Automatically Generating the Dynamic Semantics of Gradually Typed Languages

Matteo Cimini Jeremy G. Siek
Indiana University, USA
mcimini@indiana.edu jsiek@indiana.edu

Abstract

Many language designers have adopted gradual typing. However, there remains open questions regarding how to gradualize languages. Cimini and Siek (2016) created a methodology and algorithm to automatically generate the type system of a gradually typed language from a fully static version of the language. In this paper, we address the next challenge of how to automatically generate the dynamic semantics of gradually typed languages. Such languages typically use an intermediate language with explicit casts.

Our first result is a methodology for generating the syntax, type system, and dynamic semantics of the intermediate language with casts. Next, we present an algorithm that formalizes and automates the methodology, given a language definition as input. We show that our approach is general enough to automatically gradualize several languages, including features such as polymorphism, recursive types and exceptions. We prove that our algorithm produces languages that satisfy the key correctness criteria of gradual typing. Finally, we implement the algorithm, generating complete specifications of gradually typed languages in lambda-Prolog, including executable interpreters.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages, Theory

Keywords gradual typing, operational semantics, type systems, methodology

1. Introduction

Programmers see many benefits in combining static and dynamic typing within one language. Gradual typing is an approach to integrating the two disciplines in a way that provides seamless interoperability [24, 30]. Many language designers have adopted gradual typing [4, 14, 15, 29, 33, 34, 37], and many others are interested in the shift to gradual typing. However, it is not an easy task for language designers to produce a gradual language [28]. Although the literature is rich with examples, language designers need precise guidance on how to proceed to gradualize their languages. Furthermore, automatic tools are desirable to help them with the shift to gradual typing.

Recent work has begun to address this challenge. Notably, Garcia et al. [11] present an approach to the derivation of the type system and dynamic semantics of gradually typed languages based on abstract interpretation. The authors show the applicability of their approach to several languages and put forward a reusable methodology. At the same time, Cimini and Siek [5] presented a methodology and automatic tool for generating gradual type systems. Despite these two efforts, some challenges remain. In particular, it is an open question whether a fully automatic tool can generate the dynamic semantics for gradually typed languages. As it has been pointed out repeatedly [12, 23], the dynamic semantics is the difficult and delicate part of gradual typing. In this paper, we address its automatic generation.

A Methodology Gradually-typed languages typically express their dynamic semantics through an intermediate language with explicit casts [1, 24, 36], here referred to as the *cast language*. The semantics of cast languages must be designed to carefully check whether a cast should succeed or fail. Introducing casts also raises questions regarding the value forms of the language. What value should result from casting a function from one function type $T_1 \to T_2$ to another $T_3 \to T_4$? One approach is to introduce a new value form for casted functions and then add the following reduction rule [8–10].

$$(v_1 : T_1 \to T_2 \Rightarrow T_3 \to T_4)\ v_2 \longrightarrow (v_1\ (v_2 : T_3 \Rightarrow T_1)) : T_2 \Rightarrow T_4$$

A function of type $T_3 \to T_4$ is simulated using the function v_1, first casting the argument to its domain and then casting the result to the target output type. The design of this rule relies on intuitions regarding the very meaning of functions. The challenge for a general methodology and automated tool is that the meaning of the language and its operators is only available in so far as it is expressed in the reduction rules.

Notwithstanding, we present a methodology for generating the syntax, type system and dynamic semantics of the cast language starting from a fully-static language. One of the intuitions behind our methodology is in detecting when new reduction rules are needed. The rule above, for example, is needed because applications are elimination forms. Functions that are wrapped in casts are new canonical forms for the function type and they are not handled by the β-rule $(\lambda x{:}T.\, e)\ v \longrightarrow e[v/x]$. We generate new reduction rules as demanded by the progress lemma. The idea is that the new reduction rules should remove casts from the value being eliminated so that the native reduction rule (e.g. β for functions) can eventually apply. The challenge is to preserve well-typedness after removing the cast. We show that this is possible by decomposing and inserting casts in a type-directed manner.

The blame theorem is a key correctness property for gradual languages. It ensures that safe casts, i.e. casts that respect subtyp-

POPL'17, January 15–21, 2017, Paris, France
ACM. 978-1-4503-4660-3/17/01...$15.00
http://dx.doi.org/10.1145/3009837.3009863

input

```
module stlc.

typeOf (abs T1 E) (arrow T1 T2)  :- (pi x\ typeOf x T1 => typeOf (E x) T2).
typeOf (app E1 E2) T2            :- typeOf E1 (arrow T1 T2), typeOf E2 T1.

value (abs T E).
step (app (abs T E1) E2) (E1 E2) :- value E2.

% context app E e
% context app v E.
% eliminator app 1.
```

output (excerpt)

```
typeOfCC (abs T1 E) (arrow T1 T2) :- (pi x\ (typeOfCC x T1 => typeOfCC (E x) T2)).
typeOfCC (app E1 E2) T2           :- typeOfCC E1 (arrow T1 T2), typeOfCC E2 T1.
typeOfCC (cast E T1 L T2) T2      :- typeOfCC E T1, consistency T1 T2.
typeOfCC (blame T L) T.

value (cast V G L (dyn))                        :- value V, ground G.
value (cast V (arrow T1 T2) L (arrow T1' T2')) :- value V.

step E E' :- stepC E E'.
stepC (cast V T L T) V                                          :- value V.
stepC (cast (cast V G L1 (dyn)) (dyn) L2 G) V                   :- value V, ground G.
stepC (cast (cast V G1 L1 (dyn)) (dyn) L2 G2) (blame G2 L2) :- value V, ground G1, ground G2,
                                                                   not (sameGround G1 G2).
stepC (cast V T L (dyn)) (cast (cast V T L G) G L (dyn))    :- value V, getGroundOf T G, not (ground T).
stepC (cast V (dyn) L T) (cast (cast V (dyn) L G) G L T)    :- value V, getGroundOf T G, not (ground T).
stepC (app (cast V (arrow T1' T2') L (arrow T1 T2)) E2)
      (cast (app V (cast E2 T1 L T1')) T2' L T2)            :- value V, value E2.
stepC E (blame T1 L)                                        :- typeOfCC E T1, contains E (blame T2 L).
```

Figure 1. Example input: STLC formulation in λ-Prolog, augmented with convenient tags for declaratively specifying evaluation contexts and the eliminators of the language. We use higher-order abstract syntax (HOAS), e.g. (`E1 E2`) encodes the substitution `E1`$[$`E2`$/$`x`$]$. Example output: the formulation of the cast language for STLC in λ-Prolog (only some relevant rules are shown).

ing, can never trigger a cast error. To ensure the blame theorem, extra care must be paid to the way types are placed in the newly introduced casts. To check this, we present a *variance checker* that guarantees that types are placed in a way that preserves subtyping.

Thanks to this and further techniques, we can provide a methodology for the generation of the whole cast language starting from a fully-static language.

An Algorithm The second contribution is to make the methodology fully automatic. The inputs and outputs of the algorithm are languages (syntax, type system, and reduction rules) represented as logic programs (specifically, the intuitionistic theory of Harrop formulae). The algorithm automatically applies our methodology by transforming logic programs, and ultimately produces the complete specification of the language with explicit casts. We have proved that the languages that our algorithm produces always satisfy the key correctness criteria of gradual typing [28].

Correct Gradual Typing for Expressive Languages We demonstrate the generality of our approach by applying it to a language with polymorphism, recursive types, exceptions, letrec, and several other features. We report on a mechanized proof, in Abella [3], that the resulting cast language satisfies the correctness criteria of Siek et al. [28]. In particular, this is the first proof of the gradual guarantee for a language with polymorphism and recursive types.

Implementation: A Tool for Generating the Dynamic Semantics We have implemented the algorithm. The program takes specifications of the fully-static language written in λ-Prolog and generates the λ-Prolog specification of its language with casts. Figure 1 shows an example input and an excerpt of the output. The input is a call-by-value simply-typed lambda calculus. As λ-Prolog specifications are executable, the output of our tool is an interpreter that is ready to run programs.

In summary, this paper makes the following contributions.

1. We present a methodology for generating the intermediate language with casts from a static language (Section 3, 4, and 5).
2. We show the applicability of our methodology with an in-depth example. We gradualize an expressive programming language that includes polymorphism, recursive types and exceptions (Section 6).
3. We describe an algorithm that automates the methodology (Section 8). We provide an implementation that accepts and produces language definitions expressed in λ-Prolog (Section 10).
4. We validate our methodology by proving that the languages generated by our algorithm always satisfy the key correctness criteria for gradual typing (Section 9).

The implementation can be found at the following repository:
`https://github.com/mcimini/GradualizerDynamicSemantics`
The repository also contains the mechanized proof that the language mentioned above in item 2 satisfies the correctness criteria of gradual typing.

2. Overview of Gradual Typing

The gradualization of a language starts from a fully-static language and consists of deriving two more languages: the gradually typed language and the cast language. (When starting from a dynamically typed language, there is the additional first step of designing the fully-static type system, but that step is not within the scope of this

Syntax

Basic Types	B	::=	$\texttt{Int}$
Types	T	::=	$B \mid T \to T$
Terms	e	::=	$n\ (\in \mathbb{N}) \mid x \mid \lambda x{:}T.\, e \mid e\ e$
Values	v	::=	$n \mid \lambda x{:}T.\, e$
Contexts	E	::=	$E\ e \mid v\ E$

$\boxed{\Gamma \vdash e : T}$

$$\Gamma \vdash n : \texttt{Int} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash (\lambda x{:}T_1.\, e) : T_1 \to T_2}$$

$$\frac{\Gamma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2}$$

$\boxed{e \longrightarrow_{\mathrm{S}} e}$

$$\frac{}{(\lambda x{:}T.\, e)\ v \longrightarrow_{\mathrm{S}} e[v/x]} \qquad \frac{e \longrightarrow_{\mathrm{S}} e'}{E[e] \longrightarrow_{\mathrm{S}} E[e']}$$

Figure 2. The Simply Typed Lambda Calculus (STLC).

Syntax

Ground	G	::=	$B \mid \star \to \star$
Types	T	::=	$\ldots \mid \star$
Terms	e	::=	$\ldots \mid e : T \Rightarrow^{\ell} T \mid \texttt{blame}_T\, \ell$
Values	v	::=	$\ldots \mid v : G \Rightarrow^{\ell} \star$
			$\mid v : T_1 \to T_2 \Rightarrow^{\ell} T_1' \to T_2'$
Contexts	E	::=	$\ldots \mid E : T \Rightarrow^{\ell} T$

Abbrv: $(e : T_1 \Rightarrow^{\ell_1} T_2) : T_2 \Rightarrow^{\ell_2} T_3 \equiv e : T_1 \Rightarrow^{\ell_1} T_2 \Rightarrow^{\ell_2} T_3$

$\boxed{T \sim T}$ Consistency

$$\texttt{Int} \sim \texttt{Int} \qquad T \sim \star \qquad \star \sim T \qquad \frac{T_1 \sim T_3 \qquad T_2 \sim T_4}{T_1 \to T_2 \ \sim\ T_3 \to T_4}$$

$\boxed{\Gamma \vdash e : T}$

Typing rules in Figure 2, and

$$\frac{\Gamma \vdash e : T_1 \qquad T_1 \sim T_2}{\Gamma \vdash (e : T_1 \Rightarrow^{\ell} T_2) : T_2} \quad \text{(T-CAST)}$$

$$\frac{}{\Gamma \vdash \texttt{blame}_T\, \ell : T} \quad \text{(T-BLAME)}$$

$\boxed{e \longrightarrow e}$

$\longrightarrow_{\mathrm{S}}$ imported for $\longrightarrow$, Rules in Figure 4, and

$$v : B \Rightarrow^{\ell} B \longrightarrow v \quad \text{(ID-BASE)}$$

$$(v_1 : T_1 \to T_2 \Rightarrow^{\ell} T_3 \to T_4)\ v_2 \longrightarrow (v_1\ (v_2 : T_3 \Rightarrow^{\ell} T_1)) : T_2 \Rightarrow^{\ell} T_4 \quad \text{(C-BETA)}$$

Figure 3. The Cast Calculus (CC) extends the STLC.

paper.) In this section we review the gradualization of the simply-typed lambda calculus (STLC).

STLC We reproduce the definition of the STLC in Figure 2. The $\texttt{Int}$ type serves as the base case for types. We write $e \longrightarrow_{\mathrm{S}} e'$ for the reduction of terms in the STLC.

