Under consideration for publication in J. Functional Programming

When locally-named becomes nominal: Technical Report

GYESIK LEE
Hankyong National University, Anseong, South Korea
gslee@hknu.ac.kr

HUGO HERBELIN
INRIA, Paris, France
Hugo.Herbelin@inria.fr

SUNYOUNG KIM
Yonsei University, Seoul, South Korea
sunyoungkim831@gmail.com

This technical report is an extended version of Section 5 in our paper. Many technical issues we encountered during our work and some experience will explained in detail.

About using traces. The elements of a trace are not per se relevant, which is reflected by the fact that trace relocation has no impact on substitution and Kripke semantics. The only important thing is their occurrence in the trace, which is tracked by proofs of list membership. This allows names and de Bruijn indices to be superimposed. Indeed their relationship can be observed when we look at the use of de Bruijn indices in McBride and McKinna's typechecker example from (McBride & McKinna, 2004, Section 7):

- The de Bruijn index 0 corresponds to a proof that $x \in x :: m$;
- The de Bruijn successor S on indices corresponds to a proof that $x \in m$ implies $x \in y :: m$.

Consequently, we have the following correspondency between our work and (McBride & McKinna, 2004, Section 7). The only difference between them lies in the representation style of binding variables, namely named variables versus de Bruijn indices:

- `formula`m corresponds to Expr n where the length of the list m of names is the natural number n.
- The provability predicate corresponds to the definition of well-typed terms based on the *find* view.

Traces and cofinite quantification. However, using named variables lead us to find out various hidden aspects of locally-named representation, among which the roles of parameters and constants. For instance, our last contribution about nominal representation is the result of a long pondering over the meta-theoretic question about the role of traces: Whether there is a connection between traces and the confinte quantification style as in

(Aydemir *et al.*, 2008; Charguéraud, 2012).[1] More concretely, whether there is a connection between traces and the finite sets L required in every use of cofinite quantification.

Indeed, there is a certain connection although it is partial. Unfortunately, the connection can be easily seen when parameters play their role as in the conventional style. Suppose we have adapted the cofinite quantification style $(\forall_R)$ rule in the following form where a list L of some parameters is involved:

$$\frac{\Gamma \vdash [a/x]A \quad \text{for all } a \notin L}{\Gamma \vdash \forall x A} \ (\textit{Cofinite-}\forall_R)$$

Every time when we apply $(\textit{Cofinite-}\forall_R)$ we have to provide a finite list L of names. This list L should certainly cover the parameters occurring free in A, but also in Γ. Moreover in some cases such as weakening and renaming, the list L should be extended or modified correspondingly. Therefore, traces certainly play certain role, it does not fully cover the range of L.

The case when proofs-as-programs is involved is similar except that the list L becomes longer, i.e., it should contain more parameters, not just those used in formulas, but also those used in proof terms. Indeed, the Coq library for (Aydemir *et al.*, 2008) uses a very simple idea: Every time when an instance of L is required, it is taken as the set of all parameters used in the given environment of the moment. The same technique is also used by the first author in (Lee *et al.*, 2012). Note that this fact however could cause some difficulties when people try to formalize more complex work at bigger scale. Some recent work such as (Garrigue, 2010), (Montagu, 2010), and (Rossberg *et al.*, 2014) reports the strength of cofinite quantification as well as its pitfalls.

Quantification styles. The quantification style we chose in all case studies we performed is the one using *one* fresh variable. As already discussed in Section 2 of our paper, there are several issues about this style of quantification as extensively discussed in (McKinna & Pollack, 1993, 1999). In spite of the well-known issues we still believe that the quantification style we chose is the best for formal works. There are several reasons for our belief:

- It is most similar to the conventional, informal use of quantification.
- The infrastructure needed to handle some issues such as weakening is very small compared for instance to that needed to handle substitutions . Moreover, some techniques provide a reasonable way to deal with them, among which the swapping technique used in (McKinna & Pollack, 1993, 1999; Pitts, 2003; Leroy, 2007) and the simultaneous renaming in our work as explained more concretely in the case study of simply-typed lambda calculus below.
- As mentioned above, some recents work reports issues about other styles of quantification.

