OOPSLA 18, Paper #131 Artifact
Step-By-Step Guide

Fabian Muehlboeck, Cornell University
Ross Tate, Cornell University
({fabianm,ross}@cs.cornell.edu)

July 2, 2018

1 About Our Artifact

As explained in Section 7 of our paper, the Coq formalization that we have is of a more general framework, a specialization of which (with a slight caveat, see below) we present in the paper. As such, the Coq proof does not match the paper precisely, but closely.

In the paper, we deal specifically with union and intersection types combined with (mutually recursively defined) user-defined literals. The user only has to define subtyping rules for literal cases, and an *intersector* that converts an intersected list of literals into another type in disjunctive normal form, plus some proofs about them that make our system work.

In the artifact, there is no template: a user is free to define a system of arbitrary types, subtyping rules, and a *preprocessor*, which is a function from types to types (we'll explain below how, and also provide example Coq code with this artifact). The constraints placed on those components correspond closely to the constraints in the paper. There is, however, one caveat, as mentioned in the paper, which we'll explain below.

1.1 The Caveat

In Figure 7 in the paper, we define *Integrated Subtyping* as *integrating* (i.e. `DNF` + applying the intersector to each intersection) the initial type in a subtyping check and the left-hand sides of each recursive call of the literal subtyping rules. The point of this is to make sure that at every recursive step in the subtyping proof, the left-hand side (LHS) satisfies the *integrated* predicate dnf_ϕ. One can omit this integration step in a given rule if one can prove that if the LHS of the conclusion of the rule is already processed, then so are all the LHSs of the recursive calls. It can easily be seen that, in our system, this is true for the union subtyping rules, since unions are the outermost layer in a `DNF` representation of types and the intersected predicate dnf_ϕ holds for a union exactly when it holds for its two constituent parts. However, it is not necessarily true of the LHS intersection subtyping rule:

$$\frac{\tau_i <: \tau}{\tau_1 \cap \tau_2 <: \tau}$$

In the current Coq framework, his means that we are forced to require that the constituent parts of an intersection that counts as intersected are intersected on their own[1] - otherwise we would have to run the intersector on them again, which would potentially just produce an intersection that is a superset of the current one, essentially containing those types that we just dispatched into other branches of our subtyping proof, thus threatening termination. However, this requirement is very strict and makes it hard to formalize some of the Ceylon extensions we discuss in the paper – in particular, one would have to be very careful about the nesting order of types that the intersector generates when generating *principial instantiations* (see Figure 11). For the proofs in the paper, we found a better adjunction to deal with this problem. Intuitively,

[1] We provide an example instantiation of our framework where that is the case, see below and `EXAMPLE.html`, in particular the requirements `intersectedl` and `intersectedr` in `ClassDistribute.v`

we can keep track of the fact that we are recursing through the parts of a larger, "'intersected"' intersection. This is the relaxation of the intersected predicate alluded to in Section 7 of the paper. Work on mechanically formalizing this is ongoing.

1.2 Supported Claims

While not directly encoding them, the presented Coq framework should support all claims made in Sections 3 and 4 of the paper (modulo the above caveat). The table below (the same as in `STARTHERE.html`, where it may be more readable) gives an overview of which parts of those sections correspond to which parts in the formalization, and how:

Concept from Paper	**Corresponding Proof Part**	**Comment**
Literals & Types	`Typ.T` in `Common.v`	Generalized to arbitrary types, all user-provided
User-Defined Subtyping Rules	`Rules.Con, Rules.Req, Rules.Ass` in `Common.v`	
Declarative Subtyping Rules	`Traditional.TransRefl.TRCon` etc. in `Tradition.v`	in our original formalization, we used the term "'Traditional"' instead of "'Declarative"'
Reductive Subtyping Rules	see User-Defined Typing Rules	While Declarative Rules add reflexivity and transitivity, Reductive Rules are all user-supplied in this more general framework
Requirement 1: Syntax-Directedness	`DecidableRules.finite_con` in `Decide.v`	
Requirement 2: Well-Foundedness	`DecidableRules.wf` in `Decide.v`	
Requirement 3: Literal Reflexivity	`DecidableRules.refl` in `Decide.v`	Since everything is a literal/type, this captures all types
Subtyping Rules with Assumptions	`ProofPV.ProofPV` etc. in `Common.v`	
Requirement 4: R-to-D Literal Conversion	`Converter.decidable_traditional_R` in `Convert.v`	
Requirement 5: D-to-R Literal Conversion	`Converter.traditional_decidable_R` in `Convert.v`	
Requirement 6: Literal Transitivity	`DecidableRules.Red` in `Decide.v`	See also `Reduction.Reduction` in `Common.v`
Decidability of Declarative Subtyping Theorem	`Conversion.decidable_traditional` and `Conversion.traditional_decidable` in `Convert.v`	Together with `Decider.decider` in `Decide.v`
Extended Subtyping	`Extension.Con` and `ExtendedRules.ECon` etc. in `Extend.v`	

