A Type-Directed Operational Semantics For a Calculus with a Merge Operator (Artifact)

Xuejing Huang
The University of Hong Kong, China
xjhuang@cs.hku.hk
Bruno C. d. S. Oliveira
The University of Hong Kong, China
bruno@cs.hku.hk

Abstract
Our companion paper proposes a \textit{type-directed operational semantics} (TDOS) for $\lambda_i^\uparrow$: a calculus with intersection types and a merge operator. The artifact contains the specification of $\lambda_i^\uparrow$ and its TDOS, and related Coq code. $\lambda_i^\uparrow$ is formalized using the locally nameless representation with cofinite quantification. The Coq definition and some infrastructure code are generated by Ott and LNgen. $\lambda_i^\uparrow$ is inspired by two closely related calculi by Dunfield (2014) and Oliveira et al. (2016), and a simple variant of it is designed to demonstrate the possibility to match with them without any modification. To relate the two calculi with $\lambda_i^\uparrow$, a sound theorem on semantics and a completeness theorem on typing are proved for each variant. In addition, we extended the bidirectional typing of Oliveira et al.'s $\lambda_i$ calculus, and designed an elaboration from it to $\lambda_i^\uparrow$, to show that many of $\lambda_i^\uparrow$’s explicit annotations can be inferred automatically.

2012 ACM Subject Classification Theory of computation → Type theory; Software and its engineering → Object oriented languages; Software and its engineering → Polymorphism

Keywords and phrases operational semantics, type systems, intersection types

Digital Object Identifier 10.4230/DARTS.6.2.9

Funding This work has been sponsored by Hong Kong Research Grant Council project numbers 17210617 and 17209519.

Acknowledgements The authors wish to thank Bingchen Gong for testing the artifact, and the anonymous artifact reviewers for their comments and suggestions.

https://doi.org/10.4230/LIPIcs.ECOOP.2020.26
Related Conference 34th European Conference on Object-Oriented Programming (ECOOP 2020), November 15–17, 2020, Berlin, Germany (Virtual Conference)

1 Scope
The artifact includes the Coq [6] formalization and the Ott [8] specification of $\lambda_i^\uparrow$. All the lemmas and theorems in the paper are proved in the artifact.

The calculus is defined via the locally nameless representation with cofinite quantification [4]. Most of the Coq definitions and some infrastructure code are generated by the Ott tool and LNgen [2], and relies on the Penn’s metatheory library [1]. We also use the LibTatics.v from the TLC Coq library [5] which defined a collection of general-purpose tactics. The proof structure and strategy is inspired by the formalization of the NeColus calculus [3].
2 Content

The artifact package includes:
- a Docker [7] image which contains the following code with environment set up
- coq directory: the Coq formalization and proofs of $\lambda^i$ with the instructions
- spec directory: the Ott specification of $\lambda^i$ and related calculus
- paper.pdf: the companion paper with appendices

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). You can directly access the Coq code and build from scratch. The offline Docker image in the artifact package offers another option. It includes the code and all dependencies. To use the image, you can execute the following two commands in your machine with Docker installed:

docker import docker_image.tar testtest
docker run -it --user=xsnow --workdir=/home/xsnow testtest /bin/bash -l

The image is also available on the Docker Hub. You can use the following command to get and run the container:

docker run -it sxsnow/ecoop2020

In addition, the latest version of the source code is available at: https://github.com/XSnow/ECOOP2020.

4 Tested platforms

To use the Docker image, any platform supporting Docker and having it installed should be enough.

To build from scratch, Coq is necessary. It is available via opam. Its installation requirements can also be found at https://github.com/coq/coq/wiki/Installation. Penn’s metatheory library needs to be installed as well. The detailed instruction can be found inside the coq directory.

The generated Coq code has been included in the artifact. But if you would like to generate the code, you need to install LNgen (from https://github.com/plclub/lngen), which requires GHC [9], and Ott.

5 License

The artifact is available under the GNU General Public License v3.0.

6 MD5 sum of the artifact

5e97dd3092724a9fd48985f9a529ce84

7 Size of the artifact

0.99 GiB
A  Proof Structure

A.1  In the spec directory

- main_version.ott: the syntax definition and rules for $\lambda_i$
- variant.ott: the syntax definition and rules for the simpler variant of $\lambda_i$
- dunfield.ott: the syntax definition and reduction rules of Dunfield’s calculus.
- icfp.ott: the typing rules of $\lambda_i$ (icfp2016). It use the same syntax definition of expressions as dunfield.ott.

A.2  In the coq/main_version or coq/variant directory

main_version directory contains the definition and proofs of the main calculus. variant directory contains the definition and proofs of the simple variant (discussed in Section 6.1 and the Appendices).

