Type-Directed Operational Semantics for Gradual Typing

Wenjia Ye
The University of Hong Kong, Hong Kong

Bruno C. d. S. Oliveira
The University of Hong Kong, Hong Kong

Xuejing Huang
The University of Hong Kong, Hong Kong

Abstract

The semantics of gradually typed languages is typically given indirectly via an elaboration into a cast calculus. This contrasts with more conventional formulations of programming language semantics, where the semantics of a language is given directly using, for instance, an operational semantics.

This paper presents a new approach to give the semantics of gradually typed languages directly. We use a recently proposed variant of small-step operational semantics called *type-directed operational semantics* (TDOS). In TDOS type annotations become operationally relevant and can affect the result of a program. In the context of a gradually typed language, such type annotations are used to trigger type-based conversions on values. We illustrate how to employ TDOS on gradually typed languages using two calculi. The first calculus, called λB^g, is inspired by the semantics of the blame calculus, but it has implicit type conversions, enabling it to be used as a gradually typed language. The second calculus, called λB^r, explores a different design space in the semantics of gradually typed languages. It uses a so-called *blame recovery semantics*, which enables eliminating some false positives where blame is raised but normal computation could succeed. For both calculi, type safety is proved. Furthermore we show that the semantics of λB^g is sound with respect to the semantics of the blame calculus, and that λB^r comes with a *gradual guarantee*. All the results have been mechanically formalized in the Coq theorem prover.

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

Keywords and phrases Gradual Typing, Type Systems, Operational Semantics

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.12

Supplementary Material *Software (ECOOP 2021 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.7.2.9`

Funding This work has been sponsored by Hong Kong Research Grant Council projects number 17209519 and 17209520.

Acknowledgements We thank the anonymous reviewers for their helpful comments.

1 Introduction

Gradual typing aims to provide a smooth integration between the static and dynamic typing disciplines. In gradual typing a program with no type annotations behaves as a dynamically typed program, whereas a fully annotated program behaves as a statically typed program. The interesting aspect of gradual typing is that programs can be partially typed in a spectrum ranging from fully dynamically typed into fully statically typed. Several mainstream languages, including TypeScript [6], Flow [11] or Dart [8] enable forms of gradual typing to various degrees. Much research on gradual typing has focused on the pursuit of sound gradual typing, where certain type safety properties, and other properties about the transition between dynamic and static typing, are preserved.

The semantics of gradually typed languages is typically given indirectly via an elaboration into a cast calculus. For instance the *blame calculus* [39, 52], the *threesome calculus* [42] or other cast calculi [15, 20, 23, 37, 39, 49] are often used to give the semantics of gradually typed languages. Since a gradual type system can accept programs with unknown types, run-time checks are necessary to ensure type safety. Thus the job of the (type-directed) elaboration is to insert casts that bridge the gap between known and unknown types. Then the semantics of a cast calculus can be given in a conventional manner.

While elaboration is the most common approach to give the semantics for gradually typed languages, it is also possible to have a direct semantics. In fact, a direct semantics is more conventionally used to provide the meaning to more traditional forms of calculi or programming languages. A direct semantics avoids the extra indirection of a target language and can simplify the understanding of the language. Garcia et al. [17], as part of their *Abstracting Gradual Typing* (AGT) approach, advocated and proposed an approach for giving a direct semantics to gradually typed languages. They showed that the cast insertion step provided by elaboration, which was until then seen as essential to gradual typing, could be omitted. Instead, in their approach, they develop the dynamic semantics as proof reductions over source language typing derivations.

This paper presents a different approach to give the semantics of gradually typed languages directly. We use a recently proposed variant of small-step operational semantics [54] called *type-directed operational semantics* (TDOS) [25]. For the most part developing a TDOS is similar to developing a standard small step-semantics as advocated by Wright and Felleisen. However, in TDOS type annotations become operationally relevant and can affect the result of a program. While there have been past formulations of small-step semantics where type annotations are also relevant [5, 14, 18], the distinctive feature of TDOS is a so-called *typed reduction* relation. Typed reduction further reduces values based on their types. While typically values are the final result of a program, in TDOS typed reduction can further transform them based on their run-time type. Thus typed reduction provides an operational interpretation to type conversions in the language, similarly to coercions in coercion-based calculi [23].

We illustrate how to employ TDOS on gradually typed languages using two calculi. The first calculus, called λB^g, is inspired by the semantics of a variant of the blame calculus (λB) [52] by Siek et al. [39]. However, unlike the blame calculus, λB^g allows implicit type conversions, enabling it to be used as a gradually typed language. Gradually typed languages can be built on top of λB using an elaboration from a source language into λB. In contrast λB^g can already act as a gradual language, without the need for an elaboration process.

The second calculus, called λB^r, explores a different design space in the semantics of gradually typed languages. It uses a so-called *blame recovery semantics*, which enables eliminating some false positives where blame is raised but normal computation could succeed. In the λB calculus, a lambda expression annotated with a chain of types is taken as a value. This means that it accumulates the type annotations, and checks if there are errors only when the function is applied to a value. This has some drawbacks. Perhaps most notably, and widely discussed in the literature [16, 24, 37, 38, 42], is that the accumulation of annotations affects space efficiency. Moreover, sometimes blame is raised quite conservatively, when a program could successfully return a value. Of course, the blame calculus semantics is justified by its origins on contracts and traditional casts (such as those commonly used in mainstream languages like Java). In such settings all casts/contracts must be valid, and any violations should raise blame.

In the λB^r calculus, the design choice that we make is to only raise blame if the initial source type of the value and final target types are not consistent. Otherwise, even if intermediate annotations trigger type conversions which would not be consistent, the final result can still be a value provided that the initial source and final target types are themselves consistent. This semantics differs from the blame calculus where intermediate types can cause blame. Technically speaking we introduce a new *saved expression/value*, that is used as an intermediate result during reduction. A saved value is generated whenever a conversion between two inconsistent types is triggered. However, if later another type conversion is applied to the value, then a saved value can recover from the brink of blame and be restored as a conventional value, provided that the new target type is consistent with the type of the saved value. A nice aspect of this semantics is that it avoids the accumulation of type annotations, being more space efficient.

For both calculi type safety is proved. Furthermore we show that the semantics of λB^g is sound with respect to the semantics of the blame calculus, and that λB^r comes with a *gradual guarantee* [41]. All the results have been mechanically formalized in the Coq theorem prover.

In summary, the contributions of this work are:

- **TDOS for gradual typing:** We show that TDOS can be employed in gradually typed languages. This enables simple, and concise specifications of the semantics of gradually typed languages, without resorting to an intermediate cast calculus. A nice aspect of TDOS is that it remains close to the simple and familiar small-step semantics approach by Wright and Felleisen.
- **The λB^g calculus** provides a first concrete illustration of TDOS for gradual typing. It follows closely the semantics of the blame calculus, but it allows implicit type conversions. We show type-safety, determinism, as well as a soundness theorem that relates the semantics of λB^g to that of the blame calculus (ignoring blame labels).
- **The λB^r calculus and blame recovery semantics.** λB^r explores the design space of the semantics of gradual typing by using a blame recovery semantics. The key idea is to only raise blame if the initial source type of the value and final target types are not consistent. Furthermore, λB^r comes with a gradual guarantee [41].
- **Coq Formalization:** Both λB^g and λB^r, and all associated lemmas and theorems, have been formalized in the Coq theorem prover. The Coq formalization can be found in the supplementary materials of this paper:

`https://github.com/YeWenjia/TypedDirectedGradualTyping`

2 Overview

This section provides background on gradual typing and the blame calculus, and then illustrates the key ideas of our work and the λB^g and λB^r calculi.

2.1 Background: Gradual Typing and the λB calculus

Traditionally, programming languages can be divided into statically typed languages and dynamically typed languages. For a statically typed language, the type of every term must be known. The language may support type inference, but it usually requires some type annotations by the programmer, which bears some extra work for a programmer. However, the benefit of static typing is that type-unsafe programs are rejected before they are executed. On the other hand, in dynamically typed languages terms do not have static types and no

type annotations are needed. This waives the burden of a strict type discipline, at the cost of type-safety.

Gradual typing [43] is like a bridge connecting the two styles. Gradual typing extends the type system of static languages by allowing terms to have a *dynamic type* $\star$, which stands for the possibility of being any type. A term with the unknown type $\star$ is not rejected in any context by the type checker. Therefore, it can be viewed as in a dynamically typed language. In a gradually typed language, programs can be completely statically typed, or completely dynamically typed, or anything in between.

To cooperate with the very flexible $\star$ type, the common practice in gradual type systems is to define a binary relation called type *consistency*. A term of type A can be assigned type B if A and B are consistent ($A \sim B$). With $\star$ defined to be consistent with any other type, dynamic snippets can be embedded into the whole program without breaking the type soundness property. Of course, the type soundness theorem is relaxed and tolerates some kinds of run-time type errors. Besides type soundness, there are some other criteria for gradual typing systems. One well-recognized standard is the gradual guarantee proposed by Siek et al. [41].

Elaboration semantics of Gradual Typing and the λB calculus. The semantics of gradually typed languages is usually given by an elaboration into a cast calculus. This approach has been widely used since the original work on gradual typing by both Siek and Taha [43] and Tobin-Hochstadt and Felleisen [49].

One of the most widely used cast calculus for the elaboration of gradually typed languages is the blame calculus [39, 52]. Figure 1 shows the definition of the blame calculus. Here we base ourselves in a variant of the blame calculus by Siek et al. [39], but ignoring blame labels. The blame calculus is the simply-typed lambda calculus extended with the dynamic type ($\star$) and the cast expression ($t : A \Rightarrow B$). Meta-variables G and H range over ground types, which include Int and $\star \to \star$. The definition of values in the blame calculus contains some interesting forms. In particular, casts ($V : A \to B \Rightarrow A' \to B'$) and $V : G \Rightarrow \star$ are included. Run-time type errors are denoted as *blame*. Besides the standard typing rules of the simply typed lambda calculus, there is an additional typing rule for casts: if term t has type A and A is consistent with B, a cast on t from A to B has type B. The consistency relation for types states that every type is consistent with itself, $\star$ is consistent with all types, and function types are consistent only when input types and output types are consistent with each other. In the premise of rule BSTEPP-DYNA, there is a function ug which says that type A should be a function type consistent with $\star \to \star$, but not $\star \to \star$ itself.

The bottom of Figure 1 shows the reduction rules that we use in this paper. The dynamic semantics of the λB calculus is standard for most rules. The semantics of casts include the noteworthy parts. For first-order values, reduction is straightforward: a cast either succeeds or it fails and raises blame. For example:

$$1 : Int \Rightarrow \star : \star \Rightarrow Int \longmapsto^* 1$$
$$1 : Int \Rightarrow \star : \star \Rightarrow Bool \longmapsto^* blame$$

For higher-order values such as functions, the semantics is more complex, since the casted result cannot be immediately obtained. For example, if we cast from $\star \to \star$ to $\mathsf{Int} \to \mathsf{Int}$, we cannot judge the cast result immediately. So the checking process is deferred until the function is applied to an argument. Rule BSTEPP-ABETA shows that process: a function with the cast is a value which does not reduce until it has been applied to a value.

