Generic Blame-Subtyping Theorem in Agda Using Abstract Binding Trees

Tianyu Chen

Computer Science, Indiana University. Advised by Professor Jeremy G. Siek

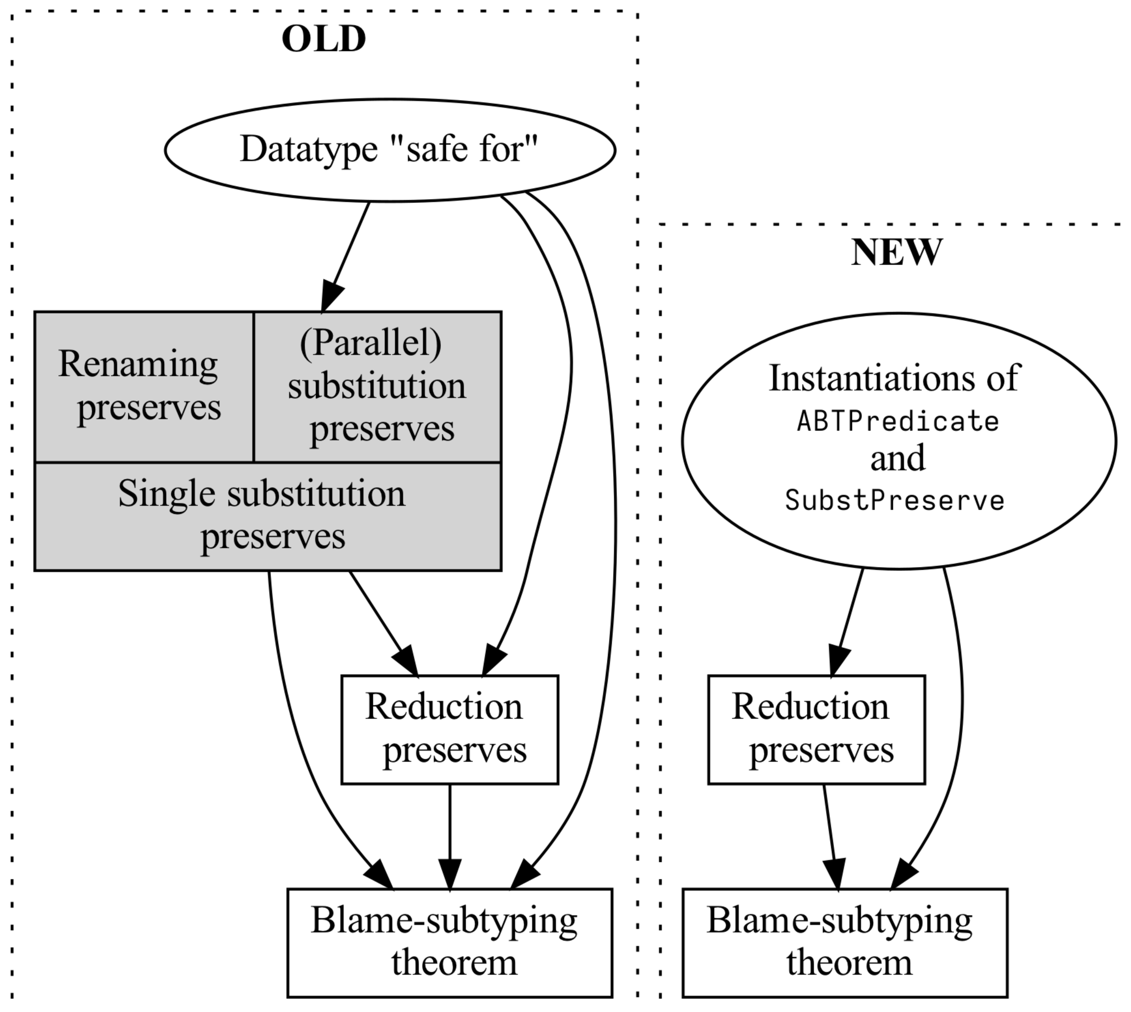

Figure: The new architecture (right) of the blame-subtyping theorem proof compared with the old one (left). The proofs of the highlighted lemmas are replaced by the instantiation of the ABT library. Edges indicate dependency.