The Gradually Typed Lambda Calculus (GTLC) GTLC [24] is the gradually typed language associated to the STLC. GTLC extends STLC with the dynamic type $\star$. Its type system takes into account the interaction between $\star$ and the other types of the language. For example, the program $(\lambda x{:}\star.\, x)\ 4$ is well-typed in GTLC because a function that accepts an argument of type dynamic also accepts integers as arguments. GTLC handles this and similar situations with the help of the consistency relation $\sim$. This paper does not focus on the gradually typed surface language, so we do not review the syntax or type system of the GTLC. However, the consistency relation also plays a role in the cast language.

The Cast Language The dynamic semantics of GTLC is defined by translation to a *cast language*, that is, a language with explicit casts. This translation is called *cast insertion*. We use the cast language of Siek et al. [28] because it respects all the criteria of gradual typing. Figure 3 shows this cast language, which we simply call Cast Calculus (CC). The CC augments the STLC with a cast expression $e : T_1 \Rightarrow^{\ell} T_2$, where ℓ denotes a label. Labels identify which cast failed; typically a compiler would use source line numbers as labels. CC also extends STLC with the blame expression $\texttt{blame}_T\, \ell$. This expression represents a cast failure and the label ℓ identifies which cast failed. In most formulations, the blame expression can have any type. Here we instead use a type-annotated form $\texttt{blame}_T\, \ell$ to prescribe the type of the blame expression. This is needed because we rely on type-uniqueness in our proof of an important correctness criterion called *the gradual guarantee*.

All casts to and from the dynamic type are factored through ground types (G). For example, integers can be directly injected and projected to $\star$ but function types factor through their ground type $\star \to \star$. This choice also drives the value forms in the cast language. For example, the following are all value forms:

- $4 : \texttt{Int} \Rightarrow^{\ell} \star$
- $(\lambda x{:}\star.\, x) : \star \to \star \Rightarrow^{\ell} \star$
- $(\lambda x{:}\texttt{Int}.\, x) : \texttt{Int} \to \texttt{Int} \Rightarrow^{\ell} \star \to \star : \star \to \star \Rightarrow^{\ell} \star$

Figure 3 defines an abbreviation for a sequence of two casts, so the last value can instead be written as

$$(\lambda x{:}\texttt{Int}.\, x) : \texttt{Int} \to \texttt{Int} \Rightarrow^{\ell} \star \to \star \Rightarrow^{\ell} \star$$

The type system of CC is shown in Figure 3 and extends that of STLC with the typing rule for the cast and blame operators. Figure 3 also defines the consistency relation $\sim$, which is employed in (T-CAST). In particular, in a cast $e : T_1 \Rightarrow^{\ell} T_2$ we require that T_1 and T_2 are consistent. This simplifies the dynamic semantics of CC[1]. Consistency is not used anywhere else in the type system of CC. For example, while $(\lambda x{:}\star.\, x)\ 4$ is well-typed in GTLC it is not in CC because $\texttt{Int}$ does not coincide with $\star$.

The dynamic semantics of CC extends that of STLC with additional reduction rules to handle casts. We divide these reduction rules into two categories. The first category comprises the reduction rules that are specific to CC. These rules are shown in Figure 3. Rule (ID-BASE) handles identity casts on base types. These casts simply succeed and leave the value unchanged. Rule (C-BETA) handles the application of casted functions in the way that we reviewed in the introduction.

The second category singles out the reduction rules that are language independent and that pertain casts. Figure 4 shows this kind of rules, which we call *cast handler reduction rules*.

Rule (ID-STAR) handles the identity cast on $\star$. Rule (SUCCEED) handles a cast of a ground type to and from $\star$. This cast leaves the value unchanged. Rule (FAIL) handles when a ground type is cast to $\star$ and then to a different ground type, which triggers blame.

[1] Alternatively, we could relax the restriction on the consistency of T_1 and T_2 at the price of adding another error transition [26].

$\boxed{e \longrightarrow e}$

$$v : \star \Rightarrow^{\ell} \star \longrightarrow v \qquad \text{(ID-STAR)}$$

$$v : G \Rightarrow^{\ell_1} \star \Rightarrow^{\ell_2} G \longrightarrow v \qquad \text{(SUCCEED)}$$

$$v : G_1 \Rightarrow^{\ell_1} \star \Rightarrow^{\ell_2} G_2 \longrightarrow \texttt{blame}_{G_2}\, \ell_2 \quad \text{if } G_1 \neq G_2 \qquad \text{(FAIL)}$$

$$v : T \Rightarrow^{\ell} \star \longrightarrow v : T \Rightarrow^{\ell} G \Rightarrow^{\ell} \star \quad \text{if } T \neq \star, T \neq G, T \sim G \qquad \text{(GROUND)}$$

$$v : \star \Rightarrow^{\ell} T \longrightarrow v : \star \Rightarrow^{\ell} G \Rightarrow^{\ell} T \quad \text{if } T \neq \star, T \neq G, T \sim G \qquad \text{(EXPAND)}$$

$$\frac{\emptyset \vdash E[\texttt{blame}_{T_1}\, \ell] : T_2}{E[\texttt{blame}_{T_1}\, \ell] \longrightarrow \texttt{blame}_{T_2}\, \ell} \qquad \text{(CTX-BLAME)}$$

Figure 4. Cast handler reduction rules

Rules (GROUND) and (EXPAND) factor function casts through the ground type $\star \to \star$. These rules do not mention function types explicitly because their formulations are meant to be general and accommodate languages with more type constructors than just function types. Rule (GROUND) applies at injecting a value of type T into $\star$ when T is not $\star$ and is not a ground type (those cases are handled by other rules). Therefore, (GROUND) applies only when function types that are not $\star \to \star$ are injected into $\star$. In this case, we insert an intermediate cast to a ground type G. As G is consistent with the function type, it must be the unique ground type $\star \to \star$. The following is an example of the (GROUND) reduction.

$$(\lambda x{:}\texttt{Int}.\, x) : \texttt{Int}{\to}\texttt{Int} \Rightarrow^{\ell} \star \longrightarrow (\lambda x{:}\texttt{Int}.\, x) : \texttt{Int}{\to}\texttt{Int} \Rightarrow^{\ell} \star{\to}\star \Rightarrow^{\ell} \star$$

Rule (EXPAND) projects functions that are not of type $\star \to \star$ out of the dynamic type. This is done, as well, with an intermediate cast, as in the following reduction

$$(\lambda x{:}\texttt{Int}.\, x) : \star \Rightarrow^{\ell} \texttt{Int}{\to}\texttt{Int} \longrightarrow (\lambda x{:}\texttt{Int}.\, x) : \star \Rightarrow^{\ell} \star{\to}\star \Rightarrow^{\ell} \texttt{Int}{\to}\texttt{Int}$$

Finally, rule (CTX-BLAME) lifts the blame error to the top level. As the blame expression may replace an expression with a different type, we preserve types by changing the type annotation on the blame expression, following Siek et al. [28].

Correctness Criteria for Gradual Typing A cast language should satisfy several criteria to enable a smooth integration of the dynamic and static typing discipline [28]. We repeat them in Figure 5 and review them here.

Conservativeness is the least we should expect: that programs of STLC run in the same way in CC.

An important property of programming languages is that of type safety. In the context of cast languages this property is delicate due to the interplay between types at run-time. In particular, cast reductions must consider all possible combinations of types, so that programs do not end in untrapped errors. Furthermore, reduction rules such as (C-BETA) are typically designed out of intuition and must be checked to be type preserving.

The blame theorem [7, 31, 35] offers a connection between casts and subtyping. Subtyping in CC is characterised by reflexivity on basic types, the contravariance for function types and the rule

$$\frac{T <: G}{T <: \star}$$

which prescribes when types are subtype of $\star$. This rule entails that basic types are subtype of $\star$. More complex types, such as function types, are subtype of $\star$ only when they are subtype of their corresponding ground type. This prevents deriving $\texttt{Int} \to \star <: \star$. Indeed, $\texttt{Int} \to \star \not<: \star \to \star$ because $\star \not<: \texttt{Int}$. The blame theorem guarantees that a cast $e : T_1 \Rightarrow^{\ell} T_2$ that respects subtyping $<:$, i.e. for which $T_1 <: T_2$, never fails at run-time. When we translate gradual programs from GTLC to CC, statically typed regions of code that interact with dynamically typed ones can only introduce safe casts. Therefore, the blame theorem guarantees that statically typed code is never blamed, a fundamental property for gradually typed languages [7, 31, 35].

Correctness Criteria [28]

Conservative Extension:
for all static e and e', $e \longrightarrow_{\mathrm{S}} e'$ iff $e \longrightarrow e'$

Type Safety:
both progress and type preservation hold.

Blame Theorem:
for all typed e, for all T and ℓ,
$e \longrightarrow^{*} \texttt{blame}_T\, \ell$ implies that ℓ is not in a safe cast of e.

Gradual Guarantee: for all typed $e_1 \sqsubseteq e_2$,
(1) if $e_2 \longrightarrow e_2'$ then $e_1 \longrightarrow^{*} e_1'$ and $e_1' \sqsubseteq e_2'$.
(2) if $e_1 \longrightarrow e_1'$ then
either $e_2 \longrightarrow^{*} e_2'$ and $e_1' \sqsubseteq e_2'$,
or $e_2 \longrightarrow^{*} \texttt{blame}_T\, \ell$.

Figure 5. Correctness Criteria for the Cast Language

The gradual guarantee is an operational semantics simulation that respects a monotonicity property w.r.t. the *precision* relation $\sqsubseteq$. Roughly, a program is less precise than another whenever the former contains more occurrences of the dynamic type. For example, we have $(\lambda x{:}\star.\, x)\ 4 \sqsubseteq (\lambda x{:}\texttt{Int}.\, x)\ 4$. However, the precision relation also derives $4 : \texttt{Int} \Rightarrow^{\ell} \star \sqsubseteq 4$ because $4 : \texttt{Int} \Rightarrow^{\ell} \star$ is a less precise (or a more dynamically typed) version of 4. Therefore, the precision relation accommodates for extra casts on the left and, similarly, on the right.

Part 1 of the gradual guarantee says that every step of the more precise program can be simulated with zero or more steps by the less precise program. This means that removing type annotations from a program that returns a value cannot make the program crash nor return a different value. The only difference is that the program may execute slower because of the presence of more casts. Part 2 says that adding type annotations can only have two possible outcomes: either the program behaves the same and returns an equivalent result or it triggers a cast error. The latter can happen because the type annotation might have been mistaken.

The correctness criteria are central for enabling a healthy interaction of the static and dynamic typing disciplines and the evolution of code from scripts to programs. In the following section, we present a methodology for generating cast languages that automatically respect these criteria.

3. A Methodology to Generate Cast Languages

The methodology starts from the fully-static language and is devised as a series of steps. Each step is described in detail in the subsequent sections. We list them below for reference.

1. Extend the input language to its *cast language template*, adding ground types, cast values, the typing rules (T-CAST) and (T-BLAME), (ID-BASE), and the cast handler reduction rules of Figure 4. Also, add casts to the evaluation contexts.
2. Classify the *eliminators* for higher-order types of the input language (such as application, `fst` and `snd`).

3. For each eliminator of a higher-order type, apply the following steps.
   (a) Building Bridges: Create the cast value to be eliminated and build the bridges (pairs of types) between the types of the value and those of the cast value.
   (b) Generating Cast Reductions: Generate a reduction rule whose left-hand side is the elimination form with the cast value of step 3(a) being placed at eliminated argument. The right-hand side is the expression that inserts casts and replaces types to the left-hand side as prescribed by the bridges. (See Section 3.4 for all the details).
   (c) Inherit valuehood conditions for the reduction rule.

We describe the steps of the methodology in detail.

3.1 Generate the Cast Language Template

We call *higher-order types* those types that have one or more (inductive) arguments such as the function type $T_1 \to T_2$, pairs $T_1 \times T_2$, the sum type $T_1 + T_2$, lists $\texttt{List}\ T$, and so on.

We adopt the style of dynamic semantics where casts from higher-order types to the dynamic type go through an intermediate cast to a *ground type*. This coincides with viewing the dynamic type as the recursive type $\mu X.\ \texttt{Int} + (X \to X)$ (Harper [13]), or equivalently, as the solution of the recursive equation $\star = \texttt{Int} + (\star \to \star)$. The following defines ground types for a given language.

Definition 1 (Ground types)**.** *Ground types consist of the basic types and the higher-order types where the dynamic type appears in all arguments.*

For example, in a language with integers, booleans, functions, pairs, sums, and lists, the ground types are defined as follows.

$$\text{Ground Types}\quad G ::= \texttt{Int} \mid \texttt{Bool} \mid \star\to\star \mid \star\times\star \mid \star+\star \mid \texttt{List}\ \star$$

Values of higher-order types are not allowed to jump from one world to another freely but only through ground types.

