Strong Normalization for Simply-Typed Lambda-Calculus in Coq: The De Bruijn Way[1]

Ezra e. k. Cooper

August 19, 2015

[1] In which I prove I am a robot

Enclosed is a proof of strong normalization (SN) for simply-typed λ-calculus (STLC) in Coq using de Bruijn indices. It uses the Tait-Girard technique of a "Reducibility" predicate which is recursive on the type. I believe it to be the smallest direct proof of that fact using those methods (i.e. de Bruijn indices in Coq), although the one by Adam Koprowski, part of the CoLoR library, is elegant and more general.

When I started on this (sometime in 2007 or 2008) I was not aware of any complete formalized proof of SN for STLC. The POPLmark challenge had already taken place, so people had begun formalizing STLC in Coq, but that challenge focused on type preservation and progress; also most of the interest was in representations other than the de Bruijn one. I don't believe anyone had embarked on a strong normalization proof, or if they had, I don't think it was in de Bruijn. In fact, the received wisdom was that de Bruijn indices would be a foolishly difficult course by which to formalize λ-calculi. To those that doubted de Bruijn, I wished to prove them wrong, or discover why they were right. Now, after some years and many hundred hours of labor, I can say with some authority: they were right. De Bruijn indices are foolishly difficult for this kind of proof. Still, I think it is worth having a full development of this basic theorem in λ-calculus in the de Bruijn representation, if only to show students clearly how foolishly difficult it really is, or for students who find de Bruijn temporarily more intuitive than the locally-nameless or higher-order abstract syntax approaches.

(Note that de Bruijn indices are still in use in compilers, where they are an efficient means to an narrow end. For formal treatment of λ-calculus, I believe they have been given an ultimate blow.)

My aim was to give the simplest and clearest direct proof of SN for STLC in Coq. No doubt my proof can still be simplified, but it is the shortest body of Coq code I have found that proves this fact with de Bruijn indices. I also hope that the result is readable. I strove to use good, non-abbreviated lemma names, and also to use little unnecessary abstraction, even if that abstraction would be useful in a wider mathematical context. This proof should be easy to read for a beginning student of λ-calculus, and of Coq. I was loathe even to introduce the so-called "setoid" rewriting features, since these call on a lot of new and abstracted Coq syntax, but the clarity which they offer their client proofs could not be argued with.

The full proof runs to 3500 lines, although that relies on a further library of 1900 lines of basic facts about lists and sets which I had to develop. This in itself was a huge part of the work of getting to the final result. I needed to prove, for example, that set union is commutative, and facts like $X = Y \Rightarrow X \cup Z = X \cup Z$. Or that removing an item from a set and then adding it back was as good as adding it in the first place. The list lemmas were used to support the representation of de Bruijn environments. Sets turned out to be useful in reasoning about the set of free variables of a term (In fact, I got very far without using sets, but was driven to free-variable sets when I started allowing reduction under a binder).

Lines per file:

27	Monomorphism.v
194	Norm.v
113	Normalize.v
18	OutsideRange.v
540	Rewrites.v
590	Shift.v
880	Subst.v
182	Term.v
30	Typing.v
208	eztactics.v
608	sn3.v
3390	total

Besides the list-manipulating routines, the cost of de Bruijn indices is partly reflected in the painful 1600 lines that are used to prove facts about “shifting” and “substitution” on de Bruijn indices, which of course are the pointy ends of that particular Dutch sword. The Tait-Girard method, by contrast, is conceptually difficult, but it occupies only about 600 lines of this proof.

I choose to treat typing in a Curry style, that is, types come after the terms rather than within them. It is possible to construct ill-typed terms, and a separate relation, `Typing`, assigns typability to them. This was no great burden; it did mean that an extra proposition, the typing relation, had to be treated with `inversion` in each proof, a minor nuisance. I’m not aware of any advantage I gained from this choice, although the potential flexibility of forming untyped terms, even temporarily, seemed appealing at the outset. A good project would be to go back through it with intrinsic Church-style types and see if anything is made harder.

Although the development was painful in its total effort, I found it extremely valuable to replicate digitally some of the work I was then doing on paper (in my PhD thesis). The rigors of Coq gave me a much keener sense of what really constitutes proof, and certainly honed my paper-proving skills.

I offer this development to students of STLC and of Coq, in hopes of lighting your way through the difficult but rewarding jungle of machine-checked proof.