Merging Gradual Typing

WENJIA YE, The University of Hong Kong, China
BRUNO C. D. S. OLIVEIRA, The University of Hong Kong, China
MATÍAS TORO, University of Chile, Chile

Programming language mechanisms with a type-directed semantics are nowadays common and widely used. Such mechanisms include *gradual typing*, *type classes*, *implicits* and intersection types with a *merge operator*. While sharing common challenges in their design and having complementary strengths, type-directed mechanisms have been mostly independently studied.

This paper studies a new calculus, called $\lambda M^\star$, which combines two type-directed mechanisms: gradual typing and a merge operator based on intersection types. Gradual typing enables a smooth transition between dynamically and statically typed code, and is available in languages such as TypeScript or Flow. The merge operator generalizes record concatenation to allow merges of values of any two types. Recent work has shown that the merge operator enables modelling expressive OOP features like *first-class traits/classes* and *dynamic inheritance* with static type-checking. These features are not found in mainstream statically typed OOP languages, but they can be found in dynamically or gradually typed languages such as JavaScript or TypeScript. In $\lambda M^\star$, by exploiting the complementary strengths of gradual typing and the merge operator, we obtain a foundation for modelling gradually typed languages with both first-class classes and dynamic inheritance. We study a static variant of $\lambda M^\star$ (called λM); prove the type-soundness of $\lambda M^\star$; show that $\lambda M^\star$ can encode gradual rows and all well-typed terms in the $GTFL_{\lesssim}$ calculus; and show that $\lambda M^\star$ satisfies gradual typing criteria. The dynamic gradual guarantee (DGG) is challenging due to the possibility of ambiguity errors. We establish a variant of the DGG using a semantic notion of precision based on a step-indexed logical relation.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Data types and structures**.

Additional Key Words and Phrases: Bidirectional Typing, Gradual Typing, Type-Directed Semantics, Merge Operator

ACM Reference Format:
Wenjia Ye, Bruno C. d. S. Oliveira, and Matías Toro. 2024. Merging Gradual Typing. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 294 (October 2024), 29 pages. https://doi.org/10.1145/3689734

1 Introduction

Programming language mechanisms with a type-directed semantics are nowadays widely used. Such mechanisms include *gradual typing* [Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006], *type classes* [Wadler and Blott 1989], *implicits* [Oliveira et al. 2010] and intersection types with a *merge operator* [Dunfield 2014; Reynolds 1997]. In all those mechanisms the semantics of a program may depend on the types assigned to the program. In other words, changing some type in the program (without changing anything else) may change the semantics of the program. Programming

Authors' Contact Information: Wenjia Ye, The University of Hong Kong, Hong Kong, China, yewenjia@connect.hku.hk; Bruno C. d. S. Oliveira, The University of Hong Kong, Hong Kong, China, bruno@cs.hku.hk; Matías Toro, PLEIAD Lab, Computer Science Department (DCC), University of Chile, Santiago, Chile, mtoro@dcc.uchile.cl.

ACM 2475-1421/2024/10-ART294
https://doi.org/10.1145/3689734

languages such as Haskell (via type classes), Scala (via implicits), gradually typed languages or even Java (via static overloading) all include language mechanisms with a type-directed semantics.

While sharing common challenges in their design and having complementary strengths, type-directed mechanisms have been mostly independently studied. In this paper we focus on integrating two type-directed mechanisms: gradual typing and the merge operator in calculi with intersection types. Gradual typing enables a gradual transition between dynamically and statically typed code, and is nowadays available in languages such as TypeScript and Flow (which are supersets of JavaScript). The merge operator generalizes record concatenation to allow merges of values of any two types. Recent work [Bi and Oliveira 2018; Zhang et al. 2021] showed that the merge operator can model expressive OOP features like *first-class traits/classes* [Takikawa et al. 2012] and *dynamic inheritance* [Ernst 2000] with static type-checking. Such features are not found in mainstream statically typed OOP languages, but they are found in dynamic languages such as JavaScript.

Due to its practical importance, there has been much research in past years on gradual typing. Nonetheless, much of the focus of research on gradual typing has been on gradualizing common statically-typed calculi that do not have a type-directed semantics. Within this line of work, Siek and Taha [2007] initiated a line of work exploring minimal gradually typed calculi for modelling objects, based on an object calculus by Abadi and Cardelli [1996]. Siek and Taha's calculus is relatively limited in that it only supports objects with a fixed number of fields/methods. More recently, gradual variants of record calculi, such as the $\mathsf{GTFL}_{\lesssim}$ calculus, have been proposed [Bañados Schwerter et al. 2021; Garcia et al. 2016]. Similarly to Siek's work, the $\mathsf{GTFL}_{\lesssim}$ calculus only supports fixed-size records. The restrictions in those calculi mean that there is still a large gap to the features that are available in JavaScript. In particular the lack of extensible objects/records prevents modelling (dynamic) multiple inheritance and more expressive OOP mechanisms that are available in languages such as JavaScript. A notable reference in this space is Takikawa et al. [2012] work, which has addressed the integration of gradual typing and first-class classes. However, this integration is at the module level, allowing dynamically typed and statically typed modules to interoperate.

Calculi with extensible records provide a natural foundation for languages with inheritance, which can be modelled by record concatenation [Cook and Palsberg 1989; Wand 1989]. Unfortunately, as identified by Cardelli and Mitchell [1991], there are important challenges to develop a *typed* language with both *record concatenation* and *subtyping*. Calculi with the merge operator and disjoint intersection types [Oliveira et al. 2016] overcome such challenges with a type-directed semantics. Recently, Huang et al. [2021] proposed a type-directed operational semantics (TDOS) approach for such calculi. The TDOS approach allows giving a direct operational semantics to calculi with the merge operator. Furthermore the TDOS approach is not tied to calculi with the merge operator, and can be used to model the semantics of other type-directed mechanisms as well. In particular, it has been adapted by Ye et al. [2021] to gradual typing. However, so far there is no calculus including *both* the merge operator and gradual typing.

This paper studies a new calculus, called $\lambda M^{\star}$, combining gradual typing and a merge operator based on intersection types. With $\lambda M^{\star}$, we obtain a foundation for modelling gradually typed languages with expressive OOP features, such as first-class classes and dynamic inheritance in a purely functional setting with records. There is still a gap between our work, and mainstream languages like JavaScript and TypeScript, since we do not consider imperative features, such as references and object identity. Nevertheless, we address fundamental questions that arise from the interaction between dynamic inheritance and method overriding. Without care, such interaction can easily lead to type unsoundness, as shown in Section 2.3 using TypeScript. Furthermore, with gradual typing, this interaction is further complicated by the possibility of runtime ambiguity errors. We make the following contributions in this paper:

- We show how to **combine two type-directed mechanisms (gradual typing and the merge operator)** into a single language. This sheds new insights such as how to integrate multiple type-directed mechanisms, and how to design *casting* relations for the dynamic semantics.
- **The λM and the $\lambda M^\star$ calculi.** We present the $\lambda M^\star$ calculus as a concrete language design integrating gradual typing and the merge operator using a TDOS. The static counterpart is a variant of a calculus with a merge operator called λM. We prove several results for λM and $\lambda M^\star$, including *type soundness*, *determinism* and the *gradual guarantee* for $\lambda M^\star$.
- **A new solution to the problem of modular type invariants for gradual rows** identified by Bañados Schwerter et al. [2021]. $\lambda M^\star$ provides a solution (inherited from previous calculi with a merge operator [Huang et al. 2021]) to preserve such modular type invariants. Moreover we relate the problem to a problem that was identified 30 years earlier by Cardelli and Mitchell [1991] for record calculi with subtyping.
- **An encoding of gradual rows and the $\text{GTFL}_{\leqslant}$ calculus** in $\lambda M^\star$. Compared to the $\text{GTFL}_{\leqslant}$ calculus, $\lambda M^\star$ does not need a special type for gradual rows, and supports *extensible records*.
- **Prototype, Coq proofs and a proof of the dynamic gradual guarantee.** All the calculi and proofs in this paper are mechanically formalized in Coq, with the exception of dynamic gradual guarantee, which employs a step-indexed logical relation and is manually proved. We also offer an interactive prototype implementation of $\lambda M^\star$ (including some simple extensions). Both the formalization, proofs and implementation are available in the artifact [Ye et al. 2024].

2 Overview

We start with an overview of the merge operator and gradual typing, motivate the combination of the two features, and give an overview of our work and the λM and $\lambda M^\star$ calculi.

2.1 Background: Gradual Typing

Gradual typing [Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006] enables programs to range from dynamic typing to static typing. To defer static type checks to runtime, gradual typing employs the unknown type $\star$ and consistency relations. The unknown type $\star$ is consistent with any type. Dynamic type errors are triggered by casts. The implicit casts in gradual typing have a type-directed semantics: the semantics of programs depends on the types used in the casts. For example, in the simple expression $1 : \star : \text{Bool}$ casting 1 to type Bool will result in *blame* (i.e. a runtime type error). However, if we have $1 : \star : \text{Int}$ instead, we obtain the integer 1 after running the program. Thus, the types used by casts can give rise to different evaluation results. Note that in the previous examples (and the examples that follow), we adopt a notation similar to type annotations to denote casts. For instance, in the expression $1 : \star : \text{Bool}$ there are two casts: a first cast from Int (the type of 1) to $\star$; and a second cast from $\star$ to Bool. We choose the use of this notation throughout the paper to be consistent with the notation in the λM and $\lambda M^\star$ calculi.