Syntax

$$
\begin{array}{ll}
\textit{Types} & A, B ::= \mathit{Int} \mid \star \mid A \to B \\
\textit{Ground types} & G, H ::= \mathit{Int} \mid \star \to \star \\
\textit{constant} & c ::= i \mid ... \\
\textit{Terms} & t ::= c \mid x \mid t : A \Rightarrow B \mid t_1\ t_2 \mid \lambda x : A.t \\
\textit{Result} & r ::= t \mid \mathit{blame} \\
\textit{Values} & V, W ::= c \mid V : A \to B \Rightarrow A' \to B' \mid \lambda x : A.t \mid V : G \Rightarrow \star \\
\textit{Context} & \Gamma ::= \cdot \mid \Gamma, x : A \\
\textit{Frame} & F ::= []\ t \mid V\ [] \mid [] : A \Rightarrow B
\end{array}
$$

$\boxed{\Gamma \vdash t : A}$ *(Additional Typing Rules)*

$$\textsc{btyp-cast}\ \frac{\Gamma \vdash t : A \qquad A \sim B}{\Gamma \vdash t : A \Rightarrow B : B}$$

$\boxed{A \sim B}$ *(Consistency of types)*

$$\textsc{s-i}\ \frac{}{\mathsf{Int} \sim \mathsf{Int}} \qquad \textsc{s-arr}\ \frac{A \sim C \qquad B \sim D}{A \to B \sim C \to D} \qquad \textsc{s-dynl}\ \frac{}{\star \sim A} \qquad \textsc{s-dynr}\ \frac{}{A \sim \star}$$

$\boxed{t \longmapsto r}$ *(Reduction for the λB Calculus)*

$$\textsc{bStepp-eval}\ \frac{t \longmapsto t'}{F.t \longmapsto F.t'} \qquad \textsc{bStepp-blame}\ \frac{t \longmapsto \mathit{blame}}{F.t \longmapsto \mathit{blame}} \qquad \textsc{bStepp-beta}\ \frac{}{(\lambda x : A.\, t)\, V \longmapsto t[x \mapsto V]}$$

$$\textsc{bStepp-vany}\ \frac{}{(V : G \Rightarrow \star) : \star \Rightarrow G \longmapsto V} \qquad \textsc{bStepp-dd}\ \frac{}{V : \star \Rightarrow \star \longmapsto V}$$

$$\textsc{bStepp-dyna}\ \frac{ug(A, \star \to \star)}{V : \star \Rightarrow A \longmapsto (V : \star \Rightarrow \star \to \star) : \star \to \star \Rightarrow A} \qquad \textsc{bStepp-blamep}\ \frac{G \nsim H}{(V : G \Rightarrow \star) : \star \Rightarrow H \longmapsto \mathit{blame}}$$

$$\textsc{bStepp-abeta}\ \frac{}{(V : A \to B \Rightarrow A' \to B')\, V \longmapsto (V\, (V : A' \Rightarrow A)) : B \Rightarrow B'} \qquad \textsc{bStepp-lit}\ \frac{}{i : \mathsf{Int} \Rightarrow \mathsf{Int} \longmapsto i}$$

$$\textsc{bStepp-anyd}\ \frac{ug(A, \star \to \star)}{V : A \Rightarrow \star \longmapsto (V : A \Rightarrow \star \to \star) : \star \to \star \Rightarrow \star}$$

Figure 1 The λB Calculus (selected rules).

2.2 Motivation for a Direct Semantics for Gradual Typing

In this paper we propose not to use an elaboration semantics into a cast calculus, but to use a direct semantics for gradual typing instead. We are not the first to propose such an approach. For instance, the AGT framework for gradual typing [17] also employs a direct semantics. In that work the authors state that "*developing dynamic semantics for gradually typed languages has typically involved the design of an independent cast calculus that is peripherally related to the source language*". They further argue that there is a gap between source gradually typed languages, and the cast calculi that they target. In particular cast calculi admit "*far more programs than those in the image of the translation procedure*". We agree with such arguments. In addition, as argued by Huang and Oliveira [25], there are some other reasons why a direct semantics is beneficial over an elaboration semantics.

A direct semantics enables simple ways for programmers and tools to reason about the behaviour of programs. For instance, with languages like Haskell it is quite common for programmers to use equational reasoning. Such reasoning steps are directly justifiable from the operational semantics of call-by-name/need languages. With a TDOS, we can easily (and justifiably) employ similar steps to reason about your source language (say GTLC or λB^g). With a semantics defined via elaboration, however, that is not an easy thing because of the indirect semantics. We refer readers to Huang and Oliveira's work, which has an extensive discussion about this point. Additionally, some tools, especially some debuggers or tools for demonstrating how programs are computed, require a direct semantics, since those tools need to show transformations that happen after some evaluation of the *source program*.

Another potential benefit of a direct semantics is simpler and shorter metatheory/implementation. For instance, with a direct semantics we can often save quite a few definitions/proofs, including a second type system, various definitions on well-formedness of terms, substitution operations and lemmas, pretty printers, etc. Though these are not arguably difficult, they do add up. Perhaps more importantly, some proofs can be simpler with a direct semantics. For example, proving the gradual guarantee is typically simpler, since some lemmas that are required with an elaboration semantics (for example, Lemma 6 in original work on the refined criteria for gradual typing [41]) are not needed with a direct operational semantics. Moreover only the precision relation for the source language is necessary.

2.3 λB^g: A Gradually Typed Lambda Calculus

Since λB requires explicit casts whenever a term's type is converted, it cannot be considered a gradually typed calculus. For comparison, the application rule for typing in the *Gradually Typed Lambda Calculus* (GTLC) [38, 41, 43]

$$\frac{\Gamma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_3 \qquad T_1 \sim T_3}{\Gamma \vdash e_1\, e_2 : T_2}\ \text{GTLC-App}$$

does not force the input term to have the same type as what the function expects. It just checks the compatibility of the two terms' types and can do implicit type conversions (casts) automatically. In a cast calculus, similar flexibility only exists when the term is wrapped with a cast, since the application rule strictly requires the argument type to be of the same type of the input of the function type. In λB, for instance, the application rule is the same as in the Simply Typed Lambda Calculus, requiring the argument type to be of the same type of the input of the function type.

Bi-directional type-checking for λB^g. As a first step to adapt a λB-like calculus into a source language for gradual typing, we turn to the bidirectional type checking [33]. Unlike in GTLC or λB, a bidirectional typing judgement may be in one of the two modes: inference or checking. In the former, a type is synthesized from the term. In the later, both the type and the term are given as input, and the typing derivation examines whether the term can be used under that type safely. In a typical bidirectional type system with subtyping, the subsumption rule is only employed in the checking mode, allowing a term to be checked by a supertype of its inferred type. That is to say, the checking mode is more relaxed than the inference mode, which typically infers a unique type. With bidirectional type-checking the application rule in such a system is not as strict as in the λB calculus, as the input term is typed with a checking mode.

Implicit type conversion in function applications. By using bidirectional type checking, we can type-check programs such as:

$(\lambda x.x)\ 1$	Accepted!
$(\lambda x.not\ x)\ 1$	Accepted!
$(\lambda x.not\ x : \star \to Bool)\ 1$	Accepted!
$(\lambda x.x + 1 : Int \to Int)\ 1$	Accepted!

and also reject ill-typed programs:

$(\lambda x.not\ x : Bool \to Bool)\ 1$	Rejected!

Note that λB^g supports *annotation expressions* of the form $e : A$. Thus, an expression like $\lambda x.not\ x : Bool \to Bool$ is a lambda expression $(\lambda x.not\ x)$ annotated with the type $Bool \to Bool$.

Explicit type conversion. Besides implicit conversions, programmers are able to trigger type conversions in an explicit fashion by wrapping the term with a type annotation $e : A$, where A denotes the target type. For instance, the two simple examples in λB in Section 2.1 can be encoded in λB^g as:

$$1 : \star : Int \longmapsto^* 1$$

$$1 : \star : Bool \longmapsto^* blame$$

with similar results to the same programs in the λB calculus. Notice that, unlike λB, there is no cast expression in λB^g. Casts are triggered by type annotations. For instance, in the first expression above $(1 : \star : Int)$, the first type annotation $(\star)$ triggers a cast from Int to $\star$. The source type Int is the type of 1, whereas the target type $\star$ is specified by the annotation. Then the second annotation Int will trigger a second cast, but now from $\star$ to Int.

Functions. One interesting change in the type system is that we handle lambdas by inference mode rather than checking mode. Our rule for lambdas is:

$$\frac{\Gamma, x : A \vdash e \Leftarrow B}{\Gamma \vdash \lambda x.\, e : A \to B \Rightarrow A \to B}\ \textsc{Typ-abs}$$

If the programmer wants to have their function statically type checked, they can write down the full annotations. Otherwise, the function can be left with no annotation, which will be desugared into a lambda with type $\star \to \star$, similarly to what happens in the GTLC for dynamically typed lambdas.

2.4 Designing a TDOS for λB^g

The most interesting aspect of λB^g is its dynamic semantics. We discuss the key ideas next.

Background: Type-Directed Operational Semantics. A type-directed operational semantics is helpful for language features whose semantics is type dependent. TDOS was originally proposed for languages that have intersection types and a merge operator [25]. To enable expressive forms of the merge operator the dynamic semantics has to account for the types, just like the semantics of gradually typed languages. In many traditional operational semantics type annotations are often ignored. In TDOS that is not the case, and the type annotations are used at runtime to determine the result of reduction. A TDOS has two parts. One part is similar to the traditional reduction rules, modulo some changes on type-related rules, like beta reduction for application, and annotation elimination for values. The second component of a TDOS is the typed reduction relation $v \longmapsto_A r$. Typed reduction has a value and a type as input and produces a value (when no run-time error is possible) as result. The resulting value is transformed from the input value.

Typed Reduction for λB^g. Due to consistency, run-time checking is needed in gradual typing. The typed reduction relation $v \longmapsto_A r$ is used when run-time checks are needed. Typed reduction compares the dynamic type of the input value with the target type. When the type of the input value (v) is not consistent to the target type (A), blame is raised. Otherwise, typed reduction adapts the value to suit the target type. Eventually, terms become more and more precise. Two easy examples to show how typed reduction work are shown next:

$$1 \longmapsto_{Int} 1$$

$$1 : \star \longmapsto_{Bool} \mathit{blame}$$

If we have an integer value 1 and we want to transform it with type Int, we simply return the original value. In contrast, attempting to transform the value $1 : \star$ under type $Bool$ will result in blame.

Typed reduction takes place in other reduction rules such as the beta reduction rule and the annotation elimination rule for values:

$$\textsc{Step-beta}\ \frac{v \longmapsto_A v'}{(\lambda x.\, e : A \to B)\, v \longmapsto e[x \mapsto v'] : B} \qquad \textsc{Step-annov}\ \frac{\mathit{not}\ (\mathit{value}\ (v : A)) \quad v \longmapsto_A r}{v : A \longmapsto r}$$

Take another example to illustrate the behavior of typed reduction in beta reduction:

$$(\lambda x.x : Bool \to Bool)\ (1 : \star)$$

If we would perform substitution directly, as conventionally done in beta-reduction, we would not check if there are run-time errors, for which blame should be raised. Since the typing rule for the argument of application is in checking mode, we need to check if the type of the argument is consistent with the target type. Therefore the argument must be further reduced with typed reduction under the expected type of the function input. When we check that the type Int is not consistent with $Bool$, blame is raised. However, if we take the example:

$$(\lambda x.x + 1 : Int \to Int)\ (1 : \star)$$

then the value 1 is substituted in the function body and the result is 2. The details of reduction and typed reduction in λB^g will be discussed in Section 3.

