Type-Directed Operational Semantics for Gradual Typing

WENJIA YE BRUNO C. D. S. OLIVEIRA
The University of Hong Kong
(e-mail: {wjye , bruno}@cs.hku.hk)

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 a TDOS, type annotations become operationally relevant and can affect the result of a program. In the context of a gradually typed language, type annotations are used to trigger type-based conversions on values. We illustrate how to employ a 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 λe, explores an eager semantics for gradually typed languages using a TDOS. For both calculi, type safety is proved. For the λB^g calculus, we also present a variant with blame labels, and illustrate how the TDOS can also deal with such an important feature of gradually typed languages. We also show that the semantics of λB^g with blame labels is sound and complete with respect to the semantics of the blame calculus, and that both calculi come with a *gradual guarantee*. All the results have been formalized in the Coq theorem prover.

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 usually behaves as a dynamically typed program[1], 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 (Bierman *et al.*, 2014), Flow (Chaudhuri *et al.*, 2017) or Dart (Bracha, 2015) enable forms of gradual typing to various degrees. Much research on gradual typing has focused on the pursuit of sound gradual typing (Siek & Taha, 2006, 2007; Wadler & Findler, 2009), 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* (Wadler & Findler, 2009; Siek

[1] In some gradually typed calculi with type inference, programs without annotations can still be statically typed (Garcia & Cimini, 2015).

et al., 2015*a*), the *threesome calculus* (Siek & Wadler, 2009) or other cast calculi (Findler & Felleisen, 2002; Gray *et al.*, 2005; Henglein, 1994; Tobin-Hochstadt & Felleisen, 2006; Siek *et al.*, 2015*a*, 2009) 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.* (2016), 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.

We present a different approach to give the semantics of gradually typed languages directly. We use a recently proposed variant of small-step operational semantics (Wright & Felleisen, 1994) called *type-directed operational semantics* (TDOS) (Huang & Oliveira, 2020). For the most part, developing a TDOS is similar to developing a standard small step-semantics as advocated by Wright and Felleisen. However, in a 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 (Bettini *et al.*, 2018; Goguen, 1994; Feng & Luo, 2011), the distinctive feature of TDOS is the use of a big-step *casting* relation for casting values under a given type. While typically values are the final result of a program, in TDOS casting can further transform them based on their run-time type. Thus, casting provides an operational interpretation to type conversions in the language, similarly to coercions in coercion-based calculi (Henglein, 1994).

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) (Wadler & Findler, 2009) by Siek *et al.* (2015*a*). 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 λe, explores a variant of the eager semantics for gradually typed languages (Herman *et al.*, 2007, 2010), inspired by the semantics adopted in AGT (Garcia *et al.*, 2016). 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 (Herman *et al.*, 2010; Siek *et al.*, 2009; Siek & Wadler, 2009; Siek & Taha, 2007; Garcia, 2013), is that the accumulation of annotations affects space efficiency. A nice aspect of the eager semantics is that it avoids the accumulation of type annotations, being more space-efficient. Furthermore, as

we shall see, the eager semantics also enables simple proofs for the gradual guarantee (Siek *et al.*, 2015*b*).

For both calculi, type safety and the gradual guarantee are proved. For λB^g, we also present a variant with *blame labels* (Wadler & Findler, 2009). Blame labels are an important feature of gradually typed languages, which enable tracking run-time type errors and pinpointing the origin of the errors. Our variant of λB^g shows how the TDOS deals with such an important feature of gradually typed languages. This is noteworthy since blame labels can be a significant challenge for some gradual typing approaches. For instance, the AGT approach has yet to provide an account for blame labels. As far as we know, our work provides the first gradually typed calculus without elaboration to an intermediate language but with blame tracking. In addition, we show that blame safety can also be proved directly for λB^g. Finally, we show that λB^g with blame labels is sound and complete with respect to the blame calculus.

A non-goal of the current paper is to investigate efficient implementations of a TDOS. The three TDOS developed in this paper are directly implementable, but inefficient. We believe that the primary strength of a TDOS is its ability to provide a direct and simple semantics for gradual languages, and focus on that in this article. However, it is currently unclear whether a TDOS can also be used to guide efficient implementations of gradually typed languages. While we believe that it is possible to adapt TDOS to account for performance, this would require introducing additional complexity, which would be in conflict with our goals here. We provide some discussion about potential directions to investigate performant TDOS-based implementations, as well as discussing other potential criticisms of the TDOS approach in Section 6.

In summary, the contributions of this work are:

- **TDOS for gradual typing:** We show that a 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 the gradual guarantee.
- **The eager λe calculus.** λe explores a TDOS design with an eager semantics for gradual typing. λe is type-safe and it comes with a gradual guarantee (Siek *et al.*, 2015*b*), which is quite simple to prove.
- **TDOS with blame labels.** We illustrate that the TDOS approach can deal with blame labels. We provide a variant of λB^g, called λB^g_l, which supports blame labels and blame tracking. λB^g_l is type-safe and deterministic, preserves the gradual guarantee and is also blame safe. Two noteworthy results are the soundness and completeness to the blame calculus, and our blame safety proof, which is done directly on λB^g_l (instead of indirectly via the blame calculus).
- **Coq formalization:** All calculi 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 article:

https://github.com/YeWenjia/TypedDirectedGradualTypingWithBlame

This article is an extended, and significantly rewritten, version of a conference paper (Ye *et al.*, 2021). There are three main novelties with respect to the conference version. Firstly, we now cover blame labels and blame tracking and introduce the new λB^g_l calculus, which is a variant of λB^g with blame labels. The conference work did not account for blame labels at all. Furthermore, this article adds several important results for λB^g_l, such as the soundness and completeness to the blame calculus, a novel blame safety theorem and the gradual guarantee for λB^g_l. Secondly, the conference version did not prove the gradual guarantee for λB^g, whereas we now show the gradual guarantee for λB^g_l. The gradual guarantee is a non-trivial result for λB^g_l, due primarily to the lazy semantics of higher-order casts. Thirdly, the λe calculus is also new, and it provides the first TDOS design for a calculus with an eager semantics. In the conference version, we explored a forgetful semantics (Greenberg, 2015), but we opted to study an eager semantics instead for the journal version since the eager semantics is more widely accepted.

2 Overview

This section provides background on gradual typing and the blame calculus, and then illustrates the key ideas of our work, the λB^g calculus (with and without blame labels) and the λe calculus.

2.1 Background: Gradual Typing and the λB Calculus

Traditionally, programming languages can be divided into statically typed languages and dynamically typed languages. In a statically typed language, a type system checks the types of terms before execution or compilation. The language may support type inference, but usually some type annotations are required for type checking. Type annotations bear 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, making programs more flexible, but at the cost of static type safety.

Gradual typing (Siek & Taha, 2006) is like a bridge connecting the two styles. Gradual typing extends the type system of static languages by allowing terms to have an *unknown type* $\star$. A term with the unknown type $\star$ is not rejected in any context by the type checker. When all terms in a program have unknown types the type checker does not reject program, similarly to 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*. Consistency offers a more relaxed notion of type compatibility compared to conventional statically typed languages where equality is used instead. The key feature of consistency is that the unknown type $\star$ is consistent with any other type. Thus, dynamic snippets of code can be embedded into the whole program without breaking type soundness. Of course, the notion of type soundness is relaxed to tolerate 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.* (2015*b*).

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 & Taha (2006) and Tobin-Hochstadt & Felleisen (2006).

One of the most widely used cast calculi for the elaboration of gradually typed languages is the blame calculus (Wadler & Findler, 2009; Siek *et al.*, 2015*a*). Figure 1 shows the definition of the blame calculus. Here we base ourselves in a variant of the blame calculus by Siek *et al.* (2015*a*). The blame calculus is the simply-typed lambda calculus extended with the unknown type ($\star$) and the cast expression ($t : A \stackrel{l}{\Rightarrow} B$). 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 ($V : A \to B \stackrel{l}{\Rightarrow} A' \to B'$) and $V : G \stackrel{l}{\Rightarrow} \star$ are included. Run-time type errors are denoted as $\mathsf{blame}\, l$. The syntactic sort l represents blame labels. A blame label $\bar{l}$ is the complement of label l and the complement is involutive, which means that $\bar{\bar{l}}$ is the same as l. Notably, when casting with a label l, the expression being cast is to blame, while when casting with $\bar{l}$, the context of the cast is to blame. Besides the standard typing rules of the simply typed lambda calculus (STLC), 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 with blame label l. 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 rule BTYP-C, $\rceil c \lceil$ is the dynamic type of the constant c. For example, $\rceil + \lceil$ is $\mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$. In the premise of rule BSTEP-DYNA, there is a predicate ug which says that type A should be consistent with a ground type G, but it should not be G itself and type $\star$. The definition of ug is:

$$\boxed{ug(A, G) \quad ::= \quad A \neq \star \wedge A \neq G \wedge A \sim G}$$

In rule BSTEP-C, if $[\![+]\!](i)$ then $+_i$ is returned, while for $[\![(+_{i_1})]\!](i_2)$, the result $(i_1 + i_2)$ is returned.

The bottom of Figure 1 shows a selection of the reduction rules of the blame calculus. The dynamic semantics of the λB calculus is standard for most rules. For first-order values, reduction is straightforward: a cast either succeeds or it fails and raises blame. For example[2]:

$$1 : \mathsf{Int} \stackrel{l_1}{\Rightarrow} \star : \star \stackrel{l_2}{\Rightarrow} \mathsf{Int} \longmapsto^* 1$$

$$1 : \mathsf{Int} \stackrel{l_1}{\Rightarrow} \star : \star \stackrel{l_2}{\Rightarrow} \mathsf{Bool} \longmapsto^* (\mathsf{blame}\, l_2)$$

For higher-order values such as functions, the semantics is more complex, since the cast result cannot be immediately obtained. For example, if we cast from $\star \to \star$ to $\mathsf{Int} \to \mathsf{Int}$,

[2] Note that throughout the paper we will assume that we have primitive types, such as Bool, for illustrating our examples more simply. While the calculi that we formalize do not contain such primitive types, they are easy to add and can be replaced by other types, such as $\mathsf{Int} \to \mathsf{Int}$, for similar effect in the examples.

Syntax

Blame Labels l	
Types	$A, B ::= \mathsf{Int} \mid \star \mid A \to B$
Ground types	$G, H ::= \mathsf{Int} \mid \star \to \star$
Constants	$c ::= i \mid + \mid +_i$
Terms	$t ::= c \mid x \mid t : A \overset{l}{\Rightarrow} B \mid t_1\ t_2 \mid \lambda x : A.t$
Results	$r ::= t \mid blame\ l$
Values	$V, W ::= c \mid V : A \to B \overset{l}{\Rightarrow} A' \to B' \mid \lambda x : A.t \mid V : G \overset{l}{\Rightarrow} \star$
Context	$\Gamma ::= \cdot \mid \Gamma, x : A$
Frame	$F ::= \square\ t \mid V\ \square \mid \square : A \overset{l}{\Rightarrow} B$

$\boxed{\Gamma \Vdash t : A}$ *(Typing Rules for the λB Calculus)*

BTYP-C
$$\frac{}{\Gamma \Vdash c :]c[}$$

BTYP-VAR
$$\frac{x : A \in \Gamma}{\Gamma \Vdash x : A}$$

BTYP-ABS
$$\frac{\Gamma, x : A \Vdash t : B}{\Gamma \Vdash \lambda x : A.\, t : A \to B}$$

BTYP-APP
$$\frac{\Gamma \Vdash t_1 : A \to B \qquad \Gamma \Vdash t_2 : A}{\Gamma \Vdash t_1\ t_2 : B}$$

BTYP-CAST
$$\frac{\Gamma \Vdash t : A \qquad A \sim B}{\Gamma \Vdash t : A \overset{l}{\Rightarrow} B : B}$$

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

S-I
$$\frac{}{\mathsf{Int} \sim \mathsf{Int}}$$

S-ARR
$$\frac{A \sim C \qquad B \sim D}{A \to B \sim C \to D}$$

S-DYNL
$$\frac{}{\star \sim A}$$

S-DYNR
$$\frac{}{A \sim \star}$$

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

BSTEP-EVAL
$$\frac{t \longmapsto t'}{F[t] \longmapsto F[t']}$$

BSTEP-BLAME
$$\frac{t \longmapsto \mathsf{blame}\ l}{F[t] \longmapsto \mathsf{blame}\ l}$$

BSTEP-BETA
$$\frac{}{(\lambda x : A.\, t)\ V \longmapsto t[x \mapsto V]}$$

BSTEP-VANY
$$\frac{}{(V : G \overset{l_1}{\Rightarrow} \star) : \star \overset{l_2}{\Rightarrow} G \longmapsto V}$$

BSTEP-DD
$$\frac{}{V : \star \overset{l}{\Rightarrow} \star \longmapsto V}$$

BSTEP-DYNA
$$\frac{ug(A, G)}{V : \star \overset{l}{\Rightarrow} A \longmapsto V : \star \overset{l}{\Rightarrow} G : G \overset{l}{\Rightarrow} A}$$

BSTEP-ANYD
$$\frac{ug(A, G)}{V : A \overset{l}{\Rightarrow} \star \longmapsto (V : A \overset{l}{\Rightarrow} G) : G \overset{l}{\Rightarrow} \star}$$

BSTEP-ABETA
$$\frac{}{(V_1 : A_1 \to B_1 \overset{l}{\Rightarrow} A_2 \to B_2)\ V_2 \longmapsto (V_1\ (V_2 : A_2 \overset{\bar{l}}{\Rightarrow} A_1)) : B_1 \overset{l}{\Rightarrow} B_2}$$

BSTEP-C
$$\frac{}{c\ v \longmapsto [\![c]\!](v)}$$

BSTEP-VANYP
$$\frac{G \neq H}{(V : G \overset{l_1}{\Rightarrow} \star) : \star \overset{l_2}{\Rightarrow} H \longmapsto \mathsf{blame}\ l_2}$$

BSTEP-LIT
$$\frac{}{i : \mathsf{Int} \overset{l}{\Rightarrow} \mathsf{Int} \longmapsto i}$$

Fig. 1. The λB Calculus (selected rules).

we cannot judge the cast result immediately. So the checking process is deferred until the function is applied to an argument. Rule BSTEP-ABETA shows that process: a function with the cast is a value which does not reduce until it has been applied to a value.

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 (Garcia *et al.*, 2016) 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*". We partly agree with such arguments. In addition, as argued by Huang & Oliveira (2020), there are some other reasons why a direct semantics is beneficial over an elaboration semantics.

A direct semantics enables a simple and direct way for programmers and tools to reason about the behavior of programs. For instance, we can reason directly about source programs by employing the operational semantics rules. With a TDOS, we can easily (and justifiably) employ similar steps to reason about the source language (say GTLC or λB^g). With a semantics defined via elaboration, however, that is not so easy because of the indirect semantics. In essence, with an elaboration semantics, one must first translate a source expression to a target calculus, then reason about the semantics in the target calculus, and finally translate the final result in the target calculus back to the source language. This process is not only more laborious, but it requires knowledge about the target language and its 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 can often be simpler, since some lemmas that are required with an elaboration semantics (for instance, Lemma 6 in the original work on the refined criteria for gradual typing (Siek *et al.*, 2015*b*)) are not needed with a direct operational semantics. Moreover, only the precision relation for the source language is necessary. We shall see some of those benefits in this paper.

Finally, another source of complication in an elaboration semantics, especially in languages with more advanced type systems, is *coherence* (Reynolds, 1991). Coherence is a desirable property for an elaboration semantics, where we show that all possible elaborations for a source program have the same semantics in the target language. For simpler languages, coherence is typically not an issue because the elaboration process is *deterministic*. That is there is always a unique target expression that is generated for the same source program. The elaboration of the GTLC into the blame calculus, for example, is deterministic. Thus, coherence is trivial in that case. However, for more advanced type

system features, including implicit polymorphism (Xie *et al.*, 2019; Bottu *et al.*, 2019) or some systems with subtyping (Huang & Oliveira, 2020), coherence becomes non-trivial because the elaboration becomes *non-deterministic*. That is, for the same source program we may be able to generate multiple distinct target expressions. Then a coherence proof has to show that despite generating different expressions, those expressions have the same meaning. Typically this is done by employing some form of contextual equivalence, and usually requires the use of *logical relations* (Reynolds, 1993). In those cases, the coherence proof may become quite non-trivial and involved.

One of the original motivations of the TDOS (Huang & Oliveira, 2020) was to avoid the need for coherence proofs. In Huang and Oliveira's work, the source calculus that they considered had subtyping with intersection types, and during elaboration coercions in the target language were generated by the subtyping relation. Unfortunately, these coercions in a system with intersection types are not unique, requiring an involved proof of coherence using contextual equivalence and logical relations. A TDOS avoids coherence proofs. Instead, we can simply prove that the semantics is deterministic, which can usually be done using standard proof techniques and simple inductions. Similar coherence issues arise with other type systems, such as for instance type systems with type classes. The proof that the elaboration of a language with type classes to a target language without type classes is coherent is highly non-trivial, as shown by the work of Bottu *et al.* (2019). Finally, there is also work in gradual typing where coherence issues also arise. For instance, Xie *et al.* (2019) has shown that a source language with gradual typing and higher-ranked type-inference can be elaborated into the polymorphic blame calculus (Ahmed *et al.*, 2011). However, coherence was not proved due to its difficulty and was left as an open problem. While in this paper the calculi involved are simple enough that no complications arise from coherence, those complications will arise in more complex settings. Thus, avoiding the complications of coherence is another argument to employ a TDOS.

2.3 λB^g: A Gradually Typed Lambda Calculus

In this subsection, we introduce some of the key ideas in λB^g, which is a gradually typed lambda calculus that is given a semantics via a TDOS. In Section 3, we provide the full details of λB^g.

Cast Calculi vs Gradually Typed Calculi. Since λB requires explicit casts whenever a term's type is converted, it cannot be considered as a gradually typed calculus. For comparison, the application rule for typing in the *Gradually Typed Lambda Calculus* (GTLC) (Siek *et al.*, 2015*b*; Siek & Taha, 2006, 2007):

$$\frac{\Gamma \Vdash e_1 : T \qquad \Gamma \Vdash e_2 : T_3 \qquad T \rhd T_1 \to T_2 \qquad T_3 \sim T_1}{\Gamma \Vdash e_1\ e_2 : T_2}\ \text{GTLC-APP}$$

does not force the input term to have the same type to what the function expects. It just checks the consistency of the two terms' types and can do implicit type conversions (casts) automatically. Note that the function $T \rhd T_1 \to T_2$ allows T to be either a function type or $\star$. When T is a function type, then T itself is returned. Otherwise, if T is $\star$, then $\star \to \star$ is returned. 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 the same as the input type of the function type.

Bidirectional Type-Checking for λB^g. In this paper, a key argument is that we do not need two calculi to formulate the semantics of a gradually typed language. Instead, we can have a single calculus that supports explicit type conversions (like casts in cast calculi) as well as implicit type conversions (like the GTLC). To introduce implicit type conversions, we turn to the bidirectional type checking (Pierce & Turner, 2000). Unlike in the 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 latter, 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. Notably, via the checking mode, the type consistency between a type annotation and a given type is checked. Furthermore, uses of the checking mode in typing essentially denote points where runtime casts need to happen to enforce type-safety.

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. For example, the rules for applications and subsumption in λB^g are:

$$\textsc{Typ-app}\quad \frac{A \rhd A_1 \to A_2 \qquad \Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Leftarrow A_1}{\Gamma \vdash e_1\ e_2 \Rightarrow A_2} \qquad\qquad \textsc{Typ-sim}\quad \frac{\Gamma \vdash e \Rightarrow A \qquad A \sim B}{\Gamma \vdash e \Leftarrow B}$$

Implicit Type Conversion in Function Applications. λB^g supports *annotated lambdas* of the form $\lambda x.e : A \to B$ with both input and output types for the function. A raw lambda $\lambda x.e$ is sugar for a lambda with a function type $\star \to \star$. Note also that, throughout this article, whenever we have an expression of the form $\lambda x.e : A \to B$, this should be interpreted as $(\lambda x.e) : A \to B$. In other words, type annotations bind more weakly than lambdas (and other forms of expressions). By using bidirectional type checking, we can type-check programs such as:

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

In addition bidirectional type checking can also reject ill-typed programs, such as:

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

In the first two examples, $\lambda x.x$ and $\lambda x.not\ x$ are equivalent to $\lambda x.x : \star \to \star$ and $\lambda x.not\ x : \star \to \star$. Thus, in the first example, the argument 1 is type checked with type $\star$. For the 3rd and 4th examples, 1 can be type checked by the function input types $\star$ and Int. However, 1 cannot be checked by type Bool in the last example, and that program is rejected.

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:

$$1 : \star : \mathsf{Int} \longmapsto^* 1$$
$$1 : \star : \mathsf{Bool} \longmapsto^* \mathit{blame}$$

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 ($1 : \star : \mathsf{Int}$), 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.

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 (Huang *et al.*, 2021). To enable expressive forms of the merge operator, the dynamic semantics has to account for types, just like the semantics of gradually typed languages. In many traditional operational semantics, type annotations are often ignored. In a TDOS that is not the case. Instead, type annotations are used at runtime to determine the result of the 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 casting relation $v \Downarrow_A r$.