Typed-Directed Operational Semantics (TDOS). Traditionally the semantics of gradual languages is given by an elaboration to an intermediate (cast) calculus [Siek and Taha 2006]. Ye et al. [2021] proposed an alternative approach to give the semantics of gradually typed calculi that avoids an elaboration. The approach is based on typed-directed operational semantics (TDOS): a variant of small-step semantics first proposed by Huang and Oliveira [2020]. A TDOS uses type annotations to determine the result of reduction at run-time. TDOS contains two main components. One is a traditional reduction relation with a few adjustments. The other one is a typed reduction relation $v \hookrightarrow_A v'$, which we call casting in our work. The casting relation takes a value and a type as the input and produces a value matching the shape of input type. Ye et al. [2021] applied the TDOS to

gradual typing successfully to two different gradually typed calculi. For gradual typing, the casting relation generalizes the result of casting ($v \hookrightarrow_A r$) to a result r, which contains not only values but also run-time errors (err_*). Compared to the elaboration approach, a benefit of the TDOS is that the dynamic semantics is defined directly for the gradually typed source language.

2.2 Background: The Merge Operator

Some calculi with intersection types employ a special operator, called the *merge operator* [Dunfield 2014; Reynolds 1997], that allows building values that can have multiple types. For example, in the following program, x has both an integer and a boolean value and has the type Int & Bool:

$$\text{let } x = 1 \,,, \text{ True in } (x + 1, \text{not } x)$$

x is built using the merge operator $(,,)$. When x is used, it can act as either an integer or a boolean. In calculi with a merge operator multi-field records are merges of single field records. As Dunfield noticed, the merge operator can encode various other programming language features, including *extensible records*, *dynamic typing* and *operator overloading*. Recent research has further shown programming language designs, such as SEDEL [Bi and Oliveira 2018] or CP [Zhang et al. 2021], based on variants of the merge operator. These designs enable applications such as first-class classes/traits [Bi and Oliveira 2018] and Compositional Programming [Zhang et al. 2021].

Type-directed Semantics of Merges and the Interaction with Subtyping. The semantics of the merge operator is type-directed: components are extracted from merges based on types. For instance, in the expression $x + 1$ above, the type Int is required by $+$. Therefore 1 should be extracted from x. While convenient, the type-directed extraction of values can lead to ambiguity. Consider $(1 \,,, 2) : \text{Int}$. This program is ambiguous because the result can be either 1 or 2. Moreover, the interaction between subtyping and the merge operator is subtle [Dunfield 2014; Huang et al. 2021]. A closely related problem was identified by Cardelli and Mitchell [1991], for calculi with *subtyping* and *record concatenation* (a special case of the general merge operator). We illustrate the issue with an example based on Cardelli and Mitchell's work:

$$\text{let } x : \{l_2 : \text{Bool}\} = \{l_1 = \text{"Boom!"}\} \,,, \{l_2 = \text{True}\} \text{ in } (\{l_1 = 2\} \,,, x).l_1 + 3$$

Variable x has type $\{l_2 : \text{Bool}\}$. The value for x includes a field l_1, which is hidden due to subtyping. The merge $\{l_1 = 2\},, x$, appears to be safe statically (since statically x does not contain l_1). However, what should happen when we do the field lookup? If the original field l_1 is preserved in x then, when we lookup l_1, there will be two l_1 fields. Naive biased lookups are problematic. For instance, in the program above, if a right-biased lookup is used, then the program would extract the string "Boom!" and try to add that to an integer, which would crash the program. In other words a naive biased lookup for merges in the presence of subtyping *is not type-sound*. Even if the two values of the field l_1 have the same type, extracting the value of the hidden field may lead to surprising behaviour to programmers, since the type of x appears to promise that no field l_1 is present. For these reasons Cardelli and Mitchell argued that biased lookups should not be used.

Disjoint Intersection Types. To address the ambiguity problems, as well as the problems arising from the interactions between merges and subtyping, Oliveira et al. [2016] proposed to have a restriction where only merges of disjoint types are accepted. Disjointness rejects ambiguous programs such as $\text{True} \,,, \text{False}$ or $1 \,,, 2$, since the types of the two values being merged are not disjoint. Similarly to gradual typing (as discussed in Section 2.1), the semantics for languages with the merge operator can also be given using a TDOS approach [Huang and Oliveira 2020; Huang et al. 2021]. Huang et al. proposed λ_i: a calculus with a merge operator and disjoint intersection types. λ_i solves the ambiguity of issues of the merge operator with disjointness and a TDOS. We

illustrate how λ_i's TDOS solves the problem next:

$$(\lambda x.\,((x\,,,\,1) + 1) : \mathsf{Bool} \to \mathsf{Int})\,(\mathsf{True}\,,,\,2) \hookrightarrow^* ((\mathsf{True}\,,,\,1) + 1) : \mathsf{Int} \hookrightarrow^* 2$$

If we just substitute $\mathsf{True}\,,,\,2$ with a normal beta reduction, a non-disjoint expression would be generated after substitution $(\mathsf{True}\,,,\,2\,,,\,1)$. Instead, True is extracted by casting $\mathsf{True}\,,,\,2$ under the function input type Bool. Thus the value that gets substituted in the body of the lambda is True instead of $\mathsf{True}\,,,\,2$. This enables the program to reduce without encountering ambiguities in the merges. Coming back to the example with records:

$$\mathsf{let}\ x : \{l_2 : \mathsf{Bool}\} = \{l_1 = \text{"Boom!"}\}\,,,\,\{l_2 = \mathsf{True}\}\ \mathsf{in}\ (\{l_1 = 2\}\,,,\,x).l_1{+}3$$

What λ_i (extended with records) does is to drop the field l_1 in x when the value is upcast to have the type $\{l_2 : \mathsf{Bool}\}$. Therefore, $(\{l_1 = 2\}\,,,\,x).l_1$ would become $(\{l_1 = 2\}\,,,\,\{l_2 = \mathsf{True}\}).l_1$ and the final result of the program would be 5. In other words, the solution of λ_i to the problem of the interaction between merges and subtyping is to ensure that values in a merge that are hidden by subtyping are dropped from the value when (up)casting.

2.3 Motivation: Combining Merges and Gradual Typing

While TDOS has been applied to both gradual typing and calculi with the merge operator separately, there is no calculus that supports *both* gradual typing and the merge operator. However there are compelling reasons to develop calculi supporting both features, which we discuss next.

Modelling Expressive Dynamic OOP features. Most mainstream implementations of gradually typed languages target languages such as JavaScript. While in gradual typing research has focused on gradualizing a variety of common type systems, there is much less effort on type systems that model highly dynamic OOP features. Yet, since languages like JavaScript are actually the most common practical focus on mainstream gradually typed language implementations (like TypeScript or Flow), this leaves open the question of how to design and implement type systems that support such features. Since one of the use-modes of gradual typing is full static typing, it is desirable to support (static) type systems that enable type-checking for (some of) the advanced OOP features of dynamic languages such as JavaScript.

For example, JavaScript supports *first-class classes* [Takikawa et al. 2012], and *dynamic inheritance* [Ernst 2000]. First-class classes are first-class values (just like lambdas in functional programming), and can be passed as arguments or returned as results. Dynamic inheritance means that the inherited classes are not statically known (they can be parametrized, for instance). With first-class classes and dynamic inheritance, the programmer can abstract over patterns in the hierarchy of classes and model mixins [Bracha and Cook 1990]. In JavaScript a mixin is as a function that takes a superclass as input and returns a subclass that extends the superclass. For example:

```
const circleMixin = shape ⇒ {
  return class extends shape { area(radius) { return PI * radius * radius; } }
};
```

In this JavaScript code, `circleMixin` extends `shape` with a method to calculate the area of a circle. The super class `shape` is a function parameter, which means that `circleMixin` can be extended by any `shape` class at runtime. In a conventional statically-typed class-based language such as Java, such parametrization by a superclass is not possible, due to restrictions of the type system.

A Type Unsound Approach to First-Class Classes in TypeScript. TypeScript supports conventional static inheritance idioms and its type system prevents type-unsafe overrides (similarly to Java or

```
class A {
  m() : number {return 5};
  n() : number {return this.m() - 4;} }
interface C {n() : number}
type GConstructor<T = {}> = new (...args: any[]) ⇒ T;
function mkB<TBase extends GConstructor<C>>(Base: TBase) {
  return class B extends Base {
    m() :  string {return "hello";}  // If m exists in Base it will be overriden
  };
}
const cl = mkB(A);   // Problem: Superclass already contains an m
const o = new cl;
console.log(o.n());
```

Fig. 1. Type unsoundness of first-class classes in TypeScript.

C#). In addition, TypeScript also supports first-class classes and dynamic inheritance[1]. However, as we shall illustrate next, the fact that with dynamic inheritance we do not have the exact type information for superclasses is problematic and leads to *type unsoundness* (without relying on dynamic types). For instance consider the TypeScript program in Figure 1. In this program, we create a class `A` with `m` and `n` methods, which return integers. Importantly, `n` is defined in terms of `m`. Then `mkB` is parametrized by a class `Base`, which is used as the superclass of `B`. We can specify the interface of the superclass as being `C`, which only contains a method `n`. Note that `B` defines another method `m`, which returns a string. TypeScript checks that there are no conflicts between `m` and the methods in the superclass interface `C`. We then create an object `o` using `A` as the superclass for `B` (via `mkB`). Unfortunately, the `m` that is present in `B` overrides the `m` from `A`. Then when we run `n` we end up subtracting an integer from a string, which results in a runtime type error (TypeScript/JavaScript actually tries to convert the string to a number and we get NaN instead).