2.5 λB^r: Gradual Typing with a Blame Recovery Semantics

An alternative semantics for Gradual Typing. In λB^r we explore an alternative semantics for gradual typing that we call *blame recovery semantics*. The main idea is to only raise blame when the initial (source) type and the final target types in a chain of type annotations are inconsistent. Intermediate inconsistent types will not lead to blame. Thus the blame recovery semantics can be viewed as being more liberal with respect to raising blame. We illustrate the difference next, with 2 programs that raise blame in λB^g, but would successfully compute a value in λB^r:

$$(\lambda x.x : Int \to Int : \star \to \star : Bool \to Bool : \star \to \star)\ 1 \longmapsto^* blame \quad \{\text{Examples in } \lambda B^g\ \}$$
$$(\lambda x.x : \star \to \star : Bool \to Bool : \star \to \star : Int \to Int)\ 1 \longmapsto^* blame$$

$$(\lambda x.x : Int \to Int : \star \to \star : Bool \to Bool : \star \to \star)\ 1 \longmapsto^* 1 : Int \quad \{\text{Same examples in } \lambda B^r\ \}$$
$$(\lambda x.2 : \star \to \star : Bool \to Bool : \star \to \star : Int \to Int)\ 1 \longmapsto^* 2 : Int$$

For the above two examples, the function being applied is wrapped on a chain of annotations that contain the inconsistent types $\mathsf{Int} \to \mathsf{Int}$ and $Bool \to Bool$. Therefore, in λB^g blame is raised in both cases. However, in λB^r, because $\star \to \star$ is consistent with $\mathsf{Int} \to \mathsf{Int}$, the annotation chain of functions is eliminated, then beta reduction applies, and it successfully reduces to an integer value.

Space Efficiency. Besides the different semantics with respect to the λB^g and λB calculi, an interesting aspect of this alternative semantics is better space efficiency. Unlike the blame calculus or λB^g, where functions with an arbitrary number of annotations are values, that is not the case in λB^r. Because of the blame recovery semantics it is possible to discard intermediate types when reducing expressions with chains of annotations. Instead of wrappers for higher-order casts, function values have just 2 annotations. Some concrete examples for higher-order casts (functions) are:

$$(\lambda x.\ x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star : \mathsf{Int} \to \mathsf{Int} \longmapsto^* (\lambda x.\ x : \mathsf{Int} \to \mathsf{Int}) : \mathsf{Int} \to \mathsf{Int}$$
$$(\lambda x.\ x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star : Bool \to Bool \longmapsto^* [\lambda x.x : Int \to Int]_{Bool \to Bool}$$
$$(\lambda x.\ x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star : Bool \to Bool : \star \to \star \longmapsto^* \lambda x.\ x : \mathsf{Int} \to \mathsf{Int} : \star \to \star$$

For the first example, because the source type $\mathsf{Int} \to \mathsf{Int}$ is consistent with the target type $\mathsf{Int} \to \mathsf{Int}$, the intermediate types are ignored and the resulting value is $(\lambda x.\ x : \mathsf{Int} \to \mathsf{Int}) : \mathsf{Int} \to \mathsf{Int}$. For the second example, because the source type $\mathsf{Int} \to \mathsf{Int}$ is not consistent with the target type $Bool \to Bool$, instead of raising blame immediately, the source type will be stored in a saved value: $[\lambda x.\ x : \mathsf{Int} \to \mathsf{Int}]_{Bool \to Bool}$. Later, if the saved value is applied to an argument, blame will be raised, since a saved value is denoting that the function has inconsistent source and target types. The third example, is similar to the second example except that there is an extra final target type $\star \to \star$. Thus, since the initial source type $\mathsf{Int} \to \mathsf{Int}$ is consistent with the final target type $\star \to \star$ the final value is $(\lambda x.\ x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star$. The above examples illustrate that at most there will be 2 type annotations in values. In contrast, for the blame calculus, the three examples are values where all the annotations are accumulated.

Saved Expressions. To realize the blame recovery semantics we introduce *saved expressions/values*, which are used to signal *potential blame*. A saved value is generated whenever

some target type arising from an annotation is inconsistent with the current value. If further annotations are processed after a saved value is generated, recovery from blame is possible. Take the third example above again. The full reduction steps for that example are:

$$
\begin{aligned}
&(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star : \mathit{Bool} \to \mathit{Bool} : \star \to \star \\
&\longmapsto [\lambda x.x : \mathit{Int} \to \mathit{Int}]_{\mathit{Bool} \to \mathit{Bool}} : \star \to \star \\
&\longmapsto \lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \star \to \star
\end{aligned}
$$

An intermediate saved value is generated, but because there is still one more consistent annotation ($\star \to \star$), the value is recovered from the saved expression, revoking the potential reason to raise blame.

Blame in λB and λB^r. While λB^r has a different semantics from λB (and λB^g), the semantics of the two calculi is still closely related. In particular, with respect to blame, a program that does not raise blame in λB^g or λB will also not raise blame in λB^r. Achieving this goal is not simple, because of the semantics of the blame calculus and λB^g for higher-order casts. For instance, consider the following λB^g program:

$$((\lambda x.\, x : (\mathit{Bool} \to \mathit{Bool}) \to (\mathit{Bool} \to \mathit{Bool})) : \star \to \star : (\mathsf{Int} \to \mathsf{Int}) \to (\mathsf{Int} \to \mathsf{Int}))\, (\lambda x.\, x : \mathsf{Int} \to \mathsf{Int})$$

In this well-typed program the lambda expression being applied has inconsistent type annotations. However, because in the blame calculus and λB^g the semantics of higher-order casts is lazy, this program will *not raise blame*. Instead, it eventually reduces to:

$$(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \mathsf{Int} \to \mathsf{Int} : \star \to \star : \mathit{Bool} \to \mathit{Bool} : \mathit{Bool} \to \mathit{Bool} : \star \to \star : \mathsf{Int} \to \mathsf{Int}$$

which is another lambda expression (arising from the argument) with inconsistent type annotations.

In λB^r the applied lambda expression in the original program would first be reduced to:

$$[\lambda x.\, x : (\mathit{Bool} \to \mathit{Bool}) \to (\mathit{Bool} \to \mathit{Bool})]_{(\mathsf{Int} \to \mathsf{Int}) \to (\mathsf{Int} \to \mathsf{Int})}$$

To achieve the same semantics as λB^g or λB, the design of λB^r has to include rules that can still perform beta-reduction for saved values being applied. In particular, λB^r has the following rule :

$$\frac{v \longmapsto_{A_1, \star, A_2} v'}{([\lambda x.\, e : A_1 \to B_1]_{A_2 \to B_2})\, v \longmapsto e[x \mapsto v'] : B_1 : \star : B_2} \textsc{VStep-apps}$$

With such a rule, the program above is reduced in λB^r as follows:

$$
\begin{aligned}
&[\lambda x.x : (\mathit{Bool} \to \mathit{Bool}) \to (\mathit{Bool} \to \mathit{Bool})]_{(\mathit{Int} \to \mathit{Int}) \to (\mathit{Int} \to \mathit{Int})} (\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) \\
&\longmapsto^* (\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \mathsf{Int} \to \mathsf{Int}
\end{aligned}
$$

3 The λB^g Calculus: Syntax, Typing and Semantics

In this section, we will introduce the gradually typed λB^g calculus. The semantics of the λB^g calculus follows closely the semantics of the λB cast calculus, and it employs a type-directed operational semantics [25] to have a direct operational semantics. λB^g uses bidirectional type-checking [33]. We prove a soundness result between the semantics of the λB^g and λB calculi (ignoring blame labels), as well as the usual type soundness property.

Syntax

Types	$A, B ::= Int \mid \star \mid A \to B$
Ground types	$G ::= Int \mid \star \to \star$
Constants	$c ::= i \mid ...$
Terms	$e ::= c \mid x \mid e : A \mid e_1\ e_2 \mid \lambda x.e : A \to B$
Result	$r ::= e \mid blame$
Values	$v ::= c \mid v : A \to B \mid \lambda x.e : A \to B \mid v : \star$
Context	$\Gamma ::= \cdot \mid \Gamma, x : A$
Frame	$F ::= [] \ e \mid v \ [] \mid [] : A$
Typing modes	$\Leftrightarrow ::= \Rightarrow \mid \Leftarrow$
Syntactic sugar	$\lambda x.e \equiv \lambda x.e : \star \to \star$

value e *(Well-formed values for λB^g calculus)*

$$\frac{}{value\ c}\ \text{VALUE-C} \qquad \frac{}{value\ \lambda x.\, e : A \to B}\ \text{VALUE-ANNO}$$

$$\frac{\rceil v \lceil = C \to D}{value\ v : A \to B}\ \text{VALUE-FANNO} \qquad \frac{Ground\ \rceil v \lceil}{value\ v : \star}\ \text{VALUE-DYN}$$

Figure 2 Syntax and well-formed values for the λB^g calculus.

3.1 Syntax

The syntax of λB^g calculus is shown in Figure 2.

Types and Ground types. Meta-variables A and B range over types. There is a basic type: the integer type Int. The calculus also has function types $A \to B$, and dynamic types $\star$. The type $\star$ is used to denote the dynamic type which is unknown. Just like in λB calculus, ground types include Int and $\star \to \star$.

Constants, Expressions and Results. Meta-variable c ranges over constants. Each constant is assigned a unique type. The constants include integers (i) of type Int. Expressions range over by the meta-variable e. There are some standard constructs which include: constants (c);variables (x); annotated expressions ($e : A$); application expressions ($e_1 e_2$) and lambda abstractions ($\lambda x.e : A \to B$). Note that lambda abstractions have the function type annotation $A \to B$, meaning that the input type is A and the output type is B. Similarly to GTLC, lambdas without type annotations are just sugar for a lambda with the annotation $\star \to \star$. Results (r) include all expressions and blame, which is used to denote cast-errors at run-time. Finally, note that, in our Coq formalization, constants such as addition of integers are implemented, but omitted here for simplicity of presentation.

Value and Contexts. The meta-variable v ranges over values. Values include constants (c); lambda abstractions ($\lambda x.e : A \to B$) and a special value with the syntax $v : A \to B$. Note that, similarly to λB, not all syntactic values are well-formed values. The *value* predicate, at the bottom of Figure 2, defines well-formed values. Lambda expressions annotated with a function type are values (rule VALUE-ANNO). A value v with a function type annotation is

a value if the dynamic type of the value is also a function type (rule VALUE-FANNO). The expression of $v : \star$ is a value only when the type of v is a ground type (rule VALUE-DYN). Constants are also values. Note that $\rceil v \lceil$ denotes the dynamic type of a value, and is defined as:

▶ **Definition 1** (Dynamic type). *$\rceil v \lceil$ denotes the dynamic type of the value v.*

$$\begin{aligned} \rceil i \lceil &= \mathsf{Int} \\ \rceil \lambda x.\, e : A \to B \lceil &= A \to B \\ \rceil v : A \lceil &= A \end{aligned}$$

The dynamic type is the most specific type of a value among all the other types. Finally, typing contexts are standard. Γ is used to track the bound variables x with their type A.

Frame and Typing modes. The meta-variable F ranges over frames [39] which is a form of evaluation contexts [31]. The frame is mostly standard, though it is perhaps noteworthy that it includes annotated expressions. $\Leftrightarrow$ is used to represent the two modes of the bidirectional typing judgment. The $\Rightarrow$ mode is the synthesis (inference) mode and $\Leftarrow$ mode is the checking mode.

3.2 Typing

We use bidirectional typing for our typing rules. The typing judgment is represented as $\Gamma \vdash e \Leftrightarrow A$, which means that the expression e could be inferred or checked by the type A under the typing environment Γ. We ignore the highlighted parts, and explain them later in Section 3.4.

Typing Relation. The typing relation of the λB^g calculus is shown in Figure 3. Most of the rules in inference mode follow the λB calculus's type system. The typing for constants (rule TYP-C) recovers the type of the constants using the definition of dynamic types. The rule TYP-VAR for variables is standard. For lambda expressions, the λB^g calculus is different from the λB calculus: in the λB^g calculus the type of a lambda expression is given. Thus the body of the lambda expression is checked with a target type that should be consistent to the type of the lambda body. For applications $e_1\ e_2$, the rule is standard for bi-directional type-checking: the type of e_1 is inferred, and the type of e_2 is checked against the domain type of e_1. The rule for annotations (rule TYP-ANNO) is also standard, inferring the annotated type, while checking the expression in the against the annotated type. All the consistency checks happen in the subsumption rule (rule TYP-SIM). However, it is important to notice that since the subsumption rule is in checking mode, all consistency checks can only happen when typing is invoked in the checking mode.