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

$$1 \Downarrow_{\mathsf{Int}} 1$$
$$1 : \star \Downarrow_{\mathsf{Bool}} \mathsf{blame}$$

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.

Casting takes place in some reduction rules, such as the beta reduction rule and the annotation elimination rule for values. Here we illustrate the beta-reduction rule, and defer

the explanation of other reduction rules to Section 3:

$$\frac{v_2 \Downarrow_A v_2'}{(\lambda x.\, e : A \to B)\; v_2 \longmapsto e[x \mapsto v_2'] : B}\ \textsc{Step-Beta}$$

Casting is used in beta-reduction to ensure that the argument value v_2 matches the expected input type of the function (A). Consider another example to illustrate the behavior of casting in beta reduction:

$$(\lambda x.\, x : \mathsf{Bool} \to \star)\ (1 : \star)$$

If we 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 the 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 casting under the expected input type of the function. When we check that the type Int (the type for the value 1) is not consistent with Bool, blame is raised. However, if we take the example:

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

then the value 1 (which arises from casting) is substituted in the function body and the result is 2.

2.5 λe: Gradual Typing with an Eager Semantics

To further illustrate the application of a TDOS to gradually typed languages, we also propose a second calculus, called λe. The λe calculus has different design choices in terms of the runtime semantics. In particular its semantics for casts is a variant of the eager semantics (Herman *et al.*, 2010) inspired by the approach employeed in AGT (Garcia *et al.*, 2016). Here we overview key ideas in λe, while the full details of λe will be given in Section 5.

Lazy versus Eager. Both the blame calculus and λB^g employ a lazy semantics for higher-order values, where functions with an arbitrary number of annotations are values. In other words, the checking process for higher-order values is deferred until the function is applied to an argument. Garcia *et al.* (2016) proposed the AGT based methodology, which has been widely applied (Toro *et al.*, 2019; Labrada *et al.*, 2022; Ye *et al.*, 2023). In AGT, a variant of the eager semantics is applied for higher-order values. In the following example, we show how higher-order values are reduced in AGT (using our notation):

$$\lambda x.\, 1 : \mathsf{Int} \to \mathsf{Int} : \star \to \star : \mathsf{Bool} \to \mathsf{Bool} \longmapsto^* \mathsf{blame}$$

Unlike the lazy semantics, where a lambda with multiple annotations would be a value, in λe the above expression is not a value and it reduces to blame. blame is raised directly because, during the reduction process, we attempt to eliminate the intermediate annotation $\star \to \star$, by checking whether $\mathsf{Int} \to \mathsf{Int}$ and $\mathsf{Bool} \to \mathsf{Bool}$ are consistent. Since these two types are not consistent blame would be raised in this case.

Meets: Making Sure that no Cast is Forgotten. A complication in the eager semantics is that we should guarantee that possible casts that arise from intermediate annotations are not forgotten. To formalize this behavior, lambda values in λe are of the form $(\!|\lambda x.\, e : A_1 \to A_2 : B_1 \to B_2 : C|\!)$ and include three annotations:

- The first annotation $(A_1 \to A_2)$ is used to type check the lambda body and to cast the argument using the input type before beta-reduction.
- The second annotation $B_1 \to B_2$ is key to the eager semantics: it stores a type, which we call the *meet type*, that is the greatest lower bound (with respect to precision) of all the intermediate annotations that have been eliminated during reduction of the lambda value. In other words, it is the most imprecise type among the types equivalent to or more precise than the eliminated types. The meet type triggers casts, subsuming the casts needed for the eliminated intermediate annotations.
- Finally, the outer annotation C is needed for type preservation.

Lets see how meet types are used in the following example:

$$
\begin{aligned}
&(\lambda x.\, x : \mathsf{Bool} \to \star : \star \to \mathsf{Bool} : \star \to \star : \star \to \mathsf{Int})\ 1 \\
&\longmapsto ((\!|\lambda x.\, x : \mathsf{Bool} \to \star : \mathsf{Bool} \to \star : \mathsf{Bool} \to \star|\!) : \star \to \mathsf{Bool} : \star \to \star : \star \to \mathsf{Int})\ 1 \\
&\longmapsto ((\!|\lambda x.\, x : \mathsf{Bool} \to \star : \mathsf{Bool} \to \mathsf{Bool} : \star \to \mathsf{Bool}|\!) : \star \to \star : \star \to \mathsf{Int})\ 1 \\
&\longmapsto ((\!|\lambda x.\, x : \mathsf{Bool} \to \star : \mathsf{Bool} \to \mathsf{Bool} : \star \to \star|\!) : \star \to \mathsf{Int})\ 1 \\
&\longmapsto \mathsf{blame}
\end{aligned}
$$

We first reduce $\lambda x.\, x : \mathsf{Bool} \to \star$ to a tagged value $(\!|\lambda x.\, x : \mathsf{Bool} \to \star : \mathsf{Bool} \to \star : \mathsf{Bool} \to \star|\!)$, which ensures that the intermediate type is more precise than the outer type. The details are explained in Section 5. Then, in the 3rd line, we are safe to remove the outer type and update the intermediate type with the meet result between $\mathsf{Bool} \to \star$ and $\star \to \mathsf{Bool}$, which is the more precise type $\mathsf{Bool} \to \mathsf{Bool}$. Finally, in the 4th line, the meet between $\mathsf{Bool} \to \mathsf{Bool}$ and $\star \to \mathsf{Int}$ is undefined and error is raised. Note that, without storing the more precise type $\mathsf{Bool} \to \mathsf{Bool}$ in the middle, then the type information of the intermediate type annotation $\star \to \mathsf{Bool}$ would be ignored.

The eager semantics does have some drawbacks compared to the lazy semantics. For instance, New *et al.* (2019) show that the η principle for the higher-order values is violated. Nevertheless, there are also some nice aspects in an eager semantics. With an eager semantics it is possible to avoid the accumulation of annotations for higher-order values. Another nice aspect of the eager semantics in the AGT approach is that the dynamic gradual guarantee can be formalized in a simple way and proved easily. This is because the number of reduction steps when evaluating a program is not affected by precision (except when runtime errors occur in the less precise version of the program). In a lazy semantics, annotations are sometimes accumulated or added during reduction. So the reduction of more precise and less dynamic programs can take different number of steps in some cases. Here, a more precise program means that the program is more statically typed. For instance, for $\lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \star$ and $\lambda x.\, x : \star \to \star : \star$, the more precise program $\lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \star$ should take a step to $\lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \star \to \star : \star$. In contrast, the less precise one $\lambda x.\, x : \star \to \star : \star$

(which is a value) takes zero steps. Since in the eager semantics annotations are not accumulated, the dynamic semantics for more and less precise programs behaves in the same way, satisfying the dynamic gradual guarantee more directly and naturally.

Functions. One interesting aspect in the type system of λe is how we deal with lambdas. If the programmer wants to have a function statically type checked, they can write down the full annotations or the annotations can be propagated by bidirectional type checking. For instance, in the term:

$$(\lambda f.\ f\ 2 : (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int})\ (\lambda x.\ x + 1)$$

The argument to the function $(\lambda x.\ x + 1)$, which is itself a lambda, has no type annotations. Here bidirectional type-checking propagates the annotations, removing the need for explicit annotations in the lambda. This is in contrast to typical gradual languages, such as the GTLC, where a raw lambda $\lambda x.\ e$ is syntactic sugar for $\lambda (x : \star).\ e$. The key difference is that in GTLC the raw lambda $(\lambda x.\ x + 1)$ in a similar program to the above would lose type information and be of type $\star \to \mathsf{Int}$. A consequence of this loss of type precision is that the elaboration of GTLC into a cast calculus would need to insert a few casts here. Firstly, one cast from $\star$ to Int would be needed to convert the type of the argument. Secondly a cast from Int to $\star$ would be needed to cast x back to an integer. These casts could potentially be avoided if there was no loss of type information. In λe, the lambda is of type $\mathsf{Int} \to \mathsf{Int}$ and no static information is lost.

To achieve this, our rule for lambda expressions, which is quite standard for bidirectional typing, is:

$$\frac{\Gamma, x : A_1 \vdash e \Leftarrow A_2 \qquad A \rhd A_1 \to A_2}{\Gamma \vdash \lambda x.\ e \Leftarrow A}\ \textsc{Typ-abs}$$

The premise $A \rhd A_1 \to A_2$ in rule TYP-ABS ensures that type A can be the unknown type $\star$ or a function type. The unknown type is interpreted as the unknown function $\star \to \star$. The typing rule TYP-ABS is employed to type-check both lambdas above: the first one because the explicit type annotation triggers the checking mode; and the second one because the application rule employs the checking mode to check that arguments conform to the right type.

2.6 Blame Tracking

An important feature of gradual typing is blame tracking, which enables pinpointing the source of blame in a program. This is helpful for programmers as they can determine the source of the problem. λB^g can support blame tracking, showing that the TDOS approach can smoothly deal with such an important feature of gradual typing. Next, we discuss the key ideas of adding blame tracking to λB^g. The full details are discussed in Section 4.

Blame Tracking. For gradually typed lambda calculi, due to the notion of consistency, runtime type errors may arise from type inconsistency. A natural question is what is the cause of the error, and which part of the program is to blame. In the λB calculus, blame labels, which are the blue parts in Figure 1, are used to track the source of errors. Blame labels include the positive blame label (l) and the negative blame label ($\bar{l}$). Positive blame

labels (l) arise when the terms contained in the cast are to blame. Negative blame labels ($\bar{l}$) arise when the context which contains the cast is to blame. An example to show the mechanism of blame tracking with different labels in λB is:

$$
\begin{aligned}
&((\lambda x : \mathsf{Int}.\, x) : \mathsf{Int} \to \mathsf{Int} \overset{l_1}{\Rightarrow} \star \to \mathsf{Int})\ (\mathsf{True} : \mathsf{Bool} \overset{l_0}{\Rightarrow} \star) \\
&\longmapsto ((\lambda x : \mathsf{Int}.\, x)\ ((\mathsf{True} : \mathsf{Bool} \overset{l_0}{\Rightarrow} \star) : \star \overset{\bar{l}_1}{\Rightarrow} \mathsf{Int})) : \mathsf{Int} \overset{l_1}{\Rightarrow} \mathsf{Int} \\
&\longmapsto^* (\mathsf{blame}\ \bar{l}_1)
\end{aligned}
$$

In this example, when the argument $(\mathsf{True} : \mathsf{Bool} \overset{l_0}{\Rightarrow} \star)$ is cast from $\star$ to Int, blame with label $\bar{l}_1$ is raised: the source type Bool is different from the target type Int causing blame to be raised.

Blame Tracking for λB^g. The λB^g_l calculus is a variant of λB^g with blame tracking. In λB^g_l blame labels are introduced in annotation expressions $(e :^l A)$, lambdas $(\lambda x.\, e :^l A \to B)$, application expressions $((e_1\ e_2)^l)$ and projections $(\pi^l_i\ e)$. Annotation expressions $(e :^l A)$ trigger casts, and are similar to cast expressions $(t : B \overset{l}{\Rightarrow} A)$ in λB. While the label in lambdas is to track the implicit cast in lambda bodies. Notably, there is a blame label for applications since the typing rule for applications is:

$$
\frac{\Gamma \vdash e_1 \Rightarrow A \qquad A \rhd A_1 \to A_2 \qquad \Gamma \vdash e_2 \Leftarrow A_1}{\Gamma \vdash (e_1\ e_2)^l \Rightarrow A_2}\ \textsc{Typ-app}
$$

In this rule, the typing for the argument e_2 is in checking mode. This means that e_2 may have a different type, which can be consistent with A_1. However, the cast induced by the application, may be the source of blame. For instance, consider the following program:

$$
((\lambda x.\, x :^{l_1} \mathsf{Int} \to \mathsf{Int})\ (\mathsf{True} :^{l_3} \star))^{l_2}
$$

For this example, the argument $(\mathsf{True} :^{l_3} \star)$ is cast under type Int, and the type of True is not consistent with type Int. Blame is raised with the blame label (l_2). Furthermore, if A is type $\star$, there is an implicit cast to $\star \to \star$ and label l is used as well. The label l in projections $\pi^l_i\ e$ has the similar behavior. The λB^g_l calculus is proved to be sound and complete with respect to the blame calculus with blame labels. Furthermore, we formalized and proved the gradual guarantee theorem for λB^g_l.

Blame Theorem. "*Well-typed programs can't be blamed*" is the slogan of Wadler & Findler (2009)'s paper. To express this slogan formally, Wadler and Findler proposed the blame theorem, which expresses the idea that we should blame the right part of a program. More specifically, a cast from a more precise type to a less precise type cannot give rise to positive blame, and negative blame cannot be triggered while casting from a less precise type to a more precise type. We have proved the blame theorem for λB^g_l, showing that this key result can be proved using a TDOS approach to gradual typing.

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

In this section, we introduce the gradually typed λB^g calculus. The semantics of the λB^g calculus follows closely the semantics of the λB cast calculus. It employs a type-directed operational semantics (Huang & Oliveira, 2020) to have a direct operational

Syntax

Types	$A, B ::= \mathsf{Int} \mid \star \mid A \to B \mid A \times B$
Constants	$c ::= i \mid + \mid +_i$
Expressions	$e ::= c \mid x \mid \lambda x.\, e : A \to B \mid e_1\ e_2 \mid e : A \mid (e_1, e_2) \mid \pi_i\ e$
Results	$r ::= e \mid \mathsf{blame}$
Values	$v ::= c \mid v : A \to B \mid v : \star \mid \lambda x.\, e : A \to B \mid (v_1, v_2)$
Context	$\Gamma ::= \cdot \mid \Gamma, x : A$
Frame	$F ::= \square\ e \mid v\ \square \mid \square : A \mid (\square, e) \mid (v, \square) \mid \pi_i\ \square$

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

VALUE-C
$$\frac{}{value\ c}$$

VALUE-ABS
$$\frac{}{value\ \lambda x.\, e : A \to B}$$

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

VALUE-DYN
$$\frac{Ground\ \rceil v \lceil \qquad value\ v}{value\ v : \star}$$

VALUE-PRO
$$\frac{value\ v_1 \qquad value\ v_2}{value\ (v_1, v_2)}$$

Ground A *(Ground types)*

GROUND-INT
$$\frac{}{Ground\ \mathsf{Int}}$$

GROUND-ARR
$$\frac{}{Ground\ \star \to \star}$$

GROUND-PRO
$$\frac{}{Ground\ \star \times \star}$$

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

semantics. λB^g uses bidirectional type-checking (Pierce & Turner, 2000). We keep the presentation light here to better illustrate the key ideas of our approach. Thus, we mainly focus on presenting the semantics and proving basic results such as determinism and type-soundness. In Section 4, we present an extension of λB^g with blame labels, and develop many more results, such as the gradual guarantee, the blame theorem and the soundness and completeness to the blame calculus.

3.1 Syntax

The syntax of the 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 $A \to B$, the unknown type $\star$, and product types $A \times B$. Compared to λB calculus, ground types not only include Int and $\star \to \star$, but also $\star \times \star$. We sometimes use the abbreviation G to denote ground types.

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 and additions ($+$, $+_i$). The type of $+$ is $\mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$. The type of $+_i$ is $\mathsf{Int} \to \mathsf{Int}$, taking an integer and

returning another integer. Meta-variable e ranges over expressions. There are some standard constructs, which include: constants (c); variables (x); annotated expressions ($e : A$); annotated lambdas ($\lambda x.\, e : A \to B$); application expressions ($e_1\ e_2$); products (e_1, e_2) and projections $(\pi_i\ e)$. Similarly to GTLC, lambdas without type annotations are just sugar for lambdas with the annotation $\star \to \star$. Results (r) include all expressions and blame, which is used to denote cast-errors at run-time.

Values and Contexts. Typing contexts are standard. Γ is used to track bound variables x with their type A. The meta-variable v ranges over well-formed values. Values contain constants, annotated values and lambda abstractions ($\lambda x.\, e : A \to B$). Similarly to λB, not all syntactic values are well-formed values. The *value* predicate, at the bottom of Figure 2, defines well-formed values, which are a subset of syntactic values. Lambda expressions are values (rule VALUE-ABS). A syntactic value v with a function type annotation is a well-formed value if the dynamic type of the value is also a function type (rule VALUE-FANNO). Note that, for well-typed values, it is guaranteed that the type of v is consistent with the annotation $A \to B$. So, the value predicate only needs to check if the type of the value has an arrow form. The expression $v : \star$ is a value only when the type of v is a ground type (rule VALUE-DYN). Products are values if there is a pair of values. Constants are also values. Note that the meta-function $\rceil v \lceil$, defined next, denotes the dynamic type of a value.

Definition 3.1 (Dynamic type). *$\rceil v \lceil$ denotes the dynamic type of the value v.*

$$\begin{aligned}
\rceil i \lceil &= \mathsf{Int} \\
\rceil \lambda x.\, e : A \to B \lceil &= A \to B \\
\rceil + \lceil &= \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \\
\rceil +_i \lceil &= \mathsf{Int} \to \mathsf{Int} \\
\rceil v : A \lceil &= A \\
\rceil (v_1, v_2) \lceil &= \rceil v_1 \lceil \times \rceil v_2 \lceil
\end{aligned}$$

Frame. The meta-variable F ranges over frames (Siek *et al.*, 2015*a*), which is a form of evaluation contexts (Felleisen & Hieb, 1992). The frame is mostly standard, though it is perhaps noteworthy that it includes annotated expressions.

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 ($\Rightarrow$) or checked ($\Leftarrow$) by the type A under the typing context Γ. The typing mode ($\Leftrightarrow$), whose definition is shown at the top of Figure 3, is used to represent the two modes of the bidirectional typing judgment.

Typing Relation. The typing relation of the λB^g calculus is shown in Figure 3. Many rules in inference mode follow the λB type system. The typing for constants (rule TYP-C) uses the dynamic type definition to infer the type. 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 function type of a lambda expression is annotated and the function body is checked with the function output type. Lambdas are annotated with full types because at runtime we want to get the type of lambdas without performing type-checking.

Typing mode $\Leftrightarrow ::= \ \Rightarrow | \Leftarrow$

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

$$\textsc{Typ-C}\quad \frac{}{\Gamma \vdash c \Rightarrow \rceil c \lceil} \qquad \textsc{Typ-Var}\quad \frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A} \qquad \textsc{Typ-Abs}\quad \frac{\Gamma, x : A \vdash e \Leftarrow B}{\Gamma \vdash \lambda x.\, e : A \to B \Rightarrow A \to B}$$

$$\textsc{Typ-App}\quad \frac{A \rhd A_1 \to A_2 \qquad \Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Leftarrow A_1}{\Gamma \vdash e_1\, e_2 \Rightarrow A_2} \qquad \textsc{Typ-Anno}\quad \frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash e : A \Rightarrow A} \qquad \textsc{Typ-Sim}\quad \frac{\Gamma \vdash e \Rightarrow A \qquad A \sim B}{\Gamma \vdash e \Leftarrow B}$$

$$\textsc{Typ-Pro}\quad \frac{\Gamma \vdash e_1 \Rightarrow A_1 \qquad \Gamma \vdash e_2 \Rightarrow A_2}{\Gamma \vdash (e_1, e_2) \Rightarrow A_1 \times A_2} \qquad \textsc{Typ-Pi}\quad \frac{\Gamma \vdash e \Rightarrow A \qquad A \blacktriangleright A_1 \times A_2 \qquad i \in \{1, 2\}}{\Gamma \vdash \pi_i\, e \Rightarrow A_i}$$

$\star$	$\rhd$	$\star \to \star$
$A \to B$	$\rhd$	$A \to B$
$\star$	$\blacktriangleright$	$\star \times \star$
$A \times B$	$\blacktriangleright$	$A \times B$

Fig. 3. Type system of the λB^g calculus.

For applications $e_1\ e_2$, the rule is almost standard for bidirectional type-checking: the type of e_1 is inferred, and the type of e_2 is checked against the domain type of e_1. However, there is some additional flexibility in the typing of applications. The expression e_1 can have type $\star$. In that case, $\star$ is interpreted as a dynamic function type $\star \to \star$, by employing a matching function (shown at the bottom of Figure 3). The rule for annotations (rule Typ-Anno) is standard, inferring the annotated type, while checking the expression against the annotated type. 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. This is the case, for instance, for the rule Typ-App, which checks whether the argument is of type A_1. In addition, to type check the product related expressions, we have rule Typ-pro and rule Typ-Pi. A pair (e_1, e_2) is well-typed with type $A \times B$, if e_1 is well-typed with type A and e_2 is well-typed with type B. When performing projections, we should make sure that the projections happen on pairs. Therefore, we use the matching relation $A \blacktriangleright B$ with product types $A_1 \times A_2$. The unknown type $\star$ matches the dynamic product type $\star \times \star$.

Consistency. Consistency plays an important role in a gradually typed lambda calculus, acting as a relaxed equality relation. The consistency relation extends λB's consistency,

which is already shown in Figure 1, with product types:

$$\frac{A_1 \sim B_1 \qquad A_2 \sim B_2}{A_1 \times A_2 \sim B_1 \times B_2}$$

In consistency, reflexivity and symmetry 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 (Siek & Taha, 2006).

Some important properties of the typing relation are that it computes dynamic types for the inference mode, and the type inference has unique types.

