Generic Blame-Subtyping Theorem in Agda Using Abstract Binding Trees

TIANYU CHEN*, Indiana University, USA

Gradually typed languages insert casts between statically typed and dynamically typed code fragments and cast calculi are intermediate representations where all casts are explicit. In this project we study parameterized cast calculus $CC(\Rightarrow)$ and build a generic proof in Agda of the Blame-Subtyping Theorem that is agnostic to cast representation. The Blame-Subtyping Theorem states that a cast whose source type and target type are subject to a subtyping relation will not get blamed during runtime. The proof involves a "safe for" predicate on cast calculus terms. We leverage the abstract binding tree (ABT) library, which is capable of reasoning about arbitrary predicates on ABT, to acquire the substitution lemma about "safe for" for free, thereby obtaining a shorter proof compared to prior work.

1 INTRODUCTION AND BACKGROUND

Gradual typing [2, 5, 8] is a paradigm that combines static and dynamic typing by inserting checks on the boundaries. Such checks are often represented as casts and intermediate languages where casts are explicit are referred to as cast calculi. When a cast fails, it gets blamed. Blames are trivial for casts between first-order types, but become complicated under a higher-order setting [4]. In practice, each cast is associated with a blame label ℓ , by which casts are identified. For example, if cast $c : A \Rightarrow^{\ell} B$ fails when the program executes, it gives rise to `(blame ℓ)`, signifying a runtime error, where ℓ is the blame label on c. The blame-subtyping theorem [6, 7, 9] states that in a cast calculus, cast $c : A \Rightarrow^{\ell} B$ will never get blamed if its source type and target type satisfy subtyping $A <: B$. It is a desirable property in that an analyzer can inspect the code *statically* and distinguish *safe* casts that are never blamed from *unsafe* ones.

The proof of blame-subtyping is in preservation-style [9]. A "safe for" predicate on term M is defined to indicate that all casts in M are *safe* with respect to blame label ℓ. Suppose we call this predicate `_SafeFor_`, we need to prove that `(M SafeFor ℓ)` is preserved during reduction. The β-rule in reduction is defined with substitution. The standard approach is to follow Chapter *Properties* of Wadler et al. [10] and include a lemma "substitution preserves the 'safe for' predicate". It makes the proof tedious, because the approach does not abstract away the similarities between type preservation and the preservation of "safe for". It motivates us to switch to representing terms using the abstract binding tree (ABT) library [1]. The ABT library has an abstraction over arbitrary predicates on ABT. Furthermore, it includes generic proofs of "operations on ABT preserve predicate", including substitution.

Our project depends on two existing developments in in Agda, combining their distinctions. The meaning of *generic* in the title is thus twofold: 1) It is *generic* regarding cast representation, like Gradual Typing in Agda (Section 1.1). 2) We define "safe for" in the style of the ABT library (Section 1.2) and get the substitution lemma for free, by instantiating the proof about *generic predicate*.

1.1 Mechanized Gradual Typing Meta-theory in Agda

The Gradual Typing in Agda project [3] [2] develops a generic cast calculus called $CC(\Rightarrow)$ that is parameterized by casts. $CC(\Rightarrow)$ is implemented as an Agda module:

```
module ParamCastCalculus (Cast : Type → Set) where -- ...
```

*Advised by Professor Jeremy G. Siek. Category: graduate

[1] Code: https://github.com/jsiek/abstract-binding-trees

[2] Code: https://github.com/jsiek/gradual-typing-in-agda

Author's address: Tianyu Chen, Computer Science, Indiana University, Bloomington, Indiana, USA, chen512@indiana.edu.