Intersector/Integrator	`Comonad.i` in `Preprocess.v`	Since everything is a literal/type, there is no distinction between intersector and integrator
Requirement 7: Intersector Completeness	`Equivocator.iextended` in `Equate.v`	
Requirement 8: Intersector Soundness	`Equivocator.unit` in `Equate.v`	
Lemma 1: Integrated Soundness	`Equivalence.ipreprocessing_extended` in `Equate.v`	
Requirement 9: Measure Preservation	`WellFoundedComonad.i_wf` in `Preprocess.v`	Requires full well-foundedness proof instead of measure preservation
Lemma 2: Integrated Decidability	`Decider.decider` in `Preprocess.v`	
Requirement 10: Literal Dereliction	`Comonad.counit` in `Preprocess.v`	
Lemma 3: Dereliction	`Comonad.counit` in `Preprocess.v`	Same as Requirement 10 due to missing type/literal distinction
Intersected Predicate	`Comonad.Preprocessed` in `Preprocess.v`	
Requirement 11: Intersector Integrated	`Comonad.i_Preprocessed` in `Preprocess.v`	
Lemma 4: Integrator Integrated	`Comonad.i_Preprocessed` in `Preprocess.v`	Same as Requirement 11 due to missing type/literal distinction
Requirement 12: Literal Promotion	`Comonad.i_promote_R` in `Preprocess.v`	
Lemma 5: Promotion	`Preprocessing.dpromote'` in `Preprocess.v`	
Lemma 6: Integrated Monotonicity	-	Helper Lemmas, established in various different forms in `Preprocess.v`
Lemma 7: Integrated Assumptions	-	
Lemma 8: Integrated Promotion	`Preprocessing.promote` in `Preprocess.v`	
Lemma 9: Integrated Reflexivity	`Comonad.derelict` and `Comonad.refl` in `Preprocess.v`	
Lemma 10: D-to-I Literal Conversion	`Equivalence.iadmitst` in `Equate.v`	
Lemma 11: Integrated Transitivity	`Preprocessing.itrans` in `Preprocess.v`	
Lemma 12: Integrated Completeness	`Equivalence.extended_ipreprocessing` in `Equate.v`	

Decidability of Extended Subtyping Theorem	`Equivalence.decider` in `Equate.v`

1.3 Claims Not Supported By The Artifact

Anything in Section 5 and after, i.e. constitutionality and specific Ceylon extensions. We do provide a formalization of a type system of generic class types with unions, reasoning about its compositionality, and a disjointness extension akin to the one described in the paper as an example of how to use the framework. However, this formalization is not complete and not meant to support the claims in the corresponding sections.

2 How To Evaluate The Artifact

We documented this Coq formalization in `coqdoc` in many places to both give an overview of how it works and how it corresponds to the claims in the paper. The file `STARTHERE.html` contains the same table of correspondences between the paper and the formalization as above, as well as the signatures of all the files contained in the portion of the artifact that is meant to support anything, and explanatory paragraphs for the important parts of those files. We believe that the largest part of evaluating this artifact will be to go through `STARTHERE.html` and examine the correspondences that we list.

3 How To Use This Framework

In general, the Coq files provide a number of *module types* that need to be instantiated, akin to the various requirements in the paper. One can use the instantiations of those module types to instantiate various modules that the framework provides, akin to the lemmas in the paper, including at the end a subtyping decider for integrated subtyping with proofs of the expected properties.

We provide a documented example of how to do all of that for a type system with generic classes, union, and intersection types, reason about its composability, and providing an extension to reason about disjointness, in `EXAMPLE.html`. The work on this example provided much of the inspiration for the current paper.