Two important properties of the typing relation is that it computes dynamic types for the inference mode, and if an expression e can be checked with type A, then e can be inferred with some type B:

▶ **Lemma 2** (Dynamic Types). *For any value v, if $\cdot \vdash v \Rightarrow A$ then $\rceil v \lceil = A$.*

▶ **Lemma 3** (Checking to inference mode). *If $\Gamma \vdash e \Leftarrow A$ then $\exists B$, $\Gamma \vdash e \Rightarrow B$.*

$\boxed{\Gamma \vdash e \Leftrightarrow A \rightsquigarrow \mathrm{t}}$ *(Typing of λB^g)*

$$\textsc{Typ-c}\ \frac{}{\Gamma \vdash c \Rightarrow \rceil c \lceil \rightsquigarrow \mathrm{c}} \qquad \textsc{Typ-var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A \rightsquigarrow x} \qquad \textsc{Typ-abs}\ \frac{\Gamma, x : A \vdash e \Leftarrow B \rightsquigarrow \mathrm{t}}{\Gamma \vdash \lambda x.\, e : A \to B \Rightarrow A \to B \rightsquigarrow \lambda x{:}A.\, t}$$

$$\textsc{Typ-app}\ \frac{\Gamma \vdash e_1 \Rightarrow A \to B \rightsquigarrow t_1 \quad \Gamma \vdash e_2 \Leftarrow A \rightsquigarrow t_2}{\Gamma \vdash e_1\, e_2 \Rightarrow B \rightsquigarrow t_1\ t_2} \qquad \textsc{Typ-anno}\ \frac{\Gamma \vdash e \Leftarrow A \rightsquigarrow \mathrm{t}}{\Gamma \vdash e : A \Rightarrow A \rightsquigarrow \mathrm{t}} \qquad \textsc{Typ-sim}\ \frac{\Gamma \vdash e \Rightarrow A \rightsquigarrow \mathrm{t} \quad A \sim B}{\Gamma \vdash e \Leftarrow B \rightsquigarrow t : A \Rightarrow B}$$

Figure 3 Type system of the λB^g calculus.

$\boxed{v \longmapsto_A r}$ *(Typed Reduction for λB^g calculus)*

$$\textsc{TReduce-abs}\ \frac{\rceil v \lceil = C \to D \quad C \to D \sim A \to B}{v \longmapsto_{A \to B} v : A \to B} \qquad \textsc{TReduce-v}\ \frac{\mathit{Ground}\ \rceil v \lceil}{v \longmapsto_\star v : \star} \qquad \textsc{TReduce-lit}\ \frac{}{i \longmapsto_{\mathsf{Int}} i} \qquad \textsc{TReduce-dd}\ \frac{}{v : \star \longmapsto_\star v : \star}$$

$$\textsc{TReduce-anyd}\ \frac{\mathit{FLike}\ \rceil v \lceil}{v \longmapsto_\star v : \star \to \star : \star} \qquad \textsc{TReduce-dyna}\ \frac{\mathit{FLike}\ A \qquad \rceil v \lceil \sim A}{v : \star \longmapsto_A v : A} \qquad \textsc{TReduce-vany}\ \frac{}{v : \star \longmapsto_{\rceil v \lceil} v} \qquad \textsc{TReduce-blame}\ \frac{\rceil v \lceil \nsim A}{v : \star \longmapsto_A \mathit{blame}}$$

Figure 4 Typed Reduction for the λB^g calculus.

Consistency. Consistency plays an important role in a gradually type lambda calculus. Consistency acts as a relaxed equality relation. The consistency relation is the same as λB, and is already shown in Figure 1. In consistency, the reflexivity and symmetry properties hold. However, it is well-known that consistency is not a transitive relation. If consistency were transitive then every type would be consistent with any other type [43].

3.3 Dynamic Semantics

The dynamic semantics of λB^g employs a type-directed operational semantics (TDOS) [25]. In TDOS, besides the usual reduction relation, there is a special *typed reduction* relation for values that is used to further reduce values based on the type of the value. Typed reduction is used by the TDOS reduction relation. In a gradually typed calculus with TDOS the typed reduction relation plays a role analogous to various cast-related reduction rules in a cast calculus. We first introduce typed reduction and then move on to the definition of reduction.

Typed Reduction. We reduce a value under a certain type using the typed reduction relation. The form of the typed reduction relation is $v \longmapsto_A r$, which means that a value v annotated with A reduces under type A to a result r. Note that the result r produced by typed reduction can only be a value or *blame*. Blame is raised during typed reduction if

we try to reduce the value under a type that is not consistent with the type of the value. For instance trying to reduce the value $1 : \star$ under the type $Bool$ will raise blame. Thus, it should be clear that typed reduction mimics the behavior of casts in cast calculi like the λB calculus. In the λB calculus, in a cast $t : B \Rightarrow A$, t should be a cast from a source type B to a target type A. Using typed reduction, the type A is the target type, whereas the dynamic type of v is the source type.

Figure 4 shows the rules of typed reduction. Rule TReduce-abs and rule TReduce-v just add a type annotation to the value. In rule TReduce-abs the dynamic type of the value is a function type, thus v annotated with $A \to B$ is a value. In rule TReduce-v, $v : \star$ is also a value when the dynamic type of v is a ground type. Rule TReduce-lit is for integer values: an integer i being reduced under the integer type results in the same integer i. A value $v : \star$ type-reduced under $\star$ returns the original value as well (rule TReduce-dd). In rule TReduce-anyd, the premise is that the dynamic type of v should be a function-like type ($FLike$). The definition of $FLike$, which plays a role analogous to $ug(A, \star \to \star)$ in λB, is:

$$\boxed{\text{FLike} \quad \text{A} \quad ::= \quad A \neq \star \wedge A \neq \star \to \star \wedge A \sim \star \to \star}$$

If a type A is $FLike$ then it is not the type $\star$ and the type $\star \to \star$, but should be consistent with $\star \to \star$. In other words, the dynamic type of v should be any function type $A \to B$ except for $\star \to \star$. In the end v is type-reduced under type $\star$ and returns the value $v : \star \to \star : \star$. In rule TReduce-vany, $v : \star$ is type-reduced under the dynamic type of v, returning v and dropping the annotation $\star$. In rule TReduce-blame, if the dynamic type of v is not consistent to the type A that we are type-reducing, then blame is raised. Finally, in rule TReduce-dyna, a value $v : \star$ being type-reduced under type A (where A is function-like and the dynamic type of v is consistent with A) results in $v : A$. That is the annotation $\star$ gets replaced by the function type A.

Properties of Typed Reduction. Some properties of typed reduction of λB^g calculus are shown next:

▶ **Lemma 4** (Typed reduction preserves well-formedness of values). *If value v and $v \longmapsto_A v'$ then value v'.*

▶ **Lemma 5** (Preservation of Typed Reduction). *If $\cdot \vdash v \Leftarrow B$ and $v \longmapsto_A v'$ then $\cdot \vdash v' \Rightarrow A$.*

▶ **Lemma 6** (Progress of Typed Reduction). *If $\cdot \vdash v \Leftarrow A$ then $\exists v'$, $v \longmapsto_A v'$ or $v \longmapsto_A$ blame.*

▶ **Lemma 7** (Determinism of Typed Reduction). *If $\cdot \vdash v \Leftarrow B$, $v \longmapsto_A r_1$ and $v \longmapsto_A r_2$ then $r_1 = r_2$.*

▶ **Lemma 8** (Typed Reduction Respects Consistency). *If $v \longmapsto_A v'$ then $\rceil v \lceil \sim A$.*

According to Lemma 4, if the result of a value type-reduced under a type A is not blame, then it should be a well-formed value. Lemma 5 shows that the target type A is preserved after typed reduction: if a value v is type-reduced by A, the result type of v' is of type A. Note that this lemma (and some others) have a premise that ensures that the value under typed reduction must be well-typed under some type B. That is, the lemma only holds for well-typed values (which are the only ones that we care about). Lemma 6 shows that if a value v is well-typed with A, then type-reducing the value will either return a

$\boxed{e \longmapsto r}$ *(Small-step Semantics)*

$$\textsc{Step-eval}\ \frac{e \longmapsto e'}{F.\,e \longmapsto F.\,e'} \qquad \textsc{Step-blame}\ \frac{e \longmapsto \mathit{blame}}{F.\,e \longmapsto \mathit{blame}} \qquad \textsc{Step-beta}\ \frac{v \longmapsto_A v'}{(\lambda x.\,e : A \to B)\,v \longmapsto e[x \mapsto v'] : B}$$

$$\textsc{Step-betap}\ \frac{v \longmapsto_A \mathit{blame}}{(\lambda x.\,e : A \to B)\,v \longmapsto \mathit{blame}} \qquad \textsc{Step-annov}\ \frac{\mathit{not}\ (\mathit{value}\ (v : A)) \quad v \longmapsto_A r}{v : A \longmapsto r}$$

$$\textsc{Step-abeta}\ \frac{\mathit{value}\ (v_1 : A \to B) \quad v_2 \longmapsto_A v_2'}{(v_1 : A \to B)\,v_2 \longmapsto (v_1\,v_2') : B} \qquad \textsc{Step-abetap}\ \frac{\mathit{value}\ (v_1 : A \to B) \quad v_2 \longmapsto_A \mathit{blame}}{(v_1 : A \to B)\,v_2 \longmapsto \mathit{blame}}$$

Figure 5 Semantics of λB^g.

well-formed value or blame. The typed reduction relation is deterministic for well-typed values (Lemma 7): if a well-typed value v is type-reduced by type A, the result will be unique. Finally, if v is type-reduced by A, the dynamic type of v should be consistent with type A (Lemma 8). Most of these lemmas are proved by induction on typed reduction relation.

Reduction. The reduction rules are shown in Figure 5. Rule Step-eval and rule Step-blame are standard evaluation context reduction rules. Rule Step-beta is the beta reduction rule. Importantly, note that typed reduction under type A is needed for v: that is we type-reduce value v to v' and replace the bound variable x in e by v'. Rule Step-betap applies when v type-reducing under type A raises blame. Rule Step-annov states that v type-reduces under type A to return r. Rule Step-abeta says v_2 type-reduces by type A to get v_2' and v_1 will erase the annotation. The expression $v_1\ v_2'$ in the result is annotated with type B. Rule Step-abetap covers the case when v_2 type-reducing under type A raises blame.

Determinism. The operational semantics of λB^g is *deterministic*: a well-typed expression reduces to a unique result. Theorem 9 is proved using Lemma 7.

▶ **Theorem 9** (Determinism of λB^g calculus). *If* $\cdot \vdash e \Leftarrow A$, $e \longmapsto r_1$ *and* $e \longmapsto r_2$ *then* $r_1 = r_2$.

Type Safety. The λB^g calculus is type safe. Theorem 10 says that if an expression is well-typed with type A, the type will be preserved after the reduction. Progress is given by Theorem 11. A well-typed expression e is either a value or there exists an expression e' which e could reduce to, or e reduces to blame.

▶ **Theorem 10** (Type Preservation of λB^g Calculus). *If* $\cdot \vdash e \Leftrightarrow A$ *and* $e \longmapsto e'$ *then* $\cdot \vdash e' \Leftrightarrow A$.

▶ **Theorem 11** (Progress of λB^g Calculus). *If* $\cdot \vdash e \Leftrightarrow A$ *then* e *is a value or* $\exists e'$, $e \longmapsto e'$ *or* $e \longmapsto \mathit{blame}$.