The cast language introduces new values, which we call *cast values*, defined as follows.

Definition 2 (Cast values)**.** *Values that are cast from a ground type to the dynamic type are cast values, these are the canonical forms of the type $\star$. Values of higher-order types that are cast to other higher-order types with the same top level type constructor are cast values, these are (additional) canonical forms of the higher-order type.*

The grammar for value forms is then extended accordingly. To accommodate the additional canonical forms for higher-order types, we cast a value meta-variable (v) with types built with distinct variables. For example, the following are the values of a cast language with functions, pairs, sums, and lists.

$$\begin{aligned}\text{Values}\quad v ::= \ldots &\mid v : G \Rightarrow^{\ell} \star\\ &\mid v : T_1 \to T_2 \Rightarrow^{\ell} T_1' \to T_2'\\ &\mid v : T_1 \times T_2 \Rightarrow^{\ell} T_1' \times T_2'\\ &\mid v : T_1 + T_2 \Rightarrow^{\ell} T_1' + T_2'\\ &\mid v : \texttt{List}\ T \Rightarrow^{\ell} \texttt{List}\ T'\end{aligned}$$

The *cast language template* is created as follows.

> *Extend the original language with the cast operator and the blame, add ground types and add cast values to the value forms of the language. Extend the original type system with* (T-CAST) *for casts, and* (T-BLAME) *for blame, as defined in Figure 3. Extend the dynamic semantics with* (ID-BASE) *and the cast handler reduction rules of Figure 4. Add the cast operator to the evaluation contexts.*

3.2 Classify Eliminators of Higher-Order Types

An eliminator is an operator that manipulates or inspects values of a certain type constructor. The typing rule of an eliminator assigns the type they eliminate to one of their argument. For example, we have highlighted the eliminated type in the following two eliminators: the application and `head` forms.

$$\frac{\Gamma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash \underline{e_1}\ e_2 : T_2} \qquad \frac{\Gamma \vdash e : \texttt{List}\ T}{\Gamma \vdash \texttt{head}_T\ \underline{e} : T}$$

We refer to the argument that is typed at the highlighted type as the *eliminated argument*. We have underlined the eliminated arguments above. We refer to arguments such as the second argument of application (e_2) as *sibling arguments*. Reduction rules of eliminators syntactically pattern-match values at the eliminated argument. For example, the β-reduction rule handles the only canonical form for the function type ($\lambda x{:}T.\ e$) and the reduction rules of `head` handle the two canonical forms for lists (`nil` and $\texttt{cons}\ v_1\ v_2$).

$$\begin{aligned}(\underline{\lambda x{:}T.\ e})\ v &\longrightarrow e[v/x]\\ \texttt{head}\ \underline{\texttt{nil}} &\longrightarrow \texttt{error}\\ \texttt{head}\ (\underline{\texttt{cons}\ v_1\ v_2}) &\longrightarrow v_1\end{aligned}$$

Why Do Eliminators Matter to Us? Answer: Retaining Progress As we have seen, the cast language introduces new canonical forms for higher-order types. These forms do not yet have associated reduction rules, which puts the progress theorem in jeopardy.

For example, when we place a higher-order cast at the eliminated argument of a function application

$$(\underline{(\lambda x{:}T_1'.\ e) : T_1' \to T_2' \Rightarrow^{\ell} T_1 \to T_2})$$

the β rule is not applicable, and since the cast expression is a value it does not contain any redexes. Therefore, the above expression is stuck. Of course, this failure is not specific to function application, but applies to all eliminators of higher-order types when we switch to the cast language. To retain progress, we must add reduction rules to handle the new canonical forms.

Only Eliminators of Higher-Order Types Matter Eliminators of basic types might have cast expressions at their eliminated argument. Those casts, however, cannot be values (as there are no cast values of basic types). Therefore they eventually reduce to one of the original canonical forms, where the native reduction applies. Hence, the progress theorem is not jeopardized for eliminators of basic types. Examples of eliminators of basic types are `if` for booleans and `pred` (predecessor) for integers.

Other reduction rules are not affected by the presence of casts. Consider for example the reduction rules for `fix` and `let`.

$$\begin{aligned}\texttt{fix}\ v &\longrightarrow v\ (\texttt{fix}\ v)\\ \texttt{let}\ x = v\ \texttt{in}\ e &\longrightarrow e[v/x]\end{aligned}$$

These operators do not pattern-match a canonical form on the left-hand side of their reduction rules, so additional reduction rules are not needed for them. In short: *only eliminators of higher-order types need an additional reduction rule*. The next sections describe how to build these rules.

3.3 Building Bridges

Given an eliminator of a higher-order type, apply the following.

> *In the typing rule for the eliminator, identify the type of the eliminated argument. Create a cast value of that type, casting a fresh value meta-variable (v) whose type is also*

fresh. Deduce bridges (pairs of types) from the cast by applying "zip" to the source and target of the cast.

As an example of the bridge construction step, we apply it to the application form starting from its typing rule.

$$\frac{\Gamma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2}$$

The type of the eliminated argument is $T_1 \to T_2$, so we create a cast value of that type applied to the fresh v_1:

$$v_1 : T_1' \to T_2' \Rightarrow^{\ell} T_1 \to T_2 \tag{1}$$

(The type of v_1 is fresh, say T', but constrained to be a function type to form a cast value, hence $T' = T_1' \to T_2'$.) Finally, we zip the source and target types, $T_1' \to T_2'$ and $T_1 \to T_2$, to obtain the bridges:

$$\{T_1' \frown T_1, T_2' \frown T_2\} \tag{2}$$

The notation $T_1' \frown T_1$ denotes a bridge from T_1' to T_1. As we shall see in the next section, bridges also work on the opposite direction.

3.4 Generating Cast Reductions

This step generates a cast reduction rule starting from the cast value and bridges produced in the previous step and referring to the typing rule for the eliminator.

The left-hand side of the reduction rule is an instance of the elimination form with the eliminated argument being the cast value from the previous step and with the sibling arguments obtained from those in the typing rule. To generate the right-hand side of the reduction rule, apply the following steps to the left-hand side:

(a) remove the cast that is around the cast value.

(b) wrap the sibling arguments with casts, using their types in the typing rule and following bridges in reverse.

(c) wrap the entire expression with a cast that uses the assigned type of the typing rule and follows (forward) bridges.

(d) type annotations are replaced by the type on the other side of their bridge.

(e) if the typing rule types an expression e with the type environment $\Gamma, x : T$ and $T' \frown T$ then e replaces x with the cast $x : T' \Rightarrow^{\ell} T$.

Continuing our example of function application, we build the following left-hand side of the reduction rule.

$$((v_1 : T_1' \to T_2' \Rightarrow^{\ell} T_1 \to T_2)\ e_2) \longrightarrow ?$$

We obtain e_2 from the typing rule of the function application. Using the siblings of the typing rule gives a direct link to the types used in the cast value and bridges. We can lookup the types of siblings in the typing rule and produce casts whose types align with those being used in the reduction rule.

Next we build the right-hand side. We apply step (a) and remove the cast on the eliminated argument.

$$(\ v_1\ \ e_2)$$

Applying step (b), we need to wrap the sibling e_2 with a cast. Referring to the typing rule for application, we have e_2 at type T_1, so we find the bridge that targets T_1, which is $T_1' \frown T_1$, and cast e_2 in the reverse direction.

$$(v_1\ \ (e_2 : T_1 \Rightarrow^{\ell} T_1')\ )$$

Finally, applying step (c), we wrap the entire expression in a cast. Referring to the typing rule, we have that the assigned type for the entire expression is T_2. As we have the bridge $T_2' \frown T_2$, we insert the following cast (in the forward direction).

$$(v_1\ (e_2 : T_1 \Rightarrow^{\ell} T_1'))\ : T_2' \Rightarrow^{\ell} T_2$$

The steps (d) and (e) do not apply to this example. Thus, we have completed the generation of the reduction rule:

$$\begin{array}{c}((v_1 : T_1' \to T_2' \Rightarrow^{\ell} T_1 \to T_2)\ \ e_2) \\ \longrightarrow \\ (v_1\ \ (e_2 : T_1 \Rightarrow^{\ell} T_1'))\ : T_2' \Rightarrow^{\ell} T_2\end{array} \qquad \text{(C-BETA}')$$

Note that we are not quite finished with this reduction rule. In Section 3.5 we show how to infer that the argument e_2 needs to be a value. Below, we explain the motivations behind the above steps and give examples that make use of steps (d) and (e).

(a) Removing the Cast From the Eliminated Argument The reason why we liberate v_1 from the cast is because we want the reduction to produce an expression that can eventually be used with one of the original reduction rules of the eliminator. It is easy to see that (C-BETA′) takes to an expression that *could* make β fire. It may, as well, take to an expression $(v_1\ (e_2 : T_1 \Rightarrow^{\ell} T_1'))\ : T_2' \Rightarrow^{\ell} T_2$ in which v_1 contains another top level cast. In this case, the same rule applies and consumes one more cast. As there can be only a finite number of wrapped casts we are guaranteed to end up to a form that can be used by β. This scenario is prescribed by the gradual guarantee: less precise programs might be slower for dealing with more casts but eventually *catch up* and simulate the more precise program. The reduction rules that we generate realize precisely this scenario, mechanically, and for all the eliminators that need it.

(b) Casts On the Siblings, Bridging in Reverse After the cast has been removed from the eliminated argument, the resulting expression is likely to be ill-typed. This is because the eliminated argument and its siblings are typed using a different set of types. For example, the intermediate expression after we apply step (a) is $(v_1\ e_2)$ and the type of v_1 uses T_1' and T_2' whereas the type of e_2 uses T_1. To form a well-typed expression, the sibling arguments need to align with using the types used by the value. Therefore, we apply casts to them and the direction of these casts corresponds with the reversed direction of the bridges. In our example, we have to wrap e_2 with the cast $(e_2 :\ T_1 \Rightarrow^{\ell} T_1'\ )$.

(c) Casts Around the Entire Expression After we have applied steps (a) and (b), the intermediate expression has a type that comes from the source of the cast value. For example, we had the intermediate expression $(v_1\ \ (e_2 : T_1 \Rightarrow^{\ell} T_1'))$ with type T_2', and T_2' was in the source of the cast in (1). However, the reduction rule that we generate must be type preserving w.r.t. the left-hand side $((v_1 : T_1' \to T_2' \Rightarrow^{\ell} T_1 \to T_2)\ \ e_2)$, which uses types from the target of the cast in (1). So we need to align the intermediate expression to use the types of the target, or type preservation would be jeopardized. We achieve this with a cast that follows the direction of the bridges. In our example, we apply the cast $(v_1\ (e_2 : T_1 \Rightarrow^{\ell} T_1'))\ :\ T_2' \Rightarrow^{\ell} T_2$.

(d) Replace Type Annotations Some elimination forms carry type annotations. Let us consider a type-annotated `head` operation for lists,[2] with the following typing and reduction rules.

$$\frac{\Gamma \vdash\ e : \texttt{List}\ T}{\Gamma \vdash\ \texttt{head}[T]\, e : T} \qquad \text{(T-HEAD)}$$

$$\texttt{head}[T](\texttt{cons}[T]\, v_1\, v_2) \longrightarrow v_1 \qquad \text{(R-HEAD)}$$

[2] When using our methodology, type annotations may appear often in the cast language because we rely on type-uniqueness to ensure the gradual guarantee (see discussion on Section 5).

The cast value for $\texttt{List}\ T$ is

$$v : \texttt{List}\ T' \Rightarrow^{\ell} \texttt{List}\ T$$

and the set of bridges is the singleton $\{T_1' \frown T_1\}$. Applying the prescribed transformations, including step (d), we obtain the reduction rule

$$\begin{array}{c} \texttt{head}[T]\,(v : \texttt{List}\ T' \Rightarrow^{\ell} \texttt{List}\ T) \\ \longrightarrow \\ (\texttt{head}[\,T'\,]\,v) : T' \Rightarrow^{\ell} T \end{array} \quad \text{(C-HEAD)}$$

The type annotation must be replaced because v is of type $\texttt{List}\ T'$, therefore $\texttt{head}[T]\,v$ is not well-typed. As expected, the produced reduction step takes to a form that could be used by (R-HEAD).