Lemma 3.1 (Dynamic Types for Values)**.** *If* $\cdot \vdash v \Rightarrow A$ *then* $\rceil v \lceil = A$.

Lemma 3.2 (Inference is Checkable)**.** *If* $\Gamma \vdash e \Rightarrow A$ *then* $\Gamma \vdash e \Leftarrow A$.

Lemma 3.3 (Uniqueness of Inference)**.** *If* $\Gamma \vdash e \Rightarrow A_1$ *and* $\Gamma \vdash e \Rightarrow A_2$ *then* $A_1 = A_2$.

3.3 Dynamic Semantics

The dynamic semantics of λB^g employs a type-directed operational semantics (TDOS) (Huang & Oliveira, 2020). In a TDOS, besides the usual reduction relation, there is a special *casting* relation for values that is used to further reduce values based on the type of the value. Casting is used by the TDOS reduction relation. In a gradually typed calculus with a TDOS, the casting relation plays a role analogous to various cast-related reduction rules in a cast calculus (Wadler & Findler, 2009; Siek *et al.*, 2015*a*; Siek & Wadler, 2009; Herman *et al.*, 2007). We first introduce casting and then move on to the definition of reduction.

Casting. We reduce a value under a certain type using the casting relation. The form of the casting relation is $v \Downarrow_A r$, which means that a value v reduces under type A to a result r. Unlike reduction, casting is defined using a big-step style. Note that the result r produced by casting can only be a value or *blame*. Blame is raised during casting 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 casting mimics the behavior of casts in cast calculi such as the λB calculus. In the λB calculus, $t : B \Rightarrow A$ casts t from source type B into target type A. Using casting in λB^g, the type A is the target type (which arises from a type annotation), whereas the dynamic type of v is the source type.

Figure 4 shows the rules of casting. Rule CAST-ABS and rule CAST-V just add a type annotation to the value. In rule CAST-ABS, the dynamic type of the value ($\rceil v \lceil$) is a function type, thus v annotated with $A \rightarrow B$ is a value. Note that several casting rules employ the dynamic type function to compute the type of a value at runtime. However, the type of a value is never computed from scratch, since we use existing type annotations to return the type of a value. In rule CAST-V, $v : \star$ is also a value when the dynamic type of v is a ground type. Rule CAST-LIT is for integer values: an integer i being reduced under the integer type results in the same integer i. A value $v : \star$ cast under $\star$ returns the original value as well (rule CAST-DD). In rule CAST-ANYD, the premise is that the dynamic type of v should be a *ug* type which is defined in λB. It means that the type of v should be any function type

$\boxed{v \Downarrow_A r}$ *(Casting)*

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

$$\frac{ug(\rceil v \lceil, G) \qquad v \Downarrow_G v'}{v \Downarrow_\star v' : \star}\ \textsc{Cast-anyd} \qquad \frac{}{v : \star \Downarrow_{\rceil v \lceil} v}\ \textsc{Cast-vany} \qquad \frac{\rceil v \lceil \nsim A}{v : \star \Downarrow_A \mathsf{blame}}\ \textsc{Cast-blame} \qquad \frac{ug(A, G) \qquad v \Downarrow_A r}{v : \star \Downarrow_A r}\ \textsc{Cast-dyna}$$

$$\frac{v_1 \Downarrow_{A_1} v_1' \qquad v_2 \Downarrow_{A_2} v_2'}{(v_1, v_2) \Downarrow_{A_1 \times A_2} (v_1', v_2')}\ \textsc{Cast-pro} \qquad \frac{v_1 \Downarrow_{A_1} \mathsf{blame} \qquad v_2 \Downarrow_{A_2} r}{(v_1, v_2) \Downarrow_{A_1 \times A_2} \mathsf{blame}}\ \textsc{Cast-l}$$

$$\frac{v_1 \Downarrow_{A_1} v_1' \qquad v_2 \Downarrow_{A_2} \mathsf{blame}}{(v_1, v_2) \Downarrow_{A_1 \times A_2} \mathsf{blame}}\ \textsc{Cast-r}$$

Fig. 4. Casting for the λB^g calculus.

$A \to B$ except for $\star \to \star$ or any product type $A \times B$ except for $\star \times \star$. In the end, v is cast under ground type G and returns the value v' annotated with $\star$.

In rule Cast-vany, $v : \star$ is cast under the dynamic type of v, returning v and dropping the annotation $\star$. In rule Cast-blame, if the dynamic type of v is not consistent to the type A that we are casting, then blame is raised. Finally, in rule Cast-dyna, a value $v : \star$ being cast under type A (where A is a ug type) returns the result of v cast under type A. For products, pairs of values are cast under the corresponding types. Rule Cast-pro models the case when both casts succeeds, while rule Cast-l and rule Cast-r model the cases where at least one of the casts fails.

Properties of Casting. Some properties of casting for the λB^g calculus are shown next:

Lemma 3.4 (Casting preserves well-formedness of values)**.** *If value v and $v \Downarrow_A v'$ then value v'.*

Lemma 3.5 (Preservation of Casting)**.** *If $\cdot \vdash v \Leftarrow A$ and $v \Downarrow_A v'$ then $\cdot \vdash v' \Rightarrow A$.*

Lemma 3.6 (Progress of Casting)**.** *If $\cdot \vdash v \Leftarrow A$ then $\exists r$, $v \Downarrow_A r$.*

Lemma 3.7 (Determinism of Casting)**.** *If $\cdot \vdash v \Leftarrow B$, $v \Downarrow_A r_1$ and $v \Downarrow_A r_2$ then $r_1 = r_2$.*

Lemma 3.8 (Casting Respects Consistency)**.** *If $\cdot \vdash v \Rightarrow B$ and $v \Downarrow_A r$ then $B \sim A$.*

According to Lemma 3.4, if the result of a value cast under a type A is a value, then it should be well-formed. Lemma 3.5 shows that the target type A is preserved after casting: if a value v is cast using 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 casting 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 3.6 shows that if a value v is well-typed with type A,

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

$$\textsc{Step-eval}\quad \frac{e \longmapsto e'}{F[e] \longmapsto F[e']} \qquad \textsc{Step-blame}\quad \frac{e \longmapsto \mathsf{blame}}{F[e] \longmapsto \mathsf{blame}} \qquad \textsc{Step-annov}\quad \frac{v \Downarrow_A r \qquad \neg value\ (v : A)}{v : A \longmapsto r}$$

$$\textsc{Step-betap}\quad \frac{\rceil v_1 \lceil = A \to B \qquad v_2 \Downarrow_A \mathsf{blame}}{v_1\, v_2 \longmapsto \mathsf{blame}} \qquad \textsc{Step-Beta}\quad \frac{v_2 \Downarrow_A v_2'}{(\lambda x.\, e : A \to B)\, v_2 \longmapsto e[x \mapsto v_2'] : B}$$

$$\textsc{Step-abeta}\quad \frac{value\ (v_1 : A \to B) \qquad v_2 \Downarrow_A v_2'}{(v_1 : A \to B)\, v_2 \longmapsto (v_1\, v_2') : B} \qquad \textsc{Step-dyn}\quad \frac{value\ (v_1 : \star)}{(v_1 : \star)\, v_2 \longmapsto (v_1 : \star : \star \to \star)\, v_2}$$

$$\textsc{Step-pjd}\quad \frac{i \in \{1, 2\}}{\pi_i\ (v : \star) \longmapsto \pi_i\ (v : \star : \star \times \star)} \qquad \textsc{Step-c}\quad \frac{\rceil c \lceil = A \to B \qquad v \Downarrow_A v'}{c\, v \longmapsto [\![c]\!](v')} \qquad \textsc{Step-pj}\quad \frac{i \in \{1, 2\}}{\pi_i\ (v_1, v_2) \longmapsto v_i}$$

Fig. 5. Semantics of λB^g.

then casting the value will either return a well-formed value or blame. The casting relation is deterministic for well-typed values (Lemma 3.7): if a well-typed value v is cast by type A, the result will be unique. Finally, if v is cast by A, the dynamic type of v should be consistent with type A (Lemma 3.8). Most of these lemmas are proved by induction on the casting relation.

Reduction. The reduction rules are shown in Figure 5. Rule Step-eval and rule Step-blame are standard rules to reduce subterms under evaluation contexts. A key difference between the reduction rules for λB^g and the blame calculus lies on the treatment of applications. λB^g supports *flexible* applications with implicit type conversions, while the blame calculus does not have such flexibility. To account for the extra flexibility, the semantics of applications needs to be more complex in λB^g.

Rule Step-beta is the beta reduction rule. Importantly, note that casting under type A is needed for v_2: that is we cast value v_2 to v_2' so that v_2' has the required input type A. Moreover, the result of the reduction is the substitution $e[x \mapsto v_2']$ annotated with type B. In rule Step-abeta, the type of the argument v_2 should also be consistent with the input type (A) of the annotated function value. Furthermore, $v_1 : A \to B$ is a well-formed value.

From typing, we know that $\star$ matches with a dynamic function type $\star \to \star$. Therefore, rule step-dyn annotates functional values of the form $v_1 : \star$ with type $\star \to \star$. Note that, when the value (v_1) does not match the required type structure, blame should be raised. For example, program $(1 : \star)\ 2$ raises blame during reduction by first applying rule Step-dyn and then rule Step-annov. Rule Step-pjd follows a similar behavior and it is annotated to product type $\star \times \star$. Rule Step-Annov is used when a value v is annotated with a type A. In that case, the value should have a type consistent with type A. So a cast is performed for the value with target type A and a result (which could be blame) is produced by casting.

Rule Step-c shows a rule that deals with primitive operations for $+$ and $+_i$ (we introduced those in Section 2), similarly to those in λB. The argument value v is cast by the

input type of the primitive operations such as Int for +. Rule STEP-PJ projects the values from products.

Determinism. The operational semantics of λB^g is *deterministic*: expressions reduce to a unique result. Theorem 3.1 is proved using Lemma 3.7.

Theorem 3.1 (Determinism of λB^g calculus)**.** *If* $\cdot \vdash e \Leftrightarrow A$, $e \longmapsto r_1$ *and* $e \longmapsto r_2$ *then* $r_1 = r_2$.

Type Soundness. The λB^g calculus is type sound. Theorem 3.2 says that if an expression is well-typed with type A, the type will be preserved after the reduction. Progress is given by Theorem 3.3. A well-typed expression e is either a value, or it reduces to a result.

Theorem 3.2 (Type Preservation of λB^g Calculus)**.** *If* $\cdot \vdash e \Leftrightarrow A$ *and* $e \longmapsto e'$ *then* $\cdot \vdash e' \Leftrightarrow A$.

Theorem 3.3 (Progress of λB^g Calculus)**.** *If* $\cdot \vdash e \Leftrightarrow A$ *then* e *is a value or* $\exists r$, $e \longmapsto r$.

4 The λB_l^g Calculus

In Section 3, we introduced the λB^g calculus. In this section, we show that the TDOS approach can also support blame labels. We introduce the λB_l^g calculus, which is the λB^g calculus with blame labels and show that it is sound and complete with respect to λB with blame labels. Furthermore, we prove the gradual guarantee and a blame safety theorem for the λB_l^g calculus.

4.1 Static Semantics

Syntax. The syntax of the λB_l^g calculus is shown in Figure 6. Most of the syntax is the same as λB^g. One small difference is that for annotations, lambdas, projections, and application expressions, there are labels.

Typing. The typing rules of λB_l^g are shown at the bottom of Figure 6. The rules follow those of λB^g. Compared to the λB^g calculus, rules TYP-ABS, TYP-APP, TYP-ANNO, and TYP-PI now deal with blame labels. In particular, for applications $(e_1\ e_2)^l$, the type of the argument (e_2) must be consistent with the input type of the function type, which is A. For this reason applications require a label for tracking the implicit cast for the application argument. Furthermore, since the type of e_1 can be $\star$, there can be an implicit cast to type $\star \to \star$. The label in applications is used to track these possible casts. Similarly, the label for projections $\pi_i^l\ e$ is used to track the possible cast from $\star$ to $\star \times \star$.

4.2 Dynamic Semantics

Casting. The form of the casting relation changes to $v \Downarrow_A^l r$. The new form means that a value v annotated with type A reduces under type A with blame label l. Blame is raised with a label during casting if we try to reduce a value with unknown type under a type that is not consistent with the type of the value. The label indicates the location in the program to blame. For instance, trying to cast the value $1 :^{l_1} \star$ under the type Bool with label l_2 indicates that this cast raises blame and we get the result blame l_2. The same example would

Syntax

Blame Labels l	
Types	$A, B ::= \mathsf{Int} \mid \star \mid A \to B \mid A \times B$
Constants	$c ::= i \mid + \mid +_i$
Expressions	$e ::= c \mid x \mid (e_1, e_2) \mid \pi_i^l\ e \mid \lambda x.\, e :^l A \to B \mid (e_1\ e_2)^l \mid e :^l A$
Result	$r ::= e \mid \mathsf{blame}\ l$
Values	$v ::= c \mid (v_1, v_2) \mid \lambda x.\, e :^l A \to B \mid v :^l A \to B \mid v :^l \star$
Context	$\Gamma ::= \cdot \mid \Gamma, x : A$
Frame	$F ::= (\square, e) \mid (v, \square) \mid \pi_i^l\ \square \mid (\square\ e)^l \mid (v\ \square)^l \mid \square :^l A$

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

VALUEL-C
$$\frac{}{value\ c}$$

VALUEL-ABS
$$\frac{}{value\ \lambda x.\, e :^l A \to B}$$

VALUEL-FANNO
$$\frac{\rceil v \lceil = C \to D \qquad value\ v}{value\ v :^l A \to B}$$

VALUEL-DYN
$$\frac{Ground\ \rceil v \lceil \qquad value\ v}{value\ v :^l \star}$$

VALUEL-PRO
$$\frac{value\ v_1 \qquad value\ v_2}{value\ (v_1, v_2)}$$

Ground A *(Ground types)*

GROUND-INT
$$\frac{}{Ground\ \mathsf{Int}}$$

GROUND-ARR
$$\frac{}{Ground\ \star \to \star}$$

GROUND-PRO
$$\frac{}{Ground\ \star \times \star}$$

$\Gamma \vdash e \Leftrightarrow A$ *(Typing)*

TYP-C
$$\frac{}{\Gamma \vdash c \Rightarrow \rceil c \lceil}$$

TYP-VAR
$$\frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A}$$

TYP-APP
$$\frac{A \rhd A_1 \to A_2 \qquad \Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Leftarrow A_1}{\Gamma \vdash (e_1\ e_2)^l \Rightarrow A_2}$$

TYP-ANNO
$$\frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash e :^l A \Rightarrow A}$$

TYP-SIM
$$\frac{\Gamma \vdash e \Rightarrow A \qquad A \sim B}{\Gamma \vdash e \Leftarrow B}$$

TYP-ABS
$$\frac{\Gamma, x : A \vdash e \Leftarrow B}{\Gamma \vdash \lambda x.\, e :^l A \to B \Rightarrow A \to B}$$

TYP-PRO
$$\frac{\Gamma \vdash e_1 \Rightarrow A_1 \qquad \Gamma \vdash e_2 \Rightarrow A_2}{\Gamma \vdash (e_1, e_2) \Rightarrow A_1 \times A_2}$$

TYP-PI
$$\frac{\Gamma \vdash e \Rightarrow A \qquad A \blacktriangleright A_1 \times A_2 \qquad i \in \{1, 2\}}{\Gamma \vdash \pi_i^l\ e \Rightarrow A_i}$$

Fig. 6. Static semantics for the λB_l^g calculus.

$\boxed{v \Downarrow^l_A r}$ *(Casting for λB^g_l calculus)*

$$\textsc{Cast-Abs}\quad \frac{\rceil v \lceil = C \to D \qquad C \to D \sim A \to B}{v \Downarrow^l_{A \to B} v :^l A \to B}$$

$$\textsc{Cast-V}\quad \frac{\mathit{Ground}\ \rceil v \lceil}{v \Downarrow^l_\star v :^l \star}$$

$$\textsc{Cast-Lit}\quad \frac{}{i \Downarrow^l_{\mathsf{Int}} i}$$

$$\textsc{Cast-DD}\quad \frac{}{v :^{l_1} \star \Downarrow^{l_2}_\star v :^{l_1} \star}$$

$$\textsc{Cast-AnyD}\quad \frac{ug(\rceil v \lceil, G) \qquad v \Downarrow^l_G v'}{v \Downarrow^l_\star v' :^l \star}$$

$$\textsc{Cast-DynA}\quad \frac{ug(A, G) \qquad v \Downarrow^{l_2}_A r}{v :^{l_1} \star \Downarrow^{l_2}_A r}$$

$$\textsc{Cast-VAny}\quad \frac{}{v :^{l_1} \star \Downarrow^{l_2}_{\rceil v \lceil} v}$$

$$\textsc{Cast-Blame}\quad \frac{\rceil v \lceil \not\sim A}{v :^{l_1} \star \Downarrow^{l_2}_A (\mathsf{blame}\ l_2)}$$

$$\textsc{Cast-Pro}\quad \frac{v_1 \Downarrow^l_{A_1} v_1' \qquad v_2 \Downarrow^l_{A_2} v_2'}{(v_1, v_2) \Downarrow^l_{A_1 \times A_2} (v_1', v_2')}$$

$$\textsc{Cast-L}\quad \frac{v_1 \Downarrow^l_{A_1} (\mathsf{blame}\ l) \qquad v_2 \Downarrow^l_{A_2} r}{(v_1, v_2) \Downarrow^l_{A_1 \times A_2} (\mathsf{blame}\ l)}$$

$$\textsc{Cast-R}\quad \frac{v_1 \Downarrow^l_{A_1} v_1' \qquad v_2 \Downarrow^l_{A_2} (\mathsf{blame}\ l)}{(v_1, v_2) \Downarrow^l_{A_1 \times A_2} (\mathsf{blame}\ l)}$$

Fig. 7. Casting for the λB^g_l Calculus.

be represented in the λB calculus as $(1 : \mathsf{Int} \overset{l_1}{\Rightarrow} \star) : \star \overset{l_2}{\Rightarrow} \mathsf{Bool}$, which also raises blame with label l_2.

Figure 7 shows the rules of casting. Except for rule CAST-BLAME, the other rules in casting are similar to the λB^g calculus. For rule CAST-BLAME, we use the blame label tracked by casting to return the blame result. The casting of λB^g_l shares all the same properties in λB^g.

Reduction. The reduction rules of λB^g_l are shown in Figure 8. They extend the rules in Figure 5 with blame labels l. The rule SSTEP-BETA deals with applications where the functions are annotated: $((\lambda x.\, e :^l A \to B)\ v)^{l_1}$. Note that this reduction rule requires casting the arguments before beta-reduction. To type such applications we use:

$$\frac{\Gamma \vdash \lambda x.\, e :^l A \to B \Rightarrow A \to B \qquad \Gamma \vdash v \Leftarrow A}{\Gamma \vdash ((\lambda x.\, e :^l A \to B)\ v)^{l_1} \Rightarrow B}$$

Due to the checking mode for the argument of an application, there is an implicit cast, which is tracked by the label l_1. For instance, consider $(((\lambda x.\, x) :^{l_0} \mathsf{Bool} \to \mathsf{Bool})\ (1 :^{l_1} \star))^l$. In this example, $1 :^{l_1} \star$ needs to be cast to type Bool with the application label l and raise $\mathsf{blame}\ l$. The label of applications $(e_1\ e_2)^l$ is helpful for two reasons. Firstly, it is used to track blame for casts of the argument. Secondly, it is also used to track blame when casting e_1 from $\star$ to the dynamic function type $\star \to \star$ (rule SSTEP-DYN). In rule SSTEP-ABETA, l is used to track blame in the cast from v_1 to $A \to B$. While removing the annotation $A \to B$, the label of application $(v_1\ v_2')$ is flipped to $\bar{l}$: if a cast is raised, then the context which contains the cast is to blame.