3.4 Soundness to λB

The judgment $\Gamma \vdash e \Leftrightarrow A \rightsquigarrow t$, shown in Figure 3 has an elaboration step from λB^g expressions to λB expressions in the gray portion of the judgement. This elaboration step is used to prove a soundness result between the semantics of λB^g and λB. A first property, given by Theorem 12, is that the elaboration is type-safe. Theorem 13 and Theorem 14 show the soundness property between the dynamic semantics of λB^g and λB. The soundness result is proved using the auxiliary lemmas 15 and 16.

▶ **Theorem 12** (Type-Safety of Elaboration). *If* $\Gamma \vdash e \Leftrightarrow A \rightsquigarrow t$ *then* $\Gamma \vdash t : A$.

▶ **Theorem 13** (Soundness of λB^g calculus semantics with respect to λB calculus semantics). *If* $\cdot \vdash e \Leftrightarrow A \rightsquigarrow t$ *and* $e \longmapsto e'$ *then* $\exists t'$, $t \longmapsto^* t'$ *and* $\cdot \vdash e' \Leftrightarrow A \rightsquigarrow t'$.

▶ **Theorem 14** (Soundness of λB^g calculus semantics with respect to λB calculus semantics). *If* $\cdot \vdash e \Leftrightarrow A \rightsquigarrow t$ *and* $e \longmapsto \mathit{blame}$ *then* $t \longmapsto^* \mathit{blame}$.

▶ **Lemma 15** (Soundness of Typed Reduction λB^g calculus with respect to λB calculus semantics). *If* $\cdot \vdash v : A \Rightarrow A \rightsquigarrow t$ *and* $v \longmapsto_A v'$ *then* $\exists t'$, $t \longmapsto^* t'$ *and* $\cdot \vdash v' \Rightarrow A \rightsquigarrow t'$.

▶ **Lemma 16** (Soundness of Typed Reduction λB^g calculus with respect to λB calculus semantics). *If* $\cdot \vdash v : A \Rightarrow A \rightsquigarrow t$ *and* $v \longmapsto_A \mathit{blame}$ *then* $t \longmapsto^* \mathit{blame}$.

4 The λB^r Calculus and the Blame Recovery Semantics

In this section, we will introduce a gradually typed calculus with a blame recovery semantics. The idea of the blame recovery semantics is essentially to ignore intermediate inconsistent types in annotations. Thus, if blame arises from intermediate type annotations, but later the final source type is found to be consistent to the final target type then blame is not raised. A nice aspect of the blame recovery semantics is that it avoids accumulating type annotations, leading to a more space-efficient representation of values. The details of syntax, typing and semantics of λB^r calculus are shown below.

4.1 Syntax

The syntax of the λB^r calculus is shown in Figure 6.

Types. Types are the same as in λB^g. A type is either a integer type Int, a function type $A \to B$ or a dynamic type $\star$.

Expressions and Results. For expressions and results, λB^r extends λB^g with two expression forms: base expressions and saved expressions. Base expressions (ss) include annotated lambda expressions and integers (i). Saved expressions ($[s]_{A \to B}$) store a lambda expression and a type $A \to B$ which is not consistent with the type of the lambda expression. The lambda expressions stored in saved expressions are denoted as s. In our Coq formalization, addition is also implemented and omitted here for simplicity of presentation.

Values. As in λB^g, v denotes values, which are base expressions ss or saved forms annotated with a type. Thus $i : A$ and $\lambda x.\, e : A \to B : C$ are examples of such expressions. Notably, in contrast with λB^g, λB^r's notion of (well-formed) values is purely syntactic: no additional constraints (besides) syntax are needed. Moreover, it should be noted that in λB^r values have a bounded number of annotations (up-to 2 for lambda and saved values), unlike the λB^g calculus.

Types	$A, B, C ::= Int \mid A \to B \mid \star$
Saved Forms	$s ::= \lambda x.e : A \to B$
Base Expressions	$ss ::= s \mid i$
Expressions	$e ::= x \mid e : A \mid e_1\ e_2 \mid ss \mid [s]_{C \to D}$
Results	$r ::= e \mid blame$
Value	$v ::= ss : A \mid [s]_A$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : A$
Frame	$F ::= v\ [] \mid []\ e$
Typing modes	$\Leftrightarrow ::= \Rightarrow \mid \Leftarrow$
Syntactic sugar	$\lambda x.u \equiv \lambda x.e : \star \to \star$

Figure 6 Syntax of the λB^r calculus (syntax that is the same as λB^g in lighter gray).

$\boxed{\Gamma \vdash e \Leftrightarrow A}$ *(New Typing Rules)*

$$\frac{\cdot \vdash s \Rightarrow C \to D \qquad A \to B \nsim C \to D}{\Gamma \vdash [s]_{A \to B} \Rightarrow A \to B}\ \textsc{Etyp-save}$$

Figure 7 Type system of the λB^r calculus. Only new typing rules are shown. All other typing rules are the same as Figure 3.

Contexts, Frame and Typing modes. Typing environments and typing modes are just the same as in the λB^g calculus. Compared to the λB^g calculus, annotation contexts are not in the frame. This change is because in the λB^g calculus we accumulate the annotations, but in the λB^r calculus we employ a blame recovery semantics. If annotated expressions were formulated in the frame, we could not formulate a rule that recovers a saved value.

4.2 Typing

As the λB^g calculus, bidirectional typing is used. Most of the rules are standard and the same as those used by the λB^g calculus in Figure 3. The only novel rule is rule Etyp-save, which states that saved forms s should be well-typed with type $C \to D$, and the type $A \to B$ in the saved expression $[s]_{A \to B}$ is not consistent with type $C \to D$. The context is empty because we only use saved expressions as intermediate results during reduction and such results must be closed.

Dynamic type for the λB^r calculus. As in the λB^g calculus, dynamic types play an important role in the calculus. $\rceil v \lceil$ denotes the dynamic type of v, and $\rceil ss \lceil$ denotes the dynamic type of ss. We need both dynamic types for values and base expressions ss, and we can define dynamic types easily as follows:

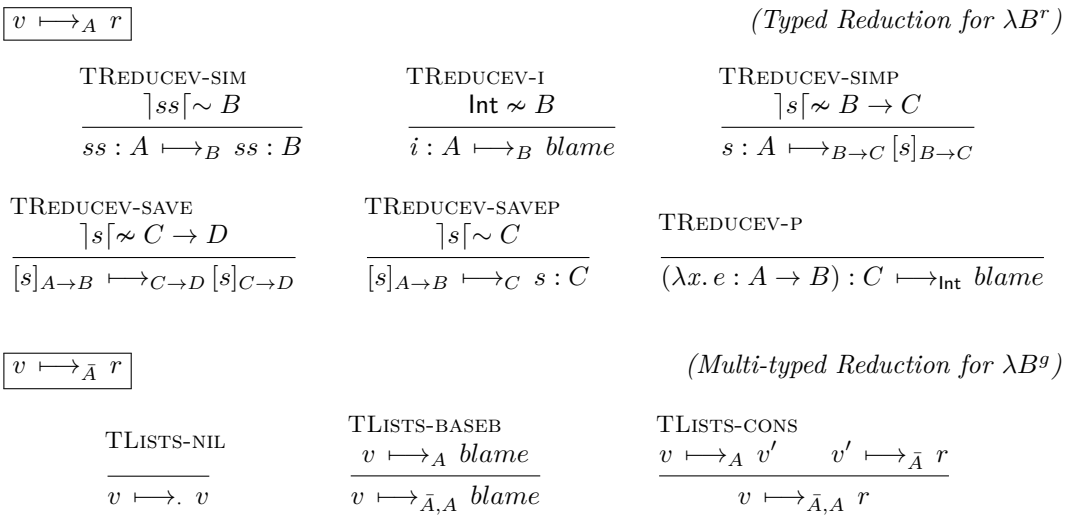

Figure 8 Typed Reduction for the λB^r Calculus.

▶ **Definition 17** (Dynamic type). *$\rceil ss \lceil$ returns the dynamic type of the base expressions ss. $\rceil v \lceil$ returns the dynamic type of the value v.*

$$\begin{aligned} \rceil i \lceil &= \mathsf{Int} \\ \rceil \lambda x.\, e : A \to B \lceil &= A \to B \\ \rceil s : A \lceil &= A \\ \rceil [s]_A \lceil &= A \end{aligned}$$

Two lemmas about dynamic types and a typing lemma about checking mode are:

▶ **Lemma 18** (Dynamic Types of Values). *If $\cdot \vdash v \Rightarrow A$ then $\rceil v \lceil = A$.*

▶ **Lemma 19** (Dynamic Types of Base Expressions). *If $\cdot \vdash ss \Rightarrow A$ then $\rceil ss \lceil = A$.*

▶ **Lemma 20** (Checked expressions can be inferred). *If $\Gamma \vdash e \Leftarrow A$ then $\exists B$, $\Gamma \vdash e \Rightarrow B$.*

4.3 Dynamic Semantics

As in the λB^g calculus, typed reduction is used in the semantics to get a direct operational semantics. Interestingly, the calculus uses not only typed reduction with single type, but also typed reduction for a collection of types.

Typed Reduction. The typed reduction rules are shown in Figure 8. Rule TReducev-sim shows that $ss : A$ type-reduces by type B to $ss : B$, if the type of ss is consistent with type B. If the type of ss is not consistent with type B, the cases for integer (i) and lambda ($\lambda x.\, e : A \to B$), which are included in ss, are different. For $i : A$, rule TReducev-i shows that if type B is not consistent with Int, it raises blame. For a value of the form $(\lambda x.\, e : A_1 \to B_1) : A$, if the type used for typed reduction is a function type $B \to C$, then the value reduces to $[\lambda x.\, e : A_1 \to B_1]_{B\to C}$ using rule TReducev-simp. However

rule TReducev-p says that if the type used for typed reduction is Int, then blame is raised. Rule TReducev-savep says that if the dynamic type of a saved form in $[s]_{A \to B}$ is consistent with C, then we can recover and return $s : C$. Otherwise, if the dynamic type of the saved form s is also not consistent with $C \to D$, then $[s]_{A \to B}$ type-reduces to another saved value $[s]_{C \to D}$ as shown by rule TReducev-save.

An Example. Lets take an example to explain behavior of typed reduction with blame recovery semantics. Suppose that we take a chain of annotations $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \mathsf{Int} \to \mathsf{Int} : \star : \mathit{Bool} \to \mathit{Bool} : \star$. Firstly, the dynamic type of $\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}$ is consistent with type $\star$ and the intermediate type $\mathsf{Int} \to \mathsf{Int}$ is erased. Then the dynamic type of $\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}$ is not consistent with $\mathit{Bool} \to \mathit{Bool}$. While reducing such an expression, an intermediate saved expression $[\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}]_{\mathsf{Int} \to \mathsf{Int}}$ is generated. However, the saved expression would later be recovered because the final type annotation $\star$ is consistent with the dynamic type of the value.

The typed reduction (and reduction) steps to reduce such an expression are shown next:

$(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \mathsf{Int} \to \mathsf{Int} : \star : \mathit{Bool} \to \mathit{Bool} : \star$
$\longmapsto$ {by STEP-ANNOV and typed reduction under $\star$}
$(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star : \mathit{Bool} \to \mathit{Bool} : \star$
$\longmapsto$ {by STEP-ANNOV and typed reduction under $\mathit{Bool} \to \mathit{Bool}$}
$([\lambda x : \mathsf{Int} \to \mathsf{Int}]_{\mathit{Bool} \to \mathit{Bool}}) : \star$
$\longmapsto$ {by STEP-ANNOV and typed reduction under $\star$}
$(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star$

Typed Reduction Properties. Typed reduction for the λB^r calculus has some interesting properties. The most interesting property is transitivity of typed reduction, which may come as a surprise since the consistency relation is not transitive, and typed reduction for λB^g is not transitive either. The transitivity lemma (Lemma 21) says that typed reduction is the same no matter whether it is type-reduced directly or indirectly via some intermediate type.

