difference between the two languages is the kinding rules for the quantification over a kind. This new kinding rules can be found in figure 5 on the next page.

This new system, while very similar to Predicative System F, seems to not have the same expressivity: every quantification raises the universe level, which disallows the closure of a universe level by universal quantification. However, it makes for an easier proof of normalization when using hereditary substitution, which is the reason why they made that change.

$$\frac{\Gamma(x) = \phi \quad \Gamma \text{ OK}}{\Gamma \vdash x : \phi} \qquad \frac{\Gamma, x : \phi_1 \vdash t : \phi_2}{\Gamma \vdash \lambda x : \phi_1.t : \phi_1 \to \phi_2}$$

$$\frac{\Gamma \vdash t_1 : \phi_1 \to \phi_2 \quad \Gamma \vdash t_2 : \phi_1}{\Gamma \vdash t_1\ t_2 : \phi_2} \qquad \frac{\Gamma, X : *_p \vdash t : \phi}{\Gamma \vdash \Lambda X : *_p.t : \forall X : *_p.\phi}$$

$$\frac{\Gamma \vdash t : \forall X : *_p.\phi_1 \quad \Gamma \vdash \phi_2 : *_p}{\Gamma \vdash t\ [\phi_2] : \phi_1[\phi_2/X]}$$

Figure 3: Typing rules for Predicative System F

$$\frac{\Gamma \vdash \phi_1 : *_p \quad \Gamma \vdash \phi_2 : *_q}{\Gamma \vdash \phi_1 \to \phi_2 : *_{\max(p,q)}} \qquad \frac{\Gamma, X : *_p \vdash \phi : *_q}{\Gamma \vdash \forall X : *_p.\phi : *_{\max(p+1,q)}}$$

$$\frac{\Gamma(X) = *_p \quad p \leq q \quad \Gamma \text{ OK}}{\Gamma \vdash X : *_q}$$

Figure 4: Kinding rules for Leivant's Predicative System F

$$\frac{\Gamma \vdash \phi_1 : *_p \quad \Gamma \vdash \phi_2 : *_q}{\Gamma \vdash \phi_1 \to \phi_2 : *_{\max(p,q)}} \qquad \frac{\Gamma, X : *_p \vdash \phi : *_q}{\Gamma \vdash \forall X : *_p.\phi : *_{\max(p,q)+1}}$$

$$\frac{\Gamma(X) = *_p \quad p \leq q \quad \Gamma \text{ OK}}{\Gamma \vdash X : *_q}$$

Figure 5: Kinding rules for Eades' and Stump's stratified System F

$$\frac{}{\cdot \text{ OK}} \qquad \frac{\Gamma \text{ OK}}{\Gamma, X : *_p \text{ OK}} \qquad \frac{\Gamma \vdash \phi : *_p \quad \Gamma \text{ OK}}{\Gamma, x : \phi \text{ OK}}$$

Figure 6: Well-formedness of contexts

3.2 Termination of hereditary substitution

The hereditary substitution function, denoted $t[u/x]^{\phi}$, takes as arguments two terms t and u, a variable x and a type ϕ which is the type of both u and x. It returns a term t' such that $t[u/x] \rightsquigarrow^* t'$. Moreover, if t and u are in normal form, then t' has to be in normal form.

The function itself is easy to implement as a standard substitution function. The only special cases are the two application cases, which can produce a beta-redex after substitution. In that case, we have to call again a substitution function, which can result in a nested call to the hereditary substitution function.

This special case makes the proof of termination of this function complicated, and the difficulty of the proof of consistency resides in that specific point. Eades and Stump introduced a lexicographical order on (ϕ, t), with the order on t being the subterm order, and the order on ϕ being an order $<_\Gamma$ defined as in figure 7.

$$\begin{array}{rcl} \phi_1 \to \phi_2 & >_\Gamma & \phi_1 \\ \phi_1 \to \phi_2 & >_\Gamma & \phi_2 \\ \forall X : *_p.\phi & >_\Gamma & \phi[\phi'/X] \text{ where } \Gamma \vdash \phi' : *_p \end{array}$$

Figure 7: The order $<_\Gamma$

Once we prove that this order decreases at each recursive call site, the only thing remaining is to prove the well-foundedness of the order $<_\Gamma$, which relies on the fact that in Eades' and Stump's stratified System F, $\phi[\phi'/X]$ always has a strictly smaller minimal kind than $\forall X : *_p.\phi$. This is not the case in Leivant's Predicative System F, which is why we will need to come up with another order on types.