(e) Types in the Type Environment Lead to Substitution Consider the sum type and the typing rule of its eliminator `case`.

$$\frac{\Gamma \vdash e_1 : T_1 + T_2 \qquad \Gamma, x : T_1 \vdash e_2 : T \qquad \Gamma, x : T_2 \vdash e_3 : T}{\Gamma \vdash (\texttt{case}\ e_1\ \texttt{of inl}\ x \Rightarrow e_2 \mid \texttt{inr}\ x \Rightarrow e_3) : T} \quad \text{(T-CASE)}$$

The cast value for the sum $T_1 + T_2$ is

$$v : T_1' + T_2' \Rightarrow^{\ell} T_1 + T_2$$

So we build the following bridges:

$$\{T_1' \frown T_1, T_2' \frown T_2\}$$

The typing rule (T-CASE) entails that e_2 and e_3 expect to receive values of type T_1 and T_2, respectively. But when we remove the cast from the cast value, obtaining

$$\texttt{case}\ v\ \texttt{of inl}\ x \Rightarrow e_2 \mid \texttt{inr}\ x \Rightarrow e_3$$

we have a mismatch to fix because v is of type $T_1' + T_2'$. Thus, we apply the substitution $[x : T_1' \Rightarrow^{\ell} T_1/x]$ to e_2 and the substitution $[x : T_2' \Rightarrow^{\ell} T_2/x]$ to e_3. So we obtain the following reduction rule.

$$\begin{array}{l} \texttt{case}\,(v : T_1' + T_2' \Rightarrow^{\ell} T_1 + T_2)\ \texttt{of inl}\ x \Rightarrow e_2 \mid \texttt{inr}\ x \Rightarrow e_3 \\ \quad\longrightarrow \\ \quad \texttt{case}\ v\ \texttt{of} \\ \qquad \texttt{inl}\ x \Rightarrow e_2[x : T_1' \Rightarrow^{\ell} T_1/x] \\ \qquad \mid \texttt{inr}\ x \Rightarrow e_3[x : T_2' \Rightarrow^{\ell} T_2/x] \end{array} \quad \text{(C-CASE)}$$

Ensuring that Casts are Determined Next we discuss the need for a natural restriction on the typing rules for eliminators to ensure that, in the generated cast reduction rule, all the types on the right-hand side come from types that appear in the left-hand side.

Let us consider the following imaginary operator `sndReplace`.

$$\frac{\Gamma \vdash e_1 : T_1 \times T_2 \qquad \Gamma \vdash e_2 : T_3}{\Gamma \vdash (\texttt{sndReplace}\ e_1\ e_2) : T_1 \times T_3} \quad \text{(T-SNDREP)}$$

$$(\texttt{sndReplace}\ (\langle v_1, v_2\rangle)\ v_3) \longrightarrow \langle v_1, v_3\rangle$$

This operator takes two arguments, a pair and an expression, and replaces the second element of the pair with the expression. Consider now the cast value $v : T_1' \times T_2' \Rightarrow^{\ell} T_1 \times T_2$ at the eliminated argument. Our transformations compute the following reduction rule.

$$\begin{array}{c} (\texttt{sndReplace}\ (v : T_1' \times T_2' \Rightarrow^{\ell} T_1 \times T_2)\ e_2) \\ \longrightarrow \\ (\texttt{sndReplace}\ v\ e_2) : T_1' \times \boxed{T_3} \Rightarrow^{l} T_1 \times \boxed{T_3} \end{array}$$

The problem here is that the right-hand side of the reduction rule mentions the type T_3, which does not appear in the left-hand side. This jeopardizes type preservation.

To ensure that casts on the right-hand side only refer to types from the left-hand side, we define when *casts are determined* for a typing rule. Intuitively, casts are not determined for (T-SNDREP) because the eliminated argument is typed at $T_1 \times T_2$ and the type of the entire expression is $T_1 \times \boxed{T_3}$, which mixes a type of the eliminated argument with a type T_3 from somewhere else. Mixing the two leads to undetermined casts: because the type of the entire expression mentions type T_1, we generate a cast on the right-hand side of the reduction, and doing so T_3 must also appear in the cast, but it is not present in the left-hand side.

Sibling arguments also may induce casts according to our transformations, so they lead to the same issue when they mix types of the eliminated argument with types from somewhere else.

When siblings and the type of the entire expression make use of types that are *exclusively* types of the eliminated argument, the casts produced are always fine. For example, in the typing rule for the application $(e_1\ e_2)$ we have the eliminated argument e_1 at type $T_1 \to T_2$, the sibling e_2 at T_1, and that the type of the entire expression is T_2. As we have seen, all the produced casts in (C-BETA) use types from the left-hand side. No issues appear also when siblings and the assigned type do not mention the types of the eliminated argument at all. For example, the rule (T-CASE) for $(\texttt{case}\ e_1\ \texttt{of inl}\ x \Rightarrow e_2 \mid \texttt{inr}\ x \Rightarrow e_3)$ types e_1 at $T_1 + T_2$, but types e_2 and e_3 at T, and also the assigned type is T. As these types do not mention any type of $T_1 + T_2$, they do not produce casts.

In short: our transformations lead to determined casts only when the types of the eliminated argument are not mixed with types from somewhere else in the types of siblings and in the type assigned to the entire expression. Our methodology accepts typing rules according to the following static check.

Definition 3. *Given a typing rule of an eliminator, we say that* casts are determined *whenever its sibling arguments and the entire expression are such that their types either contains only types from the type of the eliminated argument or contains no types from it.*

Handling the Tension Between Identity and Proxies In imperative languages with notions of observable object identity, the cast values introduced by CC, which can be viewed as a kind of proxy, represent a challenge. The standard semantics for casts does not preserve object identity [9, 32, 34]. For example, consider the reduction rules

$$\begin{array}{ll} v == v \longrightarrow \texttt{true} & \\ v == v' \longrightarrow \texttt{false} & \text{if } v \neq v' \end{array}$$

Then, given the pair $p = \langle 3, 4\rangle$, we have

$$p == p \longrightarrow \texttt{true}$$

but with the less precise $p' = p : \texttt{Int} \times \texttt{Int} \Rightarrow^{\ell_1} \star \times \texttt{Int} \Rightarrow^{\ell_2} \texttt{Int} \times \texttt{Int}$, we have

$$p' == p \longrightarrow \texttt{false}$$

This contradicts the gradual guarantee: a less precise program evaluates to a completely different value. This issue only concerns higher-order types because there are no cast values at basic types.

There are recent proposals for making proxies transparent with respect to identity [19], but incorporating such proposals into our methodology is beyond the current scope. In this paper we rule out languages with observable notions of object identity by placing a restriction on reduction rules. We require that *a meta-variable may appear no more than once on the left-hand side of a reduction rule*. For example, the reduction rule

$$v == v \longrightarrow \texttt{true}$$

is not allowed because the meta-variable v appears twice on the left-hand side.

3.5 Inheriting Valuehood Conditions for Fireability

Inherit the valuehood conditions from the native reduction rules in the new reduction rules for casts.

For example, from the call by value β-rule we inherit that also the generated reduction rule requires that the second argument be a value. That is, (C-BETA′) is transformed to

$$\begin{array}{c}((v_1 : T_1' \to T_2' \Rightarrow^\ell T_1 \to T_2)\ \ v_2) \\ \longrightarrow \\ (v_1\ \ (v_2 : T_1 \Rightarrow^\ell T_1')) : T_2' \Rightarrow^\ell T_2\end{array} \quad \text{(C-BETA)}$$

which is the reduction rule that we add to the cast language. As expected, this rule coincides with the reduction found in the literature.

4. Type Variance Matters for the Blame Theorem

Safe casts are those that respect subtyping, that is, $e : T_1 \Rightarrow^\ell T_2$ is safe whenever $T_1 <: T_2$. The blame theorem guarantees that safe casts never fail at run-time [7, 31, 35]. The key to proving the blame theorem is to show that expressions with safe casts only reduce to expressions with safe casts. When generating reduction rules we must take care to guarantee this property. To help our presentation, in this section we decorate types with the superscripts $^{(+)}$ and $^{(-)}$ when types appear in covariant and contravariant positions, respectively. Plain occurrences of variables are used positively and are labeled as covariant. Consider the following erroneous example.

$$\frac{\Gamma \vdash e_1 : T_1^{(+)} \times T_2^{(+)} \qquad \Gamma \vdash e_2 : T_1^{(+)}}{\Gamma \vdash (\texttt{fstEqual}\ e_1\ e_2) : \texttt{Bool}^{(+)}} \quad \text{(T-FSTEQUAL)}$$

$$(\texttt{fstEqual}\ \langle v_1, v_2\rangle\ v_3) \longrightarrow (\texttt{fst}\ \langle v_1, v_2\rangle) == v_3$$

`fstEqual` checks whether the first element of a pair e_1 is equal to e_2. When we apply our methodology, we obtain the cast value $v : T_1' \times T_2' \Rightarrow^\ell T_1 \times T_2$, bridges $\{T_1' \curvearrowright T_1, T_2' \curvearrowright T_2\}$ and ultimately the reduction rule

$$\begin{array}{c}(\texttt{fstEqual}\ (v_1 : T_1' \times T_2' \Rightarrow^\ell T_1 \times T_2)\ v_2) \\ \longrightarrow \\ (\texttt{fstEqual}\ v_1\ (v_2 : T_1 \Rightarrow^\ell T_1'))\end{array}$$

Starting from a safe cast $(v_1 : T_1' \times T_2' \Rightarrow^\ell T_1 \times T_2)$ we know that $T_1' \times T_2' <: T_1 \times T_2$ and, because of the covariance of $\times$, we also derive that $T_1' <: T_1$ and $T_2' <: T_2$. Above, we introduced a new cast $(v_2 : T_1 \Rightarrow^\ell T_1')$ that is not ensured to be safe because we do not have $T_1 <: T_1'$ but instead the opposite. The cast $(v_2 : T_1 \Rightarrow^\ell T_1')$ is *necessary* for $(\texttt{fstEqual}\ v_1\ (v_2 : T_1 \Rightarrow^\ell T_1'))$ to be well-typed. This scenario happens whenever a type is used covariantly by the type of the eliminated argument and then used covariantly by a sibling argument. To ensure type preservation, the types of the sibling must be cast to the source type of the cast value which are in the opposite direction of the available subtyping information. This scenario does not happen when types are used contravariantly by the type of the eliminated argument and covariantly by the siblings. For example, the rule for application

$$\frac{\Gamma \vdash e_1 : T_1^{(-)} \to T_2^{(+)} \qquad \Gamma \vdash e_2 : T_1^{(+)}}{\Gamma \vdash e_1\ e_2 : T_2^{(+)}}$$

types e_2 at T_1 in covariant position. When we start from a safe cast $(v_1 : T_1' \to T_2' \Rightarrow^\ell T_1 \to T_2)$ we derive $T_1' \to T_2' <: T_1 \to T_2$, and because $\to$ is contravariant we have $T_1 <: T_1'$ and $T_2' <: T_2$. Now it is easy to check that the left-hand side of (C-BETA), i.e., $(v_1\ (v_2 : T_1 \Rightarrow T_1')) : T_2' \Rightarrow T_2$, has a safe cast around v_2.

Something similar happens for the outer cast, from T_2' to T_2. As the cast around the entire expression is built from forward bridges, the variance requirement here is that the type of the entire expression (e.g. T_2) only appears in covariant positions in the type of the eliminated argument. For example, it is easy to check that the outer cast from T_2' to T_2 is safe.

Were we to change the type of the entire application expression to T_1 (which would be bizarre), then we would have T_1 contravariant in the eliminated type and covariant in the type of the entire expression. The produced reduction rule would be the following.

$$\begin{array}{c}((v_1 : T_1' \to T_2' \Rightarrow^\ell T_1 \to T_2)\ \ v_2) \\ \longrightarrow \\ (v_1\ \ (v_2 : T_1 \Rightarrow^\ell T_1')) : T_1' \Rightarrow^\ell T_1\end{array}$$

Again the outer cast *must* be there for type preservation. On the other hand, we cannot ensure that this cast is safe because we do not have $T_1' <: T_1$. Thus, it is good that our restrictions catch this scenario as an error.

Finally, some siblings are typed with an environment that contains an added assumption such as $x : T$. As we have seen, these types may induce a cast that aligns with the forward direction of the bridges. Then, they work relatively reversed to sibling arguments. Therefore, we simply consider those types to be in contravariant position, which is in many ways their natural role as they appear in negative position, i.e. on the left-hand side of the turnstyle.

Based on the above considerations, we define a static check to ensure that types occur in the typing rules of eliminators in a way that induce safe casts. Typing rules that satisfy the following conditions afford the *type variance adequacy w.r.t. the blame theorem.*

Definition 4. *Given the typing rule of an eliminator, we say that* type variance is adequate w.r.t. the blame theorem for the rule *whenever*

1. *Covariant types of the eliminated type are used only in contravariant position for typing siblings and in covariant position for typing the entire expression.*
2. *Contravariant types of the eliminated type are used only in covariant position for typing siblings and in contravariant position for typing the entire expression.*

Definition 4 allows invariant types to be used freely.

