Proof Pearl: The power of higher-order encodings in the logical framework LF

Brigitte Pientka

School of Computer Science
McGill University
bpientka@cs.mcgill.ca

Abstract. In this proof pearl, we demonstrate the power of higher-order encodings in the logical framework Twelf[PS99] by investigating proofs about an algorithmic specification of bounded subtype polymorphism, a problem from the POPLmark challenge [ABF$^+$05]. Our encoding and representation of the problem plays to the strengths of the logical framework LF. Higher-order abstract syntax is used to deal with issues of bound variables. More importantly, we exploit the full advantage of parametric and higher-order judgments. As a key benefit we get a tedious narrowing lemma, which must normally be proven separately, for free. Consequently, we obtain an extremely compact and elegant encoding of the admissibility of general transitivity and other meta-theoretic properties.

1 Introduction

Stimulated by the POPLmark challenge [ABF$^+$05], there have been many discussions over the last two years about what support current proof assistants provide for specifying programming languages and proofs about them. In this paper we present an implementation of bounded subtype polymorphism within the logical framework Twelf [PS99], inspired by the first problems in the POPLmark challenge[ABF$^+$05]. The logical framework LF [HHP93] provides an elegant meta-language for the specification of deductive systems together with the proofs about them. It has been particularly successful in encoding the meta-theory of programming languages. Encodings in LF typically rely on ideas of higher-order abstract syntax where object variables and binders are implemented by variables and binders in the meta-language (i.e. logical framework). One of the key benefits behind higher-order abstract syntax representations is that one can avoid implementing common and tricky routines dealing with variables, such as capture-avoiding substitution, renaming and fresh name generation.

However concentrating on encoding object variables and binders by variables and binders in the meta-language only utilizes part of the power of higher-order encodings. One important and often neglected aspect of higher-order encodings is the direct support for implementing hypothetical and parametric judgments via higher-order functions. This has two important consequences: 1) A context or environment to keep track of assumptions is unnecessary. This eliminates the need of building up a context and managing it explicitly. More importantly, it eliminates the need to explicitly reason about the properties of the context such as weakening, exchange, or strengthening. 2) Since we can view hypothetical judgments as functions, applying a substitution lemma amounts to function application. As a consequence tedious substitution lemmas need not be proven separately. Encodings within the logical framework LF usually try to exploit both properties extensively. This approach typically greatly reduces the size of the final proofs. Moreover, the meta-theoretic development scales better when we extend the language by including new constructors.

In this proof pearl we play to the strengths of logical frameworks, and we follow in spirit the philosophy expressed by Bob Harper's comments subsequent to posting our solution to the POPLmark mailing list:

> " You have to listen to the logical framework, as it were, and take its advice in guiding you towards a better way to formulate your system. We learned this lesson many years ago when we first invented LF — the exercise of formalizing a logic in LF does wonders for the logic." (Bob Harper's post to the POPLmark-list 2 May 2006).

In other words, the representation of our problem determines our success or failure to prove properties about it. Hence, we start by taking a close look at the original problem posed in the POPLmark challenge, and explain our choice of representation in the logical framework LF to take full advantage of the logical framework LF. The pay-off will be substantial, compared to other solutions which follow the problem specification more literally. We use not only higher-order abstract syntax to encode bound variables in our object language, but we model parametric and hypothetical judgments as higher-oder functions thereby eliminating the need to prove substitution and narrowing lemmas separately. Our reasoning is purely structural and does not require any extra arguments to reason explicitly about the size of arguments and derivations. And finally, our solutions unlike others does not require tedious explicit proofs to show that certain cases are impossible. As we will demonstrate our proof of admissibility of transitivity is remarkably short and fits on less than a page.

Organization of the paper

The paper is organized as follows: We first introduce our algorithmic subtyping relation for bounded polymorphism, and comment on the difference between this formulation and the original challenge in [ABF$^+$05]. Next, we will discuss its encoding in the logical framework Twelf, develop the proof that transitivity is admissible and show its implementation in Twelf. Finally, we will compare and discuss other POPLmark challenge solutions to this problem.

2 Bounded polymorphic subtyping

In this section, we will briefly introduce bounded subtype polymorphism and the basic meta-theory of System $F_{\texttt{sub}}$ (see also Ch. 26 [Pie02b]). In this system, we enrich polymorphic types such as $\forall\alpha.T$ with a subtype relation and refine the universal quantifier to carry a subtyping constraint. The syntax of types can be defined as follows:

$$\begin{array}{ll} \text{Types} & T ::= \text{top} \mid \alpha \mid T_1 \Rightarrow T_2 \mid \forall\alpha \leq T_1.T_2 \\ \text{Context} & \Gamma ::= \cdot \mid \Gamma, w{:}\alpha \leq T \end{array}$$

In $\forall\alpha \leq T_1.T_2$, the type variable α only binds occurrences of α in T_2. We will use small Greek letters such as α, β etc. to denote type variables. The typing context Γ keeps track of constraints such as $\alpha \leq T$. A context $\Gamma, w{:}\alpha \leq T$ is well-formed if T is a well-formed type in the context Γ and there exists no other assumption $\alpha \leq S$ in Γ, i.e. there is a unique constraint $\alpha \leq T$ for each type variable. Next, we describe a subtyping algorithm using the judgment:

$$\Gamma \vdash T \leq S \qquad \text{Type } T \text{ is a subtype of } S \text{ in the context } \Gamma$$

$$\frac{}{\Gamma \vdash T \leq \text{top}}\ \text{sa-top} \qquad \frac{\alpha \leq T \in \Gamma}{\Gamma \vdash \alpha \leq T}\ \text{sa-hyp} \qquad \frac{}{\Gamma \vdash \alpha \leq \alpha}\ \text{sa-ref-tvar}$$

$$\frac{\Gamma \vdash T_1 \leq S_1 \quad \Gamma \vdash S_2 \leq T_2}{\Gamma \vdash S_1 \Rightarrow S_2 \leq T_1 \Rightarrow T_2}\ \text{sa-arr} \qquad \frac{\Gamma \vdash \alpha \leq U \quad \Gamma \vdash U \leq T}{\Gamma \vdash \alpha \leq T}\ \text{sa-tr-tvar}$$

$$\frac{\Gamma \vdash T_1 \leq S_1 \quad \Gamma, w{:}\alpha \leq T_1 \vdash S_2 \leq T_2}{\Gamma \vdash \forall\alpha \leq S_1.S_2 \leq \forall\alpha \leq T_1.T_2}\ \text{sa-all}^{\alpha,w}$$

