Mechanizing Type Soundness Proofs for Damas-Milner Type Systems using the Locally Nameless Representation in Coq

Zhiyuan Wu Gary Chen Ernest Ng
{wuzed, hanxic, ngernest}@seas.upenn.edu

CIS 6700 | Spring 2023

May 8, 2023

1 Introduction

Nowadays, polymorphic type influence is ubiquitous in almost every practical programming language, and our understanding of its theoretical foundations has been constantly evolving since its inception [13, 7, 10]. While this is well understood by experts, the original soundness proofs of the Damas-Milner type system [13] are not the most immediately accessible. In this project, we build on existing efforts that extend and modernize the Damas-Milner type system [10], and attempt to reproduce aspects of the soundness proofs in the *Coq* proof assistant [6]. We present our statements and proofs in a format familiar to many of us through the *Software Foundations* series [14]. Our explorations reaffirm the validity of the soundness of the Damas-Milner type system and raise a few notable yet subtle points that can aid one's comprehension of the type soundness proofs.

2 Background & Related Work

2.1 Locally Nameless Representation

Our mechanization uses a locally nameless representation in Coq. The locally nameless representation involves representing bound variables using De Bruijn indices and free variables using names [4].

Aydemir et al. [1] proposed using the locally nameless representation with cofinite quantification of free variable names in inductive relations on terms. This allows one to reason up to alpha-equivalence, i.e. for proofs about λ-terms to not depend on the specific names chosen for free variables.

One of the first uses of the locally nameless representation was by Leroy [12], who used this representation in Coq for the POPLMark Challenge, a series of tasks designed to evaluate various techniques for mechanizing programming languages metatheory.

As discussed by Charguéraud [4], the advantages to using a locally nameless representation are as follows: Firstly, the use of names for free variables allows one to avoid the typical shifting operations that are required in a pure De Bruijn representation. At the same time, the use of De Bruijn indices for bound variables ensures that equivalence classes of alpha-equivalent λ-terms have the same representation. Additionally, the syntactic distinction between free and bound variables prevents variable capture from occurring.

2.2 Damas-Milner Type System

The Damas-Milner type system [13] brings parametric polymorphism to the lambda calculus extended with let-expressions.

In this type system, types are stratified into *polytypes* and *monotypes*. Polytypes (also called *type schemes*) contain zero or more type variables. A monotype does not contain type variables. We refer the reader to Appendix A for an overview of the term language and the syntax of types.

The typing relation for this system is presented in figure 2. The typing rules for variables, abstractions and applications are the same as in the STLC. Additionally, we note that the rule LET allows variables to be bound to expressions with polytypes.

This notion of let-polymorphism is enabled by the rules INST and GEN. INST takes a polytype and instantiates the universally quantified variables with monotypes τ. In particular, quantifiers must be placed at the outermost (*prenex*) position [18]. On the other hand, GEN takes a polytype and universally quantifies over type variables α which are *not* free in the context. Notably, the Damas-Milner type system is predicative, i.e. type variables can only be instantiated to monotypes [10].

One notable property of the Damas-Milner type system is that a type inference algorithm can infer the *principal type* (most general type) for an expression without the need for any programmer intervention [7]. That is, if an expression e typechecks in the context Γ, the algorithm can find some polytype σ such that $\Gamma \vdash e : \sigma$ and any other type for e is an instance of σ [18].

When extending Damas-Milner type inference to impredicative type systems, Jones et al. [10] contrast the non-syntax-directed and syntax-directed presentations of

the DM type system. In the non-syntax-directed system, the fact that the conclusion of the GEN rule and the premise of the INST rule share the same syntactic form (see figure 2) means that one can interleave GEN and INST ad infinitum. This limitation prevents one from being able to specify a type inference algorithm directly from the non-syntax-directed type system.

Clément et al. [5] provide a proof of equivalence between the non-syntax-directed and syntax-directed type systems, discussing the tradeoffs of the two systems. In the syntax-directed system, all typing rules have a distinct syntactic form in their conclusion. This ensures that the structure of a term's typing derivation directly reflects the term's syntax, which lends itself well to a type inference algorithm [10]. However, several auxiliary judgements are needed to infer the most general type for a term. In particular, an auxiliary instantiation judgement is used in the VAR rule to instantiate the type of a polymorphic variable when it appears. Moreover, generalization is used in the LET rule to infer a polytype for the RHS of a `let`-expression [18]. We refer the reader to Jones et al. [10] for the typing rules of the syntax-directed system.