▶ **Lemma 21** (Transitivity of typed reduction). *If* $v \longmapsto_A v_1$, *and* $v_1 \longmapsto_B v_2$ *then* $v \longmapsto_B v_2$.

Lets take an example, firstly using the typed reduction of λB^g:

1) $\lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \mathsf{Int} \to \mathsf{Int} \longmapsto_{\star \to \star} \lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \mathsf{Int} \to \mathsf{Int} : \star \to \star$
2) $\lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \mathsf{Int} \to \mathsf{Int} : \star \to \star$
$\longmapsto_{\mathit{Bool} \to \mathit{Bool}} \lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \mathsf{Int} \to \mathsf{Int} : \star \to \star : \mathit{Bool} \to \mathit{Bool}$
3) $\lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \mathsf{Int} \to \mathsf{Int} \not\longmapsto_{\mathit{Bool} \to \mathit{Bool}} \lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \mathsf{Int} \to \mathsf{Int} : \star \to \star : \mathit{Bool} \to \mathit{Bool}$

The three typed reductions correspond to the two premises and the conclusion in the transitivity lemma. The last typed reduction does not hold, and is a counter-example to transitivity of typed reduction in λB^g. Although $\mathsf{Int} \to \mathsf{Int}$ is consistent with $\star \to \star$ and $\star \to \star$ is consistent with $\mathit{Bool} \to \mathit{Bool}$, $\mathsf{Int} \to \mathsf{Int}$ is not consistent with $\mathit{Bool} \to \mathit{Bool}$. Since transitivity does not hold in type consistency and the annotations are accumulated in λB^g, the transitivity of typed reduction does not hold in λB^g. However in λB^r, the annotations are not accumulated and saved expressions are used to save the source type, so the transitivity

of typed reduction holds. The following three typed reductions illustrate what happens for the above example in λB^r:

1) $\lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \mathsf{Int} \to \mathsf{Int} \longmapsto_{\star\to\star} \lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \star \to \star$
2) $\lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \star \to \star \longmapsto_{Bool\to Bool} [\lambda x.x : Int \to Int]_{Bool\to Bool}$
3) $\lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \mathsf{Int} \to \mathsf{Int} \longmapsto_{Bool\to Bool} [\lambda x.x : Int \to Int]_{Bool\to Bool}$

Additionally, typed reduction has several of the other properties for typed reduction shown in Section 3:

▶ **Lemma 22** (Preservation of Typed Reduction). *If* $\cdot \vdash v \Leftarrow B$ *and* $v \longmapsto_A v'$ *then* $\cdot \vdash v' \Rightarrow A$.

▶ **Lemma 23** (Progress of Typed Reduction). *If* $\cdot \vdash v \Leftarrow A$ *then* $\exists v'$, $v \longmapsto_A v'$.

▶ **Lemma 24** (Determinism of Typed Reduction). *If* $\cdot \vdash v \Leftarrow B$, $v \longmapsto_A r_1$ *and* $v \longmapsto_A r_2$ *then* $r_1 = r_2$.

Multi-Typed Reduction. The bottom of Figure 8 shows multi-typed reduction. If a value v is multi-type reduced with an empty type collection, then the original v is returned as shown in rule TLists-nil. Rule TLists-baseb states that value v multi-type reducing with a type collection $(\bar{A}, A)$ raises blame when v type-reduced under type A raises blame. Rule TLists-cons says that value v multi-type reducing with a type collection $(\bar{A}, A)$ returns r if v type-reduces under type A return v' and further reduction of v' under $\bar{A}$ returns r.

Reduction. Figure 9 shows the reduction rules of the λB^r calculus. Rule vstep-eval and rule vstep-blame are standard rules. Annotation expressions are not in the frame because we aim at having a blame recovery semantics. Rule vstep-annop and rule vstep-anno are standard rules. Rule vstep-abs and rule vstep-i add an extra annotation with the dynamic type to produce a value. In rule vstep-annov if v type-reduces to v' under type A then $v : A$ reduces to v'.

There are four rules related to beta-reduction. Rule vstep-beta is the main form of beta-reduction. However, the argument v needs to first be (multi)type reduced with the input types, and the annotations with the output types are added in the final expression. If the multi-typed reduction of v raise blame, then the final result is also blame as shown in rule vstep-betap. Rule vstep-apps is another form of beta-reduction that recovers the lambda expression in the saved value when the argument value v successfully type-reduces to another value. Importantly, because the dynamic type of the lambda expression and the saved value are inconsistent, an intermediate type $\star$ is added in between the inputs/output types in both multi-typed reduction and the annotations for the resulting expression. The reason why $\star$ is needed in multi-typed reduction is that without $\star$, the result of typed reduction would not be well-typed. Take an example where v is $1 : \star$ and we are multi-type reducing using the inconsistent types $(Bool, \mathsf{Int})$. The final value $1 : Bool$ is not well-typed:

$1 : \star \longmapsto_{Bool,\mathsf{Int}} 1 : \mathsf{Int} \longmapsto_{Bool} 1 : Bool$ $\qquad\qquad$ $1 : Bool$ is not well-typed!

When the multi-typed reduction of v raises blame, the final result is blame as shown in rule vstep-appsp. Note that an alternative to rule vstep-apps and rule vstep-appsp is to have a rule that always raises blame for any saved expression being applied. Such alternative rule would be significantly simpler, but would raise blame in some cases where

$\boxed{e \longmapsto r}$ *(Small-step Semantics for the λB^r calculus)*

$$\text{VSTEP-EVAL}\ \frac{e \longmapsto e'}{F.\,e \longmapsto F.\,e'} \qquad \text{VSTEP-BLAME}\ \frac{e \longmapsto \mathit{blame}}{F.\,e \longmapsto \mathit{blame}} \qquad \text{VSTEP-ANNOP}\ \frac{e \longmapsto \mathit{blame} \quad \neg(\mathit{value}\ e : A)}{e : A \longmapsto \mathit{blame}} \qquad \text{VSTEP-ANNOV}\ \frac{v \longmapsto_A v'}{v : A \longmapsto v'}$$

$$\text{VSTEP-APPS}\ \frac{v \longmapsto_{A_1,\star,A_2} v'}{([\lambda x.\,e : A_1 \to B_1]_{A_2 \to B_2})\,v \longmapsto e[x \mapsto v'] : B_1 : \star : B_2} \qquad \text{VSTEP-ANNO}\ \frac{e \longmapsto e' \quad \neg(\mathit{value}\ e : A)}{e : A \longmapsto e' : A}$$

$$\text{VSTEP-APPSP}\ \frac{v \longmapsto_{A_1,\star,A_2} \mathit{blame}}{([\lambda x.\,e : A_1 \to B_1]_{A_2 \to B_2})\,v \longmapsto \mathit{blame}} \qquad \text{VSTEP-ABS}\ \frac{}{\lambda x.\,e : A \to B \longmapsto (\lambda x.\,e : A \to B) : A \to B}$$

$$\text{VSTEP-BETA}\ \frac{v \longmapsto_{A_1,A_2} v'}{((\lambda x.\,e : A_1 \to B_1) : A_2 \to B_2)\,v \longmapsto e[x \mapsto v'] : B_1 : B_2} \qquad \text{VSTEP-I}\ \frac{}{i \longmapsto i : \mathsf{Int}}$$

$$\text{VSTEP-BETAP}\ \frac{v \longmapsto_{A_1,A_2} \mathit{blame}}{((\lambda x.\,e : A_1 \to B_1) : A_2 \to B_2)\,v \longmapsto \mathit{blame}}$$

Figure 9 Semantics of the λB^r Calculus.

the blame calculus or λB^g do not. The more complex rules VSTEP-APPS and VSTEP-APPSP are necessary to ensure that blame is not raised when a analogous program in λB^g would not raise blame. The last example in Section 2 shows this situation and illustrates the benefit of having rule VSTEP-APPS to respect the blame semantics of λB^g.

One important property is that the reduction relation is deterministic:

▶ **Theorem 25** (Determinism of λB^r calculus). *If $\cdot \vdash e \Leftarrow A$, $e \longmapsto r_1$ and $e \longmapsto r_2$ then $r_1 = r_2$.*

Type Safety. Another important property is that the λB^r calculus is type safe. Theorem 26 says that if an expression is well-typed with type A, the type will be preserved after the reduction. Progress is shown by Theorem 27. A well-typed expression e will be a value or there exists an expression e' which e could reduce to e' or e could raise blame.

▶ **Theorem 26** (Type Preservation of λB^r Calculus). *If $\cdot \vdash e \Leftrightarrow A$ and $e \longmapsto e'$ then $\cdot \vdash e' \Leftrightarrow A$.*

▶ **Theorem 27** (Progress of λB^r Calculus). *If $\cdot \vdash e \Leftrightarrow A$ then e is a value or $\exists e'$, $e \longmapsto e'$ or $e \longmapsto \mathit{blame}$.*

Less blame. λB^r raises blame strictly less often than λB^g. As we have seen in Section 2 we can find programs that raise blame in λB^g, but will result in values in λB^r. Moreover we have proved the following theorem:

$$
\begin{array}{rcl}
|\,i\,| & = & i \\
|\,\lambda x.\, e : A \to B\,| & = & \lambda x.\, |\,e\,| : A \to B \\
|\,e : A\,| & = & |\,e\,| : A \\
|\,e_1\, e_2\,| & = & |\,e_1\,|\,|\,e_2\,| \\
|\,[s]_{A \to B}\,| & = & |\,s\,| : \star \to \star : A \to B
\end{array}
$$

Figure 10 Translating λB^r expressions to λB^g.

▶ **Theorem 28** (Conformance to the blame semantics of λB^g). *If* $\cdot \vdash |\,e\,| \Leftrightarrow^g A$ *and* $e \longmapsto^r$ *blame then* $|\,e\,| \longmapsto^{g*}$ *blame.*

which states that if a λB^r expression e reduces to blame, and the corresponding λB^g expression $|e|$ is well-typed, then reducing $|e|$ also raises blame. Note that in the theorem, for disambiguation, we annotate the relations with g or r to clarify which calculus does the relation belong to. In other words a program that results in blame in λB^r will also result in blame in λB^g. Moreover, because of the soundness lemma between λB^g and λB, λB^r also raises blame less often than λB.

To prove Theorem 28 we need a translation function between λB^r expressions and λB^g expressions. In λB^r, we have saved expressions/values, while there is no such expression in λB^g. The translation function is shown in Figure 10. For instance, the λB^r expression $[\lambda x.\, x : \mathit{Bool} \to \mathit{Bool}]_{\mathsf{Int} \to \mathsf{Int}}$ would translate to $(\lambda x.\, x : \mathit{Bool} \to \mathit{Bool}) : \star \to \star : \mathsf{Int} \to \mathsf{Int}$ in λB^g.

4.4 Gradual Guarantee

Siek et al. [41] suggested that a calculus for gradual typing should also enjoy the gradual guarantee, which ensures that programs can smoothly move from being more/less dynamically typed into more/less statically typed.

