Proof Reuse

Antoine Gaulin

June 12, 2022

It is common in the literature to reuse proofs of previously established results in order to derive new theorems. A common pattern in papers is to start from a well understood language (often System F, LF, or the Calculus of Construction), add a new construct to it (e.g. subtyping, inductive types, etc.), and then show that the desirable properties of the original system are preserved. Most proofs (at least for the basic properties) are by induction on the structure of the hypothesized derivation. To conclude that the properties hold in the extension, it is then clearly sufficient to consider only the cases relevant to the new construct. However, in mechanization, one would need to work through all the previously established cases once more. This task is tedious and unnecessary.

We investigate a few ways to simplify the development of mechanized proofs, the key idea being to reuse proofs when possible. The aim is to start from the type system of Beluga [6, 7] and look at a few extensions that allow various forms of proof reuse. Through this process, we can also fix a major problem with how contexts are represented in Beluga, namely the inability to recover premisses needed for the formation of assumptions.

The first direction is to extend the data-level type theory (i.e. the logical framework LF [2]) with refinements, thus allowing a restricted form of subtyping to the language. This can then be lifted to context schemas, and then to the computation-level (i.e. the dependent contextual modal type theory [5]) in a mostly straightforward way. The main idea behind refinements is to "separate" a type into *sorts*. While types express syntactic properties of terms, sorts express semantic properties. They can therefore be used to enforce various properties on terms, while preserving type uniqueness. In this case, we obtain a notion of subsorting rather than subtyping. Ultimately, refinements allow a very limited form of proof reuse, and their usefulness is more in simplifying proofs.

The second direction is to add constructor subtyping [8, 1], which would be more accurately called *supertyping*. This idea is simple : if a type B has all the constuctors of another type A and possibly more, then B can be viewed as a supertype of A. In this setting, we get a notion of co-inheritance, similar to the inheritance mechanism found in object oriented programming.

The third direction is to add ornaments [4]. Here, we obtain systematic ways to enhance a type and/or its constructors with additional dependencies, as well as a lifting mechanism to lift proofs on a type to its ornamented type. Combining this with constructor subtyping, we should be able to present incremental development of languages and of their meta-theory, which would be closer to what is found in the literature.

1 A refinement type system for Beluga

We start with refinements because it is the most invasive change to the language. This is due to the fact that we now want to assign both sorts and types to terms, which is done in a single judgment. This change is also present at the level of types, which are classified by both classes and kinds. Thus, almost

$$
\begin{array}{rrcl}
\text{Signatures} & \Sigma & ::= & \cdot \mid \Sigma, D \\
\text{Declarations} & D & ::= & \mathbf{s} \sqsubset \mathbf{a}{:}L \sqsubset K \mid \mathbf{c}{::}S \sqsubset A \mid \mathbf{s}_1 \leq \mathbf{s}_2 \sqsubset \mathbf{a} \mid \mathbf{w}{:}W \mid \boldsymbol{\xi}{:}\Xi \\
\text{Meta-contexts} & \Delta & ::= & \cdot \mid \Delta, u{::}S[\Psi] \sqsubset A[\Psi] \mid \Delta, p{::}S[\Psi] \sqsubset A[\Psi] \mid \Delta, s{:}\Psi_1[\Psi_2] \\
\text{Schema contexts} & \Omega & ::= & \cdot \mid \Omega, \psi{:}\Xi \\
\\
\text{Kinds} & K & ::= & \texttt{Type} \mid \texttt{Rec} \mid \Pi x{:}A.K \\
\text{Classes} & L & ::= & \texttt{Sort} \mid \Pi x{::}S \sqsubset A.L \mid \top \mid L_1 \wedge L_2 \\
\\
\text{Atomic type families} & P & ::= & \mathbf{a} \mid P\ \vec{M} \\
\text{Canonical type families} & A & ::= & P \mid \Pi x{:}A_1.A_2 \\
\text{Atomic sort families} & Q & ::= & \mathbf{s} \mid Q\ \vec{M} \\
\text{Canonical sort families} & S & ::= & Q \mid \Pi x{:}S_1 \sqsubset A_1.S_2 \mid \top \mid S_1 \wedge S_2 \\
\\
\text{Worlds} & W & ::= & \langle \ell_i{::}S_i \sqsubset B_i \rangle_n \mid \Pi x{::}S \sqsubset A.W \\
\text{Schema} & \Xi & ::= & \varepsilon \mid \Xi + W \\
\\
\text{Heads} & H & ::= & \mathbf{c} \mid x \mid \texttt{proj}\ k\ x \mid \texttt{Clo}(x, s[\sigma]) \mid \#p[\sigma] \mid \texttt{proj}\ k\ \#p \\
\text{Spines} & \vec{M} & ::= & \varepsilon \mid N; \vec{M} \\
\text{Normal terms} & N & ::= & R \mid \lambda x.N \\
\text{Neutral terms} & R & ::= & H\ \vec{M} \mid u[\sigma] \\
\\
\text{LF contexts} & \Psi & ::= & \cdot \mid \Psi, x{::}S \sqsubset A \mid \Psi, x{:}(W\vec{M}) \\
\text{Substitutions} & \sigma & ::= & \cdot \mid \texttt{wk}_\psi \mid s[\sigma] \mid \sigma; N
\end{array}
$$

Figure 1: Syntax of data-level

every inference rule must be adapted, although they keep the same flavor. The core theory presented in this section is based on [6] and [7], and the addition of refinements closely follows what is shown in [3].