Simultaneous substitution. In all of our case studies, simultaneous substitution is used instead of single substitution. We confirmed two things:

[1] This question is made by an anonymous referee of a previous version of this paper.

- In some cases such as ours presented in the paper, simultaneous substitutions is not a choice, but a must. For instance, single substitution cannot be used in showing the correspondence between syntax and semantics,
- Simultaneous substitution works well also with de Bruijn indices, see the case study of simply-typed lambda calculus below. Most importantly, we figured out how to deal with number shifting during a simultaneous substitution operation.

We also introduced a way how to simultaneously deal with multiple simultaneous substitutions. As an example, we present the definition of substitution used in our case study of simply-typed lambda calculus using the locally nameless representation:

```
Fixpoint subst_term (a:term) (ep:p_assoc) (ev:v_assoc) : term :=
  match a with
  | Param v   => match ep ^^ v with
                 | Some t => t
                 | None => Param v
                 end
  | Var n     => match ev ^^ n with
                 | Some t => t
                 | None => Var (n - (less_no ev n))
                 end
  | Fun t a1  => Fun t (subst_term a1 ep (shifting ev))
  | App a1 a2 => App (subst_term a1 ep ev) (subst_term a2 ep ev)
  end.
```

The main point in the definition above is that simultaneous substitution both for de Bruijn indices and parameters are simultaneously defined. `ep` (resp. `ev`) stands for an association for de Bruijn indices (resp. for parameters). Consequently, one can save some infrastructure needed to handle substitutions.

Dependently typed programming in Coq. With traces one can nicely work with syntax, for instance well-formedness and provability. However, it requires a good support of dependently typed programming. For instance, one could have defined the simultaneous substitution

$$[\ell \Uparrow \eta]t \in \mathtt{term}\,\ell$$

with the side condition $\mathtt{OV}(t) \subseteq \mathsf{dom}(\rho)$. Then the redundant case of (**??**) in the definition of simultaneous substitution need not be considered.

However, in Coq, working with dependent types is sometimes heavy-going. In particular, the substitution lemma does not work smoothly:

$$[\ell \Uparrow u/y]([y :: \ell \Uparrow \eta]e) = [\ell \Uparrow (y,u) :: \eta]e\,.$$

When we tried to prove it, we encountered many cases where some tactics did not work because Coq complained about type mismatch. Every time we met such a case, we then had to formulate a lemma which is a generalized form of that case. We have not tested yet, but it could work smoothly with other tools such as Agda.

Case studies. In order to check the plausibility of our ideas introduced in this paper, we took several targets to formalize in Coq. Here we give summary of our experience.

1. Trace-Based Locally-Named Representation

   A. LJT and Kripke Semantics without Parameters:

      This is the version used in our paper presentation. It is a version of Coquand-McKinna-Pollack's locally-named representation. There is only one sort of variables which stand for local variables. Parameters do not belong to the syntax at all. We let constants play the role of parameters instead. Traces are used in order to control the *unbound* occurrence of local variables in terms and formulas although they are supposed to be bound by a quantification.
      Only this version is mainly presented in the paper because our main concern is to check the role of parameters when the proofs-as-programs correspondence is not involved. Why the proofs-as-programs correspondence matters is explained in the introductory part of the paper. Note that using traces is orthogonal to the issue about the role of parameters.

   B. LJT and Kripke Semantics with Parameters.

      This version differs from the version described above only in the definition of variables. There are two sorts of variables, one for local variables and the other for parameters. Parameters play their usual role. We formalized this version in order to confirm that two versions are meta-theoretically equivalent under some conditions such as including infinitely many constants. Technically, there are nothing special extra to report except some changes caused by the presence of parameters. We intentionally wrote the Coq library following the same structure as for the version without parameters.

2. Coquand-McKinna-Pollack's Locally-Named Representation

   We took the same target as in the first case study, but this time followed Coquand-McKinna-Pollack's locally-named representation, that is, we did not make use of traces. There are also two versions, one with parameters and the other without parameters.