Precision. The top of Figure 11 shows the precision relation on types. $A \sqsubseteq B$ means that A is more precise than B. Every type is more precise than type $\star$. A function type $A_1 \to B_1$ is more precise than $A_2 \to B_2$ if type A_1 is more precise then A_2 and type B_1 is more precise than B_2. The bottom of Figure 11 shows the precision relation of expressions. $e_1 \sqsubseteq e_2$ means that e_1 is more precise than e_2. The precision relation of expressions is derived from the precision relation of types. Every expression has a precision relation with itself. $\lambda x.\, e_1 : A_1 \to B_1$ is more precise than $\lambda x.\, e_2 : A_2 \to B_2$ if e_1 is more precise than e_2 and the types are in the precision relation. For application expressions, precision holds if $e_1 \sqsubseteq e_2$ holds and $e_1' \sqsubseteq e_2'$ holds. For annotated expressions $e_1 : A$ is more precise than $e_2 : B$ if e_1 is more precise than e_2 and A is more precise than B. For saved expressions, the precision relation is similar to annotation expressions.

Notably, a saved expression $[s_1]_{A \to B}$ is more precise than an expression $s_2 : C \to D$ if s_1 is more and precise than s_2 and the type $A \to B$ is more precise than $C \to D$ (rule EP-SA). Lets take an example, to see the use of such precision rule. The expression $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star : \mathit{Bool} \to \mathit{Bool}$ is more precise than $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star : \star \to \star$ by rule EP-ANNO. According to the reduction rules, $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star : \mathit{Bool} \to \mathit{Bool}$ reduces to $[\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}]_{\mathit{Bool} \to \mathit{Bool}}$, while $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star : \star \to \star$ reduces to $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star$. In addition, $e_1 : \star : A$ is more precise than $e_2 : B$ if e_1 is more precise than e_2 and type A is more precise than B by rule EP-ANNOL. As an example to illustrate the usefulness of this rule the expression $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star \to$

$\boxed{A \sqsubseteq B}$ *(Precision relation for types)*

$$\text{TP-I}\ \frac{}{\mathsf{Int} \sqsubseteq \mathsf{Int}} \qquad \text{TP-DYN}\ \frac{}{A \sqsubseteq \star} \qquad \text{TP-ABS}\ \frac{A_1 \sqsubseteq A_2 \qquad B_1 \sqsubseteq B_2}{(A_1 \to B_1) \sqsubseteq (A_2 \to B_2)}$$

$\boxed{e_1 \sqsubseteq e_2}$ *(Precision relation for expressions)*

$$\text{EP-REFL}\ \frac{}{e \sqsubseteq e} \qquad \text{EP-ABS}\ \frac{e_1 \sqsubseteq e_2 \qquad A_1 \sqsubseteq A_2 \qquad B_1 \sqsubseteq B_2}{\lambda x.\, e_1 : A_1 \to B_1 \sqsubseteq \lambda x.\, e_2 : A_2 \to B_2} \qquad \text{EP-APP}\ \frac{e_1 \sqsubseteq e_1' \qquad e_2 \sqsubseteq e_2'}{(e_1\, e_2) \sqsubseteq (e_1'\, e_2')}$$

$$\text{EP-ANNO}\ \frac{A \sqsubseteq B \qquad e_1 \sqsubseteq e_2}{e_1 : A \sqsubseteq e_2 : B} \qquad \text{EP-SAVE}\ \frac{A \sqsubseteq C \qquad B \sqsubseteq D \qquad s_1 \sqsubseteq s_2}{[s_1]_{A \to B} \sqsubseteq [s_2]_{C \to D}} \qquad \text{EP-SA}\ \frac{A \sqsubseteq C \qquad B \sqsubseteq D \qquad s_1 \sqsubseteq s_2}{[s_1]_{A \to B} \sqsubseteq s_2 : C \to D}$$

$$\text{EP-ANNOL}\ \frac{A \sqsubseteq B \qquad e_1 \sqsubseteq e_2}{e_1 : \star : A \sqsubseteq e_2 : B} \qquad \text{EP-SAVER}\ \frac{A \sqsubseteq C \qquad B \sqsubseteq D \qquad e_1 \sqsubseteq s_2}{e_1 : \star : A \to B \sqsubseteq [s_2]_{C \to D}}$$

Figure 11 Precision relations.

$\star : \star : \mathit{Bool} \to \mathit{Bool}$ is more precise than $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star : \mathit{Bool} \to \mathit{Bool}$. The expression $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star : \star : \mathit{Bool} \to \mathit{Bool}$ reduces to $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star : \mathit{Bool} \to \mathit{Bool}$ while $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star : \mathit{Bool} \to \mathit{Bool}$ would reduce to a saved value $[\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}]_{\mathit{Bool} \to \mathit{Bool}}$. Rule EP-SAVER shows the precision relation of these two results. Rule EP-SAVER says that $e_2 : \star : A \to B$ is more precise than $[s_2]_{C \to D}$ while e_1 is more precise than s_2 and type $A \to B$ is more precise than $C \to D$.

Static Gradual Guarantee. Theorem 29 shows that the static criteria of the gradual guarantee holds for the λB^r calculus. It says that if e is more precise than e', e has type A and e' is has type B, then type A is more precise than B.

▶ **Theorem 29** (Static Gradual Guarantee of λB^r Calculus). *If* $e \sqsubseteq e'$, $\cdot \vdash e \Rightarrow A$ *and* $\cdot \vdash e' \Rightarrow B$ *then* $A \sqsubseteq B$.

Dynamic Gradual Guarantee. The λB^r calculus has a dynamic gradual guarantee. Here we formulate a theorem for the dynamic gradual guarantee. Theorem 31 shows that if e_1 is more precise than e_2, e_1 and e_2 are well-typed, and if e_1 reduces to e_1', then e_2 reduces (in multiple steps) to e_2'. Note that e_1' is guaranteed to be more precise than e_2'. Theorem 31 is similar to the one formalized in the AGT approach [17]. A small difference is that we use a multi-step relation in the conclusion because in the precision relation we have rules like rule EP-ANNOL. If we have that $1 : \star : \mathsf{Int}$ is more precise than $1 : \star$, then $1 : \star : \mathsf{Int}$ needs to reduce to $1 : \mathsf{Int}$ while $1 : \star$ is already a value for which no step is required. Theorem 32 is derived easily from Theorem 31. The auxiliary Lemma 30, which shows the property of dynamic gradual guarantee for typed reduction, is helpful to prove Theorem 32.

▶ **Lemma 30** (Dynamic Gradual Guarantee for Typed Reduction). *If* $v_1 \sqsubseteq v_2$, $\cdot \vdash v_1 \Leftarrow A$, $\cdot \vdash v_2 \Leftarrow B$, $A \sqsubseteq B$ *and* $v_1 \longmapsto_A v_1'$ *then* $\exists v_2'$, $v_2 \longmapsto_A v_2'$ *and* $v_1' \sqsubseteq v_2'$.

▶ **Theorem 31** (Dynamic Gradual Guarantee). *If* $e_1 \sqsubseteq e_2$, $\cdot \vdash e_1 \Leftrightarrow A$, $\cdot \vdash e_2 \Leftrightarrow B$ *and* $e_1 \longmapsto e_1'$ *then* $\exists e_2'$, $e_2 \longmapsto^* e_2'$ *and* $e_1' \sqsubseteq e_2'$.

▶ **Theorem 32** (Dynamic Gradual Guarantee). *If* $e_1 \sqsubseteq e_2$, $\cdot \vdash e_1 \Leftrightarrow A$, $\cdot \vdash e_2 \Leftrightarrow B$ *and* $e_1 \longmapsto^* v_1$ *then* $\exists v_2$, $e_2 \longmapsto^* v_2$ *and* $v_1 \sqsubseteq v_2$.

5 Related Work

This section discusses related work. We focus on gradual typing criteria, cast calculi, gradually typed calculi, the AGT approach and typed operational semantics.

Gradual Typing Languages and Criteria. There is a growing number of research work focusing on combining static and dynamic typing [2, 7, 22, 29, 30, 34, 45, 46, 48, 53]. Many mainstream programming languages have some form of integration between static and dynamic typing. These include TypeScript [6], Dart [8], Hack [51], Cecil [10], Bigloo [35], Visual Basic.NET [30], ProfessorJ [20], Lisp [44], Dylan [36] and Typed Racket [50].

Much work in the research literature of gradual typing focuses on the pursuit of sound gradual typing. In sound gradual typing the idea is that some form of type-safety should still be preserved. This often requires some dynamic checks that arise from static type information. Furthermore, gradually typed languages should provide a smooth integration between dynamic and static typing. For instance, one of the criteria for gradual typing is that a program that has static types should behave equivalently to a standard statically typed program [43]. Siek et al. [41], proposed the *gradual guarantee* to clarify the kinds of guarantees expected in gradually typed languages. The principle of the gradual guarantee is that static and dynamic behavior changes by changing type annotations. For the static (gradual) guarantee, the type of a more precise term should be more precise than the type of a less precise term. For the dynamic (gradual) guarantee, any program that runs without errors should continue to do so with less precise types.

Cast calculi. Due to the unknown type and consistency of the gradual typing, more programs are accepted by a gradual type system compared to their analogous static type system. Therefore, some runtime checks are required at run-time to ensure type-safety. The most common approach to give the semantics to a gradually typed language is by translating to a cast calculus, which has a standard dynamic semantics. The process of the translation to cast calculi involves inserting casts whenever type consistency holds.

There are several varieties of cast calculi. Findler and Felleisen [15] introduced assertion-based contracts for higher-order functions. Based on mirrors and contracts, Gray et al. [20] shown a new model to implement Java and Scheme. Henglein's dynamically typed λ-calculus [23] is an extention of the statically typed λ-calculus with a dynamic type and explicit dynamic type coercions. Tobin-Hochstadt and Felleisen [49] presented a framework of interlanguage migration, which ensures type-safety.

Wadler and Findler [52] introduced the blame calculus. The blame comes from Findler and Felleisen's contracts and tracks the locations where cast errors happen using blame labels. Siek et al. [37] explored the design space of higher-order casts. For first-order casts (casts on base types), the semantics is straightforward. But there are issues for higher-order casts (functions): a higher-order cast is not checked immediately. For higher-order casts,

checking is deferred until the function is applied to an argument. After application, the cast is checked against the argument and return value. A cast is used as a wrapper and splitted until the wrapped function is applied to an argument. Wrappers for higher-order casts can lead to unbounded space consumption [24].

There are some different designs for the dynamic semantics for casts calculi in the literature. Herman et al. [24] and Wadler et al. [52] use a lazy error detection strategy. With this strategy, run-time type checking is not performed when a higher-order cast is applied to a value. Instead, lazy error detection coerces the arguments of a function to the target type, and checking is only done when the argument is applied. Siek et al. [43] use a different strategy where checking higher-order casts is performed immediately when the source type is the dynamic type ($\star$). Otherwise, the later strategy is the same as lazy error detection. In the λB^r calculus, we introduce the blame recovery semantics, which is essentially to ignore intermediate type annotations in a chain of type annotations for higher-order functions. The idea is to only raise blame if the initial source type of the value and final target types are not consistent. Otherwise, even if intermediate annotations trigger type conversions, which would not be consistent, the final result can still be a value provided that the initial source and final target types are themselves consistent. This alternative approach has a bounded number of annotations, which avoids the accumulation of type annotations (up-to 2 for higher-order values).

Siek and Wadler [42] introduced threesomes, where a cast consists of three types instead of two types (twosomes) of the blame calculus. The threesome calculus is proved to be equivalent to blame calculus and a coercion-based calculus without blame labels but with space efficiency. The three types in a threesome contain the source, intermediate and target types. The intermediate type is computed by the greatest lower bound of all the intermediate types. For example, in a chain of casts:

$$1 : Int \Rightarrow \star : \star \Rightarrow Int : Int \Rightarrow Int$$

the source type and target are both Int and the intermediate type is computed to be the greatest lower bound of $\star$ and Int, resulting in Int. Compared to our λB^r calculus, function values are twosomes (borrowing Siek and Wadler's terminology). Instead of accumulating annotations, and computing the intermediate types, we simply discard them.