The description is algorithmic in the sense that general rules for reflexivity and transitivity are admissible, and for each type constructor, top, $\forall$ and $\Rightarrow$ there is one rule which can be applied. Following standard design principles found in many logics, we allow hypothetical reasoning, i.e. given an assumption $\alpha \leq T$ we can directly conclude $\alpha \leq T$. While the resulting algorithm is syntax directed, it does not eliminate all non-determinism, if viewed as a bottom-up search algorithm. In the rule sa-tr-tvar for example we still need to guess the type U. To illustrate how a proof can be derived, consider the following example:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{w{:}\alpha \leq \text{top}, v{:}\beta \leq \alpha \vdash \alpha \leq \alpha}\ \text{sa-ref-tvar} \quad \dfrac{}{w{:}\alpha \leq \text{top}, v{:}\beta \leq \alpha \vdash \beta \leq \alpha}\ \text{sa-hyp}}{w{:}\alpha \leq \text{top}, v{:}\beta \leq \alpha \vdash (\alpha \Rightarrow \beta) \leq (\alpha \Rightarrow \alpha)}\ \text{sa-arr}}{w{:}\alpha \leq \text{top} \vdash (\forall\beta \leq \alpha.\alpha \Rightarrow \beta) \leq (\forall\beta \leq \alpha.\alpha \Rightarrow \alpha)}\ \text{sa-all}^{\beta,v}}{\cdot \vdash (\forall\alpha \leq \text{top}.\forall\beta \leq \alpha.\alpha \Rightarrow \beta) \leq (\forall\alpha \leq \text{top}.\forall\beta \leq \alpha.\alpha \Rightarrow \alpha)}\ \text{sa-all}^{\alpha,w}$$

We note that the formulation of the transitivity rule given here is different from the challenge problem stated in [ABF$^+$05]. Instead of the sa-hyp and the sa-tr-tvar rule, we typically find the following specialized transitivity rule:

$$\frac{(\alpha \leq U) \in \Gamma \quad \Gamma \vdash U \leq T}{\Gamma \vdash \alpha \leq T}\ \text{sa-tr'}$$

In contrast to the rule sa-tr-tvar, there is no need to guess U. Clearly, the rule sa-tr' is a derived rule in the system we stated. In other words, the system we consider is slightly more general. The lack of a general hypothesis rule is compensated by using sa-tr' together with reflexivity. For example to prove $w{:}\alpha \leq \beta \vdash \alpha \leq \beta$, we will use first transitivity to look up $\alpha \leq \beta$ in the context, and then prove that $\beta \leq \beta$ by reflexivity. As we will show, our logically motivated algorithmic description of subtyping will lead to a much cleaner meta-theoretic development and is a nice example how the original problem specification influences the proofs about it.

3 Representing bounded polymorphism using HOAS

In this section, we encode bounded polymorphic types together with the algorithmic subtyping system in the Twelf system, which is an implementation of the logical framework LF. Twelf supports the specification of deductive systems, given via axioms and inference rules, together with the proofs about them, and is hence ideally suited for this task. The LF language, a dependently typed lambda-calculus, can be briefly described as follows:

$$\begin{array}{ll}
\text{Kinds } K & ::= \mathsf{type} \mid \Pi x{:}A.K \\
\text{Types } A & ::= a\ M_1 \ldots M_n \mid A_1 \rightarrow A_2 \mid \Pi x : A_1.A_2 \\
\text{Objects } M & ::= x \mid c \mid M_1\ M_2 \mid \lambda x{:}A.M
\end{array}$$

Objects provided by the logical framework LF include lambda-abstraction, application, constants and variables. The type label at the lambda-abstraction can be omitted in practice. Types classify objects, and range over type constants a which may be indexed by objects $M_1 \ldots M_n$, as well as non-dependent and dependent function types. Viewing types as propositions, LF types can be interpreted as logical propositions. The atomic type $a\ M_1 \ldots M_n$ corresponds to an atomic proposition, the non-dependent function type $A_1 \rightarrow A_2$ corresponds to an implication, and the dependent function type $\Pi x{:}A.B$ can be interpreted as the universal quantifier. We will use types and formulas interchangeably. Kinds classify types. For a thorough introduction to LF including the typing rules and meta-theoretic development we refer the reader to [Pfe97]. Here we focus on how to use the full power of LF to formalize the example of bounded polymorphism and proofs about it.

3.1 Encoding bounded polymorphic types

To represent the previous system describing bounded subtype polymorphism, we start with representing the language of types as objects in LF. The first obvious question is how to represent the polymorphic type $\forall\alpha \leq T_1.T_2$, in particular how to represent type variables α and the fact that any occurrence of the type variable α in the type T_2 is bound by the universal quantifier. Encodings in the logical framework LF are typically based on higher-order abstract syntax where we represent variables in the object language (i.e. the type variables α in

$\forall\alpha \leq T_1.T_2$) with variables in the meta-language (i.e. bound variables in LF). In other words, type variables α bound by $\forall$ in the object-language will be bound by λ-abstraction in the meta-language. With this representation the framework provides α-conversion and substitution for the object language.

