How small can we make a useful type theory?

Pedro da Costa Abreu Jr.

Purdue University, USA
pdacost@purdue.edu

Wait, are you also hearing Für Elise?

Universe Hierarchy, **datatypes** and **large elimination** are well known features of modern theorem provers. However, they are also known to **complicate metatheorical properties**, such as soundness, type inference, and normalization.
This work aims to assess how much these features are actually necessary in practice. To achieve this we are building **Coquedille, a compiler from a subset of Gallina to Cedille**[1], and we will probe it by translating as much as possible of the first volume Software Foundations Series.

What do you mean by Universe Hierarchy?

Recall Russell's paradox: Imagine the set of all sets that does not include itself

$$\Delta = \{x \mid x \notin x\}$$

The million dollar question is: $\Delta \in \Delta$?
If $\Delta \in \Delta$ then $\Delta \notin \Delta$, and conversely, if $\Delta \notin \Delta$ then $\Delta \in \Delta$!. This contradiction indicates that such **Δ simply cannot exist**.
This means that a sound theory needs to disallow such definitions. **One fix to this problem is to only allow definitions to quantify over terms smaller than what is being defined.** This gives rise to a *hierarchy* of larger and larger types. In Coq this is Prop, Set, Type(1), Type(2), ...
Another fix to this problem is to disallow these definitions altogether, resulting in a weaker theory.

Cool, what about Large Eliminations?

Large Eliminations are functions that computes types from terms. The canonical example is computing a proposition from an arbitrary term, such as:

```
Definition Bool_To_Prop (b : bool): Prop :=
  match b with
  | true ⇒ True
  | false ⇒ False
  end.
```

See to the right for an example of these in action.

By Datatypes you mean Algebraic Datatypes, right?

Yep! Here are two cherrypicked examples for you:

```
Inductive Vec {A : Type} : nat → Type :=
| vnil : Vec 0
| vcons : A → forall (n : nat),
            Vec n → Vec (S n).
```

```
Inductive eq {A : Type} (x : A) : A → Prop :=
    eq_refl : eq x x.
```

So you are saying Cedille has none of this?

Right. Cedille has **no large eliminations**, and **no primitive datatypes**, and like Haskell and Agda, **no universe universe hierarchy**.

Ask me how can we get away without datatypes.

Let's talk about Coquedille

Coquedille is a translator from Coq to Cedille. **We implement Coquedille in Coq** itself, using **metacoq**[2] to get a handle on the abstract syntax tree of the Gallina terms.
The translation is mostly straightforward. The main challenge is that in Coq, kinds, types and terms are syntactically the same. In Cedille we have to clearly differentiate between the three. With this in mind we access the metacoq AST and translate it to the Cedille counterpart by using the monadic constructs provided by ext-lib[3].
Our ultimate goal is to evaluate the effectiveness of Coquedille by translating as much of the Software Foundation Series [4] as possible.

Ok... And what can you do with it?

Here is an example for you: let's **translate the following non-trivial proof: List nil and Vector vnil are not the same**. Since we are comparing terms of two different types we cannot use the regular equality, so we use the JMeq heterogeneous equality instead (notated as '$\sim=$').

Don't waste your time trying to understand this proof! All I want you to notice is that **the highlighted part is a large elimination**.

If you are curious to see the proof written with tactics, take a look at the project github[4].

```
Lemma Vector_nil_neq_List_nil
    : forall A (a: A), vnil ∼= nil.
Proof.
refine (fun (A : Type) (a : A)
        (H :  vnil A ∼= nil) ⇒
let H0 : forall x y : Vec A 0,
           x = y := eq_vnil in
let H1 : nil ∼= vnil := JMeq_sym H in
(fun H2 : nil ∼= vnil ⇒
let v := vnil in
let T := Vec A 0 in
match H2 in (JMeq x)
  return ((forall x0 y : B, x0 = y) → False)
with
| JMeq_refl ⇒
    fun H3 : forall x y : list A, x = y ⇒
    let H4 : nil = a :: nil :=
      H3 nil (a ::  nil) in
    let H5 : False :=
      Logic.eq_ind nil
        (fun e :  list  A ⇒
         match e with
         | nil  ⇒ True
         | _ ::  _ ⇒ False
         end) I (a ::  nil) H4 in
    False_ind False H5
end H0) H1).
Defined.
```

Hey! You just said Cedille doesn't support Large Eliminations!

And it doesn't! But worry not, not all is doomed. We use a nice trick of translating it to a Cedille term **δ**, which basically **checks if two lambda terms are $\beta\eta$-equivalent** via the Böhm-out algorithm[5]. If the terms being compared are not $\beta\eta$-equivalent, δ allows us to apply the explosion principle to prove anything.
In order to do this translation, we first prove a lemma relating Coq equality `eq` to Cedille built-in equality `{x ≃ y}`

```
eq_primeq : ∀ A : ⋆ . Π x : A . Π y : A . eq ·A x y → { x ≃ y }
      = Λ A . λ x . λ y . λ eq . μ' eq { eq_refl → β }.
```

Now we can translate the large elimination by using δ on the generated proof that `nil = cons a nil` (look at the type of H4 at the proof above).

```
Vector_nil_neq_List_nil : ∀ A : ⋆ . Π a : A . not ·(JMeq ·(Vec ·A 0) (vnil ·A)
·(list ·A) (nil ·A))
    = Λ A : ⋆ . λ a : A . λ H : JMeq ·(Vec ·A 0) (vnil ·A)
·(list ·A) (nil ·A) . [ H0 : Π x : Vec ·A 0 . Π y : Vec ·A 0 . eq ·(Vec ·A 0) x
y = eq_vnil ·A ] - [ H1 : JMeq ·(list ·A) (nil ·A) ·(Vec ·A 0) (vnil ·A) =
JMeq_sym ·(Vec ·A 0) ·(list ·A) (vnil ·A) (nil ·A) H ] - (λ H' : JMeq ·(list ·A)
(nil ·A) ·(Vec ·A 0) (vnil ·A) . [ v : Vec ·A 0 = vnil ·A ] - [ T : ⋆ = Vec ·A 0
] - μ' H' @(λ B : ⋆ . λ x : B . λ _ : JMeq ·(list ·A) (nil ·A) ·B x . Π H0' : Π
x' : B . Π y : B . eq ·B x' y . False) { | JMeq_refl → λ H0' : Π x : list ·A . Π
y : list ·A . eq ·(list ·A) x y . [ H' : eq ·(list ·A) (nil ·A) (cons ·A a (nil
·A)) = H0' (nil ·A) (cons ·A a (nil ·A)) ]
- δ - (eq_primeq (nil ·A) (cons ·A a (nil ·A)) H') } H0) H1.
```

What's Next?

- Translate induction and recursion
- Check how much of Software Foundations we are able to translate even with these shortcomings
- Prove the correctness of the translation
  - Translation Validation
  - Logical Relations

Wait, I still have a question

I'm more than happy to hear any insights, doubts, feedbacks, or I don't know, "hi"s.

Useful links and citations

[1] `https://github.com/cedille/cedille`.
[2] `https://github.com/MetaCoq/metacoq`.
[3] `https://softwarefoundations.cis.upenn.edu/`.
[3] `https://github.com/coq-community/coq-ext-lib`.
[4] `https://github.com/pedrotst/coquedille`.
[5] H. Barendregt, The Lambda Calculus; Its Syntax and Semantics.