With $CC(\Rightarrow)$, the project develops the meta-theory of the Gradually Typed Lambda Calculus (GTLC) that is reusable across multiple variants of cast representations. Each cast calculus variant instantiates `module ParamCastCalculus`, by passing its respective definition of cast ($\Rightarrow$), i.e., the inhabitant of `(Cast : Type → Set)`. Siek and Chen [3] instantiate $CC(\Rightarrow)$ for 6 representations and its space efficient counterpart $SC(\Rightarrow)$ for 2 coercion-based representations. In addition, The paper proves three major theoretic results about $CC(\Rightarrow)$: 1) type safety 2) blame-subtyping, and 3) the dynamic gradual guarantee.

Our work fits within the Gradual Typing in Agda framework, inheriting its benefit of being cast representation agnostic. We improve upon it and simplify the blame theorem proof by turning to the ABT library as our representation of $CC(\Rightarrow)$.

1.2 The Abstract Binding Tree Library

The abstract binding tree (ABT) library is an Agda implementation of Chapter 1 of Harper [1].

```
1 module AbstractBindingTree (Op : Set) (sig : Op → List ℕ) where
2   data Args : List ℕ → Set
3   data ABT : Set where
4     `_ : Var → ABT
5     _⦅_⦆ : (op : Op) → Args (sig op) → ABT
```

The ABT module (line 1) takes two parameters: 1) `data Op : Set` where each constructor corresponds to one kind of operator 2) function `sig : Op → List ℕ` which maps each operator to its signature, where the length of the list corresponds to the number of sub-terms (arity) and each element is the number of free variables in each respective sub-term. An ABT may contain two types of nodes: variable nodes and operator nodes. A variable node (line 4) contains no child (in other word must be leaf node) while an operator node (line 5) can have arbitrary number of children, depending on its arity. Consider λ-calculus as an instantiation of ABT:

```
1 data Op : Set where                 4 sig : Op → List ℕ
2   op-lam   : Type → Op              5 sig (op-lam A)     = 1 ∷ []
3   op-app   : Op                     6 sig op-app         = 0 ∷ 0 ∷ []
```

Abstraction ($\lambda x : A. M$) contains a single sub-term M. M has one binding that introduces the variable x into scope. We use de Bruijn indices to represent variables. The operator for λ-abstraction `op-lam` takes one parameter, type annotation A (line 2). The body M of λ-abstraction has one free variable, so its signature is 1 (line 5). Neither sub-term in application has free variable, so their signatures are both 0 (line 6).

The ABT has a notion of generic predicate, whose definition follows that of ABT, by having two constructors, `var-p` for variable nodes and `op-p` for operator nodes. The predicate on the arguments of an operator node is isomorphic to list, having two constructors `nil-p` and `cons-p`. For each argument, recall that we use a ℕ to represent the number of variable bindings. If the term contains a binder, we use `bind-p` to construct the predicate, corresponding to `suc`; otherwise we use `ast-p` for the ABT of the child, corresponding to `zero`. In `module SubstPreserve`, the `preserve-substitution` lemma is proved for arbitrary ABTs, M and N, that `N [ M ]` preserves the generic predicate.

2 APPROACH AND IMPLEMENTATION

Instead of defining the typing rules as a data type where each constructor takes the well-typedness of sub-terms, we abstract the constraints $\mathscr{C}$ away and get the benefit of reusing the same set of rules for various predicates, including typing and "safe for". Again consider λ-abstraction and application as examples, the extra parameter `C` is for constraints $\mathscr{C}$:

```
pattern ⊢ƛ A  ⊢N C = op-p {op-lam A} (cons-p (bind-p (ast-p ⊢N)) nil-p) C
pattern ⊢· ⊢L ⊢M C = op-p {op-app}   (cons-p (ast-p ⊢L) (cons-p (ast-p ⊢M) nil-p)) C
```

Predicate "safe for" is created by instantiating `module ABTPredicate`. The module takes two additional parameters V and P; they correspond to variable nodes and operator nodes respectively. The constraints of the "safe for" predicate are passed in by creating the syntax directed definitions of V_s and P_s [3] . The main trick is to define the predicate as $\Psi \vdash M : \ell$, where ℓ is the blame label that "safe for" is quantified over and Ψ represents an environment whose length equals to the levels of binders. We elaborate on three interesting cases:

[Variable] Variable `(` x)` is blame-safe for any label $\ell$ (Rule 1 Appendix A.1). We deliberately choose $\Psi$ to be a list of $\ell$ that the "safe for" predicate is quantified over, so we simply relate $\ell$ with the lookup result $\ell'$ in environment $\Psi$ : `Vₛ _ _ ℓ′ ℓ = ℓ ≡ ℓ′`.