Let us look at two small examples to illustrate reduction behavior and blame tracking. The first example is the term $(((\lambda x.\, x :^{l_1} \star \to \star) :^{l_2} \mathsf{Bool} \to \mathsf{Bool})\ (1 :^{l_4} \star))^{l_3}$, which reduces

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

$$\textsc{Sstep-eval}\quad \frac{e \longmapsto e'}{F[e] \longmapsto F[e']} \qquad \textsc{Sstep-blame}\quad \frac{e \longmapsto \mathsf{blame}\ l}{F[e] \longmapsto \mathsf{blame}\ l} \qquad \textsc{Sstep-annov}\quad \frac{v \Downarrow_A^l r \qquad \neg(\mathit{value}\ v :^l A)}{v :^l A \longmapsto r}$$

$$\textsc{Sstep-betap}\quad \frac{\rceil v_1 \lceil = A \to B \qquad v_2 \Downarrow_A^l (\mathsf{blame}\ l)}{(v_1\ v_2)^l \longmapsto \mathsf{blame}\ l} \qquad \textsc{Sstep-Beta}\quad \frac{v \Downarrow_A^{l_1} v'}{((\lambda x.\, e :^l A \to B)\ v)^{l_1} \longmapsto e[x \mapsto v'] :^l B}$$

$$\textsc{Sstep-Abeta}\quad \frac{\mathit{value}\ (v_1 :^l A \to B) \qquad v_2 \Downarrow_A^{l_1} v_2'}{((v_1 :^l A \to B)\ v_2)^{l_1} \longmapsto ((v_1\ v_2')^{\bar{l}}) :^l B} \qquad \textsc{Sstep-pi}\quad \frac{i \in \{1,2\}}{\pi_i^l\ (v_1, v_2) \longmapsto v_i}$$

$$\textsc{Sstep-dyn}\quad \frac{\mathit{value}\ (v_1 :^{l_1} \star)}{((v_1 :^{l_1} \star)\ v_2)^{l_2} \longmapsto ((v_1 :^{l_1} \star :^{l_2} \star \to \star)\ v_2)^{l_2}} \qquad \textsc{Sstep-C}\quad \frac{\rceil v \lceil = A \to B \qquad v \Downarrow_A^l v'}{(c\ v)^l \longmapsto [\![c]\!](v')}$$

$$\textsc{Sstep-pd}\quad \frac{i \in \{1,2\}}{\pi_i^{l_1}\ (v :^{l_0} \star) \longmapsto \pi_i^{l_1}\ (v :^{l_0} \star :^{l_1} \star \times \star)}$$

Fig. 8. Semantics of λB_l^g.

to $(\mathsf{blame}\ l_3)$.

$$\begin{aligned} & (((\lambda x.\, x :^{l_1} \star \to \star) :^{l_2} \mathsf{Bool} \to \mathsf{Bool})\ (1 :^{l_4} \star))^{l_3} \\ \longmapsto\ & \{\text{by rule } \textsc{Sstep-betap}\} \\ & \mathsf{blame}\ l_3 \end{aligned}$$

The second example is a term $(((\lambda x.\, x :^{l_1} \star \to \star) :^{l_2} \mathsf{Int} \to \star)\ (1 :^{l_4} \star))^{l_3}$, which reduces to $1 :^{l_2} \star$. We show the detailed reduction steps below:

$$\begin{aligned} & (((\lambda x.\, x :^{l_1} \star \to \star) :^{l_2} \mathsf{Int} \to \star)\ (1 :^{l_4} \star))^{l_3} \\ \longmapsto\ & \{\text{by rule } \textsc{cast-vany} \text{ and rule } \textsc{Sstep-abeta}\} \\ & (((\lambda x.\, x :^{l_1} \star \to \star)\ 1)^{\bar{l}_2}) :^{l_2} \star \\ \longmapsto\ & \{\text{by rule } \textsc{Sstep-eval}\text{, rule } \textsc{Sstep-beta}\} \\ & ((1 :^{\bar{l}_2} \star) :^{l_1} \star) :^{l_2} \star \\ \longmapsto\ & \{\text{by rule } \textsc{Sstep-eval}\text{, rule } \textsc{Sstep-annov} \text{ and rule } \textsc{cast-dd}\} \\ & (1 :^{\bar{l}_2} \star) :^{l_2} \star \\ \longmapsto\ & \{\text{rule } \textsc{Sstep-annov} \text{ and rule } \textsc{cast-dd}\} \\ & 1 :^{\bar{l}_2} \star \end{aligned}$$

Correspondence with λB^g. We proved that after removing the blame labels in λB_l^g, the static and dynamic semantics are the same as in λB^g (Theorem 4.1). Function $|\cdot|$ removes the blame labels. To distinguish between the typing and reduction relations of

Expressions $e ::= \ldots \mid e_1\ e_2$

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

$$\frac{\rceil v_1 \lceil = A \to B \qquad v_2 \Downarrow_A^l v_2'}{(v_1\ v_2)^l \longmapsto v_1\ v_2'}\ \text{STEP-APP} \qquad \frac{}{(\lambda x.\, e :^l A \to B)\ v \longmapsto e[x \mapsto v] :^l B}\ \text{STEP-BETA}$$

$$\frac{\rceil v_1 \lceil = C \to D}{(v_1 :^l A \to B)\ v_2 \longmapsto (v_1\ (v_2 :^{\bar{l}} C)) :^l B}\ \text{STEP-ABETA} \qquad \frac{}{c\ v \longmapsto [\![c]\!](v)}\ \text{STEP-C}$$

Fig. 9. New reduction steps.

λB^g and λB^g_l, we use their name as a superscript in the relation. For example, $\longmapsto^{\lambda B^g_l}$ is the reduction of λB^g_l.

Theorem 4.1 (Relation between λB^g and λB^g_l). *$\Gamma \vdash^{\lambda B^g_l} e \Leftrightarrow A$ and $e \longmapsto^{\lambda B^g_l} r$ iff $\Gamma \vdash^{\lambda B^g}$ $|e| \Leftrightarrow A$ and $|e| \longmapsto^{\lambda B^g} |r|$.*

4.3 A More Refined Reduction Relation for Proofs

The reduction relation in the previous section is fine for an implementation. However, for some of our proofs, including the completeness to the blame calculus and the gradual guarantee, it is more convenient to have a more refined reduction relation that introduces some additional reduction steps. Here we present the more refined reduction relation. We proved that this new relation is equivalent to the reduction relation in Section 6.3.

In the new reduction relation (shown in Figure 9), the original rule SSTEP-BETA, rule SSTEP-ABETA and rule SSTEP-C are divided into two separate steps. While other rules are the same. Figure 10 illustrates the relationship between the original rules and the new rules. Take rule STEP-BETA as an example. There are two steps: 1) cast the argument; and 2) do the substitution. To record that the argument has been cast, we use *strict applications* $e_1\ e_2$. In strict applications, the type of the argument matches the input type of function e_1. The typing rule for strict applications is:

$$\frac{\Gamma \vdash e_1 \Rightarrow A \to B \qquad \Gamma \vdash e_2 \Rightarrow A}{\Gamma \vdash e_1\ e_2 \Rightarrow B}\ \text{TYP-APPV}$$

Other rules follow the same principle as rule STEP-BETA. The first step is to cast the argument, and then do the corresponding operations such as substitution and unwrapping type annotations. The bottom-left triangle in Figure 10 regards that $(v_1 :^l A \to B)\ v_2'$ reduces to $((v_1\ v_2')^{\bar{l}}) :^l B$. Strictly speaking, this does not hold because $(v_1 :^l A \to B)\ v_2'$ reduces to $(v_1\ (v_2' :^{\bar{l}} C)) :^l B$ for some C such that $\rceil v_1 \lceil = C \to D$ by rule STEP-ABETA. However, the result can be regarded as $((v_1\ v_2')^{\bar{l}}) :^l B$ because by rule STEP-APP, applications $(v_1\ v_2')^{\bar{l}}$ convert to strict applications $v_1\ v_2''$, which can also be written as strict

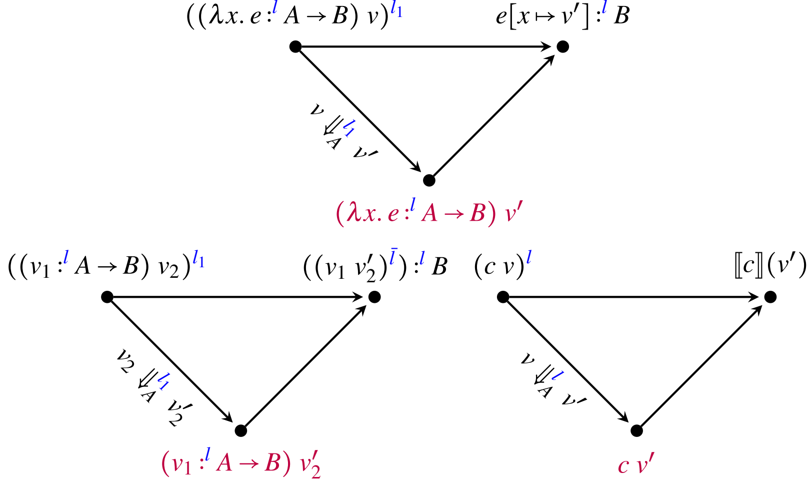

Fig. 10. Decomposition of reduction steps.

applications $v_1\ (v_2' :^{\bar{l}} C)$ and the casting is then triggered by annotations. We illustrate the new reduction on an earlier example shown in Section 6.3.

$$
\begin{aligned}
& (((\lambda x.\, x :^{l_1} \star \to \star) :^{l_2} \mathsf{Int} \to \star)\ (1 :^{l_4} \star))^{l_3} \\
\longmapsto\ & ((\lambda x.\, x :^{l_1} \star \to \star) :^{l_2} \mathsf{Int} \to \star)\ 1 \\
\longmapsto\ & ((\lambda x.\, x :^{l_1} \star \to \star)\ (1 :^{\bar{l}_2} \star)) :^{l_2} \star \\
\longmapsto\ & ((1 :^{\bar{l}_2} \star) :^{l_1} \star) :^{l_2} \star \\
\longmapsto\ & (1 :^{\bar{l}_2} \star) :^{l_2} \star \\
\longmapsto\ & 1 :^{\bar{l}_2} \star
\end{aligned}
$$

4.4 Soundness and Completeness to λB

We use an elaboration step in the typing relation to prove the soundness result between the semantics of λB^g_l and λB. The elaboration from λB^g_l to λB is shown at the top of Figure 11. The elaborating typing relation $\Gamma \vdash e \overset{l}{\Leftrightarrow} A \rightsquigarrow t$ shows how to translate expression e in λB^g_l to the term t in λB. The translation does not include products and projections, since they are not part of λB. The typing mode $\overset{l}{\Leftrightarrow}$ includes the infer mode $\Rightarrow$ and the checking mode with labels $\overset{l}{\Leftarrow}$. However, in λB^g_l, every rule in checking mode induces a cast and we need the label for the elaboration into λB. Therefore, the blame labels of λB^g_l are transferred to λB via the checking mode. In particular, in rule TYP-SIM, when we produce a λB cast expression we need to use the label that was tracked by the checking mode. There are three rules where the blame labels are propagated using the checking mode in the premises (rule TYP-ABS, rule TYP-APP and rule TYP-ANNO). Note that the λB^g_l calculus can also act as a source language. An application of expression e_1 can, not only

Typing mode $\overset{l}{\Leftrightarrow} ::= \Rightarrow | \overset{l}{\Leftarrow}$

$\boxed{\Gamma \vdash e \overset{l}{\Leftrightarrow} A \rightsquigarrow t}$ *(Elaboration from λB^g_l to λB)*

$$\text{TYP-C}\quad \frac{}{\Gamma \vdash c \Rightarrow \rceil c \lceil \rightsquigarrow c}
\qquad
\text{TYP-VAR}\quad \frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A \rightsquigarrow x}$$

$$\text{TYP-APP}\quad \frac{A \rhd A_1 \to A_2 \qquad \Gamma \vdash e_1 \Rightarrow A \rightsquigarrow t_1 \qquad \Gamma \vdash e_2 \overset{l}{\Leftarrow} A_1 \rightsquigarrow t_2}{\Gamma \vdash (e_1\ e_2)^l \Rightarrow A_2 \rightsquigarrow \langle t_1, l, A\rangle\, t_2}$$

$$\text{TYP-ANNO}\quad \frac{\Gamma \vdash e \overset{l}{\Leftarrow} A \rightsquigarrow t}{\Gamma \vdash e :^l A \Rightarrow A \rightsquigarrow t}
\qquad
\text{TYP-SIM}\quad \frac{\Gamma \vdash e \Rightarrow A \rightsquigarrow t \qquad A \sim B}{\Gamma \vdash e \overset{l}{\Leftarrow} B \rightsquigarrow t : A \overset{l}{\Rightarrow} B}$$

$$\text{TYP-ABS}\quad \frac{\Gamma, x : A \vdash e \overset{l}{\Leftarrow} B \rightsquigarrow t}{\Gamma \vdash \lambda x.\, e :^l A \to B \Rightarrow A \to B \rightsquigarrow \lambda x : A.\, t}
\qquad
\text{TYP-APPV}\quad \frac{\Gamma \vdash e_1 \Rightarrow A \to B \rightsquigarrow t_1 \qquad \Gamma \vdash e_2 \Rightarrow A \rightsquigarrow t_2}{\Gamma \vdash e_1\ e_2 \Rightarrow B \rightsquigarrow t_1\ t_2}$$

$\langle t, l, \star\rangle$	$=$	$t : \star \overset{l}{\Rightarrow} \star \to \star$	$\star$	$\rhd$	$\star \to \star$
$\langle t, l, A \to B\rangle$	$=$	t	$A \to B$	$\rhd$	$A \to B$
$\langle\langle t, l, \star\rangle\rangle$	$=$	$t : \star \overset{l}{\Rightarrow} \star \times \star$	$\star$	$\blacktriangleright$	$\star \times \star$
$\langle\langle t, l, A \times B\rangle\rangle$	$=$	t	$A \times B$	$\blacktriangleright$	$A \times B$

$\boxed{\Gamma \Vdash t : A \rightsquigarrow e}$ *(Elaboration from λB to λB^g_l)*

$$\text{BTYP-C}\quad \frac{}{\Gamma \Vdash c : \rceil c \lceil \rightsquigarrow c}
\qquad
\text{BTYP-VAR}\quad \frac{x : A \in \Gamma}{\Gamma \Vdash x : A \rightsquigarrow x}$$

$$\text{BTYP-ABS}\quad \frac{\Gamma, x : A \Vdash t : B \rightsquigarrow e :^l B}{\Gamma \Vdash \lambda x : A.\, t : A \to B \rightsquigarrow (\lambda x.\, e) :^l A \to B}$$

$$\text{BTYP-APP}\quad \frac{\Gamma \Vdash t_1 : A \to B \rightsquigarrow e_1 \qquad \Gamma \Vdash t_2 : A \rightsquigarrow e_2}{\Gamma \Vdash t_1\ t_2 : B \rightsquigarrow e_1\ e_2}
\qquad
\text{BTYP-CAST}\quad \frac{\Gamma \Vdash t : A \rightsquigarrow e \qquad A \sim B}{\Gamma \Vdash t : A \overset{l}{\Rightarrow} B : B \rightsquigarrow e :^l B}$$

Fig. 11. Elaboration between λB^g_l and λB.

infer a function type, but also infer the unknown type $\star$. Thus, the function $\langle t, l, A\rangle$ adds a cast from $\star$ to $\star \to \star$ to the λB term. At the bottom of Figure 11, we show the elaboration from λB to λB^g_l. The elaboration typing relation $\Gamma \Vdash t : A \rightsquigarrow e$ shows the expression t in λB to the expression e in λB^g_l. The elaboration is straightforward. One thing that needs to be mentioned is that for rule BTYP-ABS the lambda body (t) is elaborated to an annotated expression, since the body is in checking mode, which triggers an implicit cast. To be aligned between these two calculi, the λB calculus adds an identity cast for the lambda body.

Theorem 4.2 states that both elaborations are type preserving. Theorem 4.3 shows the soundness and completeness property between the dynamic semantics of λB_l^g and λB. We use $\Uparrow$ to represent that a term diverges. The soundness and completeness result are proved using the auxiliary lemmas 4.1, 4.2 , 4.3 and 4.4. In Lemma 4.3, $t \longmapsto^j t'$ means that t takes j steps to evaluate to t'. Lets take an example that shows how can t' reduce to error. In λB, we can have a program $(i : \mathsf{Int} \overset{l_1}{\Rightarrow} \star) : \star \overset{l_2}{\Rightarrow} \mathsf{Int} \to \mathsf{Int}$ which corresponds to $i :^{l_1} \star :^{l_2} \mathsf{Int} \to \mathsf{Int}$ in our λB_l^g. In λB_l^g, error is raised directly but λB first reduces to $((i : \mathsf{Int} \overset{l_1}{\Rightarrow} \star) : \star \overset{l_2}{\Rightarrow} \star \to \star) : \star \to \star \overset{l_2}{\Rightarrow} \mathsf{Int} \to \mathsf{Int}$ and then raises the error.

Theorem 4.2 (Type Preservation of Elaboration)**.**

- *If* $\Gamma \vdash e \overset{l}{\Leftrightarrow} A \rightsquigarrow t$ *then* $\Gamma \Vdash t : A$.
- *If* $\Gamma \Vdash t : A \rightsquigarrow e$ *then* $\Gamma \vdash e \Rightarrow A$.

Lemma 4.1 (Soundness of λB_l^g Casting with respect to λB for Values)**.** *If* $\cdot \vdash v :^l A \Rightarrow A \rightsquigarrow t$ *and* $v \Downarrow_A^l v'$ *then* $\exists t', t \longmapsto^* t'$ *and* $\cdot \vdash v' \Rightarrow A \rightsquigarrow t'$.

Lemma 4.2 (Soundness of λB_l^g Casting with respect to λB for Blame)**.** *If* $\cdot \vdash v :^l A \Rightarrow A \rightsquigarrow t$ *and* $v \Downarrow_A^l$ $(\mathsf{blame}\ l)$ *then* $t \longmapsto^* \mathsf{blame}\ l$.

Lemma 4.3 (Completeness of λB_l^g Casting with respect to λB for Values)**.** *If* $\cdot \Vdash t : B \rightsquigarrow v$ *and* $t : B \overset{l}{\Rightarrow} A \longmapsto t'$ *then* $(\exists j\ t''\ v',\ 0 \le j \le 1,\ t' \longmapsto^j t'',\ v \Downarrow_A^l v'$ *and* $\cdot \Vdash t'' : A \rightsquigarrow v')$ *or* $(t' \longmapsto \mathsf{blame}\ l$ *and* $v \Downarrow_A^l (\mathsf{blame}\ l))$.

Lemma 4.4 (Completeness of λB_l^g Casting with respect to λB for Blame)**.** *If* $\cdot \Vdash t : B \rightsquigarrow v$ *and* $t : B \overset{l}{\Rightarrow} A \longmapsto \mathsf{blame}\ l$ *then* $v \Downarrow_A^l (\mathsf{blame}\ l)$.

Theorem 4.3 (Soundness and Completeness between λB_l^g and λB)**.**

1) *if* $\cdot \vdash e \Rightarrow A \rightsquigarrow t$ *and* $e \longmapsto^* v$ *then* $\exists v', t \longmapsto^* v'$ *and* $\cdot \vdash v \Rightarrow A \rightsquigarrow v'$.
2) *if* $\cdot \vdash e \Rightarrow A \rightsquigarrow t$ *and* $e \longmapsto^* \mathsf{blame}\ l$ *then* $t \longmapsto^* \mathsf{blame}\ l$.
3) *if* $\cdot \vdash e \Rightarrow A \rightsquigarrow t$ *and* $e \Uparrow$ *then* $t \Uparrow$.
4) *if* $\cdot \Vdash t : A \rightsquigarrow e$ *and* $t \longmapsto^* v$ *then* $\exists v', e \longmapsto^* v'$ *and* $\cdot \Vdash v : A \rightsquigarrow v'$.
5) *if* $\cdot \Vdash t : A \rightsquigarrow e$ *and* $t \longmapsto^* \mathsf{blame}\ l$ *then* $e \longmapsto^* \mathsf{blame}\ l$.
6) *if* $\cdot \Vdash t : A \rightsquigarrow e$ *and* $t \Uparrow$ *then* $e \Uparrow$.

4.5 Gradual Guarantee

Siek *et al.* (2015*b*) 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. We show how to prove the gradual guarantee for λB_l^g next.