For more details on the work of Eades and Stump, you can read their article, published for the PSTT workshop in 2010.

3.3 A novel ordering on types

As mentioned above, we need to define another ordering on types which will be well-founded in Leivant's Predicative System F, and will serve to prove the termination of the hereditary substitution function.

This order is a lexicographical ordering on two components. The first one is the multiset order on the multiset of the kinds present in a type; the second one is defined as the number of type variables and universal quantifications in a type, which we will call its *depth*.

For a type ϕ, we consider $\texttt{kinds}(\phi)$, the multiset of kinds appearing in ϕ. For instance, we have:

$$\texttt{kinds}\left((\forall X : *_0.\ \forall Y : *_1.\ X \to Y) \to (\forall Y : *_0.Y)\right) = \{0, 0, 1\}$$

We consider the usual ordering on multisets for this component of our ordering on types so that, for example:

$$\{\} < \{0\} < \{0, 0\} < \{1\} < \{0, 1\} < \{0, 0, 1\}$$

The well-foundedness of this ordering is a well-known fact, we did not re-prove it and just used the part of the Coq library CoLoR[3] on multisets.

The easiest way to formally define this function `kinds` is to give the following equations:

$$\begin{aligned} \texttt{kinds}(X) &:= \{\} \\ \texttt{kinds}(\phi_1 \to \phi_2) &:= \texttt{kinds}(\phi_1) \uplus \texttt{kinds}(\phi_2) \\ \texttt{kinds}(\forall X : *_p.\phi) &:= \{p\} \uplus \texttt{kinds}(\phi) \end{aligned}$$

The depth of a type is very simply defined by the following equations:

$$\begin{aligned} \texttt{kinds}(X) &:= 1 \\ \texttt{kinds}(\phi_1 \to \phi_2) &:= \texttt{kinds}(\phi_1) + \texttt{kinds}(\phi_2) \\ \texttt{kinds}(\forall X : *_p.\phi) &:= 1 + \texttt{kinds}(\phi) \end{aligned}$$

3.4 Implementation with Coq and Equations

The full implementation can be found at http://equations-fpred.gforge.inria.fr/ and more details about it are in the paper itself. Here we will just show quickly how the formalization of such a language can look like with Coq and Equations.

3.4.1 Definition of terms

We start by defining the syntax of the language. This development is based on Vouillon's solution[13] to the POPLmark challenge for System F^{sub}. We use a standard de Bruijn encoding for the type and term variables. Kinds are encoded as simple natural numbers.

```
Definition kind := nat.

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

As the next part will explain, if we wish to avoid the axiom K in our development, we need to prove it for some of the types that we define. A way to do it is to prove that our types have decidable equality, which is really easy in that case. We will skip the statement of these decidability theorems from now on.

```
Instance eqdec_typ : EqDec typ.

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

3.4.2 Shiftings and substitutions

After defining the syntax, we can define some standard shifting and substitution operations, using EQUATIONS to do so. We only show here one that uses a `with` construct, type substitution in a type, as an example.

```
Equations(nocomp) tsubst (T : typ) (X : nat) (T' : typ) : typ :=
tsubst (tvar Y) X T' ⇐ lt_eq_lt_dec Y X ⇒ {
  | inleft (left _) ⇒ tvar Y;
  | inleft (right _) ⇒ T';
  | inright _ ⇒ tvar (Y - 1) };
tsubst (arrow T1 T2) X T' ⇒ arrow (tsubst T1 X T') (tsubst T2 X T');
tsubst (all k T2) X T' ⇒ all k (tsubst T2 (1 + X) (tshift 0 T')).
```

In that case, due to the `with`, EQUATIONS will actually generate equations such as this one, about the main function being defined:

```
Check (tsubst_equation_2 :
  ∀ (T1 T2 : typ) (X : nat) (T' : typ),
  tsubst (arrow T1 T2) X T' = arrow (tsubst T1 X T') (tsubst T2 X T')).
```

...but also equations about the auxiliary function which is defined to handle the `with`, such as that one:

```
Check tsubst_helper_1_equation_1 :
  ∀ (Y X : nat) (P : Y < X) (T' : typ),
  tsubst_obligation_3 tsubst (inleft in_left) T' = tvar Y.
```

...where tsubst_obligation_3 is that auxiliary function.