In short, the TypeScript approach to deal with first-class classes is *type unsound*. The reason for unsoundness is `mkB(A)`. `A` is a subtype of `C` with an extra `m() : number` field, but when type-checking `B` we do not know about the extra members (`m() : number`) of the subtype. Thus the type system of TypeScript fails to detect the problematic override. This problem is a manifestation of the problem identified by Cardelli and Mitchell [1991] discussed in the Section 2.2. Note also that in languages with static inheritance and top-level classes only (such as Java or C#) there is no such flexibility and the issue above does not arise. The problem is more pervasive with the dynamic type, where we may be able to inherit from a supertype with an unknown interface, but then we cannot statically prevent overrides since there is no information at all about the supertype.

First-class Traits and Dynamic Inheritance with Merges. Calculi and languages with the merge operator can model mechanisms such as first-class classes. For instance, in the CP language [Zhang et al. 2021], we can rewrite the `circle` mixin as:

```
type Shape = { name : String }
circle (super: Trait<Shape>) =
   trait inherits super ⇒ { area radius = PI * radius * radius; };
```

[1]https://www.typescriptlang.org/docs/handbook/mixins.html

The CP language supports a form of first-class traits [Bi and Oliveira 2018] (which are analogous to classes), and supports dynamic inheritance like JavaScript. However, unlike JavaScript, CP is statically typed. In the program above, the trait being inherited is parametrized. For simplicity, in the code above we assume that the interface of the trait being inherited is `Shape`, but CP also supports dynamic inheritance even when the interface of the superclass is *not fully know* using *disjoint polymorphism* [Alpuim et al. 2017]. Section 2.4 gives a brief overview of how the encoding of first-class classes in CP in terms of merges works. We also show how the semantics of casting and merges together with disjointness solve the problem in Figure 1, and enable type-sound and expressive designs of languages with first-class classes. We refer the reader interested in more advanced features of CP and the full details of how CP elaborates first-class traits and dynamic inheritance to a calculus with a merge operator to the work by Bi and Oliveira [2018] and Zhang et al. [2021].

In this work we propose to combine the merge operator with gradual typing. Thus we envision a language like CP, supporting gradual typing. In such language, we could have a variation of the program above that combines first-class traits, dynamic inheritance and gradual typing:

```
circle (super: Trait<⋆>) = trait inherits super ⇒ {
   area (radius : ⋆) : number = PI * radius * radius;
};
```

In the program above, we mix static and dynamic typing. There are two noteworthy points. Firstly, we make the type of `radius` dynamic (or unknown), and implicitly cast `radius` from $\star$ to a number. Secondly, and more interestingly, the inherited trait has an *unknown* interface. We cannot rule out conflicts statically, like in CP, because there is no static type information about the interface of the supertype. Therefore, how can we deal with possible method conflicts? For instance, what if the `super` class/trait has an `area` method, taking one argument, already? Adopting an overriding semantics would be prone to issues similar to those identified by Cardelli and Mitchell. So, instead, we propose to detect ambiguity at runtime: if the superclass contains a conflicting method, then a runtime error may be raised to indicate ambiguity. By allowing programs like the above, we can have a language, which supports very dynamic OOP features similar to those in JavaScript, while at the same time supporting gradual typing.

Unified Foundation for Type-Directed Mechanisms. A second reason to have a unified framework for type-directed mechanisms is that it is beneficial to avoid duplication of efforts in addressing common problems. For example, performant sound gradual typing is currently a hot topic [Greenman 2023; Greenman et al. 2019; Kuhlenschmidt et al. 2019; Muehlboeck and Tate 2017, 2021; Takikawa et al. 2016], since there is a high cost imposed by casting. Because calculi with the merge operator have casting, this is an issue for such calculi as well. Thus, leveraging on the developments for gradually typed languages is helpful to address similar problems in calculi with merges. In the current work we do not address the important issue of performance. However, we hope to leverage on the existing work on gradual typing in the future to improve the performance in calculi with the merge operator. Section 7 briefly sketches some possible directions for performance improvements. Furthermore, designs for the semantics of calculi with the merge operator can also lead to new developments that are useful for gradual typing. For instance, the TDOS approach to gradual typing originated from developments in the semantics of languages with the merge operator.

2.4 Key Ideas and Challenges

In this paper we propose two new calculi. The λM calculus is a statically typed calculus, which is a variant of the λ_i calculus with the merge operator and disjoint intersection types. We created

$\lambda\mathrm{M}$ because, to integrate gradual typing with the merge operator more easily, we need to modify some details of the semantics. In particular $\lambda\mathrm{M}$ has a different form of values and a lazy semantics for higher-order values [Wadler and Findler 2009] that is not present in λ_i. A side-benefit of the changes in $\lambda\mathrm{M}$ is that it leads to a standard type preservation theorem, whereas in λ_i the reduction increases the precision of types and preservation has to be relaxed. The $\lambda\mathrm{M}^\star$ calculus is a gradual version of the $\lambda\mathrm{M}$ calculus, and adds the unknown type $\star$ to $\lambda\mathrm{M}$. The addition of $\star$ to the calculus is nontrivial and leads to several changes in the semantics and the metatheory. We describe some of the key ideas next. The details are presented in Sections 3, 4 and 5.

Gradual Disjointness. In λ_i disjointness of types implies that Int is disjoint with Bool, but Int is not disjoint with Int or Int & Bool. Disjointness has a simple specification: two types are disjoint if they have no common supertypes, except for top-like types. Top-like types include $\top$ itself and types isomorphic to $\top$, such as $\top\&\top$. When using a simple subtyping relation with intersection types, this definition of disjointness means that two function types are never disjoint: we can always find common supertypes that are not top-like for any two functions [Oliveira et al. 2016]. Oliveira et al. shows some alternatives to allow functions to be disjoint. However, for simplicity here, we adopt the simpler formulation by Oliveira et al. where functions cannot be disjoint.

When adding $\star$ to a calculus with disjointness, an interesting question is: *How should $\star$ behave with respect to disjointness?* To define *gradual disjointness*, we use the existential lifting of the static relation from the Abstracting Gradual Typing (AGT) approach [Garcia et al. 2016]. With an existential lifting we know that $\star$ is disjoint with A, if there exists some disjoint pair of static types more precise than $\star$ and A. As $\top$ is more precise than $\star$, and $\top$ is disjoint with any other type, then this means that $\star$ is disjoint to any other type.

Ambiguity Errors and Type Errors. As expected, if we consider imprecise types, we need to check at runtime if disjointness is violated. For instance, $(1 : \star ,, 2 : \star) : \mathsf{Int}$ reduces to an error (Int is a possible supertype of $\star$). Otherwise the reduction would be non-deterministic: we could choose any of the two integers. Now consider the reduction of $(\mathsf{True} : \star ,, 1 : \star) : \mathsf{Int}$. First, as both components of the merge operator are suitable for casting, we cast both components to Int: $\mathsf{True} : \star \hookrightarrow_{\mathsf{Int}} \mathsf{err}$ and $1 : \star \hookrightarrow_{\mathsf{Int}} 1$. Then as the left component reduces to an error, we keep the right component and reduce the whole expression to 1. This approach is motivated by the fact that Bool & Int is a subtype of Int. Therefore, by the type safety property of the static type discipline, a program such as $(\mathsf{True} ,, 1) : \mathsf{Int}$ does not fail, using similar reasoning. Now consider the expression $((1 : \star ,, 2 : \star) : \star ,, 3) : \mathsf{Int}$. We would like this expression to reduce to an error. However, the approach that we have adopted so far does not work. Let's see why. First, $(1 : \star ,, 2 : \star) : \star \hookrightarrow_{\mathsf{Int}} \mathsf{err}$ due to ambiguity, and then $3 : \star \hookrightarrow_{\mathsf{Int}} 3$. Then, as the left component reduces to an error, we keep the right component and reduce the whole expression to 3. However, we would like an error instead.

To avoid this problem, we differentiate two kinds of errors: ambiguity errors err_a and type errors err_t. The expression $(1 : \star, 2 : \star) : \mathsf{Int}$ reduces to an ambiguity error err_a, and $1 : \star : \mathsf{Bool}$ reduces to a type error err_t. Going back to the last example $((1 : \star ,, 2 : \star) : \star ,, 3) : \mathsf{Int}$, as the left component reduces to an ambiguity error, we propagate this error to the whole expression and reduce to err_a.