To clarify the translation between object-level types and their representation in LF, we introduce the following function $\ulcorner\cdot\urcorner$ which maps object-level types to their representation in the logical framework LF. On the left, we show how to translate object-level types to their representation into LF, and on the right we show the definition of the constructors in LF.

Encoding object-level types to LF objects		Data-type definition in LF
		`tp : type.`
$\ulcorner\alpha\urcorner$	$= \alpha$	
$\ulcorner\text{top}\urcorner$	$= \texttt{top}$	`top : tp.`
$\ulcorner T_1 \Rightarrow T_2\urcorner$	$= \texttt{arr}\,\ulcorner T_1\urcorner\,\ulcorner T_2\urcorner$	`arr : tp -> tp -> tp.`
$\ulcorner\forall\alpha \leq T_1.T_2\urcorner$	$= \texttt{all}\,\ulcorner T_1\urcorner\,\lambda\alpha.\ulcorner T_2\urcorner$	`all : tp -> (tp -> tp) -> tp.`

On the right, we define an LF type called `tp`, with the constructors `top`, `arr`, and `all`. The type for the constructor `all` takes in two arguments, an argument of type `tp` and another argument of function type (`tp -> tp`). Intuitively, the first argument stands for the bound, while the second argument represents the body of the forall-expression. The key idea to note is that in the type $\forall\alpha \leq T_1.T_2$ the type variable α is a binding occurrence with scope T_2 and is not allowed to occur in T_1. Variables α in the object-level are represented by variables α in the meta-level. To illustrate the encoding of a concrete type, consider the following examples.

Object-level type	LF object
$\forall\alpha \leq \text{top}.\alpha \Rightarrow \alpha$	`all top` ($\lambda\alpha$. `arr` α α)
$\forall\alpha \leq \text{top}.\forall\beta \leq \alpha.\beta \Rightarrow \alpha$	`all top` ($\lambda\alpha$. `all` α ($\lambda\beta$. `arr` β α))

To prove the adequacy of this encoding we need to show that there is a bijection between the types represented in our object-language and the types described in the logical framework LF. This follows standard principles which are extensively discussed for example in [HL06]. We need to prove that every object-level type T which may contain the free type variables $\alpha_1, \ldots, \alpha_n$ has a representation as an LF object `T` of type `tp` in an LF context $\texttt{a}_1$`:tp, ...,` $\texttt{a}_n$`:tp`, and vice versa. Here we will highlight only the role of the LF context, since Twelf provides us with the ability to specify the shape of it and check that indeed every LF object of type `tp` will be generated in this context. This is done by the following combination of block and world declarations

```
%block g : block a:tp.
%worlds (g) (tp).
```

The block and world declaration states that the LF context is essentially of the form $\texttt{a}_1$`:tp, ...,` $\texttt{a}_n$`:tp`. This guarantees that every valid LF object of type `tp` is either generated by a constant from the signature or by an assumption in the LF context.

3.2 Encoding subtyping relation

A succinct feature of higher-order abstract syntax encodings is that we do not represent variables as a syntactic category explicitly, but variables are implicit. This will play a important role when we encode the subtyping relation described in the previous section. Recall that the algorithmic description is syntactically defined by cases for each possible type. Moreover there are two rules, sa-ref-tvar and sa-tr-tvar, which are specifically restricted to type variables. A direct immediate encoding seems problematic since trying to distinguish on variables as syntactic entities is inherently against the idea of higher-order abstract syntax! The benefit of HOAS is that it eliminates the need for object level variables, but the price is that we now cannot directly access them and we cannot even state a generic rule for variables.

There are different approaches to solve this problem – for example one could introduce a separate judgment to check if a given object is in fact a type variable. This approach has been followed in the implementation by Ashley-Rollman, Crary, and Harper [MARH]. However, this has also some disadvantages. In particular, we have to prove substitution and narrowing lemmas separately, as well as various lemmas about impossible cases. Here we will take a different approach: Instead of a general variable rule, we will add *rules for reflexivity and transitivity for each type variable.* In other words, every time we introduce a type variable α, we also introduce the corresponding reflexive and transitive properties such as $\alpha \leq \alpha$ and an assumption which says for all U and V, if $\alpha \leq U$ and $U \leq V$ then $\alpha \leq V$. Taking full advantage of the power of our meta-language, we allow not only atomic assumptions such as $\alpha \leq T$ or $\alpha \leq \alpha$, but we also allow more complex assumptions which from a meta-logic point of view can be described using universal quantifiers and implications in the meta-logic. So "for all U and V, if $\alpha \leq U$ and $U \leq V$ then $\alpha \leq V$" is represented as $\Pi U.\Pi V.\alpha \leq U \rightarrow U \leq V \rightarrow \alpha \leq V$. Now we can rewrite our formal algorithmic system given in the previous section.

$$\frac{}{\Gamma \vdash T \leq \text{top}}\ \text{sa-top} \qquad \frac{\alpha \leq T \in \Gamma}{\Gamma \vdash \alpha \leq T}\ \text{sa-hyp} \qquad \frac{\Gamma \vdash T_1 \leq S_1 \quad \Gamma \vdash S_2 \leq T_2}{\Gamma \vdash S_1 \Rightarrow S_2 \leq T_1 \Rightarrow T_2}\ \text{sa-arr}$$

$$\frac{\Gamma \vdash T_1 \leq S_1 \qquad \begin{array}{l}\Gamma,\ tr{:}\Pi U.\Pi V.\alpha \leq U \rightarrow U \leq V \rightarrow \alpha \leq V \\ \qquad\qquad w{:}\alpha \leq T_1,\ \ ref{:}\alpha \leq \alpha \vdash S_2 \leq T_2\end{array}}{\Gamma \vdash \forall \alpha \leq S_1.S_2 \leq \forall \alpha \leq T_1.T_2}\ \text{sa-all}$$

It is worth keeping in mind that in order to use a universally quantified assumption $\Pi U.\Pi V.\alpha \leq U \rightarrow U \leq V \rightarrow \alpha \leq V$ we first must instantiate U and V appropriately.