5. Type-Uniqueness and the Gradual Guarantee

Recall from Section 2 that the precision relation [28] compares expressions with others that can have extra casts on the left and on the right. For example, the following allows an extra cast on the left.

$$\frac{\Gamma_2 \vdash e_2 : T \qquad \Gamma_1, \Gamma_2 \vdash e_1 \sqsubseteq e_2 \qquad T_1 \sqsubseteq T \qquad T_2 \sqsubseteq T}{\Gamma_1, \Gamma_2 \vdash (e_1 : T_1 \Rightarrow^\ell T_2) \sqsubseteq e_2}$$

Thanks to this rule we have $(4 : \texttt{Int} \Rightarrow^\ell \star) \sqsubseteq 4$. The type of e_2 is used to constrain the source and target of the cast because we want

$$(\lambda x{:}\texttt{Int}.\, x) : \texttt{Int} \to \texttt{Int} \Rightarrow^\ell \star \to \star \not\sqsubseteq \lambda x : \star.x$$

When proving the gradual guarantee, we start from two programs $e \sqsubseteq e_2$, of which e_2 is typed at, say, T_3. When $e \sqsubseteq e_2$ is proved by the rule above we can derive that e is of the form $(e_1 : T_1 \Rightarrow^\ell T_2)$ and that e_2 has *some type* T that is more precise than T_1 and T_2. It is important, however, that T_3 as well be more precise than T_1 and T_2 because we started with steps from e_2 as typed at T_3, and those are the steps that e needs to simulate and relate by precision. To cope with this, the proof of the gradual guarantee of CC uses type-uniqueness to have $T_3 = T$ [28]. For our generalized methodology, type-uniqueness is also crucial for satisfying the gradual guarantee. Therefore we consider here only languages that afford this property. We leave it as an open problem whether the gradual guarantee can be proved without type-uniqueness.

Basic Types	B	::=	$\texttt{Bool} \mid \texttt{Int}$
Types	T	::=	$B \mid T \to T \mid T \times T \mid \texttt{List}\ T \mid T + T$
			$\mid X \mid \forall X.T \mid \mu X.T$
Expressions	e	::=	$\texttt{true} \mid \texttt{false} \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$
			$\mid \texttt{z} \mid \texttt{succ}\ e \mid \texttt{pred}\ e \mid \texttt{isZero}\ e$
			$\mid x \mid \lambda x{:}T.\, e \mid e\ e$
			$\mid \langle e, e\rangle \mid \texttt{fst}\ e \mid \texttt{snd}\ e$
			$\mid \texttt{nil}_T \mid \texttt{cons}_T\ e\ e \mid$
			$\mid \texttt{head}_T\ e \mid \texttt{tail}_T\ e \mid \texttt{isNil}_T\ e$
			$\mid \texttt{inl}_T\ e \mid \texttt{inr}_T\ e \mid \texttt{case}\,(x)\,e\,e\,e$[3]
			$\mid \Lambda X.\, e \mid e\ [T]$
			$\mid \texttt{fold}_T\ e \mid \texttt{unfold}\ e$
			$\mid \texttt{fix}\ e \mid \texttt{letrec}_T\ x = e\ \texttt{in}\ e$
			$\mid \texttt{raise}_T\ e \mid \texttt{try}\ e\ \texttt{with}\ e$
Values	v	::=	$\texttt{true} \mid \texttt{false}$
			$\mid \texttt{z} \mid \texttt{succ}\ v$
			$\mid \lambda x{:}T.\, e$
			$\mid \langle v, v\rangle$
			$\mid \texttt{nil}_T \mid \texttt{cons}_T\ v\ v$
			$\mid \texttt{inr}_T\ v \mid \texttt{inl}_T\ v$
			$\mid \Lambda X.\, e$
			$\mid \texttt{fold}_T\ v$
Errors	er	::=	$\texttt{raise}_T\ v$
Contexts	E	::=	$\texttt{if}\ E\ \texttt{then}\ e\ \texttt{else}\ e$
			$\mid \texttt{succ}\ E \mid \texttt{pred}\ E \mid \texttt{isZero}\ E$
			$\mid E\ e \mid v\ E$
			$\mid \langle E, e\rangle \mid \langle v, E\rangle \mid \texttt{fst}\ e \mid \texttt{snd}\ e$
			$\mid \texttt{cons}_T\ E\ e \mid \texttt{cons}_T\ v\ E$
			$\mid \texttt{head}_T\ E \mid \texttt{tail}_T\ E \mid \texttt{isNil}_T\ E$
			$\mid \texttt{inl}_T\ E \mid \texttt{inr}_T\ E \mid \texttt{case}\,(x)\,E\,e\,e$
			$\mid e\ [T] \mid \texttt{fold}_T\ E \mid \texttt{unfold}\ E$
			$\mid \texttt{fix}\ E \mid \texttt{letrec}_T\ x = E\ \texttt{in}\ e$
			$\mid \texttt{raise}_T\ E \mid \texttt{try}\ E\ \texttt{with}\ e$
Error Contexts	F	$\equiv$	Contexts $- \{\texttt{try}\ E\ \texttt{with}\ e\}$

Excerpt of Dynamic Semantics $\boxed{e \longrightarrow e}$

$$\texttt{pred z} \longrightarrow_{\mathrm{S}} \texttt{raise}_{\texttt{Int}}\ \texttt{z}$$

$$\texttt{head}_T\ (\texttt{nil}_T) \longrightarrow_{\mathrm{S}} \texttt{raise}_T\ \texttt{z}$$

$$\texttt{tail}_T\ (\texttt{nil}_T) \longrightarrow_{\mathrm{S}} \texttt{raise}_{(\texttt{List}\ T)}\ (\texttt{succ z})$$

$$\frac{\Gamma \vdash F[\texttt{raise}_{T_1}\ v] : T_2}{F[\texttt{raise}_{T_1}\ v] \longrightarrow_{\mathrm{S}} \texttt{raise}_{T_2}\ v} \quad \text{(CTX-ERR)}$$

Figure 6. The syntax of `fpl`. The language is not chosen for its *minimality*. For example, recursive types can define booleans and lists. On the contrary, `fpl` shows a handful of features to demonstrate how they are all in the scope of our approach.

While we require type-uniqueness in the cast calculus to prove the gradual guarantee, we do not require it for the source language. It is straightforward for the compiler from the source language to the cast language to insert the explicit casts and type annotations required by a type-unique cast language.

6. Application of the Methodology

We have applied our methodology to a number of languages. We show here the application of our methodology to the gradualization of a rich intermediate language which we call `fpl`. (The source language would provide type inference and other conveniences.)

The syntax of `fpl` is shown in Figure 6. The language includes several features: integers, booleans, `if`–`then`–`else`, pairs, sums, lists, universal types, recursive types, `fix`, `letrec` and exceptions. The typing and reduction rules for the operators of `fpl` are standard and here omitted. We show the rules for type safe lists and predecessor operator. For the sake of type-uniqueness, `raise` is type-annotated. Similarly to the blame expression of CC, `raise` replaces expressions through error contexts. Rule (CTX-ERR) employs the same solution as (CTX-BLAME) for blame and ensure type preservation by changing the type annotation of `raise`.

Applying our methodology we obtain the cast language in Figure 7. The eliminators of higher-order types of `fpl` are: the application, `fst` , `snd` , `head`, `tail`, `isNil`, `case`, type application and `unfold`. The corresponding reduction rules in Figure 7 coincide with those found in literature for each of them.

The following theorem states the correctness of the cast language produced by our methodology.

Theorem 5. *The cast language of* `fpl` *in Figure 7 satisfies the correctness criteria of gradual typing.*

We have proved Theorem 5 on paper. We are currently completing a mechanized proof of this theorem in the Abella proof assistant [3]. The current status of the proof is that conservativeness, type safety and the blame theorem are fully mechanized, including all their dependencies. The proof for the gradual guarantee is about to be mechanized in full. Only two lemmas are admitted: reflexivity of the precision relation, and that less precise programs are typed at less precise types[4]. Despite these missing pieces in the mechanized proof, we have checked that Theorem 5 holds.

7. Typed Languages as Logic Programs

We give a formal account of our methodology. This makes the methodology automatic in the form of an algorithm, which suggests implementations. Furthermore, the algorithm can be the subject of proofs. We model typed languages as logic programs in the intuitionistic type theory of Harrop formulae. Such logic is executable and underlies the λ-Prolog language.

Logic programs are equipped with a signature (Σ) and a set of rules (D). The signature defines the entities that are involved in the program. For example, STLC may have declarations $\mathsf{term} : \mathsf{kind}$ and $\mathsf{type} : \mathsf{kind}$ for expressions and types, respectively. Other declarations impose a typing discipline. For example, function type may have the declaration $\mathsf{arrow} : \mathsf{type} \to \mathsf{type} \to \mathsf{type}$, and the typing judgment may have $\mathsf{typeof} : \mathsf{term} \to \mathsf{type} \to \mathsf{o}$ (o is the kind for propositions). In our setting, logic programs make use of higher-order abstract syntax (HOAS). As such, we can use a primitive notion of λ-abstraction and application and the declaration of the STLC abstraction is $\mathsf{abs} : \mathsf{type} \to (\mathsf{term} \to \mathsf{term}) \to \mathsf{term}$, i.e. the second argument is an abstraction.

Given a signature Σ, the *terms* of logic programs are those that can be constructed from λ-terms, logical variables (X) and constructors (f) from Σ. Below we give definitions for terms, formulas, and our simplified forms for premises and rules.

Definition 6 (Terms, Formulas, Premises, and Rules over Σ)**.**

$$\begin{array}{rrcl} \textit{term} & t & ::= & x \mid \lambda x.t \mid (t\ t) \mid X \mid (\mathsf{f}\ t\ \ldots\ t) \\ \textit{formula} & F & ::= & \mathsf{pred}\ t\ \ldots\ t \\ \textit{premise} & P & ::= & F \mid \forall x.(F \Rightarrow F) \mid \forall x.F \end{array}$$

A rule r over Σ is of the form $\dfrac{P_1\ \ldots\ P_n}{F}$

where f *and* pred *are constants from the signature Σ. We define* $\texttt{prem}(r) = \{P_1, \ldots, P_n\}$*, and* $\texttt{conclusion}(r) = F$*. We use the notation $\widetilde{X}$ to denote a finite number of distinct variables as arguments, e.g.* $(\mathsf{f}\ \widetilde{X}) \equiv (\mathsf{f}\ X_1 \cdots X_n)$.

[3] $\texttt{case}\,(x)\,e\,e\,e$ is short for $\texttt{case}\ e\ \texttt{of}\ \texttt{inl}\ x \Rightarrow\ e \mid \texttt{inr}\ x \Rightarrow\ e$.

[4] Precision for recursive types has a treatment that stems from recursive subtyping (Amadio and Cardelli [2]) and those lemmas seem to require some workaround in our HOAS-based formulation.

$$
\begin{array}{llll}
\text{Types} & T & ::= & \ldots \mid \star \\
\text{Expressions} & e & ::= & \ldots \mid e : T \Rightarrow^{\ell} T \mid \mathtt{blame}_T\ \ell \\
\text{Ground types} & G & ::= & B \mid\ \star \to \star \mid \star \times \star \mid \mathtt{List}\ \star \mid \star + \star \mid \forall X.\star \mid \mu X.\star \\
\text{Values} & v & ::= & \ldots \mid v : G \Rightarrow^{\ell} \star \\
 & & & \mid v : T_1 \to T_2 \Rightarrow^{\ell} T_1' \to T_2' \ \mid\ v : T_1 \times T_2 \Rightarrow^{\ell} T_1' \times T_2' \\
 & & & \mid\ v : \mathtt{List}\ T \Rightarrow^{\ell} \mathtt{List}\ T' \ \mid\ v : T_1 + T_2 \Rightarrow^{\ell} T_1' + T_2' \\
 & & & \mid v : \forall X.T \Rightarrow^{\ell} \forall X.T' \ \mid\ v : \mu X.T \Rightarrow^{\ell} \mu X.T' \\
\text{Contexts} & E & ::= & \ldots \mid E : T \Rightarrow^{\ell} T
\end{array}
$$

Static Semantics: the typing rules of `fpl`, (T-CAST) and (T-BLAME).
Dynamic Semantics:

- the reduction rules of `fpl` imported for $\longrightarrow$, (ID-BASE), and the cast handler reduction rules of Figure 4.
- the cast reduction rules (C-BETA), (C-CASE) and (C-HEAD) (in Section 3.4, not repeated in this figure).
- the reduction rules in the third column below.