Encoding the $\mathsf{GTFL}_{\lesssim}$ Calculus and Modular Type-based Invariants. Garcia et al. [2016] developed a gradually-typed lambda calculus with records and subtyping ($\mathsf{GTFL}_{\lesssim}$) using the AGT methodology. They use gradual rows ($\{\overline{l_i : S_i}, \star\}$) to represent records with incomplete type information. Extra

fields, which are not reflected in the type, can be typed with $\star$. Two examples are given next.

$$(\{l_1 = 1, l_2 = \mathsf{True}, l_3 = \ldots, \ldots\} : \{l_1 : \mathsf{Int}, l_2 : \mathsf{Bool}\}).l_2 \hookrightarrow^* \mathsf{True}$$
$$(\{l_1 = 1, l_2 = \mathsf{True}, l_3 = \ldots, \ldots\} : \{l_1 : \mathsf{Int}, \star\}).l_2 \hookrightarrow^* \mathsf{True} : \star$$

In the first program, we have a record with multiple fields, but where only two fields are statically known. The other fields are hidden via subtyping. The projection label l_2 is contained in the record type and value. Thus the program is well-typed. If we try to project l_3 instead, the program is ill-typed and it is statically rejected. The second program illustrates gradual rows. Although the projected field is missing in the type, the value of l_2 field can still be projected. Since the l_2 field is contained in the extra unknown part $\star$. Gradual rows allow extra fields to be projected and checking whether fields are present is performed at runtime.

Gradual rows can be encoded easily in $\lambda M^\star$ via merges, intersection types and the unknown type in $\lambda M^\star$. The above programs are encoded in $\lambda M^\star$ as follows.

$$((\{l_1 = 1\} ,, \{l_2 = \mathsf{True}\} ,, \{l_3 = \ldots\} ,, \ldots) : \{l_1 : \mathsf{Int}\} \,\&\, \{l_2 : \mathsf{Bool}\}).l_2 \hookrightarrow^* \mathsf{True} \quad (1)$$
$$((\{l_1 = 1\} ,, \{l_2 = \mathsf{True}\} ,, \{l_3 = \ldots\} ,, \ldots) : \{l_1 : \mathsf{Int}\} \,\&\, \star).l_2 \hookrightarrow^* \mathsf{True} : \star \quad (2)$$

Compared to $\mathrm{GTFL}_{\lesssim}$, $\lambda M^\star$ has extensible records (via the merge operator), whereas $\mathrm{GTFL}_{\lesssim}$ only supports fixed size records. Thus, $\mathrm{GTFL}_{\lesssim}$ cannot immediately encode multiple inheritance directly (which can be supported via record concatenation) and, it cannot encode first-class classes and dynamic inheritance either. An important difference between $\mathrm{GTFL}_{\lesssim}$ and our work is that $\mathrm{GTFL}_{\lesssim}$ does *not* allow records with the same label to be present, even if these are in the dynamic parts of the rows: $\mathrm{GTFL}_{\lesssim}$ *statically* rejects records with repeated labels. This approach is possible to adopt in $\mathrm{GTFL}_{\lesssim}$ because, with fixed-sized records, *all labels are statically known.* However, this approach is problematic with extensible records and concatenation. Let us look at the following program:

$$\mathsf{let}\ \mathsf{f}\ (x : \star)(\mathsf{y} : \star) = x ,, \mathsf{y}\ \mathsf{in}\ \mathsf{f}\ \{l_1 = 1\}\{l_1 = 2\}$$

Here two dynamically typed expressions (x and y) are merged. If two records $\{l_1 = 1\}$ and $\{l_1 = 2\}$ are passed as arguments to f, there will be ambiguity. There are two possible designs. We could conservatively reject concatenation/merges with dynamic components. But this would be undesirable as it would prevent programs such as the gradual circle trait with an unknown superclass presented earlier. The other option is to allow concatenating two records with unknown fields at runtime and check ambiguity errors at runtime, which is the approach that we take.

The dynamic semantics of $\lambda M^\star$ does *not* preserve the semantics of $\mathrm{GTFL}_{\lesssim}$. Thus we do not prove an operational correspondence result. The first reason for this is that $\lambda M^\star$ employs a lazy semantics, whereas $\mathrm{GTFL}_{\lesssim}$ uses an eager semantics for higher-order casts. The second reason is that the original semantics of $\mathrm{GTFL}_{\lesssim}$ [Garcia et al. 2016] fails to preserve some expected *modular type invariants*. Although this definition has never been formally stated, it is associated with the static guarantees that types can provide regarding programs, such as parametricity granted by polymorphism. Subtyping also provides modular type invariants. Consider $\mathsf{A} <: \mathsf{B}$ and program: $\mathsf{let}\ \mathsf{x} : \mathsf{B} = \mathsf{new}\ \mathsf{A}()\ \mathsf{in}\ \mathsf{e}$. By looking at the type of x, we know that e cannot use x as an A.

In the context of gradual typing, Bañados Schwerter et al. [2021] pointed out that the semantics of $\mathrm{GTFL}_{\lesssim}$ fails to preserve expected modular type invariants. Let us consider program $\mathsf{let}\ \mathsf{x} : \{l_1 : \mathsf{Int}\} = \{l_1 = 5, l_2 = \mathsf{True}\}\ \mathsf{in}\ \mathsf{x}$. According to subtype-based reasoning of static typing, the l_2 field should not be accessed in the body of the let. However, for a gradually typed variant of the program: $\mathsf{let}\ \mathsf{x} : \{l_1 : \mathsf{Int}\} = \{l_1 = 5, l_2 = \mathsf{True}\} : \star\ \mathsf{in}\ (\mathsf{x} : \star).l_2$. the original formulation of $\mathrm{GTFL}_{\lesssim}$ should signal a run-time type error, but it does not. Instead it accesses the l_2 field of the record. In essence, the record preserves the hidden fields and allows them to be accessed later. When casting to $\star$ and back to the original record type, the l_2 field is exposed.

As discussed in Section 2.2, calculi with the merge operator and disjoint intersection types provide a solution for similar problems by enforcing the expected invariants using casting. This solution extends to a setting with gradual typing. The earlier example can be encoded in $\lambda M^\star$:

$$(\{l_1 = 5\},, \{l_2 = \mathsf{True}\}) : \star : \{l_1 : \mathsf{Int}\} : \star : \{l_1 : \mathsf{Int}\} \,\&\, \{l_2 : \mathsf{Bool}\}).l_2$$
$$\hookrightarrow (\{l_1 = 5\} : \star : \{l_1 : \mathsf{Int}\} \,\&\, \{l_2 : \mathsf{Bool}\}).l_2 \hookrightarrow \mathsf{err}_\mathsf{t}$$

When $(\{l_1 = 5\},, \{l_2 = \mathsf{True}\}) : \star$ is cast under $\{l_1 : \mathsf{Int}\}$, the field l_1 is selected and l_2 field is dropped. Then, trying to cast the resulting record under $\{l_1 : \mathsf{Int}\} \,\&\, \{l_2 : \mathsf{Bool}\}$, a type error is raised since the record no longer contains l_2.

The Dynamic Gradual Guarantee in $\lambda M^\star$. Siek et al. [2015b] proposed a set of criteria for gradual typing that encompasses several properties. An important property is referred to as the *dynamic gradual guarantee* (DGG), which relates to the notion of *(im)precision*. We say that one type is more precise than another ($A \sqsubseteq B$) if it provides more static information. For instance, $\mathsf{Int} \,\&\, \mathsf{Bool} \sqsubseteq \mathsf{Int} \,\&\, \star \sqsubseteq \star \,\&\, \star \sqsubseteq \star$. Similarly, we say that one program is more precise than another if it has more precise types. The DGG ensures that reduction is monotone with respect to imprecision.

The DGG requires special attention as it is in conflict with determinism. If we define the DGG as: decreasing precision does not alter the behavior of the program (and does not introduce new **errors of any kind**), then the DGG is not satisfied. To illustrate this, we provide a minimal example that demonstrates this incompatibility. Consider the following program:

$$((1,, \mathsf{True}) : \mathsf{Int},, (2,, \mathsf{False}) : \mathsf{Bool}) : \mathsf{Bool}$$
$$\hookrightarrow (1,, (2,, \mathsf{False}) : \mathsf{Bool}) : \mathsf{Bool} \hookrightarrow (1,, \mathsf{False}) : \mathsf{Bool} \hookrightarrow \mathsf{False}$$

If we consider a less precise version of this program, $((1,, \mathsf{True}) : \star,, (2,, \mathsf{False}) : \star) : \mathsf{Bool}$, a significant problem arises. We cannot determine which of the two merges should provide the required boolean. Arbitrarily selecting the left merge would yield True, breaking the DGG. Arbitrarily choosing the right merge does not address this problem either. For instance, a slightly different program $((1,, \mathsf{True}) : \mathsf{Int},, (2,, \mathsf{False}) : \mathsf{Bool}) : \mathsf{Int}$ reduces to 1, but a less precise program $((1,, \mathsf{True}) : \star,, (2,, \mathsf{False}) : \star) : \mathsf{Int}$ would reduce to 2, also violating the DGG. Hence, in $\lambda M^\star$, this expression reduces to an ambiguity error err_a. However, if the program is modified to $((1,, \mathsf{False}) : \star,, (2,, \mathsf{False}) : \star) : \mathsf{Bool}$, $\lambda M^\star$ reduces to False since any path would yield the same result. In our work we prove a variant of the DGG: decreasing precision does not alter the behavior of the program *modulo* ambiguity errors (i.e. new **type errors** are not introduced).