[Cast] By definition, cast $c : A \Rightarrow^{\ell'} B$ is blame-safe for ℓ if $A <: B$ or $\ell \neq \ell'$. The term `(M ⟨ c ⟩)` that consists of sub-term M and cast c is blame-safe for ℓ if 1) cast c is blame-safe for ℓ 2) sub-term M is blame-safe for ℓ (Rule 2 Appendix A.1). We summarize Rule 2 in P_s, the predicate on operator nodes : `Pₛ (op-cast c) (ℓₘ ∷ []) ⟨ tt , tt ⟩ ℓ = CastBlameSafe c ℓ × ℓ ≡ ℓₘ`.

[Blame] According to Rule 4, we require that the label on "blame" is not equal to the label that "safe for" is quantified over: `Pₛ (op-blame A ℓ′) [] tt ℓ = ℓ ≢ ℓ′`.

With V_s and P_s defined, we instantiate "safe for" by initializing with an empty Ψ :

```
open import ABTPredicate Op sig Vₛ Pₛ renaming (_⊢_⦂_ to predicate-SafeFor)
_SafeFor_ : Term → Label → Set    -- M SafeFor ℓ
_SafeFor_ = predicate-SafeFor []   -- Start with an empty environment
```

The table-setting is complete and now we instantiate `module SubstPreserve` and get "substitution preserves predicate" for free:

```
open import SubstPreserve Op sig Label Vₛ Pₛ
  (λ _ → refl) (λ { refl refl → refl }) (λ x → x) (λ { refl pM → pM })
  renaming (preserve-substitution to substitution-SafeFor)
```

With the substitution lemma in hand, "reduction preserves the 'safe for' predicate" becomes straightforward to prove. We include the full Agda formalization of the proofs in Appendix B.

3 CONCLUSION AND FUTURE WORK

In this work we take advantage of abstract binding trees and define the "safe for" predicate on the parameterized cast calculus $CC(\Rightarrow)$ as an instantiation of the generic predicate on ABT. We gain the substitution lemma about "safe for" for free and thus obtain a shorter proof of the blame-subtyping theorem, one of the core properties of gradual typing, while inheriting prior work's benefit of being generic regarding cast representation.

In future, we plan to continue our exploration in two directions:

[The ABT library] Currently, the ABT library does not provide generic theorems about reduction. We plan to extend the library and handle the reduction of ABT generically. In this way, in addition to "substitution preserves predicate", we can obtain generic proofs of properties about rewriting like confluence. Another possible extension to the library is adding binary relations beside predicates.

[Gradual Typing in Agda] We plan to make the transition to using ABTs across the entire project and port all existing proofs about the parameterized cast calculus $CC(\Rightarrow)$. 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.

[3] The subscription "s" is for "subtyping".

REFERENCES

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

[2] Jacob Matthews and Robert Bruce Findler. 2007. Operational Semantics for Multi-Language Programs. In *The 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.*

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

[4] Jeremy G. Siek, Ronald Garcia, and Walid Taha. 2009. Exploring the Design Space of Higher-Order Casts. In *European Symposium on Programming (ESOP).* 17–31.

[5] Jeremy G. Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop.* 81–92.