These rewriting equations can be used to reason about a function defined with EQUATIONS, without ressorting to actual computation, which can, in some cases, be impossible because of the use of axioms.

3.4.3 Kinding and typing rules

We skip some definitions about contexts and their well-formedness to show the kinding rules themselves. As said before, 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).
```

3.4.4 Metatheory

Then, we need to prove some metatheory results, like substitution lemmas, narrowing and so on. Again, we will skip most of them, just showing an example which also showcases another functionality of Equations, functional elimination.

```
Lemma tsubst_preserves_kinding :
  ∀ (e e' : env) (X : nat) (T U : typ) k,
  env_subst X T e e' → kinding e U k → kinding e' (tsubst U X T) k.
Proof.
  intros; funelim (tsubst U X T).
```

This lemma expresses the fact that type substitution in a type preserves kinding judgments. The usual way to do this proof would be an induction on the type U, and it would of course work. An alternative way to do it proceeds by functional elimination.

For every function defined with Equations, a functional elimination principle for that function is generated, allowing this kind of reasoning. In that case, instead of generating three cases for the three constructors of typ, as would a normal induction, it will generate one case for every equation generating the function tsubst – in that case, five. This often allows for a more natural and direct proof, along with making the proof term smaller, since it will use a fine-tuned elimination principle.

3.4.5 Reduction and canonical inhabitants of a type.

We add a notion of reduction to our language in order to define normalization. First, the beta-redexes for this calculus are the two possible applications.

```
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 consider 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').
```

Finally, we define its reflexive closure.

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

Now we define the normal forms of our language as a subset of the terms, using the mutually-inductive judgments normal and neutral.

```
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).
```

The canonical terms of a type T are the terms of that type which are also normal terms.

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

Finally, we say that the interpretation of a term t in a type T is a canonical term n of that type such that t reduces to n.

```
Definition interp e t T n := reds t n ∧ canonical e n T.
```

3.4.6 Hereditary substitution

We will now show the definition of hereditary substitution itself and make a few comments about it below.

```
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 } } }.
```

First of all, with no surprise, hsubst takes as arguments two terms, an index and a type. It also takes another, logical, argument, which carries a typing environment and a proof that the terms are in normal form. Those are required to prove the termination, and the correctness of the function.

This function returns a term, along with a postcondition stating that this term has the properties that we would expect. It also carries some more information, required again to prove the termination. Specifically, in the case where the term in which we substitute has the form $t1\ t2$, and the result of calling hsubst on $t1$ is a lambda-astraction abs $T'\ t'$, then we need a nested recursive call to hsubst. To prove that this call is well-founded, we need some kind of relationship between the type T', and the type U of the original call. This relationship is provided by the postcondition, because it is not always true! Indeed, it can be the case that X is not a free variable in $t1$, and then we don't have any kind of relationship between U and an hypothetical T'. This is summarized in the definition of the postcondition, that we show here:

```
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)).
```

This definition is a predicate on the inputs of the function (a pair (U, t) of a type U and a term t in which we want to substitute, a term u of type U that we want to substitute, and the index X where we want to substitute), its output (just a term r) and a typing environment (a pair (e, T) of an actual environment, and the type T of the term t) which is provided by the precondition of the function. The predicate post asserts two things:

- The term r is an interpretation of the term $t[u/X]$, which means it has the same type, is in normal form, and $t[u/X]$ reduces to r. This is what we need when we use the hereditary substitution function to prove normalization.
- If the term t was not an abstraction, but the result r is an abstraction, then we have a relationship between the type U and the type T. Recall that U is the type on which we call the function hsubst, whereas T is the type of both t and r. This is the relationship that we need in the specific case that we mentioned above.

The next thing to notice is the use of by *rec*, which is a construct added by Equations to specify how we want to prove the termination of the function being defined. In that case, the decreasing argument is the pair (ϕ, t) of a type and a term, and it decreases for the order called her_order, which is a lexicographical combination of the order on types that we described, and the direct subterm order on terms – actually we use directly the size of a term, as it is simpler for our purpose.

Then the function itself is quite easy to write. It is just standard pattern-matching on the term t and should be quite easy to read. Notice that even though hsubst returns a

dependent pair, we use its return value as if it was directly a term. This is allowed by the use of Program, a utility which also manages *obligations*, which are delayed proofs.

In that case, Equations and Program will generate two proofs at each recursive call – one for the precondition, one for proving that the argument actually decreases – and one proof for each leaf of this function – for the postcondition. This is a total of $2 \times 6 + 15 = 27$ obligations, that we can prove separately as we wish.

This clean separation between computation and logic makes for both a readable hereditary substitution function, and a comfortable way of proving each obligation on its own.

In the same way, it is then possible to write a normalization function, which leads easily to the proof of consistency of Predicative System F. The precondition and postcondition of this function are more straightforward than the previous ones.

```
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 }.
```

Consistency relies still on one additional fact, which is that neutral terms cannot inhabit any type in an environment with only a type variable, or an empty environment.

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

Finally, we can prove our main result: falsehood at any level k is an empty type.

```
Corollary consistency k : ¬ ∃ t, typing empty t (all k (tvar 0)).
Proof.
  intro. destruct H as [t Htyp].
  destruct (normalize t (ex_intro _ (empty, (all k (tvar 0))) Htyp))
    as [nf H].
  specialize (H (empty, (all k (tvar 0))) Htyp).
  destruct H as [_ [Htyp' Hnorm]].
  depelim Hnorm.
  - depelim Htyp'.
```