We are now ready to show the encoding of these subtyping rules in LF. We first define the constant `sub` which describes the subtyping relation. Next, we represent each inference rule in the object-language as a clause consisting of nested universal quantifiers and implications. Upper-case letters denote logic variables which are implicitly bound by a Π-quantifier at the outside.

```
sub : tp -> tp -> type.
sa_top : sub S top.
sa_arr : sub S2 T2 -> sub T1 S1
          -> sub (arr S1 S2) (arr T1 T2) .
sa_all : (Πa:tp.
           (ΠU.ΠV.sub U V ->  sub a U -> sub a V) ->
           sub a T1 -> sub a a ->
           sub (S2 a) (T2 a))
        -> sub T1 S1
        -> sub (all S1 (λv.(S2 v))) (all T1 (λv.(T2 v))).

%block l0 :
some {T:tp}
block {a:tp} {tr:ΠU.ΠV. sub U V -> sub a U -> sub a V}
      {w: sub a T}{ref: sub a a}.
%worlds  (l0) (sub _ _).
```

Using a higher-order logic programming interpretation, we can read the clause `sa_arr` as follows: To prove the goal `sub (S1 => S2) (T1 => T2)`, we must prove `sub T1 S1` and then `sub S2 T2`. Similarly we can read the clause `sa_all`: To prove `sub (all S1 (λv.(S2 v))) (all T1 (λv.(T2 v)))`, we need to prove first `sub T1 S1`, and then assuming `tr: ΠU.ΠV. sub U V -> sub a U -> sub a V`, `w:sub a T1`, and `ref:sub a a`, prove that `sub (S2 a) (T2 a)` is true where `a` is a new parameter of type `tp`.

LF objects belonging to the type `sub T S` are derivations showing that indeed `T` is a subtype of `S`. All derivations for `sub T S` (i.e. all LF objects belonging to the type `sub T S`) are generated either by a constant in the signature or by an element in the LF context which assert the following four assumptions:`a:tp`, `tr:ΠU.ΠV.sub U V->sub a U->sub a V`, `w:sub a T`, `ref:sub a a`. The block declaration describes the shape of the LF context, and the world declaration verifies that these are the only LF contexts constructed. Next we show the derivation for the example considered earlier encoded as an LF object:

```
ex1 : sub (all top (λa. all a (λ b.arr a b)))
          (all top (λa. all a (λ b.arr a a))) =
sa_all
  (λa:tp. λtr_a:ΠU:tp.ΠV:tp.sub U V -> sub a U -> sub a V.
   λw:sub a top. λref_a:sub a a.
    sa_all
     (λb:tp.λtr_b:ΠU:tp.Π.V:tp.sub U V -> sub b U -> sub b V
      λv:sub b a.λref_b:sub b b. sa_arr v ref_a) ref_a)
  sa_top.
```

3.3 Encoding admissibility of transitivity

Finally, we can turn our attention to encoding the proof for transitivity.

Theorem 1 (Admissibility of transitivity).
If $\Gamma \vdash S \leq Q$ and $\Gamma \vdash Q \leq T$ then $\Gamma \vdash S \leq T$.

Typically, this requires the proof of the following narrowing lemma which needs to be proven simultaneously.

Lemma 1 (Narrowing).
If $\Gamma, w{:}\alpha \leq Q \vdash J$ and $\Gamma \vdash T \leq Q$ then $\Gamma, v{:}\alpha \leq T \vdash J$.

However, in our setting, since we localized all rules involving type variables, we will see this is unnecessary. Following the general LF philosophy, we now encode the proof that transitivity is admissible as a relation between derivations. The implementation of the proof is shown in Figure 1.

```
% Transitivity proof:
trans: ΠQ:tp.sub S Q -> sub Q T -> sub S T -> type.
%mode trans +Q +D +E -F.

tr-top: trans T D sa_top sa_top.

tr-ar: trans Q2 D2 E2 F2 ->
       trans Q1 E1 D1 F1 ->
       trans (arr Q1 Q2) (sa_arr D2 D1) (sa_arr E2 E1) (sa_arr F2 F1).

tr-all:
(Πa:tpΠv:sub a T1.Πref: sub a a
 Πtr:ΠU.ΠV.sub a U -> sub U V -> sub a V.
% Assumption v
(ΠP:tp.ΠE:sub T1 P.trans T1 v E (tr T1 P v E)) ->
(% Reflexivity
ΠP:tp.ΠE:sub a P. trans a ref E E) ->
(% Transitivity
ΠT':tp.ΠS':tp.ΠU':tp.ΠD':sub a U'.ΠD'':sub U' T'.
ΠE':sub T' S'.ΠF':sub U' S'.
trans T' D'' E' F' ->
trans T' (tr U' T' D' D'') E' (tr U' S' D' F')) ->
% i.h. on D2' and E2
trans (Q2 a) (D2 a (tr _ _ v E1) ref tr) (E2 a v ref tr) (F2 a v ref tr)) ->
% i.h. on E1 and D1
trans Q1 E1 D1 F1 ->
trans (all Q1 Q2) (sa_all D2 D1) (sa_all E2 E1) (sa_all F2 F1).
```