- syntax_ott.v: generated from the Ott files in spec, using the locally nameless encoding. It involves the typing and semantics of $\lambda_i$, the semantics of Dunfield’s calculus, and the typing of $\lambda_i$ (icfp2016).
- rules_inf.v and rules_inf2.v: the LNgen generated code.
- Infrastructure.v: the type systems of the calculi and some lemmas.
- Subtyping_inversion.v: some properties of the subtyping relation.
- Key_properties.v: some necessary lemmas about typed reduction, top-like relation and disjointness.
- Deterministic.v: the proofs of the determinism property.
- Type_Safety.v: the proofs of the type preservation and progress properties.
- dunfield.v: the proofs of the soundness theorem with respect to Dunfield’s calculus.
- icfp.v: the proofs of the completeness theorem with respect to $\lambda_i$ (icfp2016).
- icfp_bidirectional.v: in coq/main_version only. It extends the bidirectional type system of $\lambda_i$ by a fixpoint rule, and uses the same definition of disjointness like our system. In it a different completeness theorem is proved.

B  Correspondence

B.1  Figures and Appendices

- Figure 1 (The non-deterministic small-step semantics of Dunfield’s calculus): DunfieldStep in variant/syntax_ott.v.
- Figure 2 (Subtyping rules of $\lambda_i$ and definition of top-like types): sub and topLike in main_version/syntax_ott.v.
- Figure 3 (Type system of $\lambda_i$): Etyping in main_version/syntax_ott.v.
- Figure 4 (Typed reduction of $\lambda_i$): TypedReduce in main_version/syntax_ott.v.
- (Ordinary types in $\lambda_i$): ord in main_version/syntax_ott.v.
- Figure 5 (Call-by-value reduction of $\lambda_i$): step in main_version/syntax_ott.v.
- Figure 6 (Type erasure for $\lambda_i$ expressions): erase_anno in dunfield.v
- Appendix B (The full rules of the extended Dunfield’s semantics): DunfieldStep in main_version/syntax_ott.v.
- Appendix E (The variant of $\lambda_i$): in variant/syntax_ott.v.
B.2 Definitions, Lemmas and Theorems

- Definition 1 (Disjoint types): disjointSpec in syntax_ott.v.
- Definition 2 (Consistency): consistencySpec in syntax_ott.v.
- Lemma 3 (Soundness and completeness of the definition of top-like types): toplike_super_top in Key_Propperties.v.
- Lemma 4 (Disjointness properties): disjoint_eqv, disjoint_domain_type, and disjoint_and in Key_Propperties.v.
- Definition 5 (Principal types): principal_type in Key_Propperties.v.
- Lemma 6 (Principal types): principal_type_sub, principal_type_disjoint, and principal_type_checks in Key_Propperties.v.
- Lemma 7 (Typed reduction on top-like types): TypedReduce_toplike in Key_Propperties.v.
- Lemma 8 (Transitivity of typed reduction): TypedReduce_trans in Type_Safety.v.
- Lemma 9 (Typed reduction respects subtyping): TypedReduce_sub in Key_Propperties.v.
- Lemma 10 (Consistency of disjoint values): disjoint_val_consistent in Key_Propperties.v.
- Lemma 11 (Determinism of typed reduction): TypedReduce_unique in Deterministic.v.
- Lemma 12 (Consistency after typed reduction): consistent_afterTR in Type_Safety.v.
- Lemma 13 (Preservation of typed reduction): TypedReduce_preservation in Type_Safety.v.
- Lemma 14 (Progress of typed reduction): TypedReduce_progress in Type_Safety.v.
- Theorem 15 (Determinism of \( \hookrightarrow \rightarrow \)): step_unique in Deterministic.v.
- Theorem 16 (Type preservation of \( \hookrightarrow \rightarrow \)): preservation in Type_Safety.v.
- Theorem 17 (Progress of \( \hookrightarrow \rightarrow \)): progress in Type_Safety.v.
- Theorem 18 (Soundness of \( \hookrightarrow \rightarrow \) with respect to Dunfield’s semantics): reduction_soundnes in main_version/dunfield.v.
- Theorem 19 (Soundness of typed reduction with respect to Dunfield’s semantics). tred_soundnes in main_version/dunfield.v.
- Theorem 20 (Completeness of typing with respect to \( \lambda i \)): typing_completeness in main_version/icfp.v.
- Theorem 21 (Completeness of typing with respect to the extended bidirectional type system of \( \lambda i \)): typing_completeness in coq/main_version/icfp_bidirectional.v.
- Theorem 22 (Soundness of \( \hookrightarrow \rightarrow \) in the simple variant): reduction_soundnes in variant/dunfield.v.
- Theorem 23 (Completeness of typing in the simple variant): typing_completeness in variant/icfp.v.

References