Formalizing, Mechanizing, and Verifying Class-Based Refinement Types (Artifact)

Ke Sun
Key Lab of HCST (PKU), MOE, School of Computer Science, Peking University, Beijing, China

Di Wang[1]
Key Lab of HCST (PKU), MOE, School of Computer Science, Peking University, Beijing, China

Sheng Chen
The Center for Advanced Computer Studies, University of Louisiana, Lafayette, LA, USA

Meng Wang
University of Bristol, UK

Dan Hao
Key Lab of HCST (PKU), MOE, School of Computer Science, Peking University, Beijing, China

Abstract

This is the artifact description of an ECOOP paper. A new expressive formalization of class-based refinement types is proposed in the paper. We enrich the formalization by analyzing its meta-theory and algorithmic verification. The meta-theory and algorithmic verification have been mechanized and implemented. We discuss details of the mechanization and implementation in this document.

2012 ACM Subject Classification Theory of computation → Type structures; Software and its engineering → Formal software verification

Keywords and phrases Refinement Types, Program Verification, Object-oriented Programming

Digital Object Identifier 10.4230/DARTS.10.2.22

Funding This work is sponsored by National Natural Science Foundation of China Grant No. 62232001, NSF Grant 1750886, and EPSRC Grant EP/T008911/1.

Acknowledgements We thank the anonymous reviewers for their helpful comments.

Related Article Ke Sun, Di Wang, Sheng Chen, Meng Wang, and Dan Hao, "Formalizing, Mechanizing, and Verifying Class-Based Refinement Types", in 38th European Conference on Object-Oriented Programming (ECOOP 2024), LIPIcs, Vol. 313, pp. 39:1–39:30, 2024.
`https://doi.org/10.4230/LIPIcs.ECOOP.2024.39`

Related Conference 38th European Conference on Object-Oriented Programming (ECOOP 2024), September 16–20, 2024, Vienna, Austria

Evaluation Policy The artifact has been evaluated as described in the ECOOP 2024 Call for Artifacts and the ACM Artifact Review and Badging Policy.

1 Scope

This artifact contains the Coq mechanization of a class-based refinement type calculus (named RFJ), as well as the Python implementation of a type checker of the calculus. We claim that calculus enjoys the metaproperty of type soundness and logical soundness. We also claim that calculus can be implemented as an efficient type checker, which is able to check many interesting example programs.

[1] Corresponding author

2 Content

This artifact contains two main components, the Coq mechanization and the Python Implementation.

Coq Mechanization. The mechanization is in the */home/ecoopsub/Desktop/refpy* folder. We list the structure of the mechanization, which contains about 15K lines of Coq code:

1. Definitions (3K): language definitions as presented in Section 3 of the paper.
2. Lemmas (11K):
   a. Basic Lemmas (5K): miscellaneous lemmas concerning basic operations, semantics, and class/interface definitions (some of which are listed in Section 4.1.
   b. Logical Lemmas (2K): lemmas concerning the logical interpretation (c.f., Section 4.2).
   c. Typing Lemmas (4K): basic, structural, and crucial lemmas of typing (c.f., Section 4.3).
3. Theorems (1K): type and logical soundness theorems (c.f., Sections 4.4 and 4.5).

To ensure that the calculus and meta-theoretical development described in the paper is actually the one mechanized in Coq. We give a correspondence between the calculus definition and lemmas/theorems and the Coq mechanization.
Definitions (Section 3):

- Figure 3 Left (Syntax): Definition/Syntax.v
- Figure 3 Right and Figure 6 (Subtyping and Logics): Definition/SubDenotation.v
- Figure 4 (Auxiliary definitions): Definition/Semantics.v, CTSanity.v
- Figure 5 (Small-step semantics of RFJ): Definition/Semantics.v
- Figure 7 (Typing relations of RFJ): Definition/Typing.v, CTSanity.v

Meta-theory (Section 4):