Our mechanization uses the non-syntax-directed system, as its terseness and simplicity lends itself well to proving type soundness properties.

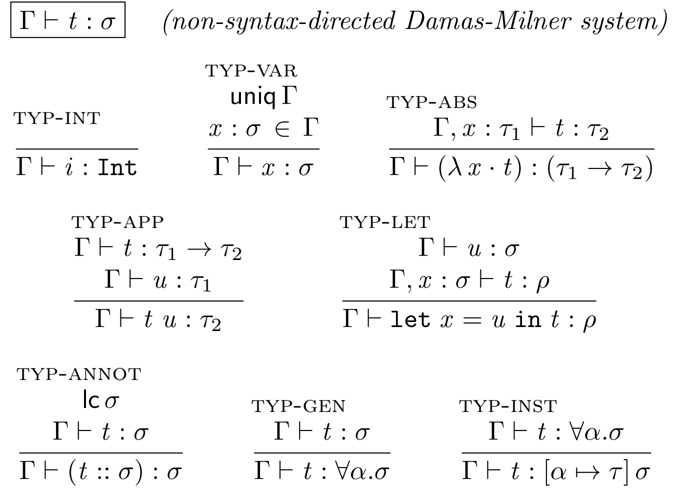
3 Definitions & Setup

We take the non-syntax-directed presentation of the Damas-Milner type system[1] directly from Jones et al. [10], which have explicit generalization and instantiation rules. We define these rules in *Ott* [15], using its locally nameless [4] backend to generate the corresponding Coq definitions. We refer the reader to figure 2 for the typing rules. Note that in the rule TYP-ANNOT, lc means "locally closed", i.e. all bound variables have bound indices in their Coq representations.

Instead of the syntactic and semantic soundness statements regarding domain semantics which are mentioned in Milner's original definitions [13], we have elected to define a simple call-by-value small-step semantics, and we have proceeded to prove progress and preservation as the criteria for type soundness. The stepping rules are as shown in Figure 1. Here we allow annotated values to step to an unannotated value.

Note that the annotated terms behave the same as unannotated ones in our usage, as they are merely an artefact of using the definitions from Jones et al. [10], who later describe bidirectional inference in their work. For convenience, we also do not consider annotated abstractions, hence the absence of a typing rule for it in figure 2.

[1]See Appendix A for the definitions of the terms and type languages



$\boxed{\Gamma \vdash t : \sigma}$ *(non-syntax-directed Damas-Milner system)*

$$\text{TYP-INT}\ \frac{}{\Gamma \vdash i : \texttt{Int}} \qquad \text{TYP-VAR}\ \frac{\mathsf{uniq}\,\Gamma \quad x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \text{TYP-ABS}\ \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\lambda\, x \cdot t) : (\tau_1 \to \tau_2)}$$

$$\text{TYP-APP}\ \frac{\Gamma \vdash t : \tau_1 \to \tau_2 \quad \Gamma \vdash u : \tau_1}{\Gamma \vdash t\ u : \tau_2} \qquad \text{TYP-LET}\ \frac{\Gamma \vdash u : \sigma \quad \Gamma, x : \sigma \vdash t : \rho}{\Gamma \vdash \texttt{let}\ x = u\ \texttt{in}\ t : \rho}$$

$$\text{TYP-ANNOT}\ \frac{\mathsf{lc}\,\sigma \quad \Gamma \vdash t : \sigma}{\Gamma \vdash (t :: \sigma) : \sigma} \qquad \text{TYP-GEN}\ \frac{\Gamma \vdash t : \sigma}{\Gamma \vdash t : \forall \alpha.\sigma} \qquad \text{TYP-INST}\ \frac{\Gamma \vdash t : \forall \alpha.\sigma}{\Gamma \vdash t : [\alpha \mapsto \tau]\,\sigma}$$

Figure 2: Typing Rules

From here, our statements of progress and preservation are largely standard. The statement for progress is constrained to closed terms.