Fig. 1. Admissibility of transitivity implemented in Twelf

We begin by defining a type family, which may be thought of as a meta-predicate corresponding to our lemma that transitivity is admissible as follows:

```
trans: ΠQ.sub S Q -> sub Q T -> sub S T -> type.
%mode +P +D +E -F.
```

Twelf binds implicitly variables `S` and `T` at the outside via Π-quantifier and reconstructs their appropriate types. We will however quantify explicitly over the variable `Q`, since our induction will proceed on the structure of the type `Q` and the first derivation `sub S Q`. The meta-predicate `trans` specifies how to translate the derivation `D:sub S Q` and `E:sub Q T` to a derivation `F:sub S T`. In other words, given `D:sub S Q` and `E:sub Q T` we aim to construct a derivation `F:sub S T`. This translation must be total, i.e. it must be defined on all possible inputs. While the theoretical foundation underlying the logical framework guarantees that only valid derivations are constructed, we must verify separately that all cases are covered and all appeals to the induction hypothesis are valid, relying on external checkers such as mode, coverage, termination and totality. The mode declaration specifies that the first two derivations together with the type `Q` are viewed as inputs. The last derivation `sub S T` is viewed as an output. To verify totality we need to show that every appeal to the induction hypothesis is valid. The induction will proceed simultaneously on the type `Q` and the derivation `D = sub S Q`, and we can apply the induction hypothesis if either `Q` is smaller or if `Q` stays the same, the derivation `D` is decreasing. Because of this we explicitly reason about the the type `Q` and we explicitly quantify over it at the beginning. Each case in the proof will correspond to a clause in our meta-program. Next, we will consider each case in the proof individually.

Case First, we consider a generic case for top.

$$\mathcal{D} = \frac{}{\Gamma \vdash S \leq Q} \quad \text{and } \mathcal{E} = \frac{}{\Gamma \vdash Q \leq \text{top}}\,\text{sa-top}$$

Clearly, there exists a derivation $\mathcal{F} : \Gamma \vdash S \leq \text{top}$ by using the rule sa-top. This is represented as a clause in our meta-program as follows:

```
tr_top : trans Q D (sa_top) (sa_top).
```

Case $$\mathcal{D} = \frac{\begin{array}{c}\mathcal{D}_1\\ \Gamma \vdash Q_1 \leq S_1\end{array} \qquad \begin{array}{c}\mathcal{D}_2\\ \Gamma \vdash S_2 \leq Q_2\end{array}}{\Gamma \vdash S_1 \Rightarrow S_2 \leq Q_1 \Rightarrow Q_2}\,\text{sa-arr}$$

This derivation is represented as `sa_arr D2 D1`.

$$\mathcal{E} = \frac{\begin{array}{c}\mathcal{E}_1\\ \Gamma \vdash T_1 \leq Q_1\end{array} \qquad \begin{array}{c}\mathcal{E}_2\\ \Gamma \vdash Q_2 \leq T_2\end{array}}{\Gamma \vdash Q_1 \Rightarrow Q_2 \leq T_1 \Rightarrow T_2}\,\text{sa-arr}$$

This derivation is represented as `sa_arr E2 E1`.

$\mathcal{F}_1 : \Gamma \vdash T_1 \leq S_1$ by i.h. on $\mathcal{E}_1$ and $\mathcal{D}_1$
$\mathcal{F}_2 : \Gamma \vdash S_2 \leq T_2$ by i.h. on $\mathcal{D}_2$ and $\mathcal{E}_2$
$\mathcal{F} : \Gamma \vdash (S_1 \Rightarrow S_2) \leq (T_1 \Rightarrow T_2)$ by rule sa-arr using $\mathcal{F}_1$ and $\mathcal{F}_2$

The appeal to the induction hypothesis corresponds to recursively applying the meta-predicate `trans` to sub-derivations `E1` and `D1` and `D2` and `E2` respectively. This case corresponds to the following case in the implementation of the proof.

```
tr-ar:
trans Q2 D2 E2 F2 ->
trans Q1 E1 D1 F1 ->
trans (arr Q1 Q2) (sa_arr D2 D1) (sa_arr E2 E1) (sa_arr F2 F1).
```

The most interesting case is the case for sa-all.

Case In the case for sa-all we have the following:

$$\mathcal{D} = \Gamma \vdash \forall\alpha \leq S_1.S_2 \leq \forall\alpha \leq Q_1.Q_2 \qquad \mathcal{E} = \Gamma \vdash \forall\alpha \leq Q_1.Q_2 \leq \forall\alpha \leq T_1.T_2$$

By inversion on $\mathcal{D}$ we get:

$\mathcal{D}_1 : \Gamma \vdash Q_1 \leq S_1$ and
$\mathcal{D}_2 : \Gamma,\ tr{:}\Pi U.\Pi V.\alpha \leq U \rightarrow U \leq V \rightarrow \alpha \leq V,\ w{:}\alpha \leq Q_1,\ ref{:}\alpha \leq \alpha \vdash S_2 \leq Q_2.$

We note that $\mathcal{D}_2$ is parametric in α and hypothetical in the three assumptions tr, w, and ref. This parametric and hypothetical derivation is encoded as a function in Twelf. The LF object describing the derivation $\mathcal{D}$ is described by the following term:

```
(sa_all (λa:tp.λtr:ΠU.ΠV.sub U V -> sub a U -> sub a V.
           λw:sub a Q1.λref:sub a a.D2 a tr w ref)
         D1)
```

By inversion on $\mathcal{E}$ we get:

$\mathcal{E}_1 : \Gamma \vdash T_1 \leq Q_1$ and
$\mathcal{E}_2 : \Gamma,\ tr{:}\Pi U.\Pi V.\alpha \leq U \rightarrow U \leq V \rightarrow \alpha \leq V,\ v{:}\alpha \leq T_1,\ ref{:}\alpha \leq \alpha \vdash Q_2 \leq T_2.$