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

[7] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *SNAPL: Summit on Advances in Programming Languages (LIPIcs: Leibniz International Proceedings in Informatics).*

[8] Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: From Scripts to Programs. In *Dynamic Languages Symposium.*

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

[10] Philip Wadler, Wen Kokke, and Jeremy G. Siek. 2020. *Programming Language Foundations in Agda.* http://plfa.inf.ed.ac.uk/20.07/

A SUPPLEMENTARY DEFINITIONS

A.1 Inference Rules of the “Safe For” Predicate

$$\text{Variable}\ \frac{}{(\grave{}\ x)\ \mathtt{SafeFor}\ \ell} \tag{1}$$

$$\text{Cast}\ \frac{\mathtt{CastBlameSafe}\ c\ \ell \qquad M\ \mathtt{SafeFor}\ \ell}{(M\ \langle\ c\ \rangle)\ \mathtt{SafeFor}\ \ell} \tag{2}$$

$$\text{Wrap}\ \frac{\mathtt{CastBlameSafe}\ c\ \ell \qquad M\ \mathtt{SafeFor}\ \ell}{(M\ \langle\ c\ _{(}\ i\ _{)}\rangle)\ \mathtt{SafeFor}\ \ell} \tag{3}$$

$$\text{Abstraction}\ \frac{N\ \mathtt{SafeFor}\ \ell}{(\text{ƛ}\ A\ \dot{}\ N)\ \mathtt{SafeFor}\ \ell} \tag{4}$$

$$\text{Application}\ \frac{L\ \mathtt{SafeFor}\ \ell \qquad M\ \mathtt{SafeFor}\ \ell}{(L\ \cdot\ M)\ \mathtt{SafeFor}\ \ell} \tag{5}$$

$$\text{Constant}\ \frac{}{(\$\ k)\ \mathtt{SafeFor}\ \ell} \tag{6}$$

$$\text{If}\ \frac{L\ \mathtt{SafeFor}\ \ell \qquad M\ \mathtt{SafeFor}\ \ell \qquad N\ \mathtt{SafeFor}\ \ell}{(\mathtt{if}\ L\ \mathtt{then}\ M\ \mathtt{else}\ N)\ \mathtt{SafeFor}\ \ell} \tag{7}$$

$$\text{Blame}\ \frac{\ell \neq \ell'}{(\mathtt{blame}\ \ell')\ \mathtt{SafeFor}\ \ell} \tag{8}$$

Our Agda development includes pairs and sums, but they are omitted here for simplicity. The rules are adapted from Siek and Chen [3]. “Wrap” is a special case of “cast”, where the attached cast must be *inert*, i.e., value-forming.

A.2 Statement of “Substitution Preserves Predicate”

```
preserve-substitution : ∀ {Γ : List I} {A B : I} (N M : ABT)
  → (A ∷ Γ) ⊢ N ⦂ B
  → Γ ⊢ M ⦂ A
    ----------------
  → Γ ⊢ N [ M ] ⦂ B
```

The `_⊢_⦂_` stands for *generic predicate*, generalized over *I*. Both the typing predicate and the “safe for” predicate in our development are instantiations of the generic predicate.

B DETAILED PROOFS OF TECHNICAL RESULTS

B.1 Plugging a Blame

If we plug a "blame" into any frame F and the result term is blame-safe for ℓ then their blame labels must be different. The proof is straightforward, by inversion on F .

```
1  plug-blame-safe-for-diff-ℓ : ∀ {A B} {ℓ ℓ′}
2    → (F : Frame A B)
3    → (plug (blame A ℓ′) F) SafeFor ℓ
4      -------------------------------------
5    → ℓ ≢ ℓ′
```

B.2 Preservation of "Safe For"

The proof for "reduction preserves the 'safe for' predicate" (line 1) is by induction on reduction (starting from line 25). The proof is mostly straightforward and follows the same overall structure of type preservation.