Theorem 1 (Progress)
$\forall\, e,\ \sigma.\ \varnothing \vdash e : \sigma \Rightarrow \mathsf{value}\ e \vee \exists\, e'.\ e \longrightarrow e'$

Theorem 2 (Preservation)
$\forall\, \Gamma,\ e,\ e',\ \sigma.\ \Gamma \vdash e : \sigma \Rightarrow e \longrightarrow e' \Rightarrow \Gamma \vdash e' : \sigma$

4 Proof Implementation

As mentioned above, we started with a locally nameless representation generated by *LNgen* [2], and stated our theorem in Coq with the help of *Metalib* [16].

4.1 Progress

```
Theorem progress : forall (e:tm) (σ:ty_poly),
  typing empty e σ ->
    is_value_of_tm e \/ exists e', step e e'.
```

Here we induct on the typing judgment `typing empty e σ` ($\varnothing \vdash e : \sigma$). Most cases are relatively straightforward,

- VAR requires a non-empty context and is thus automatically discharged by contradiction.
- In the case of INT and ABS, they are both already values, so `(left; constructor)` solves the cases.
- All cases where there are congruence stepping rules can step using said rules.
- For LET and ANNOT, there exist directly corresponding syntactic rules STEP-LET and STEP-ERASE as defined above.

$\boxed{t \longrightarrow u}$ *(small-step evaluation)*

$$\frac{u \longrightarrow u'}{\texttt{let } x = u \texttt{ in } t \longrightarrow \texttt{let } x = u' \texttt{ in } t}\ \textsc{step-let1} \qquad \frac{}{\texttt{let } x = v \texttt{ in } t \longrightarrow t\,[x \rightsquigarrow v]}\ \textsc{step-let} \qquad \frac{t \longrightarrow t'}{t\ u \longrightarrow t'\ u}\ \textsc{step-app1}$$

$$\frac{u \longrightarrow u'}{(\lambda\, x \cdot t)\ u \longrightarrow (\lambda\, x \cdot t)\ u'}\ \textsc{step-app2} \qquad \frac{}{(\lambda\, x \cdot t)\ v \longrightarrow t\,[x \rightsquigarrow v]}\ \textsc{step-app} \qquad \frac{u \longrightarrow u'}{(\lambda(x :: \sigma) \cdot t)\ u \longrightarrow (\lambda(x :: \sigma) \cdot t)\ u'}\ \textsc{step-annot-app2}$$

$$\frac{}{(\lambda(x :: \sigma) \cdot t)\ v \longrightarrow t\,[x \rightsquigarrow v]}\ \textsc{step-annot-app} \qquad \frac{t \longrightarrow t'}{t :: \sigma \longrightarrow t' :: \sigma}\ \textsc{step-erase1} \qquad \frac{}{v :: \sigma \longrightarrow v}\ \textsc{step-erase}$$

Figure 1: Small-Step Semantics

- GEN follows from the derivation without the $\forall$ generalization, and INST follows from GEN by induction.

The less obvious case is ABS where both t and u are already values. In this case, in order to take a step, we need to assert that t has the syntactic form of a lambda abstraction. This brings us to state the canonical form lemma for function types.

Lemma 1 (Canonical Forms - Functions)

$$\forall\, e,\, \sigma.\ \varnothing \vdash e : \sigma$$
$$\Rightarrow \forall\, \tau_1,\, \tau_2.\ \vdash^{inst} \sigma \leq \tau_1 \to \tau_2$$
$$\Rightarrow \mathsf{value}\ e\ \Rightarrow\ \exists\, x,\, u.\ t = \lambda x \cdot u$$

```
Lemma canonical_forms_fun:
  forall (e:tm) (σ:ty_poly),
    typing empty e σ
      -> forall (τ1 τ2:ty_mono), inst σ (τ1 → τ2)
      -> is_value_of_tm e
      -> exists u, t = <{ λ_ · u }>.
```

For syntax-directed systems such as many common descriptions of STLC, this would be trivial. However, in our case, after inverting the typing judgment, we are left with the INST case, where we cannot make both $\forall\alpha.\sigma$ and $[\alpha \mapsto \tau]\,\sigma$ both equal to $\tau_1 \to \tau_2$, which is why we generalize our induction using the *inst* relation described below, adapted again from Jones et al. [10], which helps us describe exactly which types should have canonical forms.