1.1 Data-level

The data-level of Beluga includes the usual terms, types, and kinds, but also contexts and substitutions. We add to the type level a notion of sorts, and to the kind level a similar notion of classes. Contexts are classified using a notion of schema, which, in our extension, are built out of world declarations. A world is a record of assumptions satisfying certain properties. In order to ensure well-formedness of worlds and schemata, we introduce an additional base kind `Rec` for records.

Note. It may be necessary to add a corresponding sort `RecSort` to characterize refinement records. This would also allow a subsorting mechanism on records, which could simplify the sub-worlds and sub-schema relationships.

1.1.1 Syntax

The updated syntax of the language is given in Figure 1. Most of the syntax that was already present in Beluga remains unchanged (kinds, types, terms, and substitution, to be precise). The main differences are in the contexts and signatures, where assumptions are endowed with a sort as well as a type. Most importantly, LF contexts have an additional construct to associate variables to a given world, instead of just a type. Finally, declarations are extended with subsorting and worlds.

1.1.2 Judgments

As previously mentionned, most of the judgments take a slightly different form in the presence of refinements. Let's first look at a quick summary of the judgments :

$\vdash \Sigma$ `sig`	Signature well-formedness
$\vdash_\Sigma \Omega$ `sctx`	Schema context well-formedness
$\Omega \vdash_\Sigma \Delta$ `mctx`	Meta-context well-formedness
$\Omega; \Delta \vdash_\Sigma \Psi$ `ctx`	LF context well-formedness
$\Omega; \Delta; \Psi \vdash_\Sigma L \sqsubset K$	Class L refines kind K
$\Omega; \Delta; \Psi \vdash_\Sigma Q \sqsubset P \Rightarrow L$	Atomic sort Q synthesizes atomic type P and class L
$\Omega; \Delta; \Psi \vdash_\Sigma S \sqsubset A \Leftarrow \mathtt{Sort}$	Sort S refines type A
$\Omega; \Delta; \Psi \vdash_\Sigma N \Leftarrow S \sqsubset A$	Normal term N checks against sort S refining type A
$\Omega; \Delta; \Psi \vdash_\Sigma R \Rightarrow S \sqsubset A$	Neutral term R synthesizes sort S refining type A
$\Omega; \Delta; \Psi \vdash_\Sigma \sigma \Leftarrow \Phi$	Substitution σ checks against LF context Φ
$\Omega; \Delta; \Psi \vdash_\Sigma S_1 \leq S_2 \sqsubset A$	S_1 is a sub-sort of S_2 as refinements of A
$\Omega; \Delta; \Psi \vdash_\Sigma W$ `world`	W is a well-formed world
$\Omega; \Delta; \Psi \vdash_\Sigma \Xi$ `schema`	Ξ is a well-formed context schema
$\Omega; \Delta; \Psi \vdash_\Sigma \Psi : \Xi$	LF context Ψ has schema Ξ
$\Omega; \Delta; \Psi \vdash_\Sigma W_1 \leq W_2$	W_1 is a sub-world of W_2
$\Omega; \Delta; \Psi \vdash_\Sigma \Xi_1 \leq \Xi_2$	Ξ_1 is a sub-schema of Ξ_2

1.2 Computation-level

We can lift the data-level refinements to the computation-level to obtain a restricted notion of refinements where the user does not directly specify any sorts. It could be interesting to have a full blown refinement type system, and it should not complicate matters too much.

2 Constructor subtyping

TBD

3 Ornaments

TBD

References

[1] Barthe, G., and Frade, M. J. Constructor subtyping. In *Programming Languages and Systems* (Berlin, Heidelberg, 1999), S. D. Swierstra, Ed., Springer Berlin Heidelberg, pp. 109–127.

[2] Harper, R., Honsell, F., and Plotkin, G. D. A framework for defining logics. In *Proceedings of the Symposium on Logic in Computer Science (LICS'87), Ithaca, New York, USA, June 22-25, 1987* (1987), IEEE Computer Society, pp. 194–204.

[3] Lovas, W., and Pfenning, F. Refinement types for logical frameworks and their interpretation as proof irrelevance. *Logical Methods in Computer Science 6*, 4 (dec 2010).

[4] McBride, C. Ornamental algebras, algebraic ornaments, 2011.

[5] Nanevski, A., Pfenning, F., and Pientka, B. Contextual modal type theory. *ACM Trans. Comput. Log. 9*, 3 (2008), 23:1–23:49.

[6] Pientka, B. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008* (2008), G. C. Necula and P. Wadler, Eds., ACM, pp. 371–382.

[7] Pientka, B., and Dunfield, J. Programming with proofs and explicit contexts. In *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 15-17, 2008, Valencia, Spain* (2008), S. Antoy and E. Albert, Eds., ACM, pp. 163–173.

[8] Poll, E. Subtyping and inheritance for inductive types. In *Proceedings of TYPES'97 Workshop on Subtyping, inheritance and modular development of proofs, Durham, UK* (September 1997).