This is described by the derivation

```
(sa_all (λa:tp.λtr:ΠU.ΠV.sub U V -> sub a U -> sub a V.
           λv:sub a T1.λref:sub a a. E2 a tr v ref)
         E1)
```

To prove that there exists a derivation for $\mathcal{F} = \Gamma \vdash \forall\alpha \leq S_1.S_2 \leq \forall\alpha \leq T_1.T_2$ we first appeal to the induction hypothesis on $\mathcal{E}_1$ and $\mathcal{D}_1$ to obtain a derivation

$$\mathcal{F}_1 = \Gamma \vdash T_1 \leq S_1$$

This can be represented in Twelf as `trans Q1 E1 D1 F1`. If we now could derive some derivation

$$\mathcal{F}_2 : \Gamma,\ tr{:}\forall U \forall T.\alpha \leq U \to U \leq T \to \alpha \leq T,\ v{:}\alpha \leq T_1,\ ref{:}\alpha \leq \alpha \vdash S_2 \leq T_2$$

then we could simply assemble a derivation $\mathcal{F}$ using the `sa-all` rule together with $\mathcal{F}_1$ and $\mathcal{F}_2$. However applying the induction hypothesis on $\mathcal{D}_2$ and $\mathcal{E}_2$ to obtain such a $\mathcal{F}_2$ will not work because $\mathcal{D}_2$ depends on the assumption $w{:}a \leq Q_1$ while $\mathcal{E}_2$ depends on the assumption $v{:}\alpha \leq T_1$. We need to first create a derivation $\mathcal{D}_2'$ which depends on $v{:}\alpha \leq T_1$! How can this be done? – This is the place where typically we must appeal to the narrowing lemma which allows us to replace the assumption $w{:}a \leq Q_1$ with the assumption $v{:}a \leq T_1$ since $T_1 \leq Q_1$. Here, we will take however a different view.

Recall $\mathcal{D}_2$ is a parametric and hypothetical derivation which can be viewed as a function which expects as inputs $tr{:}\Pi U.\Pi V.\alpha \leq U \to U \leq V \to \alpha \leq V$, an object of type $\alpha \leq Q_1$, and $ref{:}\alpha \leq \alpha$. In other words, we need to turn a function which expects an object of type $\alpha \leq Q_1$ into a function which expects an object v of type $\alpha \leq T_1$. The idea is to substitute a derivation $\mathcal{W}$ which depends on $v{:}\alpha \leq T_1$ for any use of $w{:}\alpha \leq Q_1$. We can construct such a derivation $\mathcal{W}$ as follows:

$$\mathcal{W}^{\alpha,v} = \dfrac{\dfrac{}{\alpha \leq T_1}\,v \qquad \dfrac{\mathcal{E}_1}{T_1 \leq Q_1}}{\alpha \leq Q_1}\,tr$$

Note that the derivation $\mathcal{W}$ is parametric in α and hypothetical in $v{:}\alpha \leq T_1$. By substituting $\mathcal{W}$ for any use of w in the hypothetical derivation $\mathcal{D}_2$, we obtain a derivation ${\mathcal{D}_2'}^{\alpha,v}$ which is parametric in α and hypothetical in $v{:}\alpha \leq T_1$. Now we are able to appeal to the i.h. on ${\mathcal{D}_2'}^{\alpha,v}$ and ${E_2}^{\alpha,v}$, and obtain

$$\mathcal{F}_2 : \Gamma,\ tr{:}\forall U \forall T.\alpha \leq U \to U \leq T \to \alpha \leq T,\ v{:}\alpha \leq T_1,\ ref{:}\alpha \leq \alpha \vdash S_2 \leq T_2$$

This derivation $\mathcal{W}$ can be represented as an object `(tr T1 Q1 v E1)` in LF. To obtain a derivation $\mathcal{D}_2'$ from $\mathcal{D}_2$ we simply apply $\mathcal{D}_2$ to `a`, `(tr T1 Q1 v E1)`, `ref`, and `tr`. This will be encoded by the following line in the proof

```
trans (Q2 a)
 (D2 a tr (tr T1 Q1 v E1) ref) (E2 a tr v ref) (F2 a tr v ref))
```

We are not done yet with the case for `sa-all`. Since our context keeps track of local rules for reflexivity and transitivity, our proof must also account for these cases locally. In particular the appeal to the i.h. on $\mathcal{D}_2'$ and E_2 takes place in a context where we have the assumptions `tr`, `v`, and `ref`. These cases will be part of the encoding of of the case for `sa_all`. We will consider each of the local assumptions next and show the encoding for each of them, before we assemble all the pieces into the encoding for the case of sa-all. We emphasize that these cases correspond to the local assumptions by writing the name of the rule in question in type-writer font. Note that these proofs about local assumptions take place in the context where we have in fact the local assumptions `tr`, `v`, and `ref`. Hence, we are free to use some of these assumptions in our proof.

Case Given $\mathcal{D} = \dfrac{}{\Gamma \vdash \alpha \leq T_1}$ v and $\mathcal{E} = \Gamma \vdash T_1 \leq P$

we must construct a derivation $\mathcal{F} = \Gamma \vdash \alpha \leq P$.

First, we note that we must prove this case for any type P and for any derivation $\mathcal{E}{:}\Gamma \vdash T_1 \leq P$, if we have a derivation $\mathcal{D}{:}\Gamma \vdash \alpha \leq T_1$ and a derivation $\mathcal{E}$ we can construct a derivation $\Gamma \vdash \alpha \leq P$. How can this be achieved? This is done by instantiating the transitivity rule `tr` with T_1 and P and using $\mathcal{D}$ and $\mathcal{E}$ to fill in the premises. Since all proofs are done in an LF context where we have the assumption `tr` denoting the transitivity rule for type variables, this is feasible. Therefore this case will be represented as

```
(ΠP:tp.Π.E:sub T1 P.trans T1 v E (tr T1 P v E))
```

We explicitly quantify universally over `P` and `E` to emphasize the fact that this translation holds universally.