$\boxed{\vdash^{inst} \sigma \leq \rho}$ *(instantiation)*

$$\frac{}{\vdash^{inst} \rho \leq \rho}\ \textsc{inst-refl} \qquad \frac{\vdash^{inst} [\alpha \mapsto \tau]\,\sigma \leq \rho}{\vdash^{inst} \forall\alpha.\sigma \leq \rho}\ \textsc{inst-trans}$$

Figure 3: Instantiation Relation

This ties our induction together, although not without significant further challenges. For example, a σ which instantiates to a function type does not always have a non-generalized principal type following the form of a function type. We will discuss this issue further in section 5 (Challenges) below.

4.2 Preservation

To prove preservation, we first needed to prove the weakening and substitution lemmas.

```
Theorem preservation: forall (E : ctx) e e' T,
  typing E e T ->
  step e e' ->
  typing E e' T.

Lemma typing_weakening: forall (E F : ctx) t T,
  typing E t T
  -> uniq (F ++ E)
  -> typing (F ++ E) t T.

Lemma typing_subst: forall (E F : ctx) e u
  S T (z : atom),
  typing (F ++ [(z, S)] ++ E) e T
    -> typing E u S
    -> typing (F ++ E) (subst_tm u z e) T.
```

As discussed by Weirich [17] and Aydemir et al [1],

there are two nuances that need to be addressed with the weakening lemma in our representation:

Firstly, since contexts are represented as association lists (where keys are `atoms` and values are `ty_polys`), a context is weakened via the list append operation. However, one limitation is that contexts can be weakened in the "middle" (as opposed to just one end of the context). Thus, the statement of the weakening lemma needs to be modified such that we quantify over all extensions to the original context `E`. Moreover, we needed to use the `remember` tactic to tell Coq to treat the extended context as a new context `E'` when inducting over typing derivations.[2]

Secondly, it is necessary to show that the function argument `x` is fresh with respect to the weakened context. Informal proofs on paper avoid this problem by adopting Barendregt's Variable Convention, which assumes that bound variables are chosen to be sufficiently fresh (i.e. different from free variables) [3]. However, in mechanized proofs, cofinite quantification is needed for the ABS, LET and GEN typing rules so that the IH for the corresponding cases in the proof is strengthened. Namely, instead of just holding for one single variable (as in the "exists-fresh" statement ofthese rules), the IH now holds for *any* fresh variable so long it is some finite set which includes the domain of the extended context.

Having proved weakening and substitution, we proceeded to prove the main preservation theorem. We initially tried proving preservation via inducting on typing judgements. However, we realized that our inductive hypothesis for the Abs case was insufficiently strong. As Harper discusses [9] [3], inducting on the typing judgement is not entirely appropriate because for each of the relevant cases (i.e. APP, GEN, INST) we may end up with multiple subcases after inversion, which makes the proof more intractable. Moreover, the typing for GEN and INST cases will simply be some arbitrary σ and some expression, with limited extra information given. We later switched to proving preservation via induction on the small-step evaluation relation instead. Inducting on the small-step evaluation means we no longer have to deal with GEN and INST cases where the typing is relatively general (some σ instead of a particular form). We can then restrict the expression into some more specific form which aids our reasoning. To prove preservation under the induction of step relation, we needed inversion lemmas for each typing rule, in particular the LET rule, with the idea being that any derivation tree for a Let-expression can be rephrased so that it ends in an application of the LET rule. This is necessary because in the non-syntax-directed type system, the GEN and INST rules can be interleaved. We discussed some challenges we faced while doing this in section 5 below.

5 Challenges

Throughout the process of mechanizing type soundness proofs, we encountered a few challenges.

The first challenge was in the definition of instantiation. We originally defined instantiation as what is suggested in Figure 3. That is, we have a base case of a ρ instantiating to itself (INST-REFL), and a INST-TRANS rule, similar to an induction step, in which the outer type variables are substituted with a τ. With the instantiation relation, we wanted to strengthen the hypothesis when proving the GEN and INST cases in the canonical forms lemma for functions. However, the problem we encountered is that the aforementioned process may not actually be valid. Consider the type $\forall\alpha.\alpha \to \alpha$ of a polymorphic identity function. In this case, one can substitute α with a type of the form $\tau_1 \to \tau_2$, which will convert the type of this polymorphic identity function into a function type τ. Therefore, we hypothesized that such definition does not work well and needs further editing.

