Locally Nameless at Scale

Stephanie Weirich
University of Pennsylvania
USA
sweirich@cis.upenn.edu

Antoine Voizard
University of Pennsylvania
USA
voizard@seas.upenn.edu

Anastasiya Kravchuk-Kirilyuk
University of Pennsylvania
USA
akravc@sas.upenn.edu

Abstract

The Corespec project is an extensive mechanization in Coq of the metatheory of System D and System DC, two related, dependently-typed languages aimed at replacing the GHC's internal language, Core [Eisenberg 2015]. In this talk, we take a retrospective look at our development through the lens of a recent addition, η-equivalence. In particular, we describe our experience with the practical application of locally nameless variable-binding representation for mechanized metatheory, supported by the Ott and LNgen tools.

ACM Reference Format:
Stephanie Weirich, Antoine Voizard, and Anastasiya Kravchuk-Kirilyuk. 2017. Locally Nameless at Scale. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Corespec

The idea behind the Corespec project is to provide a replacement for GHC's Core language FC [Sulzmann et al. 2007] that supports full spectrum dependent types [Weirich et al. 2017] . Specifically, this project required untangling the notions of type on the one hand and of erasable component (computationally irrelevant) on the other - which current GHC essentially conflates.

As part of this process, the Corespec project developed two related dependently-typed languages: System D and System DC. The former is Curry-style and includes only computationally relevant information in its syntax; the latter includes much more: typing annotations, irrelevant arguments, and coercions in support of decidable type checking. Thus, the implicit language provides an uncluttered specification of the semantics of the language as well as a simplified context for certain parts of the metatheory.

We have shown that these two languages are related via annotation and erasure theorems. Our progress result about System D can be translated to System DC through the first theorem, and the preservation result of System DC can be similarly translated to System D via the second.

Our Coq formalization has been invaluable both in the confidence that it gives us in our results, but also in the design of the system in the first place. Although Systems D and DC draw on prior work [Eisenberg 2016; Gundry 2013; Weirich et al. 2013], we designed this language in conjunction with its mechanization.

After completing the initial design, which we reported in ICFP 2017, we have been extending the equational theory with rules for η-equivalence. For this part of the project, we have been joined by Kravchuk-Kirilyuk who did not participate in the original design.

1.1 Tool support for Locally Nameless representation

Our formalization of Systems D and DC uses a locally nameless representation for variable binding, and employs cofinite quantification [Aydemir et al. 2008] in the rules that manipulate binders.

Our choice was strongly influenced by the availability of two tools:

The first, Ott [Sewell et al. 2010], is a specification tool for programming languages semantics. One can think of it as a compiler for its specification language. It takes as input rules similar to the one in figure 1 - that is, a user-definable ASCII representation of the syntax, judgment, and rules of the language. Ott compiles this to a representation in another format - for instance, LaTeX, or Coq with a locally-nameless representation (Coq/LN in the following).

For example, here is the inductive type Ott generates to represent the syntax of the lambda-calculus:

```
Induction exp :=
   | var_f : atom -> exp
   | var_b : nat -> exp
   | abs   : exp -> exp
   | app   : exp -> exp -> exp
```

(`atom` is the set of names used for free variables)

Ott also translates rule specifications of inductive definitions to Coq inductive datatypes, as in the example below. In addition, it automatically defines operations for free variable calculation (called `fv`, free variable substitution (called `subst`) and bound variable substitution (called `open`).

The second tool, LNgen [Aydemir and Weirich 2010], is a complement to Ott when used to output Coq/LN. This tool

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

uses the same specification to derive additional definitions and helper lemmas about the Coq/LN code generated by Ott. For example, one lemma generated by LNgen asserts that

```
Lemma subst_fresh :
   forall x, x `notin` fv e -> subst x e = e.
```

2 Case study: Eta reduction

One part of extending Systems D and DC with eta-equivalence rules requires proving the confluence of a parallel reduction system extended with rules for *eta-reduction*. This confluence proof justifies the consistency of the type equivalence rules of our language and is the main component of our canonical forms theorem.

Our Ott specification of this rule, in ASCII syntax, appears in Figure 1.

```
 |- b => b'
 a = b x
---------------------------- :: Eta
 |- \ x. a => b'
```

Figure 1. Ott input for rule Eta

At first glance, this rule seems a bit strange. What has become of the free variable condition (`x `notin` fv b`) required by Eta-reduction?

When we look at the generated Coq definition, it is also not immediately obvious that `x` is not allowed to appear in `b`. Here is the case generated for the η rule above:

```
Par_Eta : forall (L:vars) (a b b' : Exp),
  Par b b' ->
  (forall x, x notin L -> open a x = app b x) ->
  Par (abs a) b'
```

This rule is the constructor of eta-equivalence within the parallel reduction relation (`Par`). It relies on co-finite quantification: the second premise generalizes over the complement of `L`, a universally quantified finite set. This means that the induction hypothesis generated by this rule is available for an infinite number of variables chosen for the binder — all except those in `L`. Furthermore, this quantification enforces the free variable condition: since `b` is quantified at the outermost level of the type, it cannot depend on the variable `x`.

With this reasoning, we see why the original Ott definition makes sense. The free variable condition *is* enforced, albeit indirectly. Notice that, in the conclusion, a appears under the binder λx. This prompts Ott to output a definition in which x can appear in a, as it is aware that λ is a binder. Now, in the second premise of the rule, no side of the equation is under a binder. Thus, in the generated definition, x can *not* appear in b. This results in the proper semantics for the rule.

Although the cofinite version of the rule generates a strong induction hypothesis, sometimes one might want an *existential* version for constructing derivations that mention a particular name for the bound variable:

```
Lemma Par_Eta_exists : forall a b b' x,
  x `notin` fv b -> x `notin` fv b' ->
  Par b b' ->
  open a x = app b x ->
  Par (abs a) b'
```

This version of the rule is proven admissible in our development. It is both handy for forward reasoning, and a good way to check that we defined the type system we intended. Note that this rule makes the free variable conditions explicit.

Finally, we have also proven one more version of this rule. This version relies on the `close` operation that replaces a free variable `x` with a bound one, suitable for `abs`.

```
Lemma Par_Eta_close : forall a b x,
  x `notin` fv a ->
  Par a b ->
  Par (abs (close x (app a (var_f x)))) b
```

While being perhaps the "most obviously correct" version of η-reduction with this representation, this rule is the least useful in proof development.

3 Lessons learned

Overall, we have been very happy with the use of Coq and the related tools for our development. In particular, the mechanization of Systems D and DC has made it easier to include new collaborators when considering extensions of the system. In this case, KK could make progress on this task without having to understand the entire development.

Furthermore, we appreciated the confidence that Coq provides in our reasoning. For example, our initial design of our η-equivalence rule depended on the following inversion lemma, which sadly, does not hold in System D, for subtle reasons.

Lemma 3.1 (Inversion). *if* $\Gamma \vDash \lambda x.b\,x : \Pi^\rho x : A \to B$ *and* $x \notin \mathrm{fv} b$ *then* $\Gamma \vDash b : \Pi^\rho x : A \to B$.

Thankfully, none of the rest of the metatheory relies on preservation for parallel reduction.

This development also relies strongly on the tool support provided by Ott and LNgen. Because we have formalized the rules of the complete system using the Ott tool, the semantics that appears typeset in our paper *exactly* corresponds to the rules that we proved in Coq. Furthermore, the extensive library of lemmas and definitions provided by LNgen were essential to our development.

Tool support has been critical to this project's very existence. Other sets of tools exist to deal with variable binding in Coq, such as Autosubst [Schäfer et al. 2015] or Needle and Knot [Keuchel et al. 2016].

References

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering Formal Metatheory. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 3–15.

Brian Aydemir and Stephanie Weirich. 2010. *LNgen: Tool Support for Locally Nameless Representations.* Technical Report MS-CIS-10-24. Computer and Information Science, University of Pennsylvania.

Richard A. Eisenberg. 2015. *System FC, as implemented in GHC.* Technical Report MS-CIS-15-09. University of Pennsylvania. https://github.com/ghc/ghc/blob/master/docs/core-spec/core-spec.pdf

Richard A. Eisenberg. 2016. *Dependent Types in Haskell: Theory and Practice.* Ph.D. Dissertation. University of Pennsylvania.

Adam Gundry. 2013. *Type Inference, Haskell and Dependent Types.* Ph.D. Dissertation. University of Strathclyde.

Steven Keuchel, Stephanie Weirich, and Tom Schrijvers. 2016. Needle & Knot: Binder boilerplate tied up. In *European Symposium on Programming Languages and Systems.* Springer, 419–445.

Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015 (LNAI)*, Xingyuan Zhang and Christian Urban (Eds.). Springer-Verlag.

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. 2010. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming* 20, 1 (Jan. 2010).

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with type equality coercions. In *Types in languages design and implementation (TLDI '07).* ACM.

Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. 2013. System FC with explicit kind equality. In *Proceedings of The 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13).* Boston, MA, 275–286.

Stephanie Weirich, Antoine Voizard, Pedro Henrique Avezedo de Amorim, and Richard A. Eisenberg. 2017. A Specification for Dependent Types in Haskell. *Proc. ACM Program. Lang.* 1, ICFP, Article 31 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110275