Precision. The top of Figure 12 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. A product type $A_1 \times B_1$ is more precise than $A_2 \times B_2$ if type A_1 is more

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

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

$\boxed{\Gamma_1; \Gamma_2 \vdash e_1 \sqsubseteq e_2}$ *(Precision relation for expressions)*

$$\text{EPRE-C}\quad \frac{}{\Gamma_1; \Gamma_2 \vdash c \sqsubseteq c} \qquad \text{EPRE-X}\quad \frac{}{\Gamma_1; \Gamma_2 \vdash x \sqsubseteq x}$$

$$\text{EPRE-ABS}\quad \frac{A_1 \to B_1 \sqsubseteq A_2 \to B_2 \qquad \Gamma_1, x : A_1; \Gamma_2, x : A_2 \vdash e_1 \sqsubseteq e_2}{\Gamma_1; \Gamma_2 \vdash \lambda x.\, e_1 :^{l} A_1 \to B_1 \sqsubseteq \lambda x.\, e_1 :^{l} A_2 \to B_2}$$

$$\text{EPRE-APP}\quad \frac{\Gamma_1; \Gamma_2 \vdash e_1 \sqsubseteq e_1' \qquad \Gamma_1; \Gamma_2 \vdash e_2 \sqsubseteq e_2'}{\Gamma_1; \Gamma_2 \vdash (e_1\ e_2)^{l} \sqsubseteq (e_1'\ e_2')^{l}} \qquad \text{EPRE-APPV}\quad \frac{\Gamma_1; \Gamma_2 \vdash e_1 \sqsubseteq e_1' \qquad \Gamma_1; \Gamma_2 \vdash e_2 \sqsubseteq e_2'}{\Gamma_1; \Gamma_2 \vdash e_1\ e_2 \sqsubseteq e_1'\ e_2'}$$

$$\text{EPRE-PRO}\quad \frac{\Gamma_1; \Gamma_2 \vdash e_1 \sqsubseteq e_1' \qquad \Gamma_1; \Gamma_2 \vdash e_2 \sqsubseteq e_2'}{\Gamma_1; \Gamma_2 \vdash (e_1, e_2) \sqsubseteq (e_1', e_2')}$$

$$\text{EPRE-ANNOL}\quad \frac{\Gamma_1 \vdash e_1 \Rightarrow A_1 \qquad \Gamma_2 \vdash e_2 \Rightarrow A_2 \qquad \Gamma_1; \Gamma_2 \vdash e_1 \sqsubseteq e_2 \qquad A \sqsubseteq A_2 \qquad A_1 \sqsubseteq A_2}{\Gamma_1; \Gamma_2 \vdash e_1 :^{l} A \sqsubseteq e_2}$$

$$\text{EPRE-ANNOR}\quad \frac{\Gamma_1 \vdash e_1 \Rightarrow A_1 \qquad \Gamma_2 \vdash e_2 \Rightarrow A_2 \qquad \Gamma_1; \Gamma_2 \vdash e_1 \sqsubseteq e_2 \qquad A_1 \sqsubseteq A_2 \qquad A_1 \sqsubseteq A}{\Gamma_1; \Gamma_2 \vdash e_1 \sqsubseteq e_2 :^{l} A}$$

$$\text{EPRE-ANNO}\quad \frac{A \sqsubseteq B \qquad \Gamma_1; \Gamma_2 \vdash e_1 \sqsubseteq e_2}{\Gamma_1; \Gamma_2 \vdash e_1 :^{l} A \sqsubseteq e_2 :^{l} B}$$

$$\text{EPRE-PI}\quad \frac{\Gamma_1; \Gamma_2 \vdash e_1 \sqsubseteq e_2}{\Gamma_1; \Gamma_2 \vdash \pi_i^{l}\ e_1 \sqsubseteq \pi_i^{l}\ e_2}$$

Fig. 12. Precision relations.

precise then A_2 and type B_1 is more precise than B_2. The bottom of Figure 12 shows the precision relation for expressions. $\Gamma_1; \Gamma_2 \vdash e_1 \sqsubseteq e_2$ means that e_1 is more precise than e_2 under typing contexts Γ_1 and Γ_2. The precision relation of expressions is derived from the precision relation of types. Every expression is related to itself. There are contexts Γ_1 for e_1 and Γ_2 for e_2. The typing contexts are needed because some rules only work for well-typed expressions, and the typing judgement requires typing contexts. We use the following abbreviation $e_1 \sqsubseteq e_2 \equiv \cdot; \cdot \vdash e_1 \sqsubseteq e_2$, when the contexts of the expressions are empty.

For annotation expressions, precision is defined as follows: $e_1 : A$ is more precise than $e_2 : B$ if e_1 is more precise than e_2 and A is more precise than B. Similarly, for projections precision holds if the precision between e_1 and e_2 holds. Two lambdas are related when their bodies and annotated types are related. In applications ($(e_1\ e_2)^{l}$ and $e_1\ e_2$) and productions ((e_1, e_2)), precision holds if the precision between e_1 and e_1' holds and the precision between e_2 and e_2' holds.

An important complication arises from the presence of values with an unknown type such as $1:\star$. The term $1:\mathsf{Int}$ is more precise than $1:\star$, but while $1:\star$ is a value, the term $1:\mathsf{Int}$ reduces to 1. Similarly, the term $(\lambda x.\,x):\mathsf{Int}\to\mathsf{Int}:\star$ is more precise than $(\lambda x.\,x):\star\to\star:\star$, while $(\lambda x.\,x):\star\to\star:\star$ is a value and $(\lambda x.\,x):\mathsf{Int}\to\mathsf{Int}:\star$ reduces to $(\lambda x.\,x):\mathsf{Int}\to\mathsf{Int}:\star\to\star:\star$. Such examples create the need for the precision relation to deal with *unaligned expressions*. These unaligned cases are covered by rule EPRE-ANNOL and rule EPRE-ANNOR. Rule EPRE-ANNOL deals with more precise programs with extra annotations, while rule EPRE-ANNOR is for less precise programs with extra annotations. To ensure that the extra type annotations preserve the type precision relation, the typing judgement is used.

Static Criteria. The dynamic counterpart of expressions ($\check{e}$) (basically expressions without type annotations) can be encoded in λB_l^g by the translation function ($\lceil \check{e} \rceil$), which annotates dynamic expressions with type $\star$ (Theorem 4.4). The translation function is:

$$
\begin{aligned}
\lceil x \rceil &= x \\
\lceil c \rceil &= c :^{l} \star \\
\lceil \lambda x.\check{e} \rceil &= \lambda x.\lceil \check{e} \rceil :^{l_1} \star \to \star :^{l_2} \star \\
\lceil \check{e}_1\ \check{e}_2 \rceil &= (\lceil \check{e}_1 \rceil\ \lceil \check{e}_2 \rceil)^{l} \\
\lceil (\check{e}_1, \check{e}_2) \rceil &= (\lceil \check{e}_1 \rceil, \lceil \check{e}_2 \rceil) :^{l} \star \\
\lceil \pi_i\ \check{e} \rceil &= \pi_i^{l} \lceil \check{e} \rceil
\end{aligned}
$$

Furthermore, if all expressions are static, the type system of λB_l^g is equivalent to a fully static type system (Theorem 4.5). The predicate *static* on e and A means that the expression and type are fully static. The static type system is represented as $\vdash_S$ which is also bidirectional but only type checks the static expressions. Theorem 4.6 shows that the static gradual guarantee holds for the λB_l^g calculus. It says that if e is more precise than e' and e has type A then e' has type B and type A is more precise than B.

Theorem 4.4 (Dynamic Embedding)**.** *If $\check{e}$ is closed then $\cdot \vdash \lceil\ \check{e}\ \rceil \Rightarrow \star$.*

Theorem 4.5 (Equivalence for Static Type System)**.** *For all static e and A, $\cdot \vdash_S e \Leftrightarrow A$ iff $\cdot \vdash e \Leftrightarrow A$.*

Theorem 4.6 (Static Gradual Guarantee of λB_l^g Calculus)**.** *If $e \sqsubseteq e'$ and $\cdot \vdash e \Leftrightarrow A$ then $\exists B$, $\cdot \vdash e' \Leftrightarrow B$ and $A \sqsubseteq B$.*

Dynamic Gradual Guarantee. We formalized and proved the dynamic gradual guarantee for the λB_l^g calculus. Theorems 4.7 and Theorem 4.8 show that less precise programs will not change the dynamic semantics of programs. Lemma 4.5 is an important auxiliary lemma for Theorem 4.7 and Theorem 4.8. Notably, the dynamic gradual guarantee (Theorem 4.9) is a corollary of Theorem 4.7 and Theorem 4.8.

Lemma 4.5 (Dynamic Gradual Guarantee for Casting)**.** *If $v_1 \sqsubseteq v_2$, $\cdot \vdash v_1 \Leftarrow A_1$, $\cdot \vdash v_2 \Leftarrow B_1$, $A \sqsubseteq B$ and $v_1 \Downarrow_A^{l_1} v_1'$ then $\exists v_2'$, $v_2 \Downarrow_B^{l_2} v_2'$ and $v_1' \sqsubseteq v_2'$.*

Theorem 4.7 (Dynamic Gradual Guarantee $\sqsubseteq$)**.** *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 4.8 (Dynamic Gradual Guarantee $\sqsupseteq$). *Suppose $e_1 \sqsubseteq e_2$, $\cdot \vdash e_1 \Leftrightarrow A$ and $\cdot \vdash e_2 \Leftrightarrow B$.*

- *If $e_2 \longmapsto e_2'$ then ($\exists e_1$, $e_1 \longmapsto^* e_1'$ and $e_1' \sqsubseteq e_2'$) or ($\exists l$, $e_1 \longmapsto^* \mathsf{blame}\ l$).*
- *If $e_2 \longmapsto \mathsf{blame}\ l_2$ then $\exists l_1$, $e_1 \longmapsto^* \mathsf{blame}\ l_1$.*

Theorem 4.9 (Dynamic Gradual Guarantee). *Suppose $e_1 \sqsubseteq e_2$, $\cdot \vdash e_1 \Leftrightarrow A$ and $\cdot \vdash e_2 \Leftrightarrow B$.*

- *If $e_1 \longmapsto^* v_1$ then $\exists v_2$, $e_2 \longmapsto^* v_2$ and $v_1 \sqsubseteq v_2$.*
- *If $e_1 \Uparrow$ then $e_2 \Uparrow$.*
- *If $e_2 \longmapsto^* v_2$ then $e_1 \longmapsto^* v_1$ and $v_1 \sqsubseteq v_2$, or $e_1 \longmapsto^* \mathsf{blame}\ l$.*
- *If $e_2 \Uparrow$ then $e_1 \Uparrow$ or $e_1 \longmapsto^* \mathsf{blame}\ l$.*

4.6 Blame Theorem

The blame theorem (Wadler & Findler, 2009) shows that a cast from a more precise type to a less precise type cannot raise positive blame, but negative blame is possible. In addition, a cast from a less precise type to a more precise type cannot raise negative blame, but positive blame is possible. The blame theorem has been proved for the λB calculus. From the soundness theorem, which we have shown above, terms in λB_l^g raise blame with label l, and the corresponding terms in λB raise blame with the same label. Thus, the blame theorem also holds in λB_l^g. However, this proof of the blame theorem relies on the existence of λB. While this proof technique works, we cannot easily use it if we extend λB_l^g with new features, because that would require similar extensions to λB. For instance, we opted to add products and projections in λB_l^g, which are not available in λB. Thus, we cannot obtain the blame theorem for λB_l^g with products via the soundness theorem. Furthermore, having to rely on another calculus makes the proof rather indirect and involved. Therefore, here we show that we can also prove the blame theorem directly on λB_l^g.

The top of Figure 13 shows the subtyping, positive subtyping, and negative subtyping relations, which are the same as those for λB. Note that our type precision is also the same as the naive subtyping relation in λB. Therefore, Lemma 4.6 and Lemma 4.7 show that subtyping and type precision factor negative and positive subtyping as in λB.

At the bottom of Figure 13 we have the safe terms for λB_l^g. The relation $\Gamma \vdash e$ safe for l denotes that e is safe for label l under typing context Γ. For safe terms, after the reduction of e, label l can never be raised. Labels are raised only when casting is performed. Therefore, safe terms require us to consider all the casts (explicit or implicit) in the expressions. In the safe term definition, we are making implicit casts explicit. There are no casts in x and c so they are safe for label l (rule SF-VAR and rule SF-C). Lambda abstractions induce a cast from the type of the lambda body to the annotated function output type. Consequently, $(\lambda x.\, e :^{l_1} A_1 \to A_2)$ is safe for label l_2 if $e :^{l_1} A_2$ is safe for label l_2 (rule SF-ABS). Obviously, if one label does not appear in the expression, the expression is safe for that label. So $e :^{l_1} A$ is safe for l_2 if l_2 is not equal to $(l_1, \bar{l}_1)$ and e is safe for l_2 (rule SF-EQ). Even when the checking label is the same as the casting label, if the type of the source type is a positive subtype of the target type then the cast is also safe for the checking label (rule SF-NEQA).

In the expression $1 :^{l_1} \star$, the cast from Int to $\star$, will not raise blame with label l_1. In general, $e :^{l} B$ is safe for l if e is safe for l and A (the type of e) is a positive subtype of B.

$\boxed{A <: B}$ *(Subtyping)*

$$\frac{}{\mathsf{Int} <: \mathsf{Int}}\ \text{SUB-INT} \qquad \frac{}{\star <: \star}\ \text{SUB-DYN} \qquad \frac{A_2 <: A_1 \qquad B_1 <: B_2}{A_1 \to B_1 <: A_2 \to B_2}\ \text{SUB-ARR} \qquad \frac{A <: G}{A <: \star}\ \text{SUB-ADYN} \qquad \frac{A_1 <: A_2 \qquad B_1 <: B_2}{A_1 \times B_1 <: A_2 \times B_2}\ \text{SUB-PRO}$$

$\boxed{A <:^+ B}$ *(Positive Subtyping)*

$$\frac{}{\mathsf{Int} <:^+ \mathsf{Int}}\ \text{SUBA-INT} \qquad \frac{}{A <:^+ \star}\ \text{SUBA-DYN} \qquad \frac{A_2 <:^- A_1 \qquad B_1 <:^+ B_2}{A_1 \to B_1 <:^+ A_2 \to B_2}\ \text{SUBA-ARR} \qquad \frac{A_1 <:^+ A_2 \qquad B_1 <:^+ B_2}{A_1 \times B_1 <:^+ A_2 \times B_2}\ \text{SUBA-PRO}$$

$\boxed{A <:^- B}$ *(Negative Subtyping)*

$$\frac{}{\mathsf{Int} <:^- \mathsf{Int}}\ \text{SUBB-INT} \qquad \frac{}{\star <:^- A}\ \text{SUBB-DYN} \qquad \frac{A_2 <:^+ A_1 \qquad B_1 <:^- B_2}{A_1 \to B_1 <:^- A_2 \to B_2}\ \text{SUBB-ARR} \qquad \frac{A <:^- G}{A <:^- B}\ \text{SUBB-ADYN}$$

$$\frac{A_1 <:^- A_2 \qquad B_1 <:^- B_2}{A_1 \times B_1 <:^- A_2 \times B_2}\ \text{SUBB-PRO}$$

$\boxed{\Gamma \vdash e \text{ safe for } l}$ *(Safe Terms)*

$$\frac{}{\Gamma \vdash x \text{ safe for } l}\ \text{SF-VAR} \qquad \frac{}{\Gamma \vdash c \text{ safe for } l}\ \text{SF-C} \qquad \frac{\Gamma, x : A_1 \vdash (e :^{l_1} A_2) \text{ safe for } l_2}{\Gamma \vdash \lambda x.\, e_1 :^{l_1} A_1 \to A_2 \text{ safe for } l_2}\ \text{SF-ABS}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \Rightarrow A \\ A \rhd A_1 \to A_2 \qquad \Gamma \vdash \langle e_1, l_1, A\rangle \text{ safe for } l_2 \\ \Gamma \vdash (e_2 :^{l_1} A_1) \text{ safe for } l_2\end{array}}{\Gamma \vdash (e_1\ e_2)^{l_1} \text{ safe for } l_2}\ \text{SF-APP} \qquad \frac{l_1 \neq l_2 \qquad \bar{l}_1 \neq l_2 \qquad \Gamma \vdash e \text{ safe for } l_2}{\Gamma \vdash e :^{l_1} A \text{ safe for } l_2}\ \text{SF-EQ}$$

$$\frac{\Gamma \vdash e_1 \text{ safe for } l \qquad \Gamma \vdash e_2 \text{ safe for } l}{\Gamma \vdash e_1\ e_2 \text{ safe for } l}\ \text{SF-APPV} \qquad \frac{\Gamma \vdash e \Rightarrow A \qquad A <:^+ B \qquad \Gamma \vdash e \text{ safe for } l}{\Gamma \vdash e :^{l} B \text{ safe for } l}\ \text{SF-NEQA}$$

$$\frac{\Gamma \vdash e \Rightarrow A \qquad A <:^- B \qquad \Gamma \vdash e \text{ safe for } l}{\Gamma \vdash e :^{\bar{l}} B \text{ safe for } l}\ \text{SF-NEQB} \qquad \frac{\Gamma \vdash e_1 \text{ safe for } l \qquad \Gamma \vdash e_2 \text{ safe for } l}{\Gamma \vdash (e_1, e_2) \text{ safe for } l}\ \text{SF-PRO}$$

$$\frac{\begin{array}{c}\Gamma \vdash e \Rightarrow A \\ A \blacktriangleright A_1 \times A_2 \qquad \Gamma \vdash \langle\!\langle e, l_1, A\rangle\!\rangle \text{ safe for } l_2\end{array}}{\Gamma \vdash \pi_i^{l_1}\ e \text{ safe for } l_2}\ \text{SF-PI}$$

Fig. 13. Safe expressions of λB_l^g.

Types	$A, B, C ::= \mathsf{Int} \mid A \to B \mid \star \mid A_1 \times A_2$
Expressions	$e ::= x \mid c \mid e_1\ e_2 \mid e : A \mid (e_1, e_2) \mid \pi_i\ e \mid \lambda x.\ e \mid (\!| p : A |\!)$
Results	$r ::= e \mid \mathsf{blame}$
Raw values	$p ::= c \mid (\lambda x.\ e : A_1 \to B_1) : A_2 \to B_2 \mid (p_1, p_2)$
Value	$v ::= (\!| p : A |\!)$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : A$
Frames	$F ::= \square\ e \mid (v, \square) \mid (\square, e) \mid \pi_i\ \square \mid \square : A \mid (\lambda x.\ e)\ \square \mid (\!| c : A |\!)\ \square$

Fig. 14. Syntax of the λe calculus.

Annotation $e :^{l} B$ is safe for $\bar{l}$ if e is safe for l and A (the type of e) is a negative subtype of B (rule SF-NEQB). For example, $(\lambda x.\ x :^{l_1} \star \to \mathsf{Int} :^{l_2} \mathsf{Int} \to \mathsf{Int})$ should be safe for $\bar{l}_2$ since when the argument is applied, which triggers a cast from Int to $\star$ with $\bar{l}_2$, blame cannot be raised. For $(e_1\ e_2)^{l_1}$, if e_1 has type $\star$, there are two implicit casts: one is for e_1, which casts from $\star$ to $\star \to \star$, and the other is for e_2 cast from the type of e_2 to the input type of function of e_1. Therefore, if e_1 has type $\star$, $(e_1\ e_2)^{l_1}$ is safe for label l_2 if $e_1 :^{l_1} \star \to \star$ is safe for l_2 and $e_2 :^{l_1} \star$ is safe for l_2. While e_1 has type $A \to B$, $(e_1\ e_2)^{l_1}$ is safe for label l_2 if e_1 is safe for l_2 and $e_2 :^{l_1} A$ is safe for l_2 (rule SF-APP). Similarly, for rule SF-PI, if e_1 has type $\star$, there is implicit cast from $\star$ to $\star \times \star$. Thus if e has type $\star$, $\pi_i\ e\ l_1$ is safe for label l_2 if $e :^{l_1} \star \times \star$ is safe for l_2. The cast is added via the function $\langle\!\langle e, l, A \rangle\!\rangle$ at the bottom of figure 11. If e has type $A \times B$, $\pi_i^{l_1}\ e$ is safe for label l_2 if e is safe for l_2. Strict applications $(e_1\ e_2)$ and productions (e_1, e_2) are safe for l, if the sub-term e_1 and e_2 are safe for l (rule SF-APPV).

Lemma 4.8 shows the preservation of safe expressions: if e is safe for l and e reduces to e' then e' is safe for l. Lemma 4.9 says that if e is safe for l then l will not be raised. By Lemma 4.8, 4.9, 4.6 and 4.7, we can derive Corollary 4.9.1, showing that if blame is raised, the less precise program is to blame.

Lemma 4.6 (Factoring Subtyping)**.** *$A <: B$ if and only if $A <:^{+} B$ and $A <:^{-} B$.*

Lemma 4.7 (Factoring Precision)**.** *$A \sqsubseteq B$ if and only if $A <:^{+} B$ and $B <:^{-} A$.*

Lemma 4.8 (Preservation of Safe Expressions)**.** *If $\cdot \vdash e \Leftrightarrow A$, $\cdot \vdash e$ safe for l and $e \longmapsto e'$ then $\cdot \vdash e'$ safe for l.*

Lemma 4.9 (Progress of Safe Expressions)**.** *If $\cdot \vdash e \Leftrightarrow A$ and $\cdot \vdash e$ safe for l then $e \not\longmapsto \mathsf{blame}\ l$.*

Corollary 4.9.1 (Well-typed programs cannot be blamed)**.** *Let e be a well typed term with a subterm $e' :^{l} B$ where e' is well typed with type A, containing the only occurrences of l in e, and $\bar{l}$ does not appear in e.*

- *If $A <:^{+} B$ then $e \not\longmapsto^{*} \mathsf{blame}\ l$.*
- *If $A <:^{-} B$ then $e \not\longmapsto^{*} \mathsf{blame}\ \bar{l}$.*
- *If $A <: B$ then $e \not\longmapsto^{*} \mathsf{blame}\ l$ and $e \not\longmapsto^{*} \mathsf{blame}\ \bar{l}$.*
- *If $A \sqsubseteq B$ then $e \not\longmapsto^{*} \mathsf{blame}\ l$.*
- *If $B \sqsubseteq A$ then $e \not\longmapsto^{*} \mathsf{blame}\ \bar{l}$.*

5 The λe Calculus

In this section, we will introduce a gradually typed calculus with a variant of the eager semantics (Herman *et al.*, 2007), inspired by AGT (Garcia *et al.*, 2016). The main idea in this variant of the eager semantics is that if a function value flows through multiple casts, the effect of those casts is preserved in the updated function value. In particular, all types involved in the multiple casts must be checked for consistency.

5.1 Syntax

The syntax of the λe calculus is shown in Figure 14.

Types, Expressions and Results. A type in λe is either an integer type Int, a function type $A \rightarrow B$, an unknown type $\star$ or a product type $A \times B$. For expressions and results, λe is extended with tagged values $(\!|p : A|\!)$ with respect to λB^g.