Like λB^r, Castagna and Lanvin [9] propose a calculus that discards annotations for higher-order functions. However their semantics is different. The key difference is that in their semantics intermediate casts are discarded after consistency checks are performed. This means that programs such as (here using our notation):

$$\lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \mathsf{Int} \to \mathsf{Int} : \star \to \star : Bool \to Bool : \star \to \star$$

will raise CastErrors (i.e. blame), whereas in λB^g, λB and λB^r that is not the case. Indeed one of the design principles of λB^r is that we do not raise blame when λB^g (and λB) does not (see also Theorem 28). Saved expressions are the key to avoiding raising blame too early (or at all) in λB^r, and are generated when Castagna and Lanvin's calculus would generate blame for higher-order casts. Greenberg [21] introduced similar semantics to Castagna and Lanvin [9]. Blame is also raised in the above example. As λB^r, the intermediate consistent type for a higher-order function will be eliminated in Greenberg [21]. While in Castagna and Lanvin [9]'s work, the consistent intermediate type will be stored.

Finally, various cast calculi have been extended with various of features of practical interest. For instance, Ahmed et al. [3] extended the blame calculus to incorporate polymorphism, based on the dynamic sealing proposed by Matthews et al. [28] and Neis et al. [32].

Gradually Typed Calculi. A gradually typed lambda calculus (GTLC) should support both fully static typed and fully dynamic typed, as well as partially typed ones. Siek and Taha [43] introduced gradual typing with the notion of unknown types $\star$ and type consistency. To support object-oriented languages, Siek and Taha [38] extended the work of Abadi and Cardelli [1] and introduced gradual typing for objects. The semantics of both gradually typed calculi are indirectly defined by typed-directed translation to an intermediate language (a cast calculi). Cast calculi are independent from the GTLC, having their own type systems and operational semantics. The only tie between them is type-directed translation from the source gradually typed language to the cast calculus. In λB^g and λB^r, by using TDOS, the semantics of a GTLC is given directly without translating to any other calculus.

Because runtime checking is needed by a gradually typed language, function types dynamically generate function proxies at runtime in most of gradually typed languages. Therefore the number of proxies in unbound. Herman et al. [24] implemented gradual typing based on coercions and combined adjacent coercions. Thus, space consumption has been limited and the type system was proved to be type-safe. Addressing the space consumption issues of gradual typing has been an ongoing research effort for gradual typing, with many works on the area [16, 24, 37, 42]. The blame recovery semantics circumvents some of the space consumption issues by employing a different semantics.

Abstracting Gradual Typing (AGT). Garcia et al. [17] introduce the abstracting gradual typing (AGT) approach, following an idea by Schwerter [4]. An externally justified cast calculus is not required in AGT. Instead the runtime checks are deduced by the evidence for the consistency judgement. For the static semantics, AGT uses techniques from abstract interpretation to lift terms of the static system to gradual terms. A concretization function is used to lift gradual types to static type sets. After that, a gradual type system can be derived according to the static type system. The gradual type system keeps type safety, and enjoys the criteria of Siek et al. [41]. For the dynamic semantics, the semantics is introduced by reasoning about consistency relations. Gradual typing derivations are represented as intrinsically typed terms [12], which correspond to typing derivations directly.

Similarly to the AGT approach, by using TDOS for the dynamic semantics and a bi-directional type system, we can design a gradually typed language with a direct semantics. While related by the fact that both the AGT approach and TDOS provide means to obtain direct operational semantics for gradually typed languages, the two approaches have different and perhaps complementary goals. The goals of TDOS are more modest than those of AGT, which aims at deriving various definitions for gradually types languages in a systematic manner. In contrast TDOS and our work have no such goals. Our main aim is to adapt the standard and well-known techniques from small-step semantics, into the design of gradually typed languages. We expect that the familiarity and simplicity of the TDOS approach would be a strength, whereas the AGT approach requires some more infrastructure, but the payoff is that many definitions can then be derived. For future work, it would be interesting to see whether it is possible to combine ideas from both approaches. Perhaps having much of the AGT infrastructure, but with an alternative model for the dynamic semantics based on TDOS.

Typed Operational Semantics. In this paper, we use the type-directed operational semantics (TDOS) approach [25]. TDOS was originally used to describe the semantics of languages with intersection types and a merge operator. Like gradual typing, such features require a type-dependent semantics. In TDOS type annotations become operationally relevant and

can affect the result of a program. *Typed reduction* is the distinctive feature in TDOS. Typed reduction is used to provide an operational interpretation to type conversions in the language, similarly to coercions in coercion-based calculi [23]. Our work shows that TDOS enables a direct semantics for gradual typing. In this paper, we explored two possible semantics for gradual typing: one following a semantics similar to the blame calculus, and another with a novel blame recovery semantics. One interesting aspect of the blame recovery semantics is that it avoids some space costs that arise in some cast calculi, while being relatively simple.

There are other variants of operational semantics that make use of type annotations. Types are used in Goguen's typed operational semantics [18] reductions, similarly to TDOS. Typed operational semantics has been applied to various calculi, including simply typed lambda calculi [19], calculi with dependent types [14] and higher-order subtyping [13]. An extensive overview of related work on type-dependent semantics is given by Huang and Oliveira [25].

6 Conclusion

In this work we proposed an alternative approach to give a direct semantics to gradually typed languages without an intermediate cast calculus. Our approach is based on TDOS [25]. TDOS is a variant of small-step semantics where type annotations are operationally relevant and a special relation, called typed reduction, gives an interpretation to such type annotations at runtime. We believe that TDOS can be a valuable technique for language designers of gradually typed languages, giving them a simple and direct way to express the semantics of their language.

We presented two gradually typed lambda calculi: λB^g and λB^r. The λB^g semantics is sound to the semantics of λB. The λB^r calculus explores the large design space in the semantics of gradually typed languages with a new semantics that we call *blame recovery semantics*. This new semantics is more liberal than the semantics of the blame calculus, while still ensuring type-safety and a form of the gradual guarantee.

There is much to be done for future work. Obviously, to prove that TDOS is a worthy alternative to existing cast calculi or other approaches for the semantics of gradually typed languages, many more features should be developed with TDOS. Cast calculi have been shown to support a wide range of features, including blame tracking [52], polymorphism [3], subtyping [38] and various other features [26, 40, 47]. We hope to explore this in the future. Another important line for future work is to see whether the blame recovery semantics provides relevant space efficiency benefits in practice. This would require a well-engineered compiler for gradual typing. Perhaps trying to modify the Grift compiler [27] would be a first step on this direction. Empirical validation and case studies would be necessary.

References

1 Martin Abadi and Luca Cardelli. *A theory of objects*. Springer Science & Business Media, 2012.

2 Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM transactions on programming languages and systems (TOPLAS)*, 13(2):237–268, 1991.

3 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*, pages 201–214, 2011.

4 Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. A theory of gradual effect systems. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 283–295, 2014.

5 Lorenzo Bettini, Viviana Bono, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri. Java & lambda: a featherweight story. *Logical Methods in Computer Science*, 14(3), 2018.

6 Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.

7 John Tang Boyland. The problem of structural type tests in a gradual-typed language. *Foundations of Object-Oriented Langauges*, 2014.

8 Gilad Bracha. *The Dart programming language*. Addison-Wesley Professional, 2015.

9 Giuseppe Castagna and Victor Lanvin. Gradual typing with union and intersection types. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–28, 2017.

10 Craig Chambers. The cecil language, specification and rationale, 1993.

11 Avik Chaudhuri. Flow: a static type checker for javascript. *SPLASH-I In Systems, Programming, Languages and Applications: Software for Humanity*, 2015.

12 Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2):56–68, 1940.

13 Adriana Compagnoni and Healfdene Goguen. Typed operational semantics for higher-order subtyping. *Information and Computation*, 184(2):242–297, 2003.

14 Yangyue Feng and Zhaohui Luo. Typed operational semantics for dependent record types. *arXiv preprint arXiv:1103.3321*, 2011.

15 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 48–59, 2002.

16 Ronald Garcia. Calculating threesomes, with blame. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 417–428, 2013.

17 Ronald Garcia, Alison M Clark, and Éric Tanter. Abstracting gradual typing. *ACM SIGPLAN Notices*, 51(1):429–442, 2016.

18 Healfdene Goguen. A typed operational semantics for type theory, 1994.

19 Healfdene Goguen. Typed operational semantics. In *International Conference on Typed Lambda Calculi and Applications*, pages 186–200. Springer, 1995.

20 Kathryn E Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. *ACM SIGPLAN Notices*, 40(10):231–245, 2005.

21 Michael Greenberg. Space-efficient manifest contracts. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 181–194, 2015.

22 Lars T Hansen. Evolutionary programming and gradual typing in ecmascript 4 (tutorial). *Lars*, 2007.

23 Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, 1994.

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

25 Xuejing Huang and Bruno C d S Oliveira. A type-directed operational semantics for a calculus with a merge operator. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

26 Lintaro Ina and Atsushi Igarashi. Gradual typing for generics. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 609–624, 2011.

27 Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G Siek. Efficient gradual typing. *arXiv preprint arXiv:1802.06375*, 2018.

28 Jacob Matthews and Amal Ahmed. Parametric polymorphism through run-time sealing. In *European Symposium on Programming (ESOP)*, pages 16–31. Citeseer, 2008.

29 Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *ACM SIGPLAN Notices*, 42(1):3–10, 2007.

30 Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*. Citeseer, 2004.

31 Andrew Myers. CS 6110 Lecture 8 Evaluation Contexts , Semantics by Translation, 2013.

32 Georg Neis, Derek Dreyer, and Andreas Rossberg. Non-parametric parametricity. *ACM Sigplan Notices*, 44(9):135–148, 2009.

33 Benjamin C Pierce and David N Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.

34 Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The ins and outs of gradual type inference. *ACM SIGPLAN Notices*, 47(1):481–494, 2012.

35 Manuel Serrano and Pierre Weis. Bigloo: a portable and optimizing compiler for strict functional languages. In *International Static Analysis Symposium*, pages 366–381. Springer, 1995.

36 Andrew Shalit. *The Dylan reference manual: the definitive guide to the new object-oriented dynamic language*. Addison Wesley Longman Publishing Co., Inc., 1996.

37 Jeremy Siek, Ronald Garcia, and Walid Taha. Exploring the design space of higher-order casts. In *European Symposium on Programming*, pages 17–31. Springer, 2009.

38 Jeremy Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2007.

39 Jeremy Siek, Peter Thiemann, and Philip Wadler. Blame and coercion: together again for the first time. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 425–435, 2015.

40 Jeremy G Siek and Manish Vachharajani. Gradual typing with unification-based inference. In *Proceedings of the 2008 symposium on Dynamic languages*, pages 1–12, 2008.

41 Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

42 Jeremy G Siek and Philip Wadler. Threesomes, with and without blame. *ACM Sigplan Notices*, 45(1):365–376, 2010.

43 G Siek Jeremy and Taha Walid. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

44 Guy Steele. *Common LISP: the language*. Elsevier, 1990.

45 T Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and impersonators: run-time support for reasonable interposition. *ACM SIGPLAN Notices*, 47(10):943–962, 2012.

46 Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. Gradual typing embedded securely in javascript. *ACM SIGPLAN Notices*, 49(1):425–437, 2014.

47 Asumu Takikawa, T Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 793–810, 2012.

48 Satish Thatte. Quasi-static typing. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 367–381, 1989.

49 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 964–974, 2006.

50 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. *ACM SIGPLAN Notices*, 43(1):395–406, 2008.

51 Julien Verlaguet. Facebook: Analyzing php statically. *Commercial Users of Functional Programming (CUFP)*, 13, 2013.

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

53 Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In *European Conference on Object-Oriented Programming*, pages 459–483. Springer, 2011.

54 Andrew K Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.