Case Given $\mathcal{D} = \dfrac{}{\Gamma \vdash \alpha \leq \alpha}$ `ref` and $\mathcal{E} = \Gamma \vdash \alpha \leq P$,

we must construct a derivation $\mathcal{F} = \Gamma \vdash \alpha \leq P$, but this is simply achieved by providing $\mathcal{E}$. Again, this must be proven for all types P and all derivations $\mathcal{E}$, and we explicitly quantify universally over `P` and `E` to emphasize the fact that this translation holds universally. Therefore this case is represented as :

```
(ΠP:tp.ΠE:sub a P.trans a ref E E)
```

Case Given $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \\ \Gamma \vdash \alpha \leq U'\end{array} \qquad \begin{array}{c}\mathcal{D}'' \\ \Gamma \vdash U' \leq T'\end{array}}{\Gamma \vdash \alpha \leq T'}$ `tr` and $\mathcal{E}' = \Gamma \vdash T' \leq S'$

we must show that there exists a derivation $\mathcal{F} = \Gamma \vdash \alpha \leq S'$.

$\mathcal{F}' : \Gamma \vdash U' \leq S'$ by i.h. on $\mathcal{D}''$ and $\mathcal{E}'$
$\mathcal{F} : \Gamma \vdash \alpha \leq S'$ by `tr` rule using $\mathcal{D}'$ and $\mathcal{F}'$.

This case will be represented as:

```
ΠT':tp.ΠS':tp.ΠU':tp.
ΠD':sub a U'.ΠD'': sub U' T'.ΠE':sub T' S'.ΠF': sub U' S'.
trans T' D'' E' F' ->           % appeal to i.h. on D'' and E'
trans T' (tr U' T' D' D'') E' (tr U' S' D' F'))
```

Again we explicitly quantify over the type variables `T'`, `S'`, `U'`, and the derivations involved in this case to emphasize that we prove this statement for all types `T'`, `S'`, and `U'`, and all derivations `D'`, `D''`, and `E'`.

Finally, we can assemble all the pieces to represent the polymorphic case which is described in Figure 1.

Checking modes, termination, coverage To verify that the encoding indeed constitutes a proof, meta-theoretic properties such as coverage and termination need to be established separetely. Coverage guarantees that all cases have been covered, and termination guarantees that all appeals to the induction hypothesis are valid. In Twelf, the user must declare specifically which checkers are to be used. This is shown in Figure 2. The mode declaration `%mode trans +Q +D +E -F` specifies inputs and outputs of the defined meta-predicate, and directly corresponds to what we assume in the statement of transitivity and what we need to prove. The mode checker [RP96] verifies that all inputs are known when the predicate is called and all output arguments are known after successful execution of the predicate. To guide the coverage checker [SP03], we must declare the context we prove the theorem in. This block and world declaration must include all the dynamic parameters and dynamic hypothesis. Then the coverage declaration `%covers trans +Q +D +E -F` verifies that we have covered all cases for the inputs `Q` and `D` in the proof for `trans`, i.e. either there is a case for objects generated by the signature or there is a case which arises due to our dynamic parameters and assumptions. The declaration `%terminates {Q D} (trans Q D E F).` verifies that in all appeals to the induction hypothesis either the type `Q` was decreasing or if `Q` stayed the same the derivation `D` decreased in size [Pie05b]. Finally, the totality declaration verifies that our proof is total relying on mode, coverage and termination[1].

```
%block l :
some  {T1:tp}
block {a:tp}
      {tr:ΠU:tp.ΠT:tp.sub U T -> sub a U -> sub a T}
      {w:sub a T1}
      {ref:sub a a}
      {tr-w  :ΠU:tp.ΠE:sub T1 U.trans T1 w E (tr T1 U E w)}
      {tr-ref:ΠU:tp.ΠE:sub a U.trans a ref E E}
      {tr-tr :(ΠT':tp.ΠS':tp.ΠU':tp.
               ΠD':sub a U'.ΠD'':sub U' T'.
               ΠE':sub T' S'.ΠF':sub U' S'.
               trans T' D'' E' F' ->
               trans T' (tr U' T' D'' D') E' (tr U' S' F' D'))}.
%worlds (l)  (trans Q D E F).
%covers trans +Q +D +E -F.
%terminates {Q D} (trans Q D E F).
%total {Q D} (trans Q D E F).
```

Fig. 2. Checking totality of the `trans` encoding

[1] Technically, we can only specify total, but we found it cleaner to specify the declarations corresponding to the different parts of the totality checker separately.

Remark 1 In addition to the proof for admissibility of transitivity, we have also encoded the proof that reflexivity is admissible, and showed that the algorithm described here is a correct implementation of the declarative description of bounded subtype polymorphism where we have general reflexivity and transitivity rules. To use the admissibility lemmas, we were required to weaken the worlds of some of the lemmas such that they all shared a common world. This is an artifact of Twelf's world checker. The full implementation is available at `http://www.cs.mcgill.ca/~bpientka/code/pearl` and is an interesting example of higher-order judgments in the logical framework and their power.

Remark 2 In addition to reasoning about a formal specification it is interesting to investigate whether we can execute and test our specification. This is important since it allows the developer to generate sample behavior and animate a given specification. In our view, whether a given specification can be directly executed is to a large extent determined by the underlying operational semantics used. Using the types-as-formulas paradigm, we can assign types a higher-order logic programming interpretation [Pfe91] which allows us to execute specification. The encoding provided for subtyping is, maybe surprisingly, executable, although at first sight, it may seem we did not gain an algorithm since there is some non-deterministic left; however, this non-determinism is very limited to the specialized transitivity rules. A simple loop detection mechanism as provided by the tabled higher-order logic programming engine [Pie02a,Pie05a] in Twelf is able to handle the remaining non-determinism in the transitivity rule.