Pairs

$$
\frac{\Gamma \vdash\ e : T_1 \times T_2}{\Gamma \vdash\ \mathtt{fst}\ e : T_1} \quad \Rightarrow \quad \begin{array}{c}(v : T_1' \times T_2' \Rightarrow^{\ell} T_1 \times T_2) \\ \{T_1' \frown T_1, T_2' \frown T_2\}\end{array} \quad \Rightarrow \quad \begin{array}{c}\mathtt{fst}\ (v : T_1' \times T_2' \Rightarrow^{\ell} T_1 \times T_2) \\ \longrightarrow \\ (\mathtt{fst}\ v) : T_1' \Rightarrow^{\ell} T_1\end{array}
$$

$$
\frac{\Gamma \vdash\ e : T_1 \times T_2}{\Gamma \vdash\ \mathtt{snd}\ e : T_2} \quad \Rightarrow \quad \begin{array}{c}(v : T_1' \times T_2' \Rightarrow^{\ell} T_1 \times T_2) \\ \{T_1' \frown T_1, T_2' \frown T_2\}\end{array} \quad \Rightarrow \quad \begin{array}{c}\mathtt{snd}\ (v : T_1' \times T_2' \Rightarrow^{\ell} T_1 \times T_2) \\ \longrightarrow \\ (\mathtt{snd}\ v) : T_2' \Rightarrow^{\ell} T_2\end{array}
$$

Lists

$$
\frac{\Gamma \vdash\ e : \mathtt{List}\ T}{\Gamma \vdash\ \mathtt{tail}[T]\ e : \mathtt{List}\ T} \quad \Rightarrow \quad \begin{array}{c}(v : \mathtt{List}\ T' \Rightarrow^{\ell} \mathtt{List}\ T) \\ \{T_1' \frown T_1\}\end{array} \quad \Rightarrow \quad \begin{array}{c}\mathtt{tail}[T]\ (v : \mathtt{List}\ T' \Rightarrow^{\ell} \mathtt{List}\ T) \\ \longrightarrow \\ (\mathtt{tail}[T']\ v) : \mathtt{List}\ T' \Rightarrow^{\ell} \mathtt{List}\ T\end{array}
$$

$$
\frac{\Gamma \vdash\ e : \mathtt{List}\ T}{\Gamma \vdash\ \mathtt{isNil}[T]\ e : \mathtt{Bool}} \quad \Rightarrow \quad \begin{array}{c}(v : \mathtt{List}\ T' \Rightarrow^{\ell} \mathtt{List}\ T) \\ \{T_1' \frown T_1\}\end{array} \quad \Rightarrow \quad \begin{array}{c}\mathtt{isNil}[T]\ (v : \mathtt{List}\ T' \Rightarrow^{\ell} \mathtt{List}\ T) \\ \longrightarrow \\ (\mathtt{isNil}[T']\ v)\end{array}
$$

Universal Types

$$
\frac{\Gamma \vdash\ e : \forall X.T_2}{\Gamma \vdash\ (e\ [T_1]) : T_2[T_1/X]} \quad \Rightarrow \quad \begin{array}{c}(v : \forall X.T_2' \Rightarrow^{\ell} \forall X.T_2) \\ \{T_2' \frown T_2\}\end{array} \quad \Rightarrow \quad \begin{array}{c}((v : \forall X.T_2' \Rightarrow^{\ell} \forall X.T_2)\ [T_1]) \\ \longrightarrow \\ (v\ [T_1]) : T_2'[T_1/X] \Rightarrow^{\ell} T_2[T_1/X]\end{array}
$$

Recursive Types

$$
\frac{\Gamma \vdash\ e : \mu X.T}{\Gamma \vdash\ \mathtt{unfold}\ e : T[\mu X.T/X]} \quad \Rightarrow \quad \begin{array}{c}(v : \mu X.T' \Rightarrow^{\ell} \mu X.T) \\ \{T' \frown T\}\end{array} \quad \Rightarrow \quad \begin{array}{c}\mathtt{unfold}\ (v : \mu X.T' \Rightarrow^{\ell} \mu X.T) \\ \longrightarrow \\ (\mathtt{unfold}\ v) : T'[\mu X.T'/X] \Rightarrow^{\ell} T[\mu X.T/X]\end{array}
$$

Figure 7. The cast language for `fpl`.

The implication $\Rightarrow$ and the universal quantification $\forall x$ play an important role in the specification of languages. Consider the following typing rule for abstraction of the STLC.

$$
\frac{(\forall x.\mathsf{typeof}\ x\ T_1 \Rightarrow\ \mathsf{typeof}\ (E\ x)\ T_2)}{\mathsf{typeof}\ (\mathsf{abs}\ T_1\ E)\ (\mathsf{arrow}\ T_1\ T_2)}
$$

An environment Γ is not necessary because it is encoded as a hypothetical call to the same typing relation for a generic variable x. Operationally speaking, encountering a goal with $\forall x$ will create a new fresh constant, and proving the implication above will temporarily augment the logic program with the fact $\mathsf{typeof}\ x\ T_1$ to prove the goal $\mathsf{typeof}\ (E\ x)\ T_2$ in the augmented logic program.

Our notion of typed languages are defined below. They are logic programs that have kinds term and type, and have suitable predicates for defining languages: value, err, typeof, step, multi-step, eval-ctx. The predicate eval-ctx tells when an expression is contained in another expression in evaluation context position.

Definition 7 (Typed Language)**.** *A* typed language $\mathbb{L}$ *is a pair* (Σ, D) *where* Σ *is a signature and* D *is a set of rules over* Σ *and such that* Σ *has kinds* term *and* type*, and contains declarations*

$$
\begin{array}{l}
\mathsf{typeof} : \mathsf{term} \to \mathsf{type} \to \mathsf{o} \\
\mathsf{step}, \mathsf{multi\text{-}step}, \mathsf{eval\text{-}ctx} : \mathsf{term} \to \mathsf{term} \to \mathsf{o} \\
\mathsf{value}, \mathsf{err} : \mathsf{term} \to \mathsf{o}
\end{array}
$$

multi-step *is the reflexive and transitive closure of* step*. Rules for* $p \in \{\mathsf{value}, \mathsf{err}\}$ *have conclusion* $p\ (op\ \widetilde{X})$ *and premises* value X*, with* $X \in \widetilde{X}$*.* eval-ctx *is reflexive and its other rules are of form*

$$
\frac{\mathsf{eval\text{-}ctx}\ E_i\ E_i' \ \cup\ \Phi}{\mathsf{eval\text{-}ctx}\ (op\ \widetilde{X_1}\ E_i\ \widetilde{X_2})\ E_i'} \qquad (op_i\text{-CTX})
$$

with distinct variables $\widetilde{X_1}$*,* E_i*,* $\widetilde{X_2}$*, and* E_i'*, and* Φ *only uses* value *on variables* $\widetilde{X_1}$ *and* $\widetilde{X_2}$*. For each rule* (op_i-CTX)*,* D *contains*

$$
\frac{\mathsf{step}\ E_i\ E_i' \ \cup\ \Phi}{\mathsf{step}\ (op\ \widetilde{X_1}\ E_i\ \widetilde{X_2})\ (op\ \widetilde{X_1}\ E_i'\ \widetilde{X_2})} \qquad (op_i\text{-R-CTX})
$$

The shape of rules for defining values (value) and errors (err) matches typical definitions in languages. In particular, they are defined for a top level operator applied to distinct variables and they may check for the valuehood of some of the arguments. The shape of rules (op_i-CTX) for eval-ctx is similar, although it prescribes a recursive call on one of the arguments. Rules (op_i-R-CTX) define steps according to the evaluation contexts declared by eval-ctx[5].

We use E for variables of kind term or $k \to$ term, for some kind k. We use T for variables of kind type or type $\to$ type. Also, we use exp for terms of kind term, and ty for terms of kind type or type $\to$ type. Whenever $\texttt{conclusion}(r) = \text{typeof } exp\ ty$, we define $\texttt{input}(r) = exp$ and $\texttt{assigned}(r) = ty$.

Well-Formed Typed Languages Our notion of typed languages is quite liberal and encompasses a large portion of logic programs. We first define a notion of *well-formed typed languages* as suitable input for our algorithm. Below, Restrictions 1, 2 and 3 serve to guarantee a correct gradualization in the way we have seen in Section 3.4, 4 and 5. Restrictions 4 and 5 make the shape of rules precise so that we can formally reason about them. Also, we have distinct variables in the right-hand side of reduction rules, as discussed in Section 3.4. Restrictions 6, and also 5, enforce a simpler syntax which simplifies our endeavors. For example, we do not deal with nested abstractions. Restriction 7 allows the language to have abstractions that accommodate binders such as λ, Λ, $\forall$ and μ.

Definition 8 (Well-formed typed language)**.** *Given a typed language $\mathbb{L}$, we say that $\mathbb{L}$ is well-formed whenever*

1. *Casts are determined for the typing rules of eliminators of higher-order types.*
2. *Type variance is adequate w.r.t. the blame theorem for the typing rules of eliminators of higher-order types.*
3. *The type system satisfies type-uniqueness.*
4. *A reduction rule may pattern-match at most one argument of an operator with a value or an error. Premises of reduction rules, except for (op_i-R-CTX), can use only the predicate* value *on variables. Variables in the right-hand side are all distinct.*
5. *Each eliminator has only one typing rule. Premises of typing rules are of the form*
   - typeof $E\ ty$, *or*
   - $\forall x.$typeof $x\ T \Rightarrow$ typeof $(E\ t)\ ty$

   where E is in the expression being typed. The expression being typed is built with a top level operator applied to distinct variables, each of which is assigned a type by a premise.
6. *Restrictions to HOAS: In typing rules of eliminators, the only usage of HOAS is with applications of the form $(X\ t)$, where t does not contain applications or abstractions.*
7. *Abstractions are of kind* (term $\to$ term)*,* (type $\to$ type) *or* (type $\to$ term)*.*

Our notion of well-formed typed languages is liberal enough for accommodating all the examples that we have considered, including the full formulation of `fpl`. At the same time, these restrictions affect the applicability of the algorithm. For example, Restriction 5 forbids the use of a subtyping relation in typing rules. Similarly, Restriction 7 rules out abstractions of kind (term $\to$ type) and so excludes dependent and refinement types. Section 12 discusses the applicability of our results and our plans for future work in this respect.

[5] Alternatively, we could have a kind ctx for contexts with a hole. However, the machinery to adopt this style is a bit more involved.

8. An Algorithm for Generating Cast Languages

Our algorithm is realized with the function `castLanguage`. This function takes in input a typed language and generates another typed language, which represents a cast language. Below, we give the definition of `castLanguage`. We assume the auxiliary function $\texttt{eliminators-ho}(\mathbb{L})$ to return the set of eliminators of higher-order types of the typed language $\mathbb{L}$. The functions `template`, `valueInh`, `castRule`, and `bridges` are defined in the subsequent sections.

Definition 9 (Cast Language Generation)**.** *Given a typed language $\mathbb{L} = (\Sigma, D)$, we define* $\texttt{castLanguage}(\mathbb{L}) =$

$$\begin{array}{c}\texttt{template}(\mathbb{L}) \\ \cup \\ \bigcup_{r \in D_{|\mathsf{elim}}} \texttt{valueInh}_{\mathbb{L}}(\texttt{castRule}_r(\texttt{bridges}(r)))\end{array}$$

where

$$D_{|\mathsf{elim}} = \{r \in D \mid \begin{array}{c}\texttt{input}(r) = (op\ \widetilde{X}) \\ \wedge \\ op \in \texttt{eliminators-ho}(\mathbb{L})\end{array}\}$$

In other words, $D_{|\mathsf{elim}}$ is the set of typing rules of eliminators of higher-order types.

The function `template` returns a typed language that describes a cast language template. We add the set of reduction rules created according to our methodology. As a slight abuse of notation, Definition 9 writes $\mathbb{L} \cup D$ for including D in the set of rules of $\mathbb{L}$.

8.1 Step 1: Cast Language Templates

We devise the definition of *cast language template*. Below, we conveniently separate the reduction rules of the original language, for which we use the predicate step, and the new cast reduction rules, for which we use the predicate step_C. A reduction rule connects the two ((C-UPTO) below).