Values. Metavariable p ranges over raw values. Constants c, annotated lambdas $(\lambda x.\, e : A_1 \rightarrow B_1) : A_2 \rightarrow B_2$ and products (p_1, p_2) are included in the raw values. Values are tagged raw values. The metavariable v denotes values, which are annotated values ($(\!|p : A|\!)$). Thus $(\!|i : A|\!)$ and $(\!|\lambda x.\, e : A_1 \rightarrow B_1 : A_2 \rightarrow B_2 : C|\!)$ are examples of values. Notably, in contrast with λB^g, λe's notion of (well-formed) values is purely syntactic: no additional constraints (besides syntax) are needed. Moreover, it should be noted that in λe values have a bounded number of annotations (up-to three for lambda values), unlike the λB^g calculus. We should remark that we choose to use tagged values for simplicity. It should be possible to use a more refined notion of values where some values would require no type tag, as in λB^g. Although this would probably require some different rules to distinguish between those different forms of values.

Contexts and Frames. Typing environments are just the same as in the λB^g calculus. In frames, compared to the λB^g calculus, the first expression in an application is not a value, but a raw lambda.

5.2 Type System

As the λB^g calculus, bidirectional typing is used. Before showing the typing rules, we present the dynamic type function.

Dynamic Types for the λe Calculus. As in the λB^g calculus, dynamic types play an important role in the calculus. $\rceil p \lceil$ denotes the dynamic type of p, and $\rceil v \lceil$ denotes the dynamic type of v. We need both dynamic types for raw values and values, and we can define dynamic types easily.

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

$$\text{TYP-LIT}\quad \frac{}{\Gamma \vdash c \Rightarrow \rceil c \lceil} \qquad \text{TYP-VAR}\quad \frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A} \qquad \text{TYP-ABS}\quad \frac{A \rhd A_1 \to A_2 \qquad \Gamma, x : A_1 \vdash e \Leftarrow A_2}{\Gamma \vdash \lambda x.\, e \Leftarrow A}$$

$$\text{TYP-APP}\quad \frac{A \rhd A_1 \to A_2 \qquad \Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Leftarrow A_1}{\Gamma \vdash e_1\, e_2 \Rightarrow A_2} \qquad \text{TYP-ANNO}\quad \frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash e : A \Rightarrow A} \qquad \text{TYP-SIM}\quad \frac{\Gamma \vdash e \Rightarrow A \qquad A \sim B}{\Gamma \vdash e \Leftarrow B}$$

$$\text{TYP-ABSV}\quad \frac{\Gamma, x : A \vdash e \Leftarrow B \qquad \Gamma \vdash e_2 \Rightarrow A}{\Gamma \vdash (\lambda x.\, e)\, e_2 \Leftarrow B} \qquad \text{TYP-PRODUCT}\quad \frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B}{\Gamma \vdash (e_1, e_2) \Rightarrow A \times B}$$

$$\text{TYP-PRODUCTI}\quad \frac{\Gamma \vdash e \Rightarrow A \qquad A \blacktriangleright A_1 \times A_2 \qquad i \in \{1, 2\}}{\Gamma \vdash \pi_i\, e \Rightarrow A_i} \qquad \text{TYP-VALUE}\quad \frac{\cdot \vdash p \Leftarrow A \qquad \rceil p \lceil \sqsubseteq A}{\Gamma \vdash (\!| p : A |\!) \Rightarrow A}$$

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

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

Fig. 15. Typing rules for λe.

Definition 5.1 (Dynamic type). $\rceil p \lceil$ *returns the dynamic type of the raw values* p. $\rceil v \lceil$ *returns the dynamic type of the annotated value* v.

$$\begin{aligned}
\rceil i \lceil &= \mathsf{Int} \\
\rceil + \lceil &= \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \\
\rceil +_i \lceil &= \mathsf{Int} \to \mathsf{Int} \\
\rceil ((\lambda x.\, e : A_1 \to B_1) : A_2 \to B_2) \lceil &= A_2 \to B_2 \\
\rceil (p_1, p_2) \lceil &= \rceil p_1 \lceil \times \rceil p_2 \lceil \\
\rceil (\!| p : A |\!) \lceil &= A
\end{aligned}$$

Typing Rules for the λe Calculus. In λe, raw lambdas are checked by function types or the unknown type $\star$. Raw lambdas are allowed in applications by using rule TYP-ABSV. This rule is important to allow beta-reductions to type-check, and is essentially a runtime checking rule. That is, rule TYP-ABSV is used to enable certain intermediate expressions that arise from reduction to type-check. Otherwise, using rule TYP-APP, in an application, the function input type is the type of the argument. In rule TYP-VALUE, a well-typed value ensures that the inner raw value p can be checked by the annotated type A. Also the dynamic type of the raw value should be more precise than annotated type A. The bottom of Figure 15 shows the precision relation on types. $A \sqsubseteq B$ means that A is more precise than

$\boxed{p \Downarrow_A r}$ *(Casting for λe)*

$$\frac{}{c \Downarrow_A c}\ \text{CAST-C} \qquad \frac{A_1 \to A_2 \sim C}{(\lambda x.\, e : A_1 \to A_2) : B \Downarrow_C (\lambda x.\, e : A_1 \to A_2) : C}\ \text{CAST-ABS}$$

$$\frac{A_1 \to A_2 \not\sim C}{(\lambda x.\, e : A_1 \to A_2) : B \Downarrow_C \mathsf{blame}}\ \text{CAST-ABSP} \qquad \frac{p_1 \Downarrow_{A_1} p_1' \qquad p_2 \Downarrow_{A_2} p_2'}{(p_1, p_2) \Downarrow_{A_1 \times A_2} (p_1', p_2')}\ \text{CAST-PRO}$$

$$\frac{p_1 \Downarrow_{A_1} \mathsf{blame} \qquad p_2 \Downarrow_{A_2} r}{(p_1, p_2) \Downarrow_{A_1 \times A_2} \mathsf{blame}}\ \text{CAST-PROL} \qquad \frac{p_1 \Downarrow_{A_1} p_1' \qquad p_2 \Downarrow_{A_2} \mathsf{blame}}{(p_1, p_2) \Downarrow_{A_1 \times A_2} \mathsf{blame}}\ \text{CAST-PROR}$$

$\boxed{A \sqcap B = C}$ *(Type Meet)*

$$\frac{}{\mathsf{Int} \sqcap \mathsf{Int} = \mathsf{Int}}\ \text{MEET-INT} \qquad \frac{}{\star \sqcap A = A}\ \text{MEET-DYNL} \qquad \frac{}{A \sqcap \star = A}\ \text{MEET-DYNR} \qquad \frac{A_1 \sqcap A_2 = A_3 \qquad B_1 \sqcap B_2 = B_3}{A_1 \to B_1 \sqcap A_2 \to B_2 = A_3 \to B_3}\ \text{MEET-ARR}$$

$$\frac{A_1 \sqcap A_2 = A_3 \qquad B_1 \sqcap B_2 = B_3}{A_1 \times B_1 \sqcap A_2 \times B_2 = A_3 \times B_3}\ \text{MEET-PRO}$$

Fig. 16. Casting for the λe Calculus.

B. Type precision is the same as in λB^g_l. Other typing rules are exactly the same as those used by the λB^g calculus in Figure 3. Lemmas about dynamic types and a typing lemma about the inference mode include:

Lemma 5.1 (Dynamic Types of Raw Values)**.** *If* $\cdot \vdash p \Rightarrow A$ *then* $\rceil p \lceil = A$.

Lemma 5.2 (Dynamic Types of Values)**.** *If* $\cdot \vdash v \Rightarrow A$ *then* $\rceil v \lceil = A$.

Lemma 5.3 (Inference Uniqueness)**.** *If* $\Gamma \vdash e \Rightarrow A_1$ *and* $\Gamma \vdash e \Rightarrow A_2$ *then* $A_1 = A_2$.

5.3 Dynamic Semantics

As in the λB^g calculus, casting is used in the semantics to get a direct operational semantics.

Casting. The casting rules are shown in Figure 16. Compared to λB^g, the rules are simpler. The casting relation casts raw values instead of values. Rule CAST-C returns the constant c. Rule CAST-ABS shows that if the type of annotated lambdas $A_1 \to A_2$ is consistent with type C, then type B is replaced by C. Otherwise, if type $A_1 \to A_2$ is not consistent with type C, casting raises blame using rule CAST-ABSP. In rule CAST-PRO, when a product (p_1, p_2) is cast under product type $A_1 \times A_2$, we cast both components under A_1

and A_2, respectively and return a product of the resulting raw values. Rule CAST-PROL and rule CAST-PROR cover the case when the casts of p_1 and p_2 raise blame.

Casting Properties. Casting for the λe calculus has some interesting properties, similar to those shown in Section 3. In these Lemmas, the meet of two types ($\sqcap$) is defined at the bottom of Figure 16. The *meet* of two types is the greatest lower bound between the types in terms of precision. That is, it is the most imprecise type among the types equivalent to or more precise than the given types.

Lemma 5.4 (Preservation of Casting). *If* $\cdot \vdash p \Leftarrow A$, $\rceil p \lceil \sqcap B = A$ *and* $p \Downarrow_A p'$ *then* $\cdot \vdash (\!|p' : B|\!) \Rightarrow B$.

Lemma 5.5 (Progress of Casting). *If* $\cdot \vdash p \Leftarrow A'$ *and* $\rceil p \lceil \sqcap A = A'$ *then* $\exists r,\ p \Downarrow_{A'} r$.

Lemma 5.6 (Determinism of Casting). *If* $\cdot \vdash p \Leftarrow B$, $p \Downarrow_A r_1$ *and* $p \Downarrow_A r_2$ *then* $r_1 = r_2$.

Reduction. Figure 17 shows the reduction rules of the λe calculus. Rule STEP-BETA is the usual beta reduction rule. For rule STEP-APP, the argument e_2 is annotated with the input types, and the annotations with the output types are added in the final expression. As the type system shows, the unknown type $\star$ can be matched as a function type $\star \to \star$ or a product $\star \times \star$. Because the $\star$ type is consistent with any types, there can be run-time type-errors. For instance $(1 : \star)\ 1$ and $\pi_i\ (1 : \star)$ raise errors at runtime. Rule STEP-DYN and rule STEP-PROIP raise blame if the raw values cannot match with function types and product types, respectively. Rule STEP-PRO extracts type annotations from a pair of values and rule STEP-PROI projects the corresponding value. Rule STEP-ABS and rule STEP-I add an extra annotation with the dynamic type to produce a value. For values $(\!|p : A|\!)$ annotated with a type B, we perform consistency checking between the dynamic type of p and B. When consistency checking fails, blame is raised using rule STEP-ANNOP. If consistency checking succeeds, the raw value p is cast by the meet result of A' and B. If the casting succeeds, the result of casting annotated with type B is returned (rule STEP-ANNOV), otherwise blame is raised using rule STEP-ANNOVP. Note that these two rules and casting are the essence of our variant of the eager semantics: consistency checking happens directly, instead of waiting for the argument to be applied for high-order values. For additions ($+$ and $+_i$), if the argument is an integer then it can be used to do the operation (rule STEP-C) otherwise an error is raised (rule STEP-CF and rule STEP-CFA).

Example. Let us use an example to explain the behavior of casting with the eager semantics. Suppose that we take a chain of annotations $\lambda x.\, x : \star \to \mathsf{Int} : \mathsf{Int} \to \star : \star : \mathsf{Bool} \to \mathsf{Bool}$.

$\boxed{e \longmapsto r}$ *(Small-step semantics for the λe calculus)*

$$\text{STEP-EVAL}\quad \frac{e \longmapsto e'}{F[e] \longmapsto F[e']} \qquad \text{STEP-BLAME}\quad \frac{e \longmapsto \mathsf{blame}}{F[e] \longmapsto \mathsf{blame}} \qquad \text{STEP-BETA}\quad \frac{}{(\lambda x.\, e)\, v \longmapsto e[x \mapsto v]}$$

$$\text{STEP-APP}\quad \frac{C \rhd C_1 \to C_2}{(\llparenthesis((\lambda x.\, e : A_1 \to A_2) : B_1 \to B_2) : C\rrparenthesis)\, e_2 \longmapsto ((\lambda x.\, e)\, (e_2 : C_1 : B_1 : A_1)) : A_2 : B_2 : C_2}$$

$$\text{STEP-DYN}\quad \frac{\rceil p \lceil \nsim \star \to \star}{\llparenthesis p : A \rrparenthesis\, v_2 \longmapsto \mathsf{blame}} \qquad \text{STEP-PROIP}\quad \frac{\rceil p \lceil \nsim \star \times \star \qquad i \in \{1, 2\}}{\pi_i\, \llparenthesis p : A \rrparenthesis \longmapsto \mathsf{blame}}$$

$$\text{STEP-PRO}\quad \frac{}{(\llparenthesis p_1 : A \rrparenthesis, \llparenthesis p_2 : B \rrparenthesis) \longmapsto \llparenthesis (p_1, p_2) : A \times B \rrparenthesis} \qquad \text{STEP-PROI}\quad \frac{A \rhd A_1 \times A_2 \qquad i \in \{1, 2\}}{\pi_i\, \llparenthesis (p_1, p_2) : A \rrparenthesis \longmapsto \llparenthesis p_i : A_i \rrparenthesis}$$

$$\text{STEP-ABS}\quad \frac{A \rhd A_1 \to A_2}{(\lambda x.\, e) : A \longmapsto \llparenthesis (\lambda x.\, e : A_1 \to A_2) : A_1 \to A_2 : A \rrparenthesis} \qquad \text{STEP-I}\quad \frac{}{c \longmapsto \llparenthesis c : \rceil c \lceil \rrparenthesis}$$

$$\text{STEP-ANNOP}\quad \frac{\rceil p \lceil \nsim B}{\llparenthesis p : A \rrparenthesis : B \longmapsto \mathsf{blame}} \qquad \text{STEP-ANNOV}\quad \frac{\rceil p \lceil \sqcap B = A' \qquad p \Downarrow_{A'} p'}{\llparenthesis p : A \rrparenthesis : B \longmapsto \llparenthesis p' : B \rrparenthesis} \qquad \text{STEP-ANNOVP}\quad \frac{\rceil p \lceil \sqcap B = A' \qquad p \Downarrow_{A'} \mathsf{blame}}{\llparenthesis p : A \rrparenthesis : B \longmapsto \mathsf{blame}}$$

$$\text{STEP-C}\quad \frac{A \rhd A_1 \to A_2 \qquad \rceil c \lceil = B_1 \to B_2 \qquad \rceil p \lceil \sim B_1}{\llparenthesis c : A \rrparenthesis\, \llparenthesis p : B \rrparenthesis \longmapsto (\llbracket c \rrbracket (p)) : A_2} \qquad \text{STEP-CF}\quad \frac{\rceil c \lceil = B_1 \to B_2 \qquad \rceil p \lceil \nsim B_1}{\llparenthesis c : A \rrparenthesis\, \llparenthesis p : B \rrparenthesis \longmapsto \mathsf{blame}}$$

$$\text{STEP-CFA}\quad \frac{}{\llparenthesis c : A \rrparenthesis\, (\lambda x.\, e) \longmapsto \mathsf{blame}}$$

Fig. 17. Semantics of the λe Calculus.

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

$$\begin{aligned}
& \lambda x.\, x : \star \to \mathsf{Int} : \mathsf{Int} \to \star : \star : \mathsf{Bool} \to \mathsf{Bool} \\
& \longmapsto \{\text{by rule STEP-ABS}\} \\
& \llparenthesis \lambda x.\, x : \star \to \mathsf{Int} : \star \to \mathsf{Int} : \star \to \mathsf{Int} \rrparenthesis : \mathsf{Int} \to \star : \star : \mathsf{Bool} \to \mathsf{Bool} \\
& \longmapsto \{\text{by rule STEP-ANNOV and casting under } \mathsf{Int} \to \mathsf{Int}\} \\
& \llparenthesis \lambda x.\, x : \star \to \mathsf{Int} : \mathsf{Int} \to \mathsf{Int} : \mathsf{Int} \to \star \rrparenthesis : \star : \mathsf{Bool} \to \mathsf{Bool} \\
& \longmapsto \{\text{by rule STEP-ANNOV and casting under } \mathsf{Int} \to \mathsf{Int}\} \\
& \llparenthesis \lambda x.\, x : \star \to \mathsf{Int} : \mathsf{Int} \to \mathsf{Int} : \star \rrparenthesis : \mathsf{Bool} \to \mathsf{Bool} \\
& \longmapsto \{\text{by rule STEP-ANNOP}\} \\
& \mathsf{blame}
\end{aligned}$$

Firstly, $\lambda x.\, x : \star \to \mathsf{Int}$ reduces to a value $(\!|\lambda x.\, x : \star \to \mathsf{Int} : \star \to \mathsf{Int} : \star \to \mathsf{Int}|\!)$. Type $\star \to \mathsf{Int}$ is consistent with type $\mathsf{Int} \to \star$ and the meet of these two types is $\mathsf{Int} \to \mathsf{Int}$. So the intermediate type $\star \to \mathsf{Int}$ is replaced by $\mathsf{Int} \to \mathsf{Int}$ and the outer type is replaced by $\star \to \mathsf{Int}$. Similarly, $\mathsf{Int} \to \mathsf{Int}$ is consistent with type $\star$ and the meet of these two types is still $\mathsf{Int} \to \mathsf{Int}$. So only the outer type is replaced by type $\star$. Finally, type $\mathsf{Int} \to \mathsf{Int}$ is not consistent with $\mathsf{Bool} \to \mathsf{Bool}$ so blame is raised.

One important property is that the reduction relation is deterministic:

Theorem 5.1 (Determinism of the λe Calculus)**.** *If* $\cdot \vdash e \Leftrightarrow A$, $e \longmapsto r_1$ *and* $e \longmapsto r_2$ *then* $r_1 = r_2$.

Type Soundess. Another important property is that the λe calculus is type sound. Theorems 5.2 and 5.3 for type preservation and progress, respectively, have the same form as in λB^g.

Theorem 5.2 (Type Preservation of the λe Calculus)**.** *If* $\cdot \vdash e \Leftrightarrow A$ *and* $e \longmapsto e'$ *then* $\cdot \vdash e' \Leftrightarrow A$.

Theorem 5.3 (Progress of the λe Calculus)**.** *If* $\cdot \vdash e \Rightarrow A$ *then* e *is a value or* $\exists r$, $e \longmapsto r$.

5.4 Gradual Guarantee

Precision. Figure 18 shows the precision relation of expressions for λe. Compared to Figure 12, lambdas are not annotated with types and rules for pairs and projections are new. For pairs (e_1, e_2) and projections $(\pi_i\ e)$, precision just employs simple structural rules. 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 two values (rule EPRE-VAL), precision holds if the precision of raw values and types hold. To make sure the less precise values $((\!|p_2 : B|\!))$ are well-typed, the type of raw values should be more precise than annotated types. Note that there is an implicit assumption that the more precise values are well-typed, which ensures that the pre-value is more precise than the type annotation. Thus no such explicit assumption is needed in rule EPRE-VAL for p_1.

One noteworthy point is that the precision relation for expressions in λe is significantly simpler than the precision for λB^g. There are a few reasons that contribute to the simpler precision relation:

1. **The form of values is similar for both dynamic and static values.** Values in λe have a more consistent form for dynamic and statically typed values. For example, in λB^g, static integers are represented as i, but dynamic integers are represented as $i : \star$. In λe both static and dynamic integers require an annotation ($(\!|i : \mathsf{Int}|\!)$ and $(\!|i : \star|\!)$).
2. **The eager semantics avoids accumulation of annotations.** In the eager semantics, we can avoid accumulating annotations for higher-order values. This means that higher-order values always have exactly 3 annotations. Therefore, in combination with the previous point, we can define precision of expressions, while avoiding alignment rules such as those in λB^g.

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

$$\frac{}{c \sqsubseteq c}\ \text{EPRE-C} \qquad \frac{}{x \sqsubseteq x}\ \text{EPRE-X} \qquad \frac{e_1 \sqsubseteq e_2}{\lambda x.\, e_1 \sqsubseteq \lambda x.\, e_2}\ \text{EPRE-ABS} \qquad \frac{e_1 \sqsubseteq e_1' \qquad e_2 \sqsubseteq e_2'}{(e_1\, e_2) \sqsubseteq (e_1'\, e_2')}\ \text{EPRE-APP} \qquad \frac{A \sqsubseteq B \qquad e_1 \sqsubseteq e_2}{e_1 : A \sqsubseteq e_2 : B}\ \text{EPRE-ANNO}$$

$$\frac{e_1 \sqsubseteq e_1' \qquad e_2 \sqsubseteq e_2'}{(e_1, e_2) \sqsubseteq (e_1', e_2')}\ \text{EPRE-PRO} \qquad \frac{e_1 \sqsubseteq e_2 \qquad i \in \{1,2\}}{\pi_i\, e_1 \sqsubseteq \pi_i\, e_2}\ \text{EPRE-PROI} \qquad \frac{A \sqsubseteq B \qquad p_1 \sqsubseteq p_2 \qquad \rceil p_2 \lceil \sqsubseteq B}{(\!|p_1 : A|\!) \sqsubseteq (\!|p_2 : B|\!)}\ \text{EPRE-VAL}$$

Fig. 18. Precision relations.