```
  - depelim Htyp'; depelim Hnorm.
    + depelim Htyp'.
    + depelim Htyp'.
    + apply (neutral_tvar H Htyp').
  - apply (neutral_empty H Htyp').
Qed.
```

The proof itself is just a matter of, first, calling the normalization function. We need to provide to this function a ghost variable containing the typing environment that it needs, which is done using the ex_intro constructor. This function gives us a normal term *nf* and a proof *H* that this term has the expected properties.

Concretely, we get back a normal term which inhabits falsehood. It is then just a matter of making the contradiction obvious, which we do mainly by using the *depelim* tactic, which is a tactic added by Equations that performs *dependent* elimination of a hypothesis.

4 Controlling the Axiom K

As we saw in the presentation of Equations, the original presentation of the elimination of dependent pattern-matching uses the heterogeneous equality, also called John Major's equality, which in turn has to be eliminated. This elimination is in general equivalent to the use of K on Type, which is absolutely inconsistent with the univalence axiom. In a context like HoTT, we have to avoid this. It also destroys the computational behaviour of the definitions, which means that most definitions by Equations do not compute inside the system.

The axiom K can also be introduced by the deletion rule in the simplification, in the case where the goal depends on the equality being deleted.

In any case, to avoid K altogether, or at least to better control its use, we chose to implement the elimination of dependent pattern-matching by using homogeneous equalities, in a similar way to what has been done by Cockx et al[5] in Agda.

We will first present the notion of telescopes, which is useful to use homogeneous equalities. Then we will show how to use it in place of heterogeneous equality.

4.1 Telescopes

A telescope is basically the same thing as a dependent pair in which the second component can also be a dependent pair, and so on. This is actually how we will implement it in Coq, by using the dependent pairs provided in the standard library.

We will write $(t; u; v)$ for a telescope whose components are some terms t, u and v. If the type of t is T, the type of u is U and the type of v is V, where U and V can depend on t, and V can depend on u, then the type of $(t; u; v)$ is $\{t : T \ \& \ \{u : U \ \& \ V\}\}$.

Note that to any Coq term in an inductive type, we can associate its telescope of arguments. For instance, in the case of the ilist inductive type, the telescope associated with the term Cons n x t would be $(n; x; t)$.

We can also associate to a term the telescope of the indices of its type. In the previous example, this would be just $(\mathsf{S}n)$.

4.2 From heterogeneous to homogeneous equalities

When eliminating dependent pattern-matching, we will sometimes generate a sequence of equalities in front of the goal, such that:

$$t = u \to v \simeq w \to P$$

where the type of v depends on t and the type of w depends on u. The telescopes $(t; v)$ and $(u; w)$, however, will have the same type, so we can equate them and replace the above equalities by this one homogeneous equality:

$$(t; v) = (u; w) \to P$$

Therefore, the generalization step, where we create a fresh variable and equate its indices with those of the old variable that we want to generalize, will now instead generate an equality between dependent pairs by packing the indices and variables for which we want to build an equality.

This change affects the simplification rules that we presented in section 2.5 on page 7 in a few ways. First, we need to change `noConfusion` so that it returns one homogeneous equality between two telescopes, and not anymore a sequence of equalities. Then we can remove the rule for simplifying heterogeneous equalities, as we don't actually need it anymore. Finally, we have to replace this rule with a rule to simplify equalities between dependent pairs.

- Simplification of dependent pairs:

  We have a rule for the first component of a dependent pair:

  $$\Gamma \vdash (p; x) = (q; y) \to P \rightsquigarrow \Gamma \vdash p = q \to (p; x) = (q; y) \to P$$

  ... and for the second one:

  $$\Gamma \vdash (p; x) = (p; y) \to P \rightsquigarrow \Gamma \vdash x = y \to P$$

While we can always use the first rule, we need K on the type of the first component of the pair, in order to apply the second one. This is unavoidable in any case, but that use of K is a lot more controlled than before: we don't suddenly introduce K on `Type` and the user can know quite intuitively on which type he will need to prove K – or, sufficiently, to provide a decidable equality.

References

[1] *The Coq Proof Assistant.* Available at https://coq.inria.fr/.

[2] *Equations.* Available at http://www.pps.univ-paris-diderot.fr/~sozeau/research/coq/equations.en.html.

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

[4] Adam Chlipala (2013): *Certified Programming with Dependent Types*. MIT Press.

[5] Jesper Cockx, Dominique Devriese & Frank Piessens (2014): *Pattern matching without K*. In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, ACM, pp. 257–268. Available at http://people.cs.kuleuven.be/~jesper.cockx/Without-K/Pattern-matching-without-K.pdf.

[6] Harley Eades & Aaron Stump (2010): *Hereditary substitution for stratified system f*. In: *International Workshop on Proof-Search in Type Theories, PSTT*, 10. Available at http://homepage.divms.uiowa.edu/~astump/papers/pstt-2010.pdf.

[7] Healfdene Goguen, Conor McBride & James McKinna (2006): *Eliminating dependent pattern matching. Algebra, Meaning, and Computation*, pp. 521–540. Available at http://cs.ru.nl/~james/RESEARCH/goguen2006.pdf.

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

[9] Cyprien Mangin & Matthieu Sozeau (2015): *Equations for Hereditary Substitution in Leivant's Predicative System F: a case study*. Available at http://www.pps.univ-paris-diderot.fr/~sozeau/research/publications/drafts/fpred.pdf.

[10] Matthieu Sozeau (2010): *Equations: A Dependent Pattern-Matching Compiler*. In: *First International Conference on Interactive Theorem Proving*, Springer. Available at http://mattam.org/research/publications/Equations:_A_Dependent_Pattern-Matching_Compiler.pdf.

[11] Matthieu Sozeau (2015): *Coq support for HoTT*. Available at http://www.pps.univ-paris-diderot.fr/~sozeau/research/publications/Coq_support_for_HoTT-HoTTUF-290615.pdf.

[12] The Univalent Foundations Program (2013): *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study.

[13] Jérôme Vouillon (2012): *A Solution to the PoplMark Challenge Based on de Bruijn Indices*. *Journal of Automated Reasoning* 49(3), pp. 327–362. Available at http://www.pps.univ-paris-diderot.fr/~vouillon/publi/poplmark.pdf.