Introduction

The Gradual Typing in Agda project develops a generic cast calculus called $CC(\Rightarrow)$ that is parameterized by casts. The meta-theory of the Gradually Typed Lambda Calculus (GTLC) developed with $CC(\Rightarrow)$ is reusable across multiple variants of cast representations. Siek and Chen [1] prove 1) type safety 2) blame-subtyping, and 3) the dynamic gradual guarantee about $CC(\Rightarrow)$. To reduce repetitive work, it is necessary to abstract away the similarities between different cast representations and the proofs of various lemmas and theorems. The former is done by leveraging the module system of Agda, which makes code and proof sharing between cast representations possible (Table 1). This poster targets the latter half of the problem: generalizing over the preservation proofs of various predicates. Many proofs involve lemmas about "substitution preserves", such as "substitution preserves type" in type preservation, "substitution preserves term precision" in the dynamic gradual guarantee, and "substitution preserves 'safe for'" in the blame-subtyping theorem [2]. Specifically in this project, we focus on the last one: proving the blame theorem [3, 4, 2] by using the ABT library to represent $CC(\Rightarrow)$ terms and get the substitution lemma *for free* (Figure 1).

Background

[Gradual Typing] Gradual typing is a paradigm that combines static and dynamic typing by inserting checks on the boundaries. Consider the following partially typed program, where * stands for the statically unknown type:

```
(define (abs (n : Int)) → Int
    (if (n < 0) (- n) n))
(define (dist (n : Int) (m : *)) → Int
    (abs (n - m)))
```

Function call `dist(0, ``yes'')` will result in a failed cast, which is caught at runtime. Intermediate languages where all casts are explicit are usually called cast calculi.

[Abstract Binding Tree] We use the ABT library, which is an Agda implementation of Chapter 1 of Harper [5]. For example, Figure 2 represents the term `(λ x : * . x) (unit ⟨ Unit ⇒ * ⟩)` in $CC(\Rightarrow)$ as an ABT:

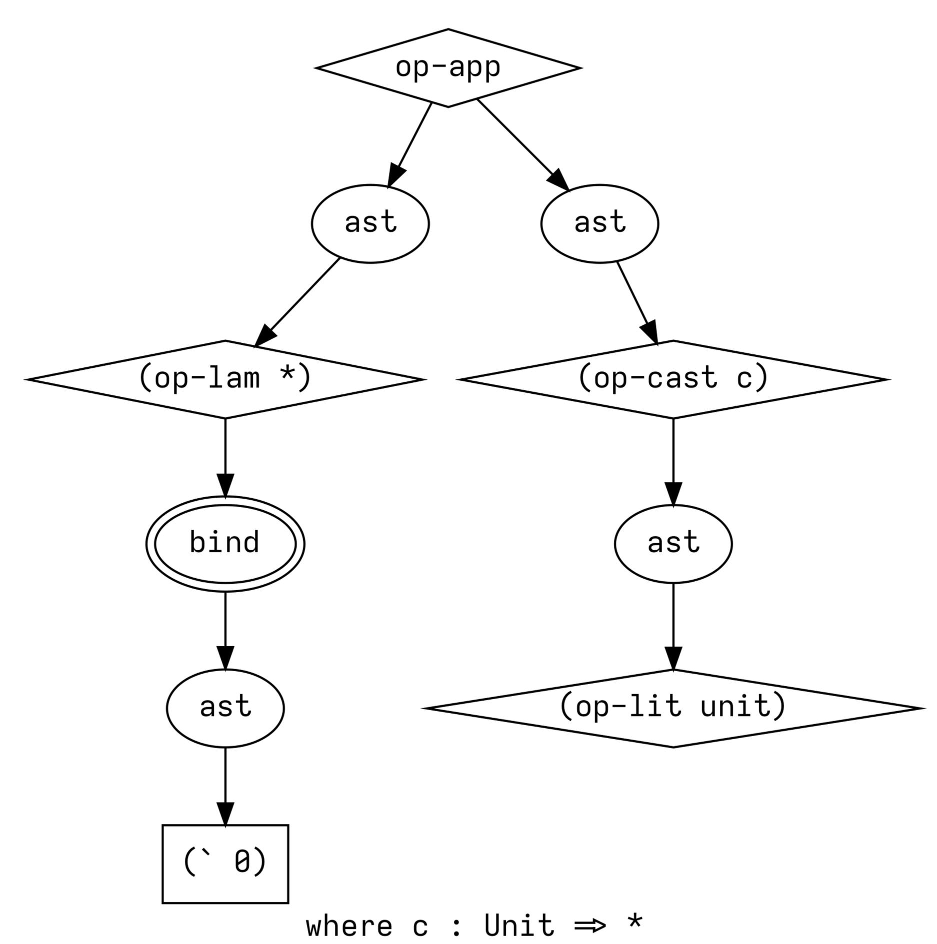

Figure: Representing `(λx:*.x) (unit⟨Unit ⇒ *⟩)` as ABT

Approach and Implementation

Instead of the usual approach of defining the "safe for" predicate as a datatype, we instantiate `module ABTPredicate` in the ABT library, which contains a notion of "generic predicate on ABT". For example, consider the rule for `cast`:

```
safefor-cast : ∀ {S T M} {c : Cast (S ⇒ T)} {ℓ}
  → CastBlameSafe c ℓ
  → M SafeFor ℓ
  ---------------------
  → (M ⟨ c ⟩) SafeFor ℓ
```

It becomes a clause of P_s, which is for the predicate on operator nodes:

```
Pₛ (op-cast c) (ℓₘ ∷ []) ⟨ tt , tt ⟩ ℓ =
  CastBlameSafe c ℓ × ℓ ≡ ℓₘ
```

Essentially, we need one such clause for *each* operator, the right hand side of which stands for the constraints. For `cast`, they say 1) cast c is safe 2) the `cast` term and its sub-term are safe for the same label. The counterpart of P_s on variable nodes is V_s. The "safe for" predicate is then created as an instantiation of the generic predicate, by passing the definitions of both V_s and P_s. Similarly, we instantiate `module SubstPreserve` and get the lemma "substitution preserves 'safe for'" for free.

We state the blame-subtyping theorem as:

```
soundness-<: : ∀ {A} {M : Term} {ℓ}
  → M SafeFor ℓ
  --------------------
  → ¬ (M —↠ blame A ℓ)
```

The proof is by induction on the reduction sequence `M —↠ blame ℓ`, which depends on the substitution lemma and "reduction preserves 'safe for'". The proof of the latter follows the overall structure of type preservation.

Theoretic Results of the Gradual Typing in Agda Project (Table 1)

We compare the proofs of various properties (rows) across cast representations (columns) before (to the left of "/") and after (to the right of "/") Gradual Typing in Agda. The status of a proof can be 1) m: has proof assistant mechanization 2) p: has pen-and-paper proof in prior literature or 3) x: has no proof.

Cast Representation	EDA	EDI	λB	EDC	LDC	λC
Type Safety	m / m	x / m	m / m	x / m	p / m	p / m
Blame-subtyping Theorem	p / m	x / m	p / m	x / m	p / m	p / m
Dynamic Gradual Guarantee	x / x	x / x	m / m	x / x	x / x	x / x

Summary of Contributions

We obtain a shorter Agda proof of the blame-subtyping theorem of the parameterized cast calculus $CC(\Rightarrow)$ by switching to the abstract binding tree (ABT) library as the representation for terms. We define the "safe for" predicate in the style of the library and acquire the substitution lemma for free. We argue that this proof technique is suitable for proving the preservation of arbitrary predicates on terms.

Future Work

There are two possible future research directions:

[The ABT library] The ABT library does not provide generic theorems about *reduction*. A possible extension to the library may handle the reduction of ABT generically. In addition to "substitution preserves predicate", we can then obtain generic proofs of properties about rewriting like confluence.

[Gradual Typing in Agda] We plan to make the transition to using ABTs across the entire project. We are also investigating ways to decouple blame-strategy, UD versus D, from our current proof of the dynamic gradual guarantee. Additionally, we would like to incorporate mutable references and different heap models into the project.

References

[1] Jeremy G. Siek and Tianyu Chen. Parameterized cast calculi and reusable meta-theory for gradually typed lambda calculi. *Journal of Functional Programming*, 31: e30, 2021. doi: 10.1017/S0956796821000241.

[2] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming*, ESOP, pages 1–16, March 2009.

[3] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *SNAPL: Summit on Advances in Programming Languages*, LIPIcs: Leibniz International Proceedings in Informatics, May 2015.

[4] Jeremy G. Siek, Peter Thiemann, and Philip Wadler. Blame and coercion: Together again for the first time. In *Conference on Programming Language Design and Implementation*, PLDI, June 2015.

[5] Professor Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2012. ISBN 1107029570, 9781107029576.

Contact Information

- Website: https://homes.luddy.indiana.edu/chen512
- Email: chen512@indiana.edu