Our second attempt was to rewrite the trans rule in the instantiation without the substitution. We hypothesized that such a form would benefit us by solving the problem above. However, the problem remains. Specifically, due to the restrictions of Ott, for a type $\forall\alpha.\sigma$, we can only observe and reason about σ when we open it. In other words, in order for us to use the induction hypothesis in the GEN and INST cases, we need to open σ with respect to α using an arbitrary but particular τ (which is what Ott would generate for us in order to maintain the consistency of the locally nameless representation in the type). However, such a restriction caused difficulties for two reasons: 1.) The previous cases for GEN and INST that we weren't able to prove are still impossible, and 2.) the INST case in the canonical forms lemma (after we induct on the typing derivation) has a weak induction hypothesis which states that we are able to open the σ type with some random τ. However, the goal we are trying to show is that we would like to open the σ type with some arbitrary free variables.

As Yiyun suggested, we could define two fixpoint functions outside of Ott that would forcefully "peel off" all the $\forall$ quantifiers in front of a τ. Specifically, we can define two functions as follows:

$$P(\sigma) = \begin{cases} \top & \text{if } \sigma = \text{Function type} \\ P(\sigma_1) & \text{if } \sigma = \forall\alpha.\sigma_1 \\ \bot & \text{otherwise} \end{cases} \tag{1}$$

[2] This issue is avoided in *Software Foundations* [14], where contexts are represented as partial maps. However, this representation requires a notion of map inclusion to be defined when discussing extensions to a context.

[3] Discussed in pg. 70 (Section 11.1) of Harper's *Programming Languages: Theory and Practice (2006).*

$$B(\sigma) = \begin{cases} \top & \text{if } \sigma = \text{Bounded type variable} \\ P(\sigma_1) & \text{if } \sigma = \forall\alpha.\sigma_1 \\ \bot & \text{otherwise} \end{cases} \quad (2)$$

This two functions go against the restrictions of the locally nameless representation because we intentionally leave bounded variables inside the expressions intact in order to aid our reasoning. Specifically, the first function allowed us to manually inspect whether the inside of a σ looks like a function type. If so, this means that the expression must always be a function (one cannot open a σ type that looks like a function and make it a non-function). The second function allowed us to identify a type with the structure of $\forall\alpha_1.\forall\alpha_2.\ldots.\forall\alpha_k.\ \alpha_i$ where $1 \leq i \leq k$. As Yiyun pointed out, such a case cannot happen in the empty context because there is no corresponding term. Given that this structure is our main concern, if we can eliminate this possibility, then we can prove in the canonical forms lemma that the rest of the terms in the GEN or INST cases must be in the form of a function (i.e. return $\top$ from $P(\cdot)$).

With the two functions, we can make the canonical forms lemma stronger and say that a σ is a function type if $P(\sigma)$ holds. Moreover, we no longer have to use the instantiation relation defined earlier since $P(\sigma)$ is a stronger relation.

However, after this change was adopted, it is not the case that the entirety of the proof of progress proceeded without issue. Some challenges remain when using $P(\sigma)$ and $B(\sigma)$ predicates. One main challenge we faced initially was the difficulty of proving the $\forall\alpha.\sigma$ case of the P and B predicates when opening. That is, we want to prove that:

$$\forall\sigma\ \forall x,\ P(\sigma\hat{\ }x) = P(\sigma)$$
$$\forall\sigma\ \forall x,\ B(\sigma\hat{\ }x) = B(\sigma)$$

where x is a free variable or a bound variable. Originally, we wrote the statement similar to what is shown as follows:

```
Lemma normalize_open_n_var_rec :
  forall (σ: ty_poly) (n : nat) (a : tyvar),
    P (open_ty_poly_wrt_ty_mono σ
        (ty_mono_var_f a))
    = P (σ).
```