Definition 10 (Cast Language Template of a Typed Language)**.** *Given a typed language $\mathbb{L} = (\Sigma', D')$, the* cast language template of $\mathbb{L}$*, written* $\texttt{template}(\mathbb{L})$ *is (itself) a typed language (Σ, D) such that $\Sigma' \subseteq \Sigma$ and $D' \subseteq D$ and in which Σ contains the declarations*

$$\begin{array}{l}\mathsf{label} : \mathsf{kind} \\ \star : \mathsf{type} \\ \mathsf{ground} : \mathsf{type} \to \mathsf{o} \\ \mathsf{cast} : \mathsf{term} \to \mathsf{type} \to \mathsf{label} \to \mathsf{type} \to \mathsf{term} \\ \mathsf{blame} : \mathsf{type} \to \mathsf{label} \to \mathsf{term} \\ \mathsf{step_C} : \mathsf{term} \to \mathsf{term} \to \mathsf{o} \\ \mathsf{consistency} : \mathsf{type} \to \mathsf{type} \to \mathsf{o}\end{array}$$

and D contains

- *the rules for consistency over constructors of kind* type *in Σ. (We omit showing the generation of these rules).*
- *the following two (typing) rules*

$$\frac{\begin{array}{c}\mathsf{typeof}\ E\ T_1 \\ \mathsf{consistency}\ T_1\ T_2\end{array}}{\mathsf{typeof}\ (\mathsf{cast}\ E\ T_1\ L\ T_2)\ T_2} \qquad \frac{}{\mathsf{typeof}\ (\mathsf{blame}\ T\ L)\ T}$$

- *the following definitions for ground types*
  for all $c : k_1 \to \ldots \to k_n \to$ type in Σ

$$\mathsf{ground}\ (c\ T_1\ \ldots\ T_n)$$

$$\text{with } T_i = \begin{cases}\star & \text{if } k_i = \mathsf{type} \\ \lambda x.\star & \text{if } k_i = \mathsf{type} \to \mathsf{type}\end{cases}$$

- *the additional definitions of values*

$$\frac{\mathsf{value}\ V \qquad \mathsf{ground}\ G}{\mathsf{value}\ (\mathsf{cast}\ V\ G\ L\ \star)}$$

and for all $c : k_1 \to \ldots \to k_n \to \mathsf{type}$ *in* Σ

$$\frac{\mathsf{value}\ V}{\mathsf{value}\ (\mathsf{cast}\ V\ (c\ T_1 \to \ldots \to T_n)\ L\ (c\ T_1' \to \ldots \to T_n'))}$$

- step *reduces up-to* $\mathsf{step_C}$*:* $\dfrac{\mathsf{step_C}\ E\ E'}{\mathsf{step}\ E\ E'}$ (C-UPTO)
- *the rule for setting casts as evaluation contexts*

$$\frac{\mathsf{eval\text{-}ctx}\ E\ E'}{\mathsf{eval\text{-}ctx}\ (\mathsf{cast}\ E\ T_1\ L\ T_2)\ E'}$$

$$\frac{\mathsf{step}\ E\ E'}{\mathsf{step}\ (\mathsf{cast}\ E\ T_1\ L\ T_2)\ (\mathsf{cast}\ E'\ T_1\ L\ T_2)}$$

- *the rule for triggering a blame*

$$\frac{\mathsf{typeof}\ E\ T_1 \qquad \mathsf{eval\text{-}ctx}\ E\ (\mathsf{blame}\ T_2\ L)}{\mathsf{step}\ E\ (\mathsf{blame}\ T_1\ L)}$$

- (ID-BASE) *and the cast handler reduction rules of Figure 4, represented as logic programming rules.*

It is worth noting that by Definition 10 a cast language template inherits declarations and rules from the typed language.

8.2 Step 3(a): Building Bridges

The function `bridges` realizes the step for building bridges. This function is from typing rules to pairs $(cvalue, B)$, where $cvalue$ is a term of kind term that represents a cast value, and B is a set of bridges. To help the presentation, we assume that the eliminated argument is always the first after the type annotation arguments that the operator might have. This is without loss of generality. In presenting our algorithm, we underline the eliminated argument.

Definition 11 (Bridge Construction)**.** *Given a (typing) rule* r *of the form*

$$\frac{\mathsf{typeof}\ E\ (c\ T_1 \cdots T_n) \cup \Phi}{\mathsf{typeof}\ (op\ \widetilde{T}\ \underline{E}\ \widetilde{E})\ \ ty}$$

where $T_1, \ldots, T_n$ *are distinct variables, (Note that variables in* $\widetilde{T}$ *may appear in* $T_1, \ldots, T_n$*), we define*

$$\mathtt{bridges}(r) = (\mathsf{cast}\ V\ (c\ T_1' \cdots T_n')\ L\ (c\ T_1 \cdots T_n), B)$$

with $B = \{T_1' \curvearrowright T_1, \ldots, T_n' \curvearrowright T_n\}$*, for fresh variables* $T_1', \ldots, T_n'$*.*

8.3 Step 3(b): Generating Cast Reductions

Now we give a formal account of the generation of cast reduction rules. A major part of the step involves inserting casts around expressions. To this aim, we define the notion of *bridge lookup* for computing the appropriate source and target type for casts. Intuitively, if a set of bridges B contains $T' \curvearrowright T$ then $(\mathtt{List}\ T)^{Br}_B = \mathtt{List}\ T'$. Note that during the generation of the reduction rule we retrieve types in the typing rule. In particular, we always have types such as `List` T at hand and not `List` T'.

Definition 12 (Bridge lookup)**.** *Given a term* ty *and a set of bridges* B*, the* bridge lookup for ty in B*, written* ty^{Br}_B*, is defined as follows.*

$$\begin{array}{rcl} T^{Br}_B & = & T',\ \textit{if}\ T' \curvearrowright T \in B. \\ (c\ ty_1\ \cdots\ ty_n)^{Br}_B & = & (c\ ty^{Br}_{1,B}\ \cdots\ ty^{Br}_{n,B}). \\ T^{Br}_B & = & T,\ \textit{otherwise}. \end{array}$$

With this ingredient, we can define the function `castRule`, which generates a cast reduction rule. This function takes as input a pair $(cvalue, B)$ that is the output of the function `bridges`. `castRule` is also parametrized by the typing rule of the eliminator we are currently handling. The function returns a reduction rule.

Definition 13 (Cast Reduction Generation)**.** *Given a (typing) rule* r *such that* $\mathtt{input}(r) = (op\ \widetilde{T}\ \underline{E}\ \widetilde{E})$*, an expression term* $cvalue = (\mathsf{cast}\ V\ ty_1\ L\ ty_2)$ *and a set of bridges* B*, we define*

$\mathtt{castRule}_r((cvalue, B)) =$

$$\frac{\mathsf{value}\ V}{\mathsf{step_C}\ (op\ \widetilde{T}\ \underline{cvalue}\ \widetilde{E})\ \ \mathtt{wrap}((op\ \mathtt{typ}(\widetilde{T})\ \underline{V}\ \mathtt{sibl}(\widetilde{E})))}$$

where (below $\Pi = \mathtt{prem}(r)$ *and F is a formula)*

$\mathtt{typ}(T_1\ \cdots\ T_n) \equiv T^{Br}_{1,B}\ \cdots\ T^{Br}_{n,B}$

$\mathtt{sibl}(E_1\ \cdots\ E_n) \equiv \mathtt{sibl}(E_1)\ \cdots\ \mathtt{sibl}(E_n)$

$\mathtt{sibl}(E) =$

$$\begin{cases} (\mathsf{cast}\ E\ ty\ L\ ty^{Br}_B) & \mathsf{typeof} E\ ty \in \Pi \\ \lambda x.\mathsf{cast}\ (E\ \mathtt{te}(F))\ ty\ L\ ty^{Br}_B & \forall x.F \Rightarrow \mathsf{typeof}(E\ x)\ ty \in \Pi \end{cases}$$

$$\mathtt{te}(\mathsf{typeof}\ x\ ty) = \begin{cases} (\mathsf{cast}\ x\ ty^{Br}_B\ L\ ty) & ty \neq ty^{Br}_B \\ x & \textit{otherwise} \end{cases}$$

$\mathtt{wrap}(exp) = (\mathsf{cast}\ exp\ \mathtt{assigned}(r)^{Br}_B\ L\ \mathtt{assigned}(r))$

To build the left-hand side of the reduction rule we retrieve the expression being typed by the typing rule. Then, we place the cast value as eliminated argument. It is convenient to use the siblings from the typing rule to have a link with the types used by $cvalue$. The right-hand side is built by placing the value that is nested in $cvalue$ as eliminated argument, and by treating type annotations with `typ`, the rest of the siblings with `sibl`, and wrapping the entire expression with `wrap`. The function `typ` simply replaces the type with its bridge lookup. Note that those types that do not have a bridge have a reflexive bridge lookup. These types remain unchanged in the right-hand side. The function `sibl` retrieves the typing premise of an expression variable. This rule exists by Restriction 5 of well-formedness. We then cast the variable to their bridge lookup. (This may produce a trivial reflexive cast for those types that do not have reverse bridge.) At the encounter of an implicational typing premise, we have to deal with a variable E that is an HOAS abstraction. In that case a cast $(\mathsf{cast}\ E\ ty\ L\ ty^{Br}_B)$ is not well-typed because E is not of kind type. Therefore, we generate a wrapped term $\lambda x.(\mathsf{cast}\ (E\ x)\ ty\ L\ ty^{Br}_B)$, using HOAS λ and application. This expression is an abstraction and replaces the abstraction E. Additionally, as x is typed at the left of an implication, it (virtually) appears in the type environment. According to step (e) of Section 3.4 we need to check whether some other type has a bridge to it. We do this with the function `te`. We simply check for a reflexive bridge lookup to see whether this bridge exists or not. In case a bridge exists (condition $ty \neq ty^{Br}_B$) then the abstraction that is fed with x is instrumented to receive a cast instead. This is the HOAS realization of the substitution explained in Section 3.4. In case a bridge does not exist the abstraction takes x. Finally, `wrap` is responsible for inserting the cast around the entire expression. This cast targets the assigned type of the typing rule.

As an optimization, implementations may, and should, avoid inserting reflexive casts.

8.4 Step 3(c): Valuehood Inheritance

The function `valueInh` takes the reduction rule produced by `castRule` and generates another reduction rule. To be precise, `valueInh` injects new value premises into the reduction rule in input. It does so for those siblings of the eliminator that need to be value to let some reduction rule fire.

Definition 14 (Valuehood Inheritance Procedure)**.** *Given a typed language* $\mathbb{L} = (\Sigma, D)$ *and a (reduction) rule* r *such that*

$$r = \frac{\mathsf{value}\ V}{\mathsf{step_C}\ (op\ \widetilde{T}\ \underline{exp_1}\ t_1\ \cdots\ t_n)\ exp_2}$$

we define

$$\mathtt{valueInh}_{\mathbb{L}}(r) = \frac{\mathsf{value}\ V \cup \Phi}{\mathsf{step_C}\ (op\ \widetilde{T}\ \underline{exp_1}\ t_1\ \cdots\ t_n)\ exp_2}$$

where Φ *is the least set such that* $\mathsf{value}\ E_i \in \Phi$ *whenever*

- $t_i = \begin{cases} E_i, \text{or} \\ (\mathsf{cast}\ E_i\ ty_i\ L\ ty_i') \end{cases}$
- $\mathsf{value}\ E_i' \in \mathtt{prem}(r')$ *for some* $r' \in D$ *such that* $\mathtt{conclusion}(r') = \mathsf{step}\ (op\ \widetilde{T'}\ \underline{exp_1'}\ E_1' \cdots E_n')\ exp_2'$. *In other words, the variable at that same position* i *is tested for valuehood in* r'.

9. Correctness of the Gradualizer

We have proved that given a type safe and well-formed typed language $\mathbb{L}$, its cast language $\mathbb{C} = \mathtt{castLanguage}(\mathbb{L})$ satisfies the key correctness criteria of gradual typing (Figure 5) except for type safety, which we conjecture. Below, we discuss these results.

Conservative Extension This property holds because static expressions do not contain casts, blame or occurrences of the dynamic type. Therefore, $\mathbb{L}$ and $\mathbb{C}$ reduce static expressions with the same part of the dynamic semantics.

Type Safety We could not formally prove this property. As the language is parameterized, we do not know the original type safety proof. Yet, step and $\mathsf{step_C}$ are interleaved with the rule (C-UPTO). This might require injecting proof cases within the main induction of the original proof and we do not currently have a means to reason about it. The literature presents some solutions to combining type safety proofs which require reasoning on a syntactic representation of proofs such as that of proof assistants [6, 22]. Here we conjecture that type safety is inherited when starting from a type safe language. We have evidence that this would be the case. Below we offer some arguments for it. First, we have proved that the reduction rules $\mathsf{step_C}$ are, stand alone, type preserving. This is thanks to our methodology, which carefully casts siblings, and the entire expressions, always in synch with type preservation. Previous work [28, 36] shows that cast handler reduction rules cover all cases for progress and are type preserving. As further evidence, we have taken the mechanized proof of type safety of `fpl` and created a mechanized proof for the type safety of its cast language, obtained by the automatic generation of our tool.