3. **No need for typing premises and typing contexts.** In addition, there are no typing premises. Therefore typing contexts can also be avoided. The need for inference typing premises in the precision relation of λB_l^g follows the blame calculus, which employs similar premises in its precision relation. However, in order to support inference typing premises, lambda expressions need to be annotated. Otherwise raw lambdas cannot be inferred in the general case, and the precision relation would not be able to deal with raw lambdas. In λe, there is no such issue because the precision relation does not include any typing premises. Therefore, raw lambdas can be easily supported in λe. We conjecture that it is also possible to support raw lambdas in λB_l^g, but this would require changing the current setup and proof for the dynamic gradual guarantee, which is based on that of the blame calculus. We leave this exploration for future work.
4. **No need for introducing strict applications.** As discussed in Section 4, the proof of the dynamic gradual guarantee for λB_l^g requires a second set of reduction rules and the introduction of strict applications. This introduces a significant burden in terms of definitions and proofs compared to λe. The need for strict applications for the dynamic gradual guarantee of λB_l^g is discussed in more detail in Section 6.3.

Static Criteria. The λe calculus can also encode the dynamic counterpart of expressions ($\check{e}$) by the translation function ($\lceil \check{e} \rceil_b$) (Theorem 5.4). However, unlike in λB_l^g, where we need to insert $\star \to \star$ every time for raw lambdas, in λe, we can do a simple optimization by exploiting bidirectional type-checking. In essence, when raw lambdas are in checking positions, the types can be inferred from the contextual type information. The translation function ($\lceil \check{e} \rceil_b$) is shown as follows and the boolean flag b indicates an unknown type annotation should be inserted or not:

$$\begin{aligned}
\lceil x \rceil_b &= x \\
\lceil c \rceil_{\mathsf{true}} &= c : \star \\
\lceil c \rceil_{\mathsf{false}} &= c \\
\lceil \lambda x.\check{e} \rceil_{\mathsf{true}} &= \lambda x.\lceil \check{e} \rceil_{\mathsf{false}} : \star \\
\lceil \lambda x.\check{e} \rceil_{\mathsf{false}} &= \lambda x.\lceil \check{e} \rceil_{\mathsf{false}} \\
\lceil \check{e}_1\ \check{e}_2 \rceil_b &= \lceil \check{e}_1 \rceil_{\mathsf{true}}\ \lceil \check{e}_2 \rceil_{\mathsf{false}}
\end{aligned}$$

For example, the dynamic expression $(\lambda x.\,\lambda y.\,x\,y)\,(\lambda x.\,x)\,1$ can be translated to $((\lambda x.\,\lambda y.\,x\,y) : \star)\,(\lambda x.\,x)\,1$. Only one unknown type $\star$ is inserted, despite the presence of 3 lambda expressions without annotations in the original expression. We should also remark that, more generally, we can use a similar idea to improve on the syntactic sugar for dynamic lambdas. Instead of blindly adding $\star \to \star$ annotations to lambdas without type annotations, we only need to add $\star \to \star$ to lambdas in inference positions.

Theorem 5.5 shows that the static gradual guarantee holds for the λe calculus. It says that if e of type A is more precise than e', then e' has some type B, and type A is more precise than B.

Theorem 5.4 (Dynamic Embedding). *If $\check{e}$ is closed then $\cdot \vdash \lceil \check{e} \rceil_{\mathsf{true}} \Rightarrow \star$.*

Theorem 5.5 (Static Gradual Guarantee of the λe Calculus). *If $e \sqsubseteq e'$ and $\cdot \vdash e \Leftrightarrow A$ then $\exists B, \cdot \vdash e' \Leftrightarrow B$ and $A \sqsubseteq B$.*

Dynamic Gradual Guarantee. The λe calculus has a dynamic gradual guarantee. Theorem 5.6 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 to e'_2. Note that e'_1 is guaranteed to be more precise than e'_2. Theorem 5.6 is similar to the one formalized in the AGT approach (Garcia *et al.*, 2016). Theorem 5.7 is derived easily from Theorem 5.6. The auxiliary Lemma 5.7, which shows the property of dynamic gradual guarantee for casting, is helpful to prove Theorem 5.7.

Lemma 5.7 (Dynamic Gradual Guarantee for Casting). *If $p_1 \sqsubseteq p_2$, $\cdot \vdash p_1 \Leftarrow A_0$, $\cdot \vdash p_2 \Leftarrow B_0$, $A_2 \sqsubseteq B_2$, $\rceil p_1 \lceil \sqcap A_2 = A_1$, $\rceil p_2 \lceil \sqcap B_2 = B_1$ and $p_1 \Downarrow_{A_1} p'_1$ then $\exists p'_2$, $p_2 \Downarrow_{B_1} p'_2$ and $(\!| p'_1 : A_2 |\!) \sqsubseteq (\!| p'_2 : B_2 |\!)$.*

Theorem 5.6 (Dynamic Gradual Guarantee (single-step)). *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 5.7 (Dynamic Gradual Guarantee). *Suppose $e_1 \sqsubseteq e_2$, $\cdot \vdash e_1 \Leftrightarrow A$ and $\cdot \vdash e_2 \Leftrightarrow B$.*

- *If $e_1 \longmapsto^* v_1$ then $\exists v_2$, $e_2 \longmapsto^* v_2$ and $v_1 \sqsubseteq v_2$.*
- *If $e_1 \Uparrow$ then $e_2 \Uparrow$.*
- *If $e_2 \longmapsto^* v_2$ then $e_1 \longmapsto^* v_1$ and $v_1 \sqsubseteq v_2$, or $e_1 \longmapsto^*$ blame.*
- *If $e_2 \Uparrow$ then $e_1 \Uparrow$ or $e_1 \longmapsto^*$ blame.*

6 Discussion

In this section, we wish to discuss our results and further compare the TDOS based approach with traditional cast calculi. In particular, we wish to go through some possible criticisms of the TDOS approach and to analyse the results obtained in this work.

To conduct this discussion, we believe that it is useful, at points, to draw an analogy with work on the semantics of object-oriented languages. During the 80s and the 90s, there were at least *two* schools of thought working on the semantics of OOP languages. One school of thought was on using lambda calculi, such as $F_{<:}$ (Cardelli *et al.*, 1994; Curien & Ghelli, 1992; Pierce, 1994), to give the semantics of OOP languages via an elaboration (Bruce *et al.*, 1999; Abadi *et al.*, 1996). Another school of thought was to give the semantics of OOP languages directly (Abadi & Cardelli, 1996; Igarashi *et al.*, 2001). The work on

Featherweight Java (FJ) (Igarashi *et al.*, 2001) is a highlight of the direct approach. Both lines of work have been highly influential and led to much innovation. We believe that the TDOS can bring some new ideas into the landscape of gradual typing, complementing already established ideas studied with the elaboration approach.

6.1 Comparing TDOS based Calculi with Traditional Cast Calculi

The semantics of applications. Perhaps the most notable difference between the TDOS and traditional cast calculi, such as the blame calculus, is on the semantics of applications. In the blame calculus, applications are standard and *strict*. Here strict means that applications accept only arguments that *exactly* match the type of the input of the function. Explicit casts, together with strict applications, are then used in the blame calculus to encode *flexible* applications, where the arguments can have different (but consistent) types and casts bridge the gap between the types.

In a TDOS, we must support flexible applications, because calculi with a TDOS model a gradual typing semantics directly. Since gradual source languages support flexible applications, the semantics of applications in a TDOS needs to be necessarily more complex than that in a cast calculus. This difference can be seen in our work. For instance, in the semantics of λB in Figure 1, there are three rules dedicated to the semantics of applications (rules BSTEP-BETA, BSTEP-ABETA, and BSTEP-C). In contrast, for λB^g (Figure 5), there are five rules for applications (rules STEP-BETAP, STEP-BETA, STEP-ABETA, STEP-DYN, and STEP-C). In a TDOS, applications may raise blame due to inconsistent arguments. Thus rule STEP-BETAP is needed in λB^g. In the blame calculus, applications themselves cannot raise blame, since they are strict. So rule STEP-BETAP or similar rules are unnecessary. Furthermore, in λB^g, a function with type $\star$ can be directly applied to an argument, since $\star \sim \star \to \star$. Thus, we also need rule STEP-DYN in λB^g. In the blame calculus, function applications require a function to always have a function type. So rule STEP-DYN or similar is also not needed in λB. In summary, in a TDOS, because applications are more flexible, they can assume less about the functions and the arguments. As a result, to ensure that applications are safe, casting must be used to check that the functions and arguments have adequate value forms at runtime.

There are some important consequences of having flexible applications that are worth noting and are discussed next.

Non-orthogonality of the semantics. A first point is that, from a language design point of view, one may question the non-orthogonality of the semantics of applications in a TDOS. In the blame calculus, applications are strict and no casting is involved. All casting that happens is delegated into a separate cast construct. In a TDOS, applications are not orthogonal to casting, since applications also perform some casting. The non-orthogonality is a direct consequence of the goal to have a direct gradual typing semantics. So, while a TDOS can eliminate the need of a source language, and an elaboration to a cast calculus, a price to pay is some additional complexity on the semantics for applications.

To build on our analogy with the semantics of OOP languages, this is similar to the semantics of method calls versus the semantics of applications in lambda calculi such as $F_{<:}$. Method calls in OOP languages have a relatively complex semantics due to *dynamic*

dispatching, where the implementation of the method to be executed may only be determined at runtime. Thus, method calls are analogous to flexible applications and require encodings that involve several constructs, including applications, in conventional lambda calculi. In general, the non-orthogonality of the semantics is one of the trade-offs that we need to make if we want to directly model a semantics similar to the source language. Often, in source languages, constructs are made more flexible to provide programming convenience.

Non-standard beta-reduction. A more technical point is that beta-reduction in λB^g is non-standard. It is well-established how to efficiently implement standard beta-reduction. But it is unclear whether the form of beta reduction in λB^g can be efficiently implemented. However, we have shown that, for all calculi presented in this paper, it is possible to recover conventional beta-reduction. For λe, we already employ a standard beta-reduction rule. For λB^g and λB^g_l, we have seen that it is possible to have an alternative design for the reduction rules (Figure 9). In this alternative design, we introduce a form of strict applications in addition to flexible applications. Then, flexible applications will reduce into strict applications, and the beta-reduction of strict applications is standard. The cost of the alternative design is one extra form of application.

Flexible applications are costly. Another important concern is that flexible applications entail extra runtime costs. Casting must be used in applications to validate the types of arguments and functions being applied. In the elaboration approach, it is possible to detect strict applications, and avoid casts for those applications when doing *cast insertion* (Siek & Taha, 2006). So, while with the elaboration approach, many applications will be strict and avoid casts, in the TDOS considered in this paper, all applications will require casts. One downside of switching to a TDOS is that it becomes less clear where the dividing line is between static checking and dynamic checking. Therefore, an important question is whether it is possible to optimize TDOS to obtain similar benefits to those of cast insertion. We will discuss this question in more detail next.

6.2 Optimizing a TDOS

We are advocating the TDOS primarily as a more direct way to provide the semantics of gradually typed languages. In doing so, we forfeit some optimization benefits that are well-established for the elaboration approach. A question that then arises is whether or not we can recover the same benefits, in terms of optimizations, that we obtain in an elaboration approach. While it is not the goal of this paper to investigate how to optimize the TDOS, here we briefly discuss some possible directions and wish to argue that it is possible to have more efficient TDOS-based implementations.

Simplicity of the semantics versus guiding efficient implementations. A first point that we wish to stress is that the main goal of the TDOS is to express the semantics and metatheory of a gradual language as directly and simply as possible. However, this goal can be in conflict with using the semantics to guide efficient implementations. We believe that this conflict is not specific to the TDOS, but arises in many other formulations of operational semantics. For instance, going back to our OOP analogy, we believe that the semantics of FJ was designed with the primary goal of simplicity in mind. The work on FJ does

not provide a direct answer for questions such as how to optimize dynamically dispatched method calls. Although, of course, this is an important concern in the implementation of OOP languages. At the same time, we believe that there is nothing preventing variants of FJ that enable answering such questions directly. For instance, it should be possible to have variants of FJ with multiple kinds of method calls (including dynamic and static), and some pre-processing analysis that detects dynamic calls that can be transformed into static calls. Of course, such an extension to FJ would require more constructs and rules, and thus would have some extra complexity compared to the original calculus. Similarly to FJ, our formulations of the TDOS in this paper express the semantics of all applications with flexible applications, even if many applications can be made strict to achieve better performance.

A point worth mentioning here is that in the elaboration approach for gradual languages, the elaboration itself is, of course, *compulsory* to give the semantics of the source language. Since the elaboration is compulsory anyway, it makes sense to exploit it and couple some optimizations with this step. In essence, this enables us to exploit static knowledge about the program to generate optimized code in cast calculi. A TDOS *does not require* an elaboration step. However, we could have an *optional* step type-checking a TDOS program and attempting to use static type information to generate more efficient code. In such a step, we could do similar optimizations to those performed in the traditional elaboration of gradual languages. For these optimizations to be possible it may be necessary to extend the TDOS with more constructs. We briefly sketch a couple of possible optimizations next.

Optimizing Strict Applications. With the alternative set of reduction rules for λB^g_l in Figure 9, we can avoid casting for arguments if we know that an application is strict. Thus, assuming an optimization pass on a program, before compilation, we could try to optimize some applications. For example, suppose that we would have the application:

$$(\lambda x.\, x + 1 : \mathsf{Int} \to \star)\ 5$$

With the current set of reduction rules, using flexible applications, we would need to cast the integer argument to an integer. However, we statically know that such a cast is unnecessary. With a pre-optimization pass, we could detect this, and then transform the flexible application above into a strict application:

$$\color{red}{(\lambda x.\, x + 1 : \mathsf{Int} \to \star)\ 5}$$

Then the cast of the argument would be avoided at runtime.

Clearly the alternative syntax and set of rules in Figure 9 does not go far enough into what we could do. For instance, in the example above, although we can avoid the cast of the argument, we still need to cast the result of beta-reduction with the return type of the function. A possible idea here would be to make lambda expressions strict, by ensuring that the return type of the function matches *exactly* with the type of the body of the function. With such a form of strict lambdas, the expression above could not type check, since $x + 1$ (the expression in the body) should have type Int, but the return type of the function is $\star$. Strict lambdas do not lose expressiveness compared to the current, more flexible, form of lambdas. This is because we can always insert a cast in the body to encode a flexible

lambda. For instance, we could have $(\lambda x.\, (x + 1 : \star) : \mathsf{Int} \to \star)$, and now the body would have type $\star$.

We have not fully investigated the optimizations discussed above. For optimizing strict applications, we believe that we could employ techniques similar to existing cast insertion techniques (Siek & Taha, 2006). For the possibility of replacing flexible lambdas by strict lambdas, we have not formalized the idea yet and we may need to pay some attention to the gradual guarantee.

Design Choices in the TDOS. Formulations of the TDOS are relatively new and under-explored. Thus, there could be different design choices with potential advantages over the current design choices. One design choice that we have made is to have lambda values with both input and output annotations. This is in contrast to the blame calculus, where lambda abstractions only include the input type annotation: that is they are of the form $\lambda x : A.\, e$. One alternative TDOS design that was explored by Fan *et al.* (2022) for calculi with the merge operator, models lambda expressions as $\lambda x : A.\, e$. However, function values need to be wrapped under an annotation. That is they need to be of the form $\lambda x : A.\, e : C$ (where C is the type of the whole function). In a TDOS, output types are needed because casts require the *source* type of the cast. In the blame calculus source types are written explicitly. For example, consider the TDOS application:

$$(\lambda x.\, x : \star \to \star)\; 5$$

which casts the argument to a dynamic value. In the blame calculus, one possible cast that we could insert to enable the program above would be:

$$(\lambda x : \star.\, x : \star \to \star \Rightarrow \mathsf{Int} \to \star)\; 5$$

Here we cast the function from the source type $\star \to \star$ to the target type $\mathsf{Int} \to \star$. In the blame calculus, both the source and target types of a cast have to be explicitly written. In a TDOS, we require functions to have output types to ensure that the source type of a cast can be easily computed, and sometimes, the target type can be left implicit. One could wonder if we could compute an imprecise type for $\lambda x : A.\, e$ and use that form of lambdas. That is, we could always return $\star$ for the output of function $\rceil \lambda x : A.\, e \lceil = A \to \star$. However, the missing output type information would bring problems in a TDOS. Suppose that we have the following program:

$$(\lambda x : \star.\, 1) : \star : \mathsf{Bool} \to \mathsf{Bool}$$

The dynamic type of the lambda expression $\lambda x : \star.\, 1$ would be $\star \to \star$, which is a ground type. Thus, the program would reduce to an ill-typed program:

$$(\lambda x : \star.\, 1) : \mathsf{Bool} \to \mathsf{Bool}$$

With our current formulation, the lambda expression takes the form $\lambda x.\, 1 : \star \to \mathsf{Int}$. Then the dynamic type would be the more precise type $\star \to \mathsf{Int}$, which is not a ground type. Thus the problem with $(\lambda x : \star.\, 1) : \star : \mathsf{Bool} \to \mathsf{Bool}$ would not happen.

6.3 Metatheory and the Gradual Guarantee

One of our arguments for the TDOS is that it can enable a more direct approach to the semantics of gradual languages, as well as potentially simpler metatheory. For most proofs in the paper, we believe that the proofs are quite simple. This includes proofs of preservation, progress, determinism and the blame theorem. One proof that deserves more discussion is the (dynamic) gradual guarantee, which we talk about next.

Gradual Guarantee Proofs. Using an elaboration approach to provide the semantics of a source gradual language usually requires two separate precision relations for the source language and the cast calculus. Furthermore, because of the elaboration, properties such as elaboration type preservation are necessary. In contrast, the λe calculus leads to a simple proof of the gradual guarantee. Only one set of precision rules is required and no elaboration theorems are needed. For λe, we believe that the gradual guarantee is simpler than with an elaboration approach.

For λB^g_l, however, we faced some issues for the dynamic gradual guarantee. Due to the generalization of the application typing rule, there are some complications in the gradual guarantee for λB^g_l. In particular, λB^g_l needs to introduce an alternative set of reduction rules with strict applications ($e_1\ e_2$). So the precision for the dynamic gradual guarantee should account for strict applications, whereas no strict applications are needed for the static gradual guarantee.

It is worthwhile discussing why strict applications are needed. From the precision relation, we know that the following two programs are in a precision relation:

$$(((\lambda x.\, x :^{l_1} \mathsf{Int} \to \mathsf{Int}) :^{l_2} \star \to \star)\ 1)^{l_3} \sqsubseteq ((\lambda x.\, x :^{l_1} \star \to \star)\ 1)^{l_3}$$

To prove the dynamic gradual guarantee, we need to satisfy Theorem 4.7, which shows that if the more precise expression reduces to a new expression, then the less precise expression, after multiple steps of reduction, must preserve the precision relation. If expression $(((\lambda x.\, x :^{l_1} \mathsf{Int} \to \mathsf{Int}) :^{l_2} \star \to \star)\ 1)^{l_3}$ takes a reduction step:

$$(((\lambda x.\, x :^{l_1} \mathsf{Int} \to \mathsf{Int}) :^{l_2} \star \to \star)\ 1)^{l_3} \longmapsto (((\lambda x.\, x :^{l_1} \mathsf{Int} \to \mathsf{Int})\ (1 :^{l_3} \star))^{\bar{l}_2}) :^{l_2} \star$$

Then $(\lambda x.\, x :^{l_1} \star \to \star)\ 1$ should be in precision with $(((\lambda x.\, x :^{l_1} \mathsf{Int} \to \mathsf{Int})\ (1 :^{l_3} \star))^{\bar{l}_2}) :^{l_2} \star$ after multiple reduction steps. However, if $((\lambda x.\, x :^{l_1} \star \to \star)\ 1)^{l_3}$ takes a step then beta-reduction is performed, and the result cannot preserve precision. So, the only option is not to reduce. However, the argument 1 is not less precise than $(1 :^{l_3} \star)$. The problem is that the argument type is not the same as the function input type. This is due to the generalization of the typing of applications, where there is an implicit cast for the arguments.

To address the issue above, we introduce strict applications to perform the implicit cast earlier. Then, for a strict application, the type of the argument is the same as the function input type. For $(\lambda x.\, x :^{l_1} \star \to \star)\ 1$, there is a cast from 1 to function input type $\star$. After we cast 1 to $\star$, the result is $(1 :^{l_1} \star)$, which is less precise than $(1 :^{l_2} \star)$ as expected. Therefore, we use strict applications to label the applications which have performed the implicit cast for arguments, and have the same type as function inputs. Another potential solution would be to use a semantic proof method based on logical relations for proving the dynamic gradual guarantee (New *et al.*, 2019). With this approach, we believe that the intermediate reduction system with strict applications would not be needed. The key observation

is that the final reduction result should be in a precision relation (even if intermediate results may not be in a precision relation). However, a semantic approach introduces its own complexity, so we have not explored this possibility.

In short, for the proof of the dynamic gradual guarantee in λB^g_l, while we can avoid some proofs and definitions required by elaboration, we faced other complications. Thus we cannot claim that the proof of the dynamic gradual guarantee for λB^g_l is simpler than the proof for the blame calculus.