To improve readability, we collapse all cases that belong to the ξ-rule produced by casing on frame into a single lemma about "plug" (line 7):

```
1  preserve-SafeFor : ∀ {M M′ : Term} {ℓ}
2    → M SafeFor ℓ
3    → M —→ M′
4      --------------------
5    → M′ SafeFor ℓ
6
7  preserve-SafeFor-plug : ∀ {A B} {M M′ : Term} {ℓ}
8    → (F : Frame A B)
9    → (plug M F) SafeFor ℓ
10   → M —→ M′
11     -----------------------------
12   → (plug M′ F) SafeFor ℓ
13
14 preserve-SafeFor-plug (F-·₁ _ _) (⊢· safeforₗ safeforₘ 𝐶ₛ-·) R =
15     ⊢· (preserve-SafeFor safeforₗ R) safeforₘ 𝐶ₛ-·
16 preserve-SafeFor-plug (F-·₂ _ _ _) (⊢· safeforₗ safeforₘ 𝐶ₛ-·) R =
17     ⊢· safeforₗ (preserve-SafeFor safeforₘ R) 𝐶ₛ-·
18 preserve-SafeFor-plug (F-if _ _ _ _) (⊢if safeforₗ safeforₘ safeforₙ 𝐶ₛ-if) R =
19     ⊢if (preserve-SafeFor safeforₗ R) safeforₘ safeforₙ 𝐶ₛ-if
20 preserve-SafeFor-plug (F-cast c) (⊢cast .c safeforₘ ⟨ safe , refl ⟩) R =
21     ⊢cast c (preserve-SafeFor safeforₘ R) ⟨ safe , refl ⟩
22 preserve-SafeFor-plug (F-wrap c i) (⊢wrap .c .i safeforₘ ⟨ safe , refl ⟩) R =
23     ⊢wrap c i (preserve-SafeFor safeforₘ R) ⟨ safe , refl ⟩
24
25 preserve-SafeFor safefor (ξ {F = F} _ rd) =
26     preserve-SafeFor-plug F safefor rd
27 preserve-SafeFor safefor (ξ-blame {F = F}) =
28     ⊢blame _ _ (plug-blame-safe-for-diff-ℓ F safefor)
29 preserve-SafeFor (⊢· (⊢ƛ _ safeforₙ 𝐶⊢-ƛ) safeforₘ 𝐶ₛ-·) (β v) =
30     substitution-SafeFor _ _ safeforₙ safeforₘ
31 preserve-SafeFor _ δ = ⊢$ _ _ 𝐶ₛ-$
32 preserve-SafeFor (⊢if _ safeforₘ _ 𝐶ₛ-if) β-if-true = safeforₘ
33 preserve-SafeFor (⊢if _ _ safeforₙ 𝐶ₛ-if) β-if-false = safeforₙ
34 preserve-SafeFor (⊢cast c safeforₘ ⟨ safe , refl ⟩) (cast v {a}) =
    applyCast-pres-SafeFor a safe saferorₘ
preserve-SafeFor (⊢· (⊢wrap c i safefor₁ ⟨ safe , refl ⟩) safeforₘ 𝐶ₛ-·)
                 (fun-cast {c = c} v w {x}) =
    ⊢cast _ (⊢· safefor₁ (⊢cast _ safeforₘ ⟨ domBlameSafe safe x , refl ⟩) 𝐶ₛ-·)
          ⟨ codBlameSafe safe x , refl ⟩
```

B.3 The Blame-Subtyping Theorem

We state the blame-subtyping theorem as "soundness of <:". The proof is by induction on reduction.