- Lemma 3: Lemmas/BasicLemmas/LemmasTypeSubstitution.v: tsubBV_invariant and tsubBV_invariant'
- Lemma 4: Lemmas/BasicLemmas/LemmasBigStepSemantics.v: evals_invariant and evals_invariant', EvalsTo_BStepEval and BStepEval_EvalsTo
- Lemma 5: Lemmas/BasicLemmas/LemmasExactness.v: exact_eval
- Lemma 6: Lemmas/BasicLemmas/LemmasExactness.v: exact_type
- Lemma 7: Lemmas/LogicalLemmas/LemmasDenotesTyping.v: typing_denotes
- Lemma 8: Lemmas/LogicalLemmas/LemmasDenotesTyping.v: denotes_typing
- Lemma 9: Lemmas/TypingLemmas/LemmasNarrowing.v: INarrow, narrow_subtyp', narrow_typ'
- Lemma 10: Lemmas/TypingLemmas/LemmasSubstitutionTyping.v: ISub2, subst_subtype2', subst_typ2
- Lemma 11: Lemmas/TypingLemmas/LemmasWeakenTyp.v: IWeak, weaken_subtype', weaken_typ'
- Lemma 12: Lemmas/TypingLemmas/Preservation_Progress.v: progress'
- Lemma 13: Lemmas/TypingLemmas/Preservation_Progress.v: preservation'
- Lemma 14: Lemmas/TypingLemmas/Closing_Substitution.v: closing_substitution
- Corollary 15: Theorems/TypeSoundness.v: type_soundness
- Corollary 16: Theorems/LogicalSoundness.v: logical_soundness
- Theorem 17: Theorems/LogicalSoundness.v: logical_soundness_closed

In the list above, the underlined words are the lemma/theorem names in the code.

Python Implementation. The implementation is in the */home/ecoopsub/Desktop/refpy* folder. The Python implementation of the calculus has a straightforward structure as an AST traverser, which performs basic type checking and SMT constraint collecting. The SMT.py file implements the SMT theory described in Section 5 of the paper.

We give a correspondence between the items listed in Figure 1 with the Python examples in the ref_test folder, with the number in parentheses showing its order in runtests.py.

- pizza: pizza.py (Example #1)
- pizza_visitor: a_little_Java/lession5_objects_pizza.py (Example #6)
- tree: a_little_Java/lession7_overloadingAndgenericVisitor_tree.py (Example #8)
- geometry: a_little_Java/lession9_dataExtensionAndfactory_geometry.py (Example #10)
- list: list.py (Example #11)
- lambda calculus: lambda.py (Example #12)
- stlc: stlc.py (Example #13)

To check all the 14 examples together, use the *runtests.py* script by running "python3 runtests.py" at the root of "refpy" folder, which should print the log (checked constraints, overall time cost) to the terminal. For the negative version of the 14 examples, please use "python3 runtests_negative.py", which should find out several injected errors of the 14 examples.

3 Getting the artifact

The artifact endorsed by the Artifact Evaluation Committee is available free of charge on the Dagstuhl Research Online Publication Server (DROPS). The VM containing the artifact is available at: `https://zenodo.org/records/12683231`. You can also get the newest version of the mechanization at `https://github.com/ksun212/RFJCoq`, and the implementation at `https://github.com/ksun212/Refpy`.

4 Tested platforms

This VM is tested on an Ubuntu 22.04 virtual machine (VirtualBox), the virtual machine is granted 8 virtual cores (the host CPU is AMD 5800H), 8G memory, and 20G virtual disk, no other resource should be needed. We expect the VM to work properly on an AMD-based VirtualBox. The computation it performs is not resource-intensive, although it would run longer (10 seconds) on a low-end AMD CPU.

We expect the mechanization to work properly on any platform with Coq 8.17, and the Implementation on any platform with Python 3.10.

5 License

The artifact is available under license Creative Commons Attribution 4.0 International.

6 MD5 sum of the artifact

d7ac9157fe33c460972f771647ed8d62

7 Size of the artifact

7.6 GiB