4 Conclusion

We have presented a higher-order encoding of subtyping algorithm for bounded polymorphism, one of the challenge problems from the POPLmark challenge. We not only use higher-order abstract syntax for implementing the types but exploit the full power of parametric and higher-order judgments. As a key benefit we get the narrowing lemma for free. This makes the encoding of the proof that general transitivity is admissible is extremely compact. Our reasoning is purely structural and does not require any extra size argument. Finally, unlike other encodings based on higher-order abstract syntax ours does not require proofs showing that some cases are impossible. It is remarkable that the proof for admissibility of transitivity fits on less than a page, and it is by far the shortest one so far submitted. We believe our development illustrates nicely how the problem formulation does wonders for the subsequent meta-theoretic development.

As mentioned, our case study is inspired by one of the POPLmark challenge problems, and there have been several solutions submitted exploring a wide range of possible variable encodings. We can categorize the solutions into several categories: encoding variables via de Bruijn indices, encoding variables via concrete names, encoding variables in a nameless approach, and giving an encoding using nominal types. All these solutions must prove the narrowing lemma and often various other properties about variables separately. Closest to our solution is the one by Ashley-Rollman, Crary, and Harper within the logical framework Twelf. While their proposal uses higher-order abstract syntax for encoding bound type variables, it does not exploit the full power of higher-order functions to implement hypothetical judgments. Our solution differs from their solution in that we push the power of hypothetical and parametric judgments and it demonstrates what is possible if we design the specification following the LF methodology.

We believe this case study provides interesting insights in how higher-order encodings can yield extremely compact implementations. The key behind this encoding is to think about derivations as higher-order functions, and compose higher-order functions to create new derivation. Due to the compactness of our encoding we also believe it is easier to scale. For example, to add new type constructors for cross products we simply add the corresponding cases in the subtyping algorithm and in the transitivity proof. However, we do not need to prove any additional lemmas.

In the future, it would be interesting to investigate whether our approach can be replicated in other systems which support higher-order abstract syntax and hypothetical and parametric judgments. For example, Momigliano and Ambler [MA03] propose an extension to Hybrid which supports meta-reasoning with higher-order abstract syntax in Isabelle HOL. In Coq, Felty has developed a two-level meta-logic to facilitate hypothetical and parametric reasoning [Fel02]. Another meta-logic capable of replicating our proof idea is $FO\lambda^{\nabla\mathbb{N}}$ by McDowell and Miller [MM02]. Replicating the meta-theoretic development in this paper in other systems would provide valuable insights into how well the methodology scales to other systems. It also would allow a direct comparison between these systems concerning their philosophy, automatic proof support, and complexity of encoding.

Acknowledgment

I would like to thank Chad E. Brown, Stefan Monnier, David Xi Li, and Amy Felty for listening and commenting on the implementation.

References

[ABF$^+$05] B. Aydemir, A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In Joe Hurd and Thomas F. Melham, editors, *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs), Oxford, UK, August 22-25*, volume 3603 of *Lecture Notes in Computer Science(LNCS)*, pages 50–65. Springer, 2005.

[Fel02] Amy P. Felty. Two-level meta-reasoning in Coq. In *Fifteenth International Conference on Theorem Proving in Higher-Order Logics*, pages 198–213. Springer-Verlag Lecture Notes in Computer Science, August 2002.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[HL06] Robert Harper and Daniel Licata. Mechanizing metatheory in a logical framework. (Submitted for publication.), October 2006.

[MA03] Alberto Momigliano and Simon J. Ambler. Multi-level meta-reasoning with higher-order abstract syntax. In *Sixth International Conference on Foundations of Software Science and Computational Structures*, pages 375–391. Springer-Verlag Lecture Notes in Computer Science, April 2003.

[MARH] Karl Crary and Michael Ashley-Rollman and Robert Harper. Twelf solution to POPLmark challenge. electronically available at `http://fling-l.seas.upenn.edu/~plclub/cgi-bin/poplmark/`.

[MM02] Raymond C. McDowell and Dale A. Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Transactions on Computational Logic*, 3(1):80–136, 2002.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Pfe97] Frank Pfenning. Computation and deduction, 1997.

[Pie02a] Brigitte Pientka. A proof-theoretic foundation for tabled higher-order logic programming. In P. Stuckey, editor, *18th International Conference on Logic Programming, Copenhagen, Denmark*, Lecture Notes in Computer Science (LNCS), 2401, pages 271 –286. Springer-Verlag, 2002.

[Pie02b] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[Pie05a] Brigitte Pientka. Tabling for higher-order logic programming. In Robert Nieuwenhuis, editor, *20th International Conference on Automated Deduction (CADE), Talinn, Estonia*, volume 3632 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2005.

[Pie05b] Brigitte Pientka. Verifying termination and reduction properties about higher-order logic programs. *Journal of Automated Reasoning*, 34(2):179–207, 2005.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag Lecture Notes in Artificial Intelligence (LNAI) 1632.

[RP96] Ekkehard Rohwedder and Frank Pfenning. Mode and termination checking for higher-order logic programs. In Hanne Riis Nielson, editor, *Proceedings of the European Symposium on Programming*, pages 296–310, Linköping, Sweden, April 1996. Springer-Verlag Lecture Notes in Computer Science (LNCS) 1058.

[SP03] Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, pages 120–135, Rome, Italy, September 2003. Springer-Verlag LNCS 2758.