Encoding First-Class Classes and Dynamic Inheritance. With $\lambda M^\star$ we can encode a form of first-class classes/traits and dynamic multiple inheritance with gradual typing. The encoding follows an existing encoding of first-class traits employed in the SEDEL and CP programming languages. The addition of gradual typing is essentially *orthogonal* to the existing encoding. The basic idea of the encoding is well-known and itself inspired by work on object encodings using records [Bruce et al. 1999; Cardelli 1988; Cook and Palsberg 1989; Wand 1989]. In the well-known record encoding records are used to model objects, record concatenation models (multiple) inheritance, classes (or traits) can be modeled as functions parametrized by self-references that return records (objects), and fixpoints model class instantiation. In the translation of those ideas to $\lambda M^\star$, records are modeled as merges of single field records, and record concatenation is just a special case of merges. For example, a simplified version of the encoding, for the `circle` trait, in $\lambda M^\star$ is:

```
let circle = λsuper. (super,, {area = (λradius. pi * radius * radius) : ⋆ → int}) in
let obj = circle {} in
obj.area 2
```

$$
\begin{array}{lll}
\text{Types} & A, B, C & ::= \mathsf{Int} \mid \top \mid \bot \mid A \to B \mid \{l : A\} \mid A \,\&\, B \\
\text{Ordinary Types} & A^{\circ} & ::= \mathsf{Int} \mid A \to B \mid \{l : A\} \\
\text{Expressions} & e & ::= x \mid i \mid \mathsf{Top} \mid \lambda x.\, e \mid \{l = e\} \mid e.l \mid e : A \mid e_1\, e_2 \mid e_1 \,,, e_2 \mid \mathsf{fix}\, x.\, e \\
\text{Functionals} & f & ::= \lambda x.\, e \mid f : A \to B \\
\text{Values} & v & ::= \mathsf{Top} \mid i \mid f : A \to B \mid v_1 \,,, v_2 \mid \{l = v\} \\
\text{Term contexts} & \Gamma & ::= \cdot \mid \Gamma, x : A \\
\text{Frames} & F & ::= (\lambda x.\, e)\, \square \mid \square\, e \mid \square : A \mid v \,,, \square \mid \square \,,, e \mid \{l = \square\} \mid \square.l \\
\text{Syntactic sugar} & & \{l_1 : A_1;\ \ldots;\ l_n : A_n\} \triangleq \{l_1 : A_1\} \,\&\, \cdots \,\&\, \{l_n : A_n\} \\
 & & \{l_1 = e_1;\ \ldots;\ l_n = e_n\} \triangleq \{l_1 = e_1\} \,,, \ldots \,,, \{l_n = e_n\}
\end{array}
$$

Fig. 2. The syntax of the λM calculus.

In this example we omit the treatment of self references for simplicity of presentation. The idea is that `circle` models the corresponding trait. To model the inheritance of a super trait, we simply use the merge operator to merge `super` with the new methods. Furthermore, note that it is easy to model multiple inheritance. For instance, we could modify the program above to take two super traits `super1` and `super2` as arguments instead of `super`. Then we could simply use the merge operator to compose all the super traits (`super1 , , super2`) and then merge that with the additional methods. Note also, that in this case `super` has an unknown type. We create an object by calling `circle` with a superclass, which for this example is empty. Then we call the area method in the object, to obtain the area as a result. If we change the second line to:

```
let obj = circle ({area = (λp. 1) : ⋆ → int}) in
```

with a super trait containing a conflicting `area` method, then an ambiguity error is raised at runtime for the program. Since the addition of the unknown type is essentially orthogonal to the encoding, we can simply reuse previous encodings in $\lambda M^{\star}$ to model a source language with first-class traits or classes. Thus we omit a formal treatment of the encoding in this paper. For the formal treatment of the encoding, and its full details, including the treatment of self-references, we refer the reader to previous work on encoding first-class traits [Bi and Oliveira 2018; Zhang et al. 2021].

3 The λM Calculus: Syntax, Typing and Semantics

This section introduces the λM calculus: a variant of the λ_i calculus [Huang et al. 2021; Oliveira et al. 2016]. The main change is the adoption of lazy dynamic semantics for annotations on higher-order values. The λM calculus is the static counterpart of the gradually typed calculus in Section 4.

3.1 Syntax

The syntax of the λM calculus is shown in Figure 2. Meta-variables A, B, and C range over types. There are base types (Int), the greatest type ($\top$), the least type ($\bot$) and compound types. Compound types are function types ($A \to B$) or intersection types ($A \,\&\, B$). A single field record type $\{l : A\}$ has a field l with type A. Multi-field record types are encoded by intersections of single field record types [Reynolds 1997]. Ordinary types (A°) are types that are not intersection types, the top type or the bottom type. They are the types of atomic values appearing in a merge.

Meta-variable e ranges over expressions. Most expressions are typical: variables (x); integers (i); a canonical top value (Top); annotated expressions ($e : A$); applications ($e_1\, e_2$); lambda expressions ($\lambda x.\, e$) and fixpoints ($\mathsf{fix}\, x.\, e$). The merge of expressions e_1 and e_2 is denoted by $(e_1 \,,, e_2)$. A record $\{l = e\}$ stands for a single field record with label l and expression e. Selection of record fields is done by the projection expression $e.l$. A merge of single field records encodes multi-field records.

Typing modes $\quad \Leftrightarrow ::= \Rightarrow | \Leftarrow$

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

$$\frac{\vdash B \qquad \Gamma \vdash e \Rightarrow A \qquad A <: B}{\Gamma \vdash e \Leftarrow B}\ \textsc{Typ-sub} \qquad \frac{\Gamma \vdash e_1 \Rightarrow A \to B \qquad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\, e_2 \Rightarrow B}\ \textsc{Typ-app} \qquad \frac{\Gamma, x : A \vdash e_1 \Leftarrow B \qquad \Gamma \vdash e_2 \Rightarrow A}{\Gamma \vdash (\lambda x.\, e_1)\, e_2 \Leftarrow B}\ \textsc{Typ-rt}$$

$$\frac{}{\Gamma \vdash \mathsf{Top} \Rightarrow \top}\ \textsc{Typ-top} \qquad \frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash e : A \Rightarrow A}\ \textsc{Typ-anno} \qquad \frac{\Gamma \vdash e \Rightarrow A}{\Gamma \vdash \{l = e\} \Rightarrow \{l : A\}}\ \textsc{Typ-rcd}$$

$$\frac{}{\Gamma \vdash i \Rightarrow \mathsf{Int}}\ \textsc{Typ-lit} \qquad \frac{\vdash A \qquad \Gamma, x : A \vdash e \Leftarrow B}{\Gamma \vdash \lambda x.\, e \Leftarrow A \to B}\ \textsc{Typ-abs} \qquad \frac{\vdash A \qquad \Gamma, x : A \vdash e \Leftarrow A}{\Gamma \vdash \mathsf{fix}\, x.\, e \Leftarrow A}\ \textsc{Typ-fix}$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A}\ \textsc{Typ-var} \qquad \frac{A * B \qquad \Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B}{\Gamma \vdash e_1 \,,\, e_2 \Rightarrow A \,\&\, B}\ \textsc{Typ-merge} \qquad \frac{A \bullet l \triangleright B \qquad \Gamma \vdash e \Rightarrow A}{\Gamma \vdash e.l \Rightarrow B}\ \textsc{Typ-proj}$$

Fig. 3. The type system of the λM calculus.

Meta-variable f ranges over functionals, which are lambdas with zero or more function type annotations. Meta-variable v ranges over values. Values include: integers i; the top value Top; annotated functionals $f : A \to B$; a merge of values $v_1 \,,\, v_2$ and records $\{l = v\}$. This is different from the λ_i calculus, where functional values only have a single annotation. This change is made to delay the combination of function type annotations, to help gradualizing the calculus. Typing context Γ tracks bound variables x with their type A. Meta-variable F ranges over frames [Siek et al. 2015a]. The frame is mostly standard but it includes annotated expressions, and merges. Additionally, the frame for application $(\lambda x.\, e)\ \square$ restricts the function to be an unannotated lambda, as applications of annotated lambdas are eliminated by annotating both the argument and the entire application.

3.2 Bidirectional Typing

Like λ_i, we use bidirectional type checking, to avoid a general subsumption rule. As shown by previous work, a general subsumption rule is known to cause ambiguity in the presence of a merge operator [Huang et al. 2021; Oliveira et al. 2016]. The typing judgment is represented as $\Gamma \vdash e \Leftrightarrow A$. The typing mode $\Leftrightarrow$ is a metavariable, whose definition is shown at the top of Figure 3, and is either inference ($\Rightarrow$) or checking ($\Leftarrow$). As in λ_i, besides disallowing non-disjoint merges, we do not support unrestricted intersections, which means that expressions like $1 : \mathsf{Int} \,\&\, \mathsf{Int}$, where the intersection in the type annotation is not disjoint, are not allowed.

Typing Relation. The typing relation of the λM calculus is shown in Figure 3. Most of the rules follow the bidirectional type system of the λ_i calculus. In these rules, to avoid the ambiguity introduced by the merge operator, the disjointness restriction on rule Typ-merge is used to reject examples such as $1 \,,\, 2$. The disjointness restriction applies to any types. We define an auxiliary judgement $\vdash A$, adopted from Oliveira et al. [2016], which defines well-formed types. The full relation is mostly straightforward and shown in the extended version of the paper. The only notable rule imposes a disjointness restriction on all intersection types. There is also a standard (omitted) relation that checks if contexts are well-formed (i.e. all bound variables have well-formed types).