7 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 and Criteria. There is a growing number of research work focusing on combining static and dynamic typing (Rastogi *et al.*, 2012; Strickland *et al.*, 2012; Matthews & Findler, 2009; Abadi *et al.*, 1991; Meijer & Drayton, 2004; Boyland, 2014; Wolff *et al.*, 2011; Swamy *et al.*, 2014; Hansen, 2007; Thatte, 1989). Many mainstream programming languages have some form of integration between static and dynamic typing. These include TypeScript (Bierman *et al.*, 2014), Dart (Bracha, 2015), Hack (Verlaguet, 2013), Cecil (Chambers, 1992), Bigloo (Serrano & Weis, 1995), Visual Basic.NET (Meijer & Drayton, 2004), ProfessorJ (Gray *et al.*, 2005), Lisp (Moon, 1989), Dylan (Shalit, 1996) and Typed Racket (Tobin-Hochstadt & Felleisen, 2008).

A gradually typed lambda calculus (GTLC) should support both fully statically typed and fully dynamically typed programs, as well as partially statically typed or dynamically typed ones. Siek & Taha (2006) introduced gradual typing with the notion of the unknown type $\star$ and type consistency. Because run-time checking is needed by a gradually typed language, function type annotations are accumulated at run-time in most of the gradually typed languages. Therefore, the number of accumulated type annotations can be unbound. Herman *et al.* (2007) 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, with many works on the area (Siek *et al.*, 2009; Siek & Wadler, 2009; Garcia, 2013; Herman *et al.*, 2010).

Much work in the research literature of gradual typing focuses on the pursuit of sound gradual typing. A language with sound gradual typing should come with a guarantee of type soundness. 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 (Siek & Taha, 2006). Siek *et al.* (2015*b*) 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. The semantics of a gradually typed calculus is normally given indirectly via a translation (or elaboration) into a cast calculus. The process of the translation to cast calculi involves inserting casts whenever type consistency holds between different types. Cast calculi are independent from the GTLC, having their own type systems and operational semantics. In λB^g and λe, by using TDOS, the semantics of a GTLC is given directly without translating to any other calculus.

There are several varieties of cast calculi. Findler & Felleisen (2002) introduced assertion-based contracts for higher-order functions. Based on the coercions and checks for higher-order values, which are implemented by an ad-hoc mixture of wrappers, reflection, and dynamic predicates, Gray *et al.* (2005) provided the following observation. First, the wrapper and reflection operations fit the profile of mirrors. Second, the checks correspond to contracts. Finally, the timing and shape of mirror operations coincide with the timing and shape of contract operations. Consequently, they presented a new model of interoperability that builds on the ideas of mirrors and contracts. Henglein's dynamically typed λ-calculus (Henglein, 1994) is an extension of the statically typed λ-calculus with a dynamic type and explicit dynamic type coercions. To port portions of programs from scripting languages to sound typed languages, Tobin-Hochstadt & Felleisen (2006) presented a framework for expressing this interlanguage migration. They proved that, for a program which consists of modules in the untyped lambda calculus, rewriting one of them in a simply typed lambda calculus can produce an equivalent program and be type safe.

Wadler & Findler (2009) 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.* (2009) 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 split when the wrapped function is applied to an argument. Wrappers for higher-order casts can lead to unbounded space consumption (Herman *et al.*, 2010).

There are some different designs for the dynamic semantics for cast calculi in the literature. Wadler & Findler (2009) use a lazy error detection strategy, which coerces the arguments of a function to the target type, and checking is only done when the argument is applied. Siek & Taha (2006) use a different strategy where checking higher-order casts is performed immediately when the source type is the unknown type ($\star$). Otherwise, the later strategy is the same as lazy error detection.

Siek & Wadler (2009) 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 a coercion-based calculus (Herman *et al.*, 2007) 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. In λe the three annotations in lambda values play a similar role to the three annotations in threesomes. Castagna & Lanvin (2017) proposed a calculus that discards annotations for higher-order functions, following an eager semantics. The dynamic semantics for higher-order values above can be summarized as two categories. One is the

lazy semantics, and the other is the eager semantics which attempts to merge intermediate annotations for higher-order casts. In our work, we study calculi with both variants.

Finally, various cast calculi have been extended with various of features of practical interest. For instance, Ahmed *et al.* (2011) extended the blame calculus to incorporate polymorphism, based on the dynamic sealing proposed by Matthews & Ahmed (2008) and Neis *et al.* (2009).

Cast calculi require explicit casts and cannot be used directly as a gradually typed language. They can be used as the target of source gradual languages via an elaboration that inserts casts where needed. In contrast, the calculi presented in this work support both explicit and implicit casts. Explicit casts are supported via type annotations, which can act as casts at runtime. Implicit casts arise from the use of bidirectional type-checking, where the checking mode denotes points in the program where casts are needed. Because implicit casts are supported in all our calculi (λB^g, λB_l^g and λe), these calculi can all be used directly as gradually typed languages and no elaboration step is necessary.

In the conference version of this work, we introduced the λB^r calculus (Ye *et al.*, 2021), which uses a forgetful semantics (Greenberg, 2015) called the blame recovery semantics. The blame recovery semantics ignores 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). In the present work, we opted to present an eager semantics, inspired by the AGT approach, instead. However, the eager semantics in λe and traditional AGT (Garcia *et al.*, 2016), is only applied to values. In other work (Herman *et al.*, 2010; Siek & Wadler, 2009; Bañados Schwerter *et al.*, 2021), the eager semantics is applied to all expressions and leads to a space-efficiency GTLC. Furthermore, both AGT and our eager semantics is more eager than the original approach by Herman *et al.* (2010), since, if two types are inconsistent, an error is raised directly. However in space efficient calculi, error is recorded and raised when arguments are applied. We leave a space-efficient formulation of the eager semantics using TDOS as future work.

Abstracting Gradual Typing (AGT). Garcia *et al.* (2016) introduce the abstracting gradual typing (AGT) approach, following an idea by Bañados Schwerter *et al.* (2014). An externally justified cast calculus is not required in AGT. Instead, 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 retains the type safety of the static type system, and enjoys the criteria of Siek *et al.* (2015*b*). The dynamic semantics is given by reasoning about the type derivations obtained from the type safety proof of the static language counterpart. Gradual typing derivations are represented as intrinsically typed terms (Church, 1940), which correspond to typing derivations directly.

One aspect that the TDOS has in common with the AGT approach is that by using the TDOS for the dynamic semantics and a bidirectional type system, we can design a gradually typed language with a direct semantics. Nevertheless, 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 typed 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. It would be interesting to reuse much of the AGT infrastructure, but with an alternative model for the dynamic semantics based on a TDOS instead. One current limitation of the AGT approach is that it does not offer an account of blame tracking and blame labels. As we have seen in Section 4, the TDOS can model languages with blame tracking and blame labels and be used to prove important theorems such as blame safety.

Typed Operational Semantics. In this paper, we use the type-directed operational semantics (TDOS) approach (Huang & Oliveira, 2020). 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 a TDOS, type annotations become operationally relevant and can affect the result of a program. Casting is used to provide an operational interpretation to type conversions in the language, similarly to coercions in coercion-based calculi (Henglein, 1994). Our work shows that a TDOS enables a direct semantics for gradual typing. We explored two possible semantics for gradual typing: one following a semantics similar to the blame calculus, and another with an eager semantics.

There are other variants of operational semantics that make use of type annotations. Types are used in Goguen's typed operational semantics (Goguen, 1994), similarly to TDOS. Typed operational semantics has been applied to various calculi, including simply typed lambda calculi (Goguen, 1995) and calculi with dependent types (Feng & Luo, 2011) and higher-order subtyping (Compagnoni & Goguen, 2003). An extensive overview of related work on type-dependent semantics is given by Huang & Oliveira (2020).

8 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 a TDOS (Huang & Oliveira, 2020). TDOS is a variant of small-step semantics where type annotations are operationally relevant and a special big-step casting relation 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 languages.

We presented two gradually typed lambda calculi: λB^g and λe. The λB^g semantics follows the semantics of λB. The λe calculus explores the use of an eager semantics for gradually typed languages using a TDOS. In addition, the λB^g_l calculus shows that blame

labels can be modeled using a TDOS, and results such as the blame theorem and the gradual guarantee can also be proved.

There is much to be done for future work. Although we argued that some benefits of the TDOS include easier and more direct proofs, we have not empirically validated those claims. To do so we would need to formalize the GTLC and a target cast calculus (such as the blame calculus or a cast calculus with an eager semantics) in Coq together with all the relevant proofs (such as the blame theorem the gradual guarantee and others). Then we could empirically compare the proofs, for instance in terms of size or number of lemmas that are required and/or other metrics. In our work, we have only informally mentioned that some lemmas and definitions would not be needed in a TDOS, but perhaps it could be the case that a TDOS would also require some extra lemmas or extra complexity in the proofs that is not necessary in the conventional approach using an elaboration semantics. One issue that we have identified in the proofs with the TDOS is the dynamic gradual guarantee proof for λB_l^g, which we discussed in Section 6. An empirical evaluation would help assessing such benefits (or not) more precisely, and it is one direction 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 the TDOS. Cast calculi have been shown to support a wide range of features such as polymorphism (Ahmed *et al.*, 2011), subtyping (Siek & Taha, 2007) and various other features (Siek & Vachharajani, 2008; Takikawa *et al.*, 2012). We hope to explore these in the future.

Acknowledgements: We are grateful to anonymous reviewers and our colleagues at the HKU PL group. This work has been sponsored by Hong Kong Research Grants Council projects number 17209520 and 17209821.

Conflicts of Interest: None.

References

Abadi, M., & Cardelli, L. (1996). *A theory of objects*. Springer-Verlag.

Abadi, Martín, Cardelli, Luca, Pierce, Benjamin, & Plotkin, Gordon. (1991). Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **13**(2), 237–268.

Abadi, Martín, Cardelli, Luca, & Viswanathan, Ramesh. (1996). An interpretation of objects and object types. *Pages 396–409 of: Proceedings of the 23rd acm sigplan-sigact symposium on principles of programming languages*.

Ahmed, Amal, Findler, Robert Bruce, Siek, Jeremy G, & Wadler, Philip. (2011). Blame for all. *Pages 201–214 of: Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Bañados Schwerter, Felipe, Clark, Alison M., Jafery, Khurram A., & Garcia, Ronald. (2021). Abstracting gradual typing moving forward: precise and space-efficient. *Proc. acm program. lang.*, **5**(POPL).

Bañados Schwerter, Felipe, Garcia, Ronald, & Tanter, Éric. (2014). A theory of gradual effect systems. *Pages 283–295 of: Proceedings of the 19th ACM SIGPLAN International Conference on Functional programming*.

Bettini, Lorenzo, Bono, Viviana, Dezani-Ciancaglini, Mariangiola, Giannini, Paola, & Venneri, Betti. (2018). Java & Lambda: a Featherweight Story. *Logical Methods in Computer Science*, **14**(3).

Bierman, Gavin, Abadi, Martín, & Torgersen, Mads. (2014). Understanding TypeScript. *Pages 257–281 of: European Conference on Object-Oriented Programming*. Springer.

Bottu, Gert-Jan, Xie, Ningning, Marntirosian, Koar, & Schrijvers, Tom. (2019). Coherence of Type Class Resolution. *Proc. acm program. lang.*, **3**(ICFP).

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

Bracha, Gilad. (2015). The Dart Programming Language.

Bruce, Kim B., Cardelli, Luca, & Pierce, Benjamin C. (1999). Comparing object encodings. vol. 155.

Cardelli, Luca, Martini, Simone, Mitchell, John C., & Scedrov, Andre. (1994). An extension of system f with subtyping. *Information and computation*, **109**(1-2), 4–56.

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

Chambers, Craig. (1992). Object-Oriented Multi-Methods in Cecil. *European conference on object-oriented programming*.

Chaudhuri, Avik, Vekris, Panagiotis, Goldman, Sam, Roch, Marshall, & Levi, Gabriel. (2017). Fast and Precise Type Checking for JavaScript. *Proc. acm program. lang.*, **1**(OOPSLA).

Church, Alonzo. (1940). A formulation of the simple theory of types. *The Journal of Symbolic Logic*, **5**(2), 56–68.

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

Curien, Pierre-Louis, & Ghelli, Giorgio. (1992). Coherence of subsumption, minimum typing and type-checking in $f_{\leq}$. *Mathematical structures in computer science*, **2**(1), 55–91.

Fan, Andong, Huang, Xuejing, Xu, Han, Sun, Yaozhu, & Oliveira, Bruno C. d. S. (2022). Direct Foundations for Compositional Programming. *Pages 18:1–18:28 of:* Ali, Karim, & Vitek, Jan (eds), *36th european conference on object-oriented programming (ecoop 2022)*. Leibniz International Proceedings in Informatics (LIPIcs), vol. 222. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Felleisen, Matthias, & Hieb, Robert. (1992). The Revised Report on the Syntactic Theories of Sequential Control and State. *Theor. comput. sci.*, **103**(2), 235–271.

Feng, Yangyue, & Luo, Zhaohui. (2011). Typed Operational Semantics for Dependent Record Types. *Electronic Proceedings in Theoretical Computer Science*, **53**(Mar), 30–46.

Findler, Robert Bruce, & Felleisen, Matthias. (2002). Contracts for higher-order functions. *Pages 48–59 of: Proceedings of the seventh ACM SIGPLAN International Conference on Functional programming*.

Garcia, Ronald. (2013). Calculating threesomes, with blame. *Pages 417–428 of: Proceedings of the 18th ACM SIGPLAN International Conference on Functional programming*.

Garcia, Ronald, & Cimini, Matteo. (2015). Principal Type Schemes for Gradual Programs. *Pages 303–315 of:* Rajamani, Sriram K., & Walker, David (eds), *Proceedings of the 42nd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2015, mumbai, india, january 15-17*.

Garcia, Ronald, Clark, Alison M., & Tanter, Éric. (2016). Abstracting Gradual Typing. *Pages 429–442 of: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. New York, NY, USA: Association for Computing Machinery.

Goguen, Healfdene. (1994). *A typed operational semantics for type theory*. Ph.D. thesis, University of Edinburgh.

Goguen, Healfdene. (1995). Typed operational semantics. *Pages 186–200 of: International Conference on Typed Lambda Calculi and Applications*. Springer.

Gray, Kathryn E., Findler, Robert Bruce, & Flatt, Matthew. (2005). Fine-Grained Interoperability through Mirrors and Contracts. *Page 231–245 of: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. New York, NY, USA: Association for Computing Machinery.

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

Hansen, Lars T. (2007). *Evolutionary programming and gradual typing in ECMAScript 4*. Tech. rept.

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

Herman, David, Tomb, Aaron, & Flanagan, Cormac. (2007). Space-efficient gradual typing. *In Trends in Functional Programming (TFP)*.

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

Huang, Xuejing, & Oliveira, Bruno C. d. S. (2020). A Type-Directed Operational Semantics For a Calculus with a Merge Operator. *Pages 26:1–26:32 of:* Hirschfeld, Robert, & Pape, Tobias (eds), *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Leibniz International Proceedings in Informatics (LIPIcs), vol. 166. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

Huang, Xuejing, Zhao, Jinxu, & Oliveira, Bruno C. d. S. (2021). Taming the Merge Operator. *Journal of Functional Programming*, **31**, e28.

Igarashi, Atsushi, Pierce, Benjamin C., & Wadler, Philip. (2001). Featherweight java: a minimal core calculus for java and gj. *Acm transactions on programming languages and systems (toplas)*, **23**(3), 396–450.

Labrada, Elizabeth, Toro, Matías, Tanter, Éric, & Devriese, Dominique. (2022). Plausible sealing for gradual parametricity. *Proc. acm program. lang.*, **6**(OOPSLA1).

Matthews, Jacob, & Ahmed, Amal. (2008). Parametric polymorphism through run-time sealing or, theorems for low, low prices! *Pages 16–31 of:* Drossopoulou, Sophia (ed), *Programming languages and systems*.

Matthews, Jacob, & Findler, Robert Bruce. (2009). Operational Semantics for Multi-Language Programs. *ACM Trans. Program. Lang. Syst.*, **31**(3).

Meijer, Erik, & Drayton, Peter. (2004). Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. Citeseer.

Moon, David A. (1989). *The Common LISP Object-Oriented Programming Language Standard*. New York, NY, USA: Association for Computing Machinery. Page 49–78.

Neis, Georg, Dreyer, Derek, & Rossberg, Andreas. (2009). Non-Parametric Parametricity. *Page 135–148 of: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*. ICFP '09. New York, NY, USA: Association for Computing Machinery.

New, Max S., Licata, Daniel R., & Ahmed, Amal. (2019). Gradual Type Theory. *Proc. ACM Program. Lang.*, **3**(POPL).

Pierce, Benjamin C. (1994). Bounded quantification is undecidable. *Information and computation*, **112**(1), 131–165.

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

Rastogi, Aseem, Chaudhuri, Avik, & Hosmer, Basil. (2012). The Ins and Outs of Gradual Type Inference. *Page 481–494 of: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '12. New York, NY, USA: Association for Computing Machinery.

Reynolds, John C. (1991). The coherence of languages with intersection types. *Pages 675–700 of: Lecture notes in computer science (lncs)*. Springer Berlin Heidelberg.

Reynolds, John C. (1993). An introduction to logical relations and parametric polymorphism (abstract). *Page 155–156 of: Proceedings of the 20th acm sigplan-sigact symposium on principles of programming languages*. POPL '93. New York, NY, USA: Association for Computing Machinery.

Serrano, Manuel, & Weis, Pierre. (1995). Bigloo: a portable and optimizing compiler for strict functional languages. *Pages 366–381 of: International Static Analysis Symposium*. Springer.

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

Siek, Jeremy, & Taha, Walid. (2007). Gradual typing for objects. *Pages 2–27 of: European Conference on Object-Oriented Programming*. Springer.

Siek, Jeremy, Garcia, Ronald, & Taha, Walid. (2009). Exploring the design space of higher-order casts. *Pages 17–31 of: European Symposium on Programming*. Springer.

Siek, Jeremy, Thiemann, Peter, & Wadler, Philip. (2015a). Blame and Coercion: Together Again for the First Time. *Page 425–435 of: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. New York, NY, USA: Association for Computing Machinery.

Siek, Jeremy G., & Taha, Walid. (2006). Gradual typing for functional languages. *Pages 81–92 of: Scheme and Functional Programming Workshop*, vol. 6.

Siek, Jeremy G, & Vachharajani, Manish. (2008). Gradual typing with unification-based inference. *Pages 1–12 of: Proceedings of the 2008 Symposium on Dynamic languages*.

Siek, Jeremy G., & Wadler, Philip. (2009). Threesomes, with and without blame. *Page 34–46 of: Proceedings for the 1st Workshop on Script to Program Evolution*. STOP '09. New York, NY, USA: Association for Computing Machinery.

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

Strickland, T. Stephen, Tobin-Hochstadt, Sam, Findler, Robert Bruce, & Flatt, Matthew. (2012). Chaperones and Impersonators: Run-Time Support for Reasonable Interposition. *Page 943–962 of: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '12. New York, NY, USA: Association for Computing Machinery.

Swamy, Nikhil, Fournet, Cedric, Rastogi, Aseem, Bhargavan, Karthikeyan, Chen, Juan, Strub, Pierre-Yves, & Bierman, Gavin. (2014). Gradual Typing Embedded Securely in JavaScript. *Page 425–437 of: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. New York, NY, USA: Association for Computing Machinery.

Takikawa, Asumu, Strickland, T Stephen, Dimoulas, Christos, Tobin-Hochstadt, Sam, & Felleisen, Matthias. (2012). Gradual typing for first-class classes. *Pages 793–810 of: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*.

Thatte, Satish. (1989). Quasi-static typing. *Pages 367–381 of: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Tobin-Hochstadt, Sam, & Felleisen, Matthias. (2006). Interlanguage migration: from scripts to programs. *Pages 964–974 of: Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*.

Tobin-Hochstadt, Sam, & Felleisen, Matthias. (2008). The Design and Implementation of Typed Scheme. *Page 395–406 of: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08. New York, NY, USA: Association for Computing Machinery.

Toro, Matías, Labrada, Elizabeth, & Tanter, Éric. (2019). Gradual parametricity, revisited. *Proceedings of the acm on programming languages*, **3**(POPL), 17:1–17:30.

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

Wadler, Philip, & Findler, Robert Bruce. (2009). Well-typed programs can't be blamed. *Pages 1–16 of: European Symposium on Programming*. Springer.

Wolff, Roger, Garcia, Ronald, Tanter, Éric, & Aldrich, Jonathan. (2011). Gradual typestate. *Pages 459–483 of: European Conference on Object-Oriented Programming*. Springer.

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

Xie, Ningning, Bi, Xuan, Oliveira, Bruno C. D. S., & Schrijvers, Tom. (2019). Consistent Subtyping for All. *Acm trans. program. lang. syst.*, **42**(1).

Ye, Wenjia, Oliveira, Bruno C. d. S., & Huang, Xuejing. (2021). Type-Directed Operational Semantics for Gradual Typing. *35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Ye, Wenjia, Toro, Matías, & Olmedo, Federico. (2023). A gradual probabilistic lambda calculus. *Proc. acm program. lang.*, **7**(OOPSLA1).