However, the issue with the above statement happens in the $\forall\alpha.\sigma$. Specifically, the induction hypothesis is not strong enough in that the induction hypothesis is for some random index (if we unfold the `open_ty_poly_wrt_ty_mono` definition), instead of for any arbitrary index. The solution, as Yiyun pointed out, is to state the lemmas with respect to `open_ty_poly_wrt_ty_mono_rec`. In this case, we are able to prove for the statement for all indices, and we were able to prove canonical forms at the end by stating a newer version of the lemma:

```
Lemma canonical_forms_fun:
  forall (e:tm) (σ:ty_poly),
    typing empty e σ
      -> P(T)
      -> is_value_of_tm e
      -> exists u, t = <{ λ_ · u }>.
```

The GEN case can be solved by recognizing that σ will remain a function when it is opened with a free variable. Similarly, the INST case can be solved with the following two lemmas:

```
Lemma not_bad_open_func_equiv:
  forall (σ:ty_poly) (n : nat) (τ : ty_mono),
      ~ B (σ)
    -> P (open_ty_poly_wrt_ty_mono_rec n τ σ)
        = P (σ).
```

which says that if σ is not in the form $\forall\alpha_1.\forall\alpha_2.\ldots.\forall\alpha_k.\alpha_j$, then if σ looks like a function, opening it with any τ will be a function. Another lemma Yiyun advised us to use was:

```
Theorem wt_no_bad_poly:
  forall (Gamma : ctx) (e : tm) (sig : ty_poly)
    (Htyping: typing Gamma e sig),
    is_value_of_tm e
      -> not (bad_fun_poly sig).
```

This lemma says that if a term is well type, and that term is a value, then it is impossible for the term to be bad. As Yiyun had explained, the reason why this hold is that there is no expression that can give the type $\forall\alpha.\alpha$.

Using both lemmas, we were able to replicate Yiyun's proof of canonical forms. We have listed this process as part of the challenges due to the struggles we faced, and we would not be able to surmount these difficulties without Yiyun's assistance.

Another challenge we faced which we weren't able to fully resolve was finding the correct statement of the auxiliary inversion lemmas for the proof of preservation as shown below:

```
Lemma let_inversion : forall Gamma u e sig',
    typing Gamma (exp_let u e) sig'
      -> exists L (sig : ty_poly),
        typing Gamma u sig
      /\ forall x, x `notin` L
      -> typing ((x, sig)::Gamma) e sig'.

Lemma annot_inversion :
  forall (Gamma : ctx) (e : tm)
    (sig sig': ty_poly),
  typing Gamma (exp_type_anno e sig) sig' ->
  sh sig sig' ->
  typing Gamma e sig.
```

We are in the process of proving these inversion lemmas, along with the inversion lemma for application, and abstraction. For the Let-expression inversion lemma, we are having trouble in making the induction hypothesis generalized enough. Currently we have some cases in which the type for u in the goal is not necessarily the same as the type for u in the context due to the existential quantifier in the statement of the lemma. We need to generalize the typing for u to strengthen the induction hypothesis.

For the inversions lemma for annotations, currently we are still struggling with some cases in which there is insufficient useful information in the context. For instance, we have struggled to reconcile the expression in the goal e, with the expressions in the context `exp_type_annot e`. We believe that some other relationship (maybe a stepping relationship), needs to be included in the lemma statement. We also suspect that we need to declare some Fixpoint functions that can reason about `open_ty_poly_wrt_ty_mono`. The situation we are facing may be similar to our earlier struggles with proving that $P(\cdot)$ and $B(\cdot)$ hold. The solution in this case may be to also define some Fixpoint functions such that we can reason about the unfolded definition of `open_ty_poly_wrt_ty_mono_rec` instead of the folded version, in which we are only supplied with an arbitrary index instead of all indices.