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

$$\frac{A_1 <: A_2}{\{l : A_1\} <: \{l : A_2\}}\ \text{S-RCD} \qquad \frac{}{\text{Int} <: \text{Int}}\ \text{S-Z} \qquad \frac{}{A <: \top}\ \text{S-TOP} \qquad \frac{B_1 <: A_1 \qquad A_2 <: B_2}{A_1 \to A_2 <: B_1 \to B_2}\ \text{S-ARR}$$

$$\frac{A_1 <: A_2 \qquad A_1 <: A_3}{A_1 <: A_2 \,\&\, A_3}\ \text{S-AND} \qquad \frac{A_1 <: A_3}{A_1 \,\&\, A_2 <: A_3}\ \text{S-ANDL} \qquad \frac{A_2 <: A_3}{A_1 \,\&\, A_2 <: A_3}\ \text{S-ANDR} \qquad \frac{}{\bot <: A}\ \text{S-BOT}$$

$\boxed{A \sqcup B}$ *(Common Ordinary Super Types (COST))*

$$\frac{}{\text{Int} \sqcup \text{Int}}\ \text{CO-INT} \qquad \frac{}{\bot \sqcup \bot}\ \text{CO-BOT} \qquad \frac{}{\bot \sqcup A^{\circ}}\ \text{CO-BO} \qquad \frac{}{A^{\circ} \sqcup \bot}\ \text{CO-OB}$$

$$\frac{}{(A_1 \to B_1) \sqcup (A_2 \to B_2)}\ \text{CO-ARR} \qquad \frac{}{\{l : A\} \sqcup \{l : B\}}\ \text{CO-RCD} \qquad \frac{A_1 \sqcup B}{(A_1 \,\&\, A_2) \sqcup B}\ \text{CO-ANDL}$$

$$\frac{A_2 \sqcup B}{(A_1 \,\&\, A_2) \sqcup B}\ \text{CO-ANDR} \qquad \frac{A \sqcup B_1}{A \sqcup (B_1 \,\&\, B_2)}\ \text{CO-RANDL} \qquad \frac{A \sqcup B_2}{A \sqcup (B_1 \,\&\, B_2)}\ \text{CO-RANDR}$$

Fig. 4. Subtyping and the COST.

Furthermore, we add two more typing rules for records and projections. The typing rule for single field records is standard (rule Typ-rcd). The type of a projection $e.l$ is obtained by inferring the type A of the expression being projected, and extracting the field type from A (rule Typ-proj) using an auxiliary relation $A \bullet l \triangleright B$, which is shown in the extended version of the paper. Finally, there is a typing rule Typ-rt that is only needed for proofs, and is used to type-check terms that only arise in intermediate steps of reduction. Since lambdas do not have annotations in beta reduction, the type information is obtained from the arguments.

3.3 Subtyping and Disjointness

Subtyping. The subtyping rules, which are mostly standard, are shown at the top of Figure 4. Our rules follow the formalization by Davies and Pfenning [2000] but with an additional rule S-rcd to incorporate record types. The extended subtyping relation is reflexive and transitive.

Disjointness. Our specification of disjointness follows one of the definitions in the original λ_i:

Definition 3.1 (Disjointness Specification). $A *_{\text{spec}} B \equiv \forall C, A <: C \land B <: C \implies \top <: C$

This definition implies that the values that inhabit the two types cannot have overlapping types, with the exception of top values. Such top values do not cause ambiguity because there is only one canonical value of type top [Alpuim et al. 2017]. Furthermore, we define a simpler algorithmic formulation based on a relation that checks whether two types have common ordinary super types (COST). To define the algorithmic formulation of disjointness, the Common Ordinary Super Types Relation (COSTR) $A \sqcup B$ is presented in the bottom of Figure 4. In essence values with ordinary types are the atomic components (i.e. they cannot themselves be merges) of merges. If two types have a COST then they overlap. For example Int & Bool and Int have the COST Int. When two types have a COST in common they cannot be disjoint, since we can obtain a different value with the same overlapping type from each value of the two types. Firstly, note that the top type is a common supertype of every other type, but it is not a COST (since the top type is not ordinary). Most rules are intuitive. One rule that deserves explanation rule co-arr: two functions have at

least one COST: $\bot \to \top$. Thus, functions cannot be disjoint. For intersections, when $A_1 \,\&\, A_2$ and one of the types A_1 or B_1 share ordinary supertypes with the other type B, we can easily conclude that $A_1 \,\&\, A_2$ has a COST with B (rules CO-ANDL, CO-ANDR, CO-RANDL, and CO-RANDR). With the help of the COSTR relation, an equivalent algorithmic formulation definition of disjointness is:

Definition 3.2 (Algorithmic Disjointness). $A * B \equiv \neg(A \sqcup B)$

3.4 Dynamic Semantics

The dynamic semantics of λM employs a type-directed operational semantics (TDOS) [Huang and Oliveira 2020]. In TDOS, besides the usual reduction relation, there is a special *casting* relation for values that is used to convert values to the specified type. Casting is used by the TDOS reduction relation, and it essentially gives an interpretation of type coercions at runtime.

Casting. The casting rules are shown at the top of Figure 5. Most of the rules directly follow λ_i. Rule CAST-TOP and rule CAST-LIT reduce the values according to the cast type. The main difference, compared to λ_i, is in rule CAST-ABS, which now employs a lazy semantics: functions accumulate the casting function type $(C \to D)$ to the functional value. We return a record value after casting the field value under the field type (rule CAST-RCD). Rule CAST-MERGEL and rule CAST-MERGER select a value from a merge of values $(v_1 ,, v_2)$ using an ordinary type A. Rule CAST-AND splits the intersection type used for the cast, and casts the value and each type separately.

Properties of Casting. Most of the properties of casting of λ_i hold here as well, and most proofs are proved by induction on the casting derivation.

Some important properties of the casting relation are shown next.

Lemma 3.3 (Casting Determinism). *If* $\cdot \vdash v \Leftarrow B, v \hookrightarrow_A v_1$ *and* $v \hookrightarrow_A v_2$ *then* $v_1 = v_2$.

Lemma 3.4 (Casting Preservation). *If* $\cdot \vdash v \Leftarrow B, \vdash A$ *and* $v \hookrightarrow_A v'$ *then* $\cdot \vdash v' \Rightarrow A$.

Lemma 3.5 (Casting Progress). *If* $\cdot \vdash v \Leftarrow A$ *then* $\exists v', v \hookrightarrow_A v'$.

Lemma 3.3 says that the result of casting is unique. Note that the determinism lemma is non-trivial and only holds for well-typed values. Its proof requires reasoning about the properties of well-typed values. The casting relation preserves the type of the cast (Lemma 3.4), and there always exists a result when the value is cast under A (Lemma 3.5).

Reduction. The reduction rules are shown at the bottom of Figure 5. Rule STEP-EVAL is a standard rule for evaluation contexts. Dealing with applications and beta reduction is interesting and different from λ_i. Firstly, rule STEP-BETA is standard beta reduction. Secondly, the top-level function annotation is eliminated by annotating the input types for arguments and output types for applications (rule STEP-APP). In rule STEP-ANNOV, annotated values $v : A$ are evaluated by casting them under the annotated types. However, $(v : A)$ can be a (functional) value. In such case, since the expression is already a value, it should not be reduced. Thus, we require the condition NotVal $(v : A)$ which is defined as not a functional value: NotVal $e \equiv e \neq (f : A \to B)$. Fixpoints substitute themselves in the body (rule STEP-FIX). Rule STEP-PROJ is for projections of record values. To project the field value, we cast the value v by the record type $\{l : A\}$. The field type A is obtained by projecting the dynamic type of v by projection label l. The dynamic type for values $\mathsf{ty}(v)$ is:

$$\mathsf{ty}(i) = \mathsf{Int} \qquad \mathsf{ty}(\mathsf{Top}) = \top \qquad \mathsf{ty}((f : A \to B)) = A \to B$$
$$\mathsf{ty}(\{l = p\}) = \{l : \mathsf{ty}(p)\} \qquad \mathsf{ty}((v_1 ,, v_2)) = (\mathsf{ty}(v_1)) \,\&\, (\mathsf{ty}(v_2))$$

An important property of a well-typed value is that its dynamic type is the inferred type of a value.

Lemma 3.6 (Dynamic Types). *For any value* v, *if* $\cdot \vdash v \Rightarrow A$ *then* $\mathsf{ty}(v) = A$.

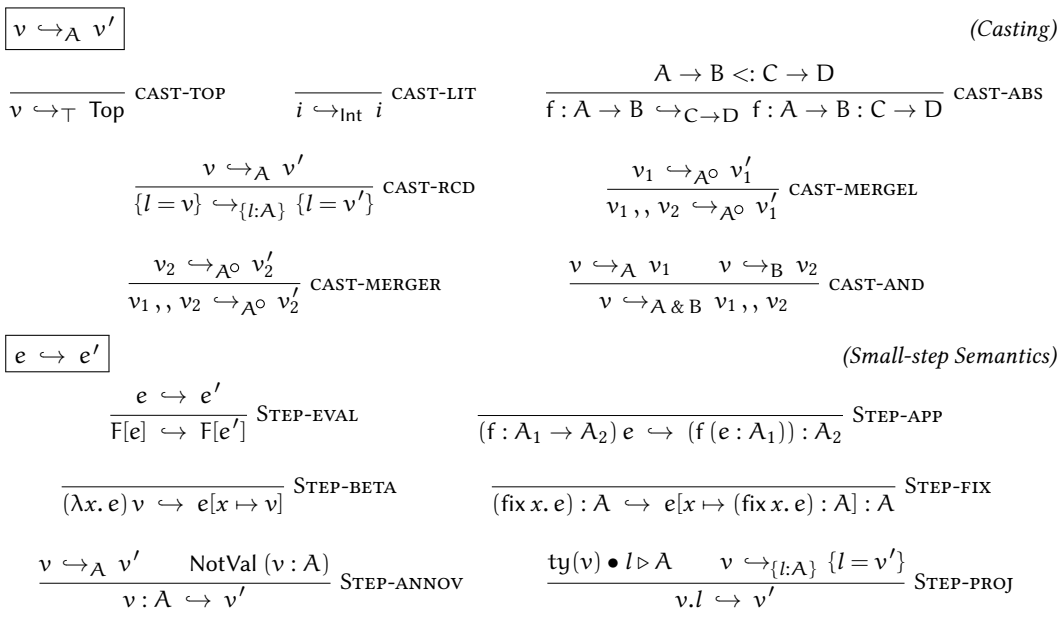

Fig. 5. Casting and small-step semantics for λM.