Blame Theorem We have proved the Blame Theorem. The notion of cast safety depends on subtyping, which varies from type to type depending on their variance. Our proofs assume functions $\mathtt{covar}(c, i)$, $\mathtt{contra}(c, i)$, and $\mathtt{invar}(c, i)$ that tells whether the i-th argument of the type constructor c is covariant, contravariant, or invariant, respectively. The proof proceeds making use of the subtyping generated accordingly, by employing the following rule (we show only the relevant rule).

$$\frac{\begin{array}{c}\forall i \text{ s.t. } \mathtt{covar}(c,i), T_i <: S_i \\ \forall i \text{ s.t. } \mathtt{contra}(c,i), S_i <: T_i \\ \forall i \text{ s.t. } \mathtt{invar}(c,i), T_i = S_i\end{array}}{c\ T_1 \ldots T_n <: c\ S_1 \ldots S_n}$$

Quantified types also need to inform whether they have to be treated recursively. For example, μ uses the approach to recursive subtype as devised by Amadio and Cardelli [2]. To prove the blame theorem, a crucial help comes from the invariant that the source of reductions starts with a higher-order cast with just one top level constructor. This is decomposed in casts that simply use variables. This allows us to match immediately the definition rules of subtyping. The only extra care is in the direction of subtyping. The adequacy of the type variance with respect to the blame theorem ensures that type variables are always placed in casts in a way that respect subtyping.

Gradual Guarantee We have established the gradual guarantee. The proof follows the two-part structure delineated by Siek et al. [28], though generalized to the level of language definitions as input. Part 1 is the core of the proof, and as observed by Siek et al. [28], it also implies Part 2. Here we give an idea of the general reasoning for the proof of Part 1. Whenever $e = (op\ e_1\ \ldots e_n) \sqsubseteq (op\ e_1'\ \ldots e_n') = e'$ and e' takes a step with a reduction rule, we first look at the valuehood conditions on arguments e_i' of the reduction rule that proved the step. For their corresponding e_i we apply a *catch-up to value* lemma. This ensures that they can take steps and become values. Let e^* denote e with its proper arguments reduced to values. Now, we have a chance that the same reduction rule applies to e^*. We need to reason on the operator op and the reduction rule. We discuss only the case where the reduction rule is native of the fully-static language. By Restriction 4 of well-formed typed languages, the reduction rule pattern-matches either zero or one of its arguments. We prove each case in turn.

If the reduction rule pattern-matches zero arguments (e.g. the reduction rule for `let` and `fix`), then the reduction rule from the more precise term also applies to e^* with the valuehood conditions being satisfied. Recall that e contains arguments related to those of e' by the precision relation and the catch-up-to-value lemma preserves this relation. Therefore e^* has arguments related to those of e'. The reduction rule used argument meta-variables to form its right-hand side. Therefore, the step from e' forms the right-hand side with arguments of e', while e^* forms the same right-hand side, but using its own arguments, which are related to those of e'. Therefore, congruence establishes that the two right-hand sides, and so the two steps, are related by the precision relation.

If the reduction rule pattern-matches one argument, that argument is a value or an error. We discuss only the case for values. In this case, the operator is an eliminator and we have that e^* contains a value v_1 at the eliminated argument while the e' has a value at the eliminated argument that is not a cast. For example, we might have $v_1 \sqsubseteq \lambda x{:}T.\, e$ or $v_1 \sqsubseteq (\mathtt{cons}_T\ v')$. We apply a simulation lemma like the one in Siek et al. [28]. This lemma depends on the operator and on the reduction rule, so it is language-dependent. The lemma proceeds by induction on the precision relation and case analysis on v_1. If v_1 is a cast, we apply the inductive hypothesis to derive that the simulation holds for the inner value of the cast. Because the operator is an eliminator and a cast is in eliminated position, we can apply the reduction rule for $\mathsf{step_C}$ that we generated. Otherwise, if v_1 is not a cast, then it is of the same canonical form as the eliminated argument in e', and it reduces by the same rule as e'.

10. Implementation

We have implemented our algorithm in Haskell. Our tool takes language specifications in λ-Prolog and produces the cast language in λ-Prolog. These specifications are executable and thus include interpreters for running programs. We have applied our tool to the types and features encountered in this paper. Remarkably, we have used the tool to generate the cast language of a language as rich as `fpl`. The implementation of our tool can be found at the repository
`https://github.com/mcimini/GradualizerDynamicSemantics`
This repository also contains the mechanized proofs.

11. Related Work

Cimini and Siek [5] present a methodology, algorithm, and implementation to mechanically transform a static type system into a gradual type system and a cast insertion procedure. In the present paper, we extend that work with the generation of the dynamic semantics of gradually typed languages. Taken together, these two results provide a complete solution, generating gradual languages with executable type checkers and interpreters, directly from the specification of a statically typed language.

Garcia et al. [11] offer Abstract Gradual Typing (AGT) as a foundation for deriving the static and dynamic semantics of gradually typed languages. The AGT approach derives a dynamic semantics for the gradually typed language directly. This differs from the literature and the current paper, which instead define the dynamic semantics of the gradually typed language via translation to a cast calculus. Thus, the AGT approach provides a direct characterization of the correct behavior of a gradually typed program, which is useful for validating and evaluating the mechanisms and designs of other translation-based approaches, of which this present work is an example. The AGT approach handles some features, such as gradual row types and gradual effects, that are not addressed in the present work. On the other hand, it remains an open problem as to whether the AGT approach can be fully automated with an algorithm and implementation.

12. Discussion of our Results and Future Work

Our methodology and algorithm target languages defined in small step operational semantics and based on evaluation contexts. We handle a simple typing judgment of the form $\Gamma \vdash e : T$ and a reduction relation of the form $e \longrightarrow e'$. Moreover, we handle a class of languages based on introduction and elimination forms that is common in language design. Overall, our results seem very stable for a fairly expressive class of languages that includes modern features such as recursive types, polymorphism and exceptions.

However, our approach has a number of limitations. Languages with more complicated forms of typing judgements and reduction rules are not currently handled. These include references, type-and-effect systems, and typestate, to name a few. In the case of references, our algorithm could be modified to simply thread the heap in reduction rules, in which case our tool would naturally generate the rules

$$!(v : \texttt{Ref}\ T \Rightarrow^{\ell} \texttt{Ref}\ T') \mid \mu \longrightarrow (!v) : T \Rightarrow^{\ell} T' \mid \mu$$

$$(v_1 : \texttt{Ref}\ T \Rightarrow^{\ell} \texttt{Ref}\ T') := v_2 \mid \mu \longrightarrow v_1 := (v_2 : T' \Rightarrow^{\ell} T) \mid \mu$$

These rules correspond to those found in literature for gradual references [16]. However, we do not yet have a general formulation for handling languages that make use of auxiliary structures such as the heap, and stating and proving the gradual guarantee on them.

Our notion of well-formed languages requires that typing rules of eliminators only rely on typing judgements. This excludes the use of a subtyping relation, which we plan to address in future work.

Some languages make use of multiple and complex binding structures. An example is the ML-style pattern matching. These languages seem to require a non-trivial encoding for fitting into our HOAS-based formulation. Therefore, they fall out of the current syntactic restrictions of our algorithm. We plan on relaxing our restrictions to a more liberal input syntax.

Some languages have their own cast operator. When those languages are given as input, those operators are treated as any other to be gradualized. A cast operator such as that in featherweight Java [18] has a formulation that is based on introduction and elimination forms and, in principle, could fit into our setting. However, that cast operator also makes use of subtyping and nominal types, which require some extension. We plan on extending our work to capture this and similar cast operators in the near future.

As discussed in Section 8, dependent and refinement types are currently out of scope. There has been research on accommodating these features in gradual typing [20, 21] which we hope to build upon in the future.

We have used a formulation of the dynamic semantics that can incur the space efficiency problem described by Herman et al. [17]. There are proposed solutions [25, 27] which we plan to automate in the future.

13. Conclusions

We have proved for the first time that a rich language (`fpl`) satisfies the correctness criteria of gradual typing. This result shows that correct gradual typing is a realistic goal for programming languages with modern features such as recursive types, polymorphism, exceptions and so on.

We have created a methodology and algorithm for automatically generating the dynamic semantics of a cast language from a fully-static language, thereby automating the creation of gradually typed languages. As pointed out [12, 23], the dynamic semantics is the most delicate aspect in gradual typing. So far, rules such as (C-BETA) have required considerable innovation. It is pleasantly surprising that striving to ensure correctness criteria led to devising an automatic algorithm to generate such reduction rules.

Overall, our results complete the work initiated by Cimini and Siek [5] and demonstrate that gradual typing can, in principle, be automated for a fairly expressive class of languages, and that it is possible to create tools for supporting the shift to gradual typing.

Acknowledgments

We thank Ronald Garcia for his clarifications on AGT. We thank Sam Tobin-Hochstadt, Michael Vitousek, and David Christiansen for their feedback. We also thank the anonymous POPL reviewers, their comments and suggestions helped us improve the presentation of the paper. This research was funded by the National Science Foundation under grant number 1518844, *SHF: Large: Gradual Typing Across the Spectrum*.

References

[1] Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 201–214, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0.

[2] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 104–118, New York, NY, USA, 1991. ACM. ISBN 0-89791-419-8.

[3] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2), 2014.

[4] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 257–281. Springer Berlin Heidelberg, 2014.

[5] Matteo Cimini and Jeremy G. Siek. The gradualizer: a methodology and algorithm for generating gradual type systems. In *Symposium on Principles of Programming Languages*, POPL, January 2016.

[6] Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 207–218, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1832-7.

[7] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. Technical Report NU-CCS-02-05, Northeastern University, 2002.

[8] Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming*, ICFP, pages 48–59, October 2002.

[9] Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. Semantic casts: Contracts and structural subtyping in a nominal world. In *European Conference on Object-Oriented Programming*, 2004.

[10] Cormac Flanagan. Hybrid type checking. In *POPL 2006: The 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 245–256, Charleston, South Carolina, January 2006.

[11] Ronal Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St Petersburg, FL, USA, January 20-22*. ACM Press, 2016.

[12] Álvaro García-Pérez, Pablo Nogueira, and Ilya Sergey. Deriving interpretations of the gradually-typed lambda calculus. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, pages 157–168, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2619-3.

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

[14] Anders Hejlsberg. C# 4.0 and beyond. Microsoft Channel 9 Blog, April 2010.

[15] Anders Hejlsberg. Introducing TypeScript. Microsoft Channel 9 Blog, 2012.

[16] David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. In *Trends in Functional Prog. (TFP)*, page XXVIII, April 2007.

[17] David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. *Higher-Order and Symbolic Computation*, 23(2):167–189, 2010.

[18] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001. ISSN 0164-0925.

[19] Matthias Keil and Peter Thiemann. Transparent object proxies in javascript. In *European Conference on Object-Oriented Programming*, 2015.

[20] Nicolás Lehmann and Éric Tanter. Gradual refinement types. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, Fr, January 18-20*. ACM Press, 2017.

[21] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types (extended abstract). In *3rd IFIP International Conference on Theoretical Computer Science*, August 2004.

[22] Christopher Schwaab and Jeremy G. Siek. Modular type-safety proofs in agda. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification*, PLPV '13, pages 3–12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1860-0.

[23] Jeremy G. Siek and Ronald Garcia. Interpretations of the gradually-typed lambda calculus. In *Scheme and Functional Programming Workshop*, 2012.

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

[25] Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. In *Proceedings for the 1st workshop on Script to Program Evolution*, STOP '09, pages 34–46, New York, NY, USA, 2009. ACM.

[26] Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 365–376, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9.

[27] 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.

[28] 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.

[29] The Dart Team. *Dart Programming Language Specification*. Google, 1.2 edition, March 2014.

[30] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In *Dynamic Languages Symposium*, 2006.

[31] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *Symposium on Principles of Programming Languages*, January 2008.

[32] Tom Van Cutsem and Mark S. Miller. Proxies: Design principles for robust object-oriented intercession apis. In *Proceedings of the 6th Symposium on Dynamic Languages*, DLS '10, pages 59–72, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0405-4.

[33] Julien Verlaguet. Facebook: Analyzing PHP statically. In *Commercial Users of Functional Programming (CUFP)*, 2013.

[34] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for Python. In *Symposium on Dynamic Languages*, DLS '14, pages 1–16, 2014.

[35] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Workshop on Scheme and Functional Programming*, pages 15–26, 2007.

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

[37] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating typed and untyped code in a scripting language. In *Symposium on Principles of Programming Languages*, POPL, pages 377–388, 2010.