Specifically, we also had some trouble integrating cofinite quantification and a shallow subsumption relation into the statement of the inversion lemmas. We attach our attempt at defining shallow subsumption in Ott below:

```
Inductive sh : ty_poly -> ty_poly -> Prop :=
  | sh_refl :
    forall (sig:ty_poly),
      lc_ty_poly sig ->
      sh sig sig
  | sh_spec :
    forall (rho1 rho2:ty_rho) (tau:ty_mono),
      sh  (open_ty_poly_wrt_ty_mono
        (ty_poly_rho rho1) tau )
        (ty_poly_rho rho2) ->
      sh (ty_poly_poly_gen (ty_poly_rho rho1))
        (ty_poly_rho rho2)
  | sh_skol :
    forall (L:vars) (sig:ty_poly) (rho:ty_rho),
      ( forall a , a \notin  L  -> sh
        ( open_ty_poly_wrt_ty_mono sig
          (ty_mono_var_f a) )
        (ty_poly_rho
          ( open_ty_rho_wrt_ty_mono rho
          (ty_mono_var_f a) ) ) )  ->
      sh (ty_poly_poly_gen sig)
        (ty_poly_poly_gen (ty_poly_rho rho)).
```

We are still unsure whether our definition of the subsumption rule is the cause of the difficulties we face. Specifically, we are unsure whether the Ott-generated cofinitely quantified version of the SKOL rule (`sh_skol`) accurately reflects the "exists-fresh" statement of the corresponding rule on paper [10]. Another uncertainty we have is that the Ott-generated version of the SPEC rule (`sh_spec`) quantifies over all monotypes τ when opening ρ_1 with respect to τ, and we are unsure whether this quantification is appropriate as there many be other conditions that τ must satisfy.

To overcome these limitations for the inversion lemma and proceed, we could examine Leroy's solution to the POPLMark Challenge which uses the locally nameless representation [12], and investigate how the inversion lemmas and type substitution lemmas in Leroy's Coq development (originally proven for System $F_{<:}$) could be adapted to the Damas-Milner type system.

6 Future Work

In our efforts to address the challenges mentioned above, we have discovered that many of our issues stemmed from the non-syntax-directed typing rules. Thus the first obvious thing to explore would be the syntax-directed version of the system also described in Jones et al. [10], as well as its equivalence with the non-syntax-directed version following the proof outline presented by Clément et al. [5]. We suspect that in the syntax-directed system, type soundness proofs would likely be more straightforward. Additionally, while the completeness theorem of the type inference algorithm is generally deemed more difficult, it could also be worthwhile for us to mechanize its proof.

We can also extend our proofs to involve more automation. As pointed out by Yiyun, we can use tactics libraries such as CoqHammer that could automate the simple destruct / induction / inversion process, in which we currently had to perform manual case analysis. Such tools may help us in help us move the expedition further.

We are also curious about the role of the locally nameless representations played in our struggles because we had to constantly deal with opening and closing the types. Thus, it may also be informative to attempt the same proofs in other representations, such as De Bruijn indices [8], although this might require more manual statements of shifting and closed-ness constraints, especially if we continue to use Coq.

One of the troubles we kept facing was considering the edge cases that may potentially falsify a statement. Thus, we are also interested in incorporating randomized property-based testing tools like QuickChick [11] as a sanity check when verifying the statement of our lemmas.

We could also expand our setup to reproduce soundness proofs of other systems in [10, Jones et al.], which include extensions such as higher-ranked types and bidirectional inference.

7 Conclusion

Through this project, we have learned how to use Ott and LNGen to mechanize type soundness proofs using the locally nameless presentation.

We have gained a much deeper appreciation of the challenges associated with variable binding, and how using the locally nameless representation with cofinite quantification enables stronger inductive hypotheses compared to standard "exists-fresh" quantification. This is because for rules which involve opening abstractions, the cofinitely-quantified IH holds for not just one single name, but instead for all variable names except those in some finite set L. Moreover, in practice, we can include the finitely many names that lead to clashes in L, which makes the cofinitely-quantified versions of introduction forms practical to use in proof developments.

In the non-syntax-directed type system, we faced the challenge whereby the GEN and INST rules could be invoked at any time during a typing derivation. Our experience mechanizing type soundness proofs for this system gave us a deeper appreciation for the syntax-directed presentation of the type system, where there is at most one typing rule that can be applied for each possible syntactic form of a term.

8 Acknowledgements

We thank Yiyun Liu for his advice on the canonical forms lemma for functions, his hints on Coq definitions, and his advice on Coq proof techniques such as utilizing CoqHammer for potential automation. We also want to thank Professor Stephanie Weirich for her advice and mentorship throughout this project, including pointing out the appropriate generalization of the canonical forms lemma, as well as the purpose of the inversion lemmas for preservation.

References

[1] Aydemir, B., Charguéraud, A., Pierce, B. C., Pollack, R., and Weirich, S. Engineering formal metatheory. *SIGPLAN Not. 43*, 1 (jan 2008), 3–15.

[2] Aydemir, B. E., and Weirich, S. Lngen: Tool support for locally nameless representations.

[3] Barendregt, H. P. *The lambda calculus - its syntax and semantics*, vol. 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985.

[4] Charguéraud, A. The locally nameless representation. *J. Automat. Reason. 49*, 3 (Oct. 2012), 363–408.

[5] Clément, D., Despeyroux, T., Kahn, G., and Despeyroux, J. A simple applicative language: Mini-ml. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (New York, NY, USA, 1986), LFP '86, Association for Computing Machinery, p. 13–27.

[6] Coq Development Team. The Coq Proof Assistant, version 8.8.0. https://doi.org/10.5281/zenodo.1219885, Apr 2018.

[7] Damas, L., and Milner, R. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '82* (New York, New York, USA, 1982), ACM Press.

[8] de Bruijn, N. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings) 75*, 5 (1972), 381–392.

[9] Harper, R. Programming Languages: Theory and Practice.

[10] Jones, S. P., Vytiniotis, D., Weirich, S., and Shields, M. Practical type inference for arbitrary-rank types. *J. Funct. Prog. 17*, 1 (Jan. 2007), 1–82.

[11] Lampropoulos, L., and Pierce, B. C. QuickChick: Property-Based Testing In Coq. Electronic textbook. https://softwarefoundations.cis.upenn.edu/qc-current/index.html, 2018.

[12] Leroy, X. *A locally nameless solution to the POPLmark challenge.* 2007.

[13] Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17*, 3 (1978), 348–375.

[14] Pierce, B. C., de Amorim, A. A., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V., and Yorgey, B. *Software Foundations.* 2008.

[15] Sewell, P., Zappa Nardelli, F., Owens, S., Peskine, G., Ridge, T., Sarkar, S., and Strnisa, R. Ott: Effective tool support for the working semanticist. *J. Funct. Program. 20* (01 2010), 71–122.

[16] Weirich, S. Language Specification & Variable Binding: The Locally Nameless Representation. https://www.seas.upenn.edu/~sweirich/dsss17/html/Stlc.Lec1.html. CIS 6700 Lecture, 2023–01-23.

[17] Weirich, S. Reasoning about LN: Typing (Preservation & Progress). https://www.seas.upenn.edu/~sweirich/dsss17/html/Stlc.Lec2.html. CIS 6700 Lecture, 2023–01-25.

[18] Weirich, S. Practical type inference for arbitrary-rank types. CIS 6700 Lecture Slides, 2023. Lecture delivered on: 2023–03-22.

A Term & Type Grammars

Monotypes, τ	$::=$		Monotypes
	$\mid$	$\texttt{Int}$	
	$\mid$	α	
	$\mid$	$\tau_1 \to \tau_2$	
Rho-types, ρ	$::=$		Rho-types
	$\mid$	τ	
Polytypes, σ	$::=$		
	$\mid$	ρ	
	$\mid$	$\forall \alpha.\sigma$	
Exp, t, u, e	$::=$		
	$\mid$	i	Literal
	$\mid$	x	Variable
	$\mid$	$e\ e'$	Application
	$\mid$	$\lambda\, x \cdot e$	Abstraction
	$\mid$	$\texttt{let}\ x = e\ \texttt{in}\ e'$	Local binding
	$\mid$	$\texttt{if}\ e\ \texttt{then}\ e'\ \texttt{else}\ e''$	Local binding
	$\mid$	$\lambda(x :: \sigma) \cdot t$	Typed abstraction
	$\mid$	$t :: \sigma$	Type annotation
Value, v	$::=$		
	$\mid$	i	
	$\mid$	$\lambda\, x \cdot t$	
Typing contexts, Γ	$::=$		
	$\mid$	$\varnothing$	empty context
	$\mid$	$\Gamma, x : \sigma$	assumption

B Up-to-Date Progress

See https://github.com/wu000168/milner_types-cis6700
(The latest code for the lemmas discussed in section 5 (Challenges) is located in the `dev-3` branch.)