Finally, the λM calculus is *deterministic* and *type sound*:

Theorem 3.7 (Determinism). *If* $\cdot \vdash e \Leftrightarrow A, e \hookrightarrow e_1$ *and* $e \hookrightarrow e_2$ *then* $e_1 = e_2$.

Theorem 3.8 (Type Preservation). *If* $\cdot \vdash e \Leftrightarrow A$ *and* $e \hookrightarrow e'$ *then* $\cdot \vdash e' \Leftrightarrow A$.

Theorem 3.9 (Progress). *If* $\cdot \vdash e \Rightarrow A$ *then* e *is a value or* $\exists e', e \hookrightarrow e'$.

4 The λM⋆ Calculus : Syntax, Typing and Semantics

This section introduces the λM⋆ calculus, the gradual counterpart of λM. We prove determinism and type soundness. Section 5 presents the gradual typing criteria satisfied by λM⋆.

4.1 Syntax

The syntax of λM⋆ calculus is shown in Figure 6. Types extend the types of λM calculus with the unknown type ($\star$). Because λM⋆ is gradually typed, runtime type errors are possible. Runtime type errors are denoted as err_t for type errors, and err_a for ambiguity errors. We use err_* when the type of the error is not important or inferred from the context. Results (r) can be any expressions or an error err_*. Meta-variable s ranges over ordinary values. Ordinary values include: integers i; the top value Top; functional values and records with a value field $\{l = v\}$. Meta-variable g ranges over ground values. Ground values are values with ground types ($\top$, Int or dynamic compound types such as $\star \,\&\, \star$). Meta-variable v stands for well-formed values. Values are either ordinary values s, a merge of values $v_1 ,, v_2$ or ground values annotated with unknown type $(g : \star)$. Compared to the λM calculus, we extend values with dynamic ground values $(g : \star)$.

To encode dynamically typed lambdas (i.e. lambdas without static type information) we need to insert $\star$ annotations. This approach is similar to the approach used in GTLC [Siek and Taha 2006], where an unannotated lambda $\lambda x.\, e$ is syntactic sugar for $\lambda(x : \star).\, e$. While we could apply a

Types	$A, B, C ::= \mathsf{Int} \mid \top \mid \bot \mid A \to B \mid A \,\&\, B \mid \{l : A\} \mid \star$
Expressions	$e ::= x \mid i \mid \mathsf{Top} \mid \lambda x.\, e \mid \{l = e\} \mid e.l \mid e : A \mid e_1\, e_2 \mid e_1 ,, e_2 \mid \mathsf{fix}\, x.\, e$
Results	$r ::= e \mid \mathsf{err}_*$
Functionals	$f ::= \lambda x.\, e \mid f : A \to B$
Ordinary values	$s ::= \mathsf{Top} \mid i \mid f : A \to B \mid \{l = v\}$
Ground values	$g ::= \mathsf{Top} \mid i \mid \lambda x.\, e : \star \to \star \mid f : \star \to \star \mid \{l = g : \star\} \mid g : \star ,, g : \star$
Values	$v ::= s \mid v_1 ,, v_2 \mid g : \star$
Term contexts	$\Gamma ::= \cdot \mid \Gamma, x : A$
Frames	$F ::= (\lambda x.\, e)\, \square \mid \square\, e \mid v ,, \square \mid \square ,, e \mid \{l = \square\} \mid \square.l \mid \square : A$

Fig. 6. Syntax of the λM⋆ calculus.

similar transformation for λM⋆ terms, simply adding a ⋆ annotation in non-annotated lambdas, we can do better in λM⋆ because of bidirectional type checking. We only need to insert annotations on unannotated lambdas that are in *inference* positions. For example, given the dynamically typed expression $(\lambda f.\lambda x.\, f\, x)(\lambda y.y)$ we can obtain a well-typed λM⋆ program by automatically annotating only one lambda abstraction: $((\lambda f.\lambda x.\, f\, x) : \star)(\lambda y.y)$. Bidirectional type checking can propagate type information to lambdas in checking positions. So, while those lambdas are unannotated, they are still statically typed. This idea extends to dynamically typed fixpoints, which can be annotated in a similar way. We show the details of this sugaring process in the extended version of the paper.

4.2 Consistent Subtyping and Disjointness

Consistent Subtyping. To integrate the type consistency and subtyping relations in gradual typing, we follow the consistent subtyping approach in Xie et al. [2019]'s work, which was inspired by an earlier approach by Siek and Taha [2007]. The type consistency rules are at the top of Figure 7. They are standard and proved to be reflexive and symmetric but not transitive. The subtyping rules extend the subtyping rules of λM with a rule for dynamic types (rule S-DYN), where a dynamic type is only a subtype of itself. Following Xie et al.'s approach, we add a premise in rule S-TOP, which restricts type A to be static. The subtyping rules are also reflexive and transitive.

The definition of consistent subtyping is supported by subtyping and consistency. Our consistent subtyping relation is extended with intersection types and (single field) record types, and is shown in Figure 7. Consistent subtyping is proved to be equivalent to the declarative formulation of consistent subtyping proposed by Xie et al. [2019]:

LEMMA 4.1 (CONSISTENT SUBTYPING). $A \lesssim B \triangleq \exists A'\, B'.\, A <: A'$ *and* $A' \sim B'$ *and* $B' <: B$.

This specification defines consistent subtyping in terms of type consistency and subtyping, and is a useful guideline for the design of consistent subtyping relations. Note that, compared to the subtyping relation, all the rules are essentially the same, with the exception of rules CS-DYNL, CS-DYNR, and CS-TOP which have a different treatment from subtyping.

Disjointness and COSTR. To establish the specification of gradual disjointness ($A *_{\text{spec}} B$), we draw inspiration from AGT and lift the disjointness definition from λM, as follows:

Definition 4.2 (Disjointness Specification). $A *_{\text{spec}} B \equiv \exists\, \mathsf{Static}\, A'\, B'.\, A' \sqsubseteq A \;\wedge\; B' \sqsubseteq B \wedge (\forall\, C,\; A' <: C \;\wedge\; B' <: C \implies \top <: C)$

We use an adapted version of the existential lifting of predicates, which relies on the precision relation $\sqsubseteq$ between types, defined in Figure 7. Every type is more precise than itself and the unknown type ⋆. The remaining rules are defined inductively. The original AGT existential lifting of predicates

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

$$\frac{}{\mathsf{Int} \sim \mathsf{Int}}\ \textsc{sim-i} \qquad \frac{}{\top \sim \top}\ \textsc{sim-top} \qquad \frac{}{\bot \sim \bot}\ \textsc{sim-bot} \qquad \frac{A \sim C \qquad B \sim D}{A \to B \sim C \to D}\ \textsc{sim-arr}$$

$$\frac{}{\star \sim A}\ \textsc{sim-dynl} \qquad \frac{}{A \sim \star}\ \textsc{sim-dynr} \qquad \frac{A \sim C \qquad B \sim D}{A \,\&\, B \sim C \,\&\, D}\ \textsc{sim-merge} \qquad \frac{A_1 \sim A_2}{\{l : A_1\} \sim \{l : A_2\}}\ \textsc{sim-rcd}$$

$\boxed{A <: B}$ *(Additional or Changed Subtyping Rules)*

$$\frac{\mathsf{Static}\ A}{A <: \top}\ \textsc{S-Top} \qquad \frac{}{\star <: \star}\ \textsc{S-dyn}$$

$\boxed{A \lesssim B}$ *(Consistent Subtyping)*

$$\frac{}{\mathsf{Int} \lesssim \mathsf{Int}}\ \textsc{CS-z} \qquad \frac{}{\star \lesssim A}\ \textsc{CS-dynl} \qquad \frac{}{A \lesssim \star}\ \textsc{CS-dynr} \qquad \frac{}{\bot \lesssim A}\ \textsc{CS-bot} \qquad \frac{}{A \lesssim \top}\ \textsc{CS-top}$$

$$\frac{B_1 \lesssim A_1 \qquad A_2 \lesssim B_2}{A_1 \to A_2 \lesssim B_1 \to B_2}\ \textsc{CS-arr} \qquad \frac{A_1 \lesssim A_3}{A_1 \,\&\, A_2 \lesssim A_3}\ \textsc{CS-andl} \qquad \frac{A_2 \lesssim A_3}{A_1 \,\&\, A_2 \lesssim A_3}\ \textsc{CS-andr}$$

$$\frac{A_1 \lesssim A_2 \qquad A_1 \lesssim A_3}{A_1 \lesssim A_2 \,\&\, A_3}\ \textsc{CS-and} \qquad \frac{A_1 \lesssim A_2}{\{l : A_1\} \lesssim \{l : A_2\}}\ \textsc{CS-rcd}$$