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

[ξ]. The case for ξ-rule is proved using the "reduction preserves" lemma in Section B.2:

```
soundness-<: safefor-plug ( _ —→⟨ ξ ⊢M M—→M′ ⟩ plugM′F↠blame ) =
    soundness-<: (preserve-SafeFor safefor-plug (ξ ⊢M M—→M′)) plugM′F↠blame
```

[ξ-blame]. The ξ-blame case is impossible, because 1) "blame" does not reduce 2) of the lemma in Section B.1.

```
soundness-<: safefor ( _ —→⟨ ξ-blame {F = F} ⟩ blame↠blame ) =
    case blame↠blame of λ where
      (_ ∎) →
        contradiction refl (plug-blame-safe-for-diff-ℓ F safefor)
      (_ —→⟨ blame→ ⟩ _) →
        contradiction blame→ (blame⌿→ refl)
```

[β]. Recall that β-rule invokes single substitution, following which we use the lemma "single substitution preserves the 'safe for' predicate" in Section 2:

```
-- Application (β)
soundness-<: (⊢· (⊢ƛ _ safeforₙ 𝐶ₛ-ƛ) safeforₘ 𝐶ₛ-·)
             ( (ƛ _ ˙ N) · M —→⟨ β vₘ ⟩ N[M]↠blame ) =
    soundness-<: (substitution-SafeFor _ _ safeforₙ safeforₘ) N[M]↠blame
```

[δ and β-if]. The δ case and both β-if cases are straightforward, directly proved by induction hypotheses about the rest of the reduction sequence:

```
-- δ
soundness-<: (⊢· _ _ 𝐶ₛ-·)
             ( ($ f # _) · ($ k # _) —→⟨ δ ⟩ f·k↠blame ) =
    soundness-<: (⊢$ (f k) _ 𝐶ₛ-$) f·k↠blame
-- If
soundness-<: (⊢if _ safeforₘ _ 𝐶ₛ-if)
             ( if ($ true # _) then M else N endif  —→⟨ β-if-true ⟩  M↠blame ) =
    soundness-<: safeforₘ M↠blame
soundness-<: (⊢if _ _ safeforₙ 𝐶ₛ-if)
             ( if ($ false # _) then M else N endif —→⟨ β-if-false ⟩ N↠blame ) =
    soundness-<: safeforₙ N↠blame
```

[Cast]. In the case for cast, we turn to field `applyCast-pres-SafeFor` in the cast representation interface and prove that applying a cast to a value preserves "safe for". Its proof is specific to each cast representation, since the implementation of applying a cast varies from representation to representation.

```
-- Cast
soundness-<: (⊢cast .c safeforₘ ⟨ safe , refl ⟩)
             ( M ⟨ c ⟩ —→⟨ cast v {a} ⟩ applyCastMc↠blame ) =
    soundness-<: (applyCast-pres-SafeFor a safe safeforₘ) applyCastMc↠blame
```

[Wrap]. Wrapping an inert cast is value-forming, so the proof is straightforward.

```
-- Wrap
soundness-<: (⊢cast .c safeforₘ ⟨ safe , refl ⟩)
             ( M ⟨ c ⟩ —→⟨ wrap v {i} ⟩ applyCastMc↠blame ) =
    soundness-<: (⊢wrap c i safeforₘ ⟨ safe , refl ⟩) applyCastMc↠blame
```

[Fun-cast]. Proved by the induction hypothesis about the rest of the reduction sequence.

```
-- Fun-cast
soundness-<: (⊢· (⊢wrap .c .i safeforₘ ⟨ safe , refl ⟩) safeforₙ 𝐶ₛ-·)
             ( M ⟨ c ₍ i ₎⟩ · N —→⟨ fun-cast vₘ vₙ {x} ⟩ M·N↠blame) =
    soundness-<: (⊢cast _ (⊢· safeforₘ
                                (⊢cast _ safeforₙ ⟨ domBlameSafe safe x , refl ⟩) 𝐶ₛ-·)
                            ⟨ codBlameSafe safe x , refl ⟩) M·N↠blame
```

[Blame]. Impossible because blaming ℓ is not blame-safe for the same label ℓ:

```
-- Blame
soundness-<: (⊢blame _ _ ℓ≢) (blame _ ℓ ∎) = ℓ≢ refl
```