   A. LJT and Kripke Semantics without Parameters:
   B. LJT and Kripke Semantics with Parameters:

   Two versions are formalized following the same structure as the corresponding versions of the first case study with traces. This case is easier to handle than the first case with traces. And there are nothing special extra to report on technical issues except that the version without parameters is meta-theoretically equivalent to the version with parameters.

3. Locally Nameless Representation

   This is the case study where we tested our idea of using simultaneous substitution and simultaneous renaming. We took Leroy's contribution to the POPLmark Challenge (Leroy, 2007) and slimmed it down in order to handle the simply-typed lambda calculus, still using the locally nameless representation.

A. Soundness of Simply-typed Lambda Calculus:

The main changes we made are as follows.

- Simultaneous substitution is used instead of single substitution.
- The quantification style is different. We used the style using one fresh instance of a binder instead of using all fresh instances of a binder.
- Simultaneous renaming is used instead of variable swapping in order to handle weakening and renaming. Note that simultaneous renaming can be considered to be a special case of simultaneous substitution.

Our main concern was to check the utility of simultaneous renaming in dealing with weakening and renaming when the conventional style of quantification is used. We could show that simultaneous renaming works well when some injectivity condition which we call we call `strong_env` is imposed:

```
Definition env (A X : Type) := list (A * X).

Inductive strong_env {A X : Type} : env A X -> Prop :=
  strong_env_nil  : strong_env nil
| strong_env_cons : forall (a : A) (x : X) (e : env A X),
    strong_env e ->
    a # dom e ->
    x # img e ->
    strong_env ((a, x) :: e).
```

The `strong_env` requires that, when we add a pair to an existing list of pairs, each element of the pair should not be already used in the same position of a pair from the given list. Note that swapping is a special case of strong environments.
However, this `strong_env` condition is used only in proving the weakening property with respect to typability. In other cases such as the weakening property with respect to the well-formedness, no such condition is required. It is also the same in the case presented in Section 2 of our paper. Therefore, we conjecture that no side condition need to imposed on simultaneous substitution so long as the proofs-as-programs correspondence is not the part of the discourse.
Our overall opinion of using simultaneous renaming is very positive. It solves the problems related to the conventional style of quantification, and the infrastructure for it is not at all heavy although some proofs became longer than expected. In this sense, the swapping technique is easier to handle. However, the Generalized Weakening in Section 2 of our paper cannot be proved by using swapping. The form of swapping is too restrictive.

Bibliography

Aydemir, Brian, Charguéraud, Arthur, Pierce, Benjamin C., Pollack, Randy, & Weirich, Stephanie. (2008). Engineering formal metatheory. *Acm sigplan notices*, **43**, 3.

Charguéraud, Arthur. (2012). The locally nameless representation. *Journal of automated reasoning*, **49**(3), 363–408.

Garrigue, Jacques. (2010). A certified implementation of ML with structural polymorphism. LNCS, vol. 6461.

Lee, Gyesik, d. S. Oliveira, Bruno C., Cho, Sungkeun, & Yi, Kwangkeun. (2012). Gmeta: A generic formal metatheory framework for first-order representations. LNCS, vol. 7211.

Leroy, Xavier. (2007). A locally nameless solution to the POPLmark challenge. *INRIA Tecnical Report*, **6908**.

McBride, Conor, & McKinna, James. (2004). The view from the left. *Journal of functional programming*, **14**(1), 69–111.

McKinna, James, & Pollack, Robert. (1993). Pure type systems formalized. LNCS, vol. 664.

McKinna, James, & Pollack, Robert. (1999). Some lambda calculus and type theory formalized. *Journal of automated reasoning*, **23**(3-4), 373–409.

Montagu, Benoît. (2010). Experience report : Mechanizing Core F using the locally nameless approach (extended abstract). *5th ACM SIGPLAN Workshop on Mechanizing Metatheory*.

Pitts, Andrew M. (2003). Nominal logic, a first order theory of names and binding. *Information and computation*, **186**(2), 165–193.

Rossberg, Andreas, Russo, Claudio V., & Dreyer, Derek. (2014). F-ing modules. *J. funct. program.*, **24**(5), 529–607.