$\boxed{A \sqsubseteq B}$ *(Type Precision)*

$$\frac{}{A \sqsubseteq A}\ \textsc{tp-refl} \qquad \frac{}{A \sqsubseteq \star}\ \textsc{tp-dyn} \qquad \frac{A_1 \sqsubseteq A_2 \qquad B_1 \sqsubseteq B_2}{(A_1 \to B_1) \sqsubseteq (A_2 \to B_2)}\ \textsc{tp-abs}$$

$$\frac{A_1 \sqsubseteq A_2 \qquad B_1 \sqsubseteq B_2}{A_1 \,\&\, B_1 \sqsubseteq A_2 \,\&\, B_2}\ \textsc{tp-and} \qquad \frac{A_1 \sqsubseteq A_2}{\{l : A_1\} \sqsubseteq \{l : A_2\}}\ \textsc{tp-rcd}$$

Fig. 7. Consistency, Subtyping, Consistent Subtyping and Type Precision.

is as follows: $\widetilde{P}(A, B) = \exists\, \mathsf{Static}\ A' \in \gamma(A), \mathsf{Static}\ B' \in \gamma(B).P(A', B')$, where γ represents a concretization function that maps gradual types to set of static types. As the precision relation in AGT is also defined in terms of concretization ($A \sqsubseteq B \equiv \gamma(A) \subseteq \gamma(B)$), the existential lifting of predicates can be equivalently expressed as $\widetilde{P}(A, B) = \exists\, \mathsf{Static}\ A' \sqsubseteq A, \mathsf{Static}\ B' \sqsubseteq B.P(A', B')$. We provide a simplified definition of Def. 4.2 in the extended version of the paper, which does not use existentials (proving its equivalence). Finally, the algorithmic definition of disjointness is syntactically identical to the one in λM (as $\star$ is not related to any other gradual type in COSTR): $A * B \equiv \neg(A \sqcup B)$. This definition has been proven to be equivalent to both formal specifications.

4.3 Bidirectional Typing

As in the λM calculus, bidirectional typing is used. The typing rules are almost the same as those used by the λM calculus in Figure 3. New rules, or rules that are changed are shown next.

$$\textsc{Typ-cs}\ \frac{\vdash B \qquad A \lesssim B \qquad \Gamma \vdash e \Rightarrow A}{\Gamma \vdash e \Leftarrow B} \qquad \textsc{Typ-app}\ \frac{A \triangleright A_1 \to A_2 \qquad \Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Leftarrow A_1}{\Gamma \vdash e_1\, e_2 \Rightarrow A_2}$$

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

In gradual typing, the unknown type is used to allow some programs with runtime type errors. We allow these kind of programs by changing the subsumption rule (rule Typ-sub) in Figure 3 to rule Typ-cs, which now uses consistent subtyping instead of subtyping. For example, $1 : \star : \mathsf{Bool}$ and $\mathsf{True} : \star : \mathsf{Int}$ are allowed by the new rule Typ-cs. Application and lambdas (rules Typ-app and Typ-abs) use the *match* partial operator to deduce a function type from a gradual type. This operator is defined as $\star \triangleright \star \to \star$, and $A_1 \to A_2 \triangleright A_1 \to A_2$. For projections, we allow programs such as $((\{l_1 = 1\} ,, \mathsf{True}) : \star).l_2$ and $((1 ,, \mathsf{True}) : \star).l$. The presence of a dynamic type relaxes the type checker to allow projections from expressions with type $\star$. In the case that the label being projected does not exist a runtime error is raised. The definitions of projection typing ($A \bullet l \triangleright B$) and well-formed types for $\lambda M^{\star}$ are shown in the extended version.

4.4 Casting

The casting rules are shown in the Figure 8. Because of runtime errors, the casting judgement $v \hookrightarrow_A r$ returns a result (r), which contains values v, type errors err_t and ambiguous errors err_a.

Casting ordinary values. Rule cast-top, rule cast-lit and rule cast-rcd are the same as λM calculus. To adapt to a gradual calculus, the subtyping premise of rule cast-Abs is updated to account for consistent subtyping.

Casting merges and intersection types. Rule cast-and mimics its static counterpart: it casts the value to both A and B. However, it also handles ambiguity errors and type errors. To achieve this, this rule utilizes the $r_1 \wedge r_2 = r_3$ meta-function defined at the bottom right side of Figure 8. The cast reduces to an error if either of the results is an error, giving priority to type errors to maintain determinism with respect to rule cast-err. Otherwise, it merges both results. Rule cast-merge handles the case where a merge is cast to an ordinary type. Compared to λM, as both components of a merge can have imprecise type annotations, the ordinary type can be a consistent supertype of both types (e.g. $(1 : \star, \mathsf{True})$ cast to Bool). Thus, we need to check *dynamically* if there is no ambiguity (e.g. $(1 : \star, 2)$ cast to Int). This rule first casts both components of the merge and then combines the results using the meta-function $r_1 \vee r_2 = r_3$ defined at the bottom left side of Figure 8. The cast reduces to a value if either both components reduce to the same value or one component reduces to a value and the other to a type error. For example, if we cast $(1 : \star, \mathsf{True})$ to Int, the left and right components reduce to 1 and err_t respectively, so we reduce to 1. Similarly, $(1 : \star ,, 1)$ cast to Int reduces to 1 as both branches reduce to the same value. However, in cases like $(1 : \star ,, 2) ,, 3$ cast to Int, the left component would result in an ambiguity error and the right component would yield 3. Instead of wrongly keeping the right component, we yield an ambiguity error. In other words, contrary to rule cast-and, we prioritize ambiguity errors over type errors.

Casting to and from unknown. Rules cast-sd and cast-mergd cast values to $\star$. In rule cast-sd, ordinary values are cast to the top-level constructor of their type with the $\mathsf{ground}(A)$ function:

$$\mathsf{ground}(\top) = \top \quad \mathsf{ground}(\mathsf{Int}) = \mathsf{Int} \quad \mathsf{ground}(A \to B) = \star \to \star \quad \mathsf{ground}(\{l : A\}) = \{l : \star\}$$

The result of this cast is a ground value, annotated with the $\star$ type to preserve types. We cast to a ground type, instead of just annotating the value with $\star$ directly, to allow dropping the $\star$ type when the ground value is used. For example, $\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}$ cast to $\star$ returns $\lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \star \to \star : \star$ then, if the value is cast to $\mathsf{Bool} \to \mathsf{Bool}$, the $\star$ type can be dropped safely to obtain $\lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \star \to \star : \mathsf{Bool} \to \mathsf{Bool}$. On the contrary, rule cast-mergd does not cast the merge value $(v_1 ,, v_2)$ to the type-level constructor $\star \,\&\, \star$. Otherwise it would create a cycle with rule cast-and. Consider program $(1 ,, \mathsf{True})$ cast to $\star$. If we cast $(1 ,, \mathsf{True})$ to $\star \,\&\, \star$, then by rule cast-and, we would yield casting from $(1 ,, \mathsf{True})$ to $\star$ again, forming a cycle. Therefore, we cast v_1 and v_2 to $\star$ separately. For example, if $(1 ,, \lambda x.\, x : \mathsf{Int} \to \mathsf{Int})$ is cast to $\star$, the dynamic

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

$$\frac{}{v \hookrightarrow_\top \mathsf{Top}}\ \textsc{cast-top} \qquad \frac{}{i \hookrightarrow_{\mathsf{Int}} i}\ \textsc{cast-lit} \qquad \frac{v \hookrightarrow_A v'}{\{l = v\} \hookrightarrow_{\{l:A\}} \{l = v'\}}\ \textsc{cast-rcd}$$

$$\frac{A \to B \lesssim C \to D}{f : A \to B \hookrightarrow_{C \to D} f : A \to B : C \to D}\ \textsc{cast-Abs} \qquad \frac{v_1 \hookrightarrow_{A^\circ} r_1 \qquad v_2 \hookrightarrow_{A^\circ} r_2}{v_1 ,, v_2 \hookrightarrow_{A^\circ} r_1 \vee r_2}\ \textsc{cast-merge}$$

$$\frac{v \hookrightarrow_A r_1 \qquad v \hookrightarrow_B r_2}{v \hookrightarrow_{A \,\&\, B} r_1 \wedge r_2}\ \textsc{cast-And} \qquad \frac{s \hookrightarrow_{\mathsf{ground}(\mathsf{ty}(s))} g}{s \hookrightarrow_\star g : \star}\ \textsc{cast-sd}$$

$$\frac{v_1 \hookrightarrow_\star v_1' \qquad v_2 \hookrightarrow_\star v_2'}{v_1 ,, v_2 \hookrightarrow_\star (v_1' ,, v_2') : \star}\ \textsc{cast-mergd} \qquad \frac{}{g : \star \hookrightarrow_\star g : \star}\ \textsc{cast-dd} \qquad \frac{g \hookrightarrow_{A^\circ} r}{g : \star \hookrightarrow_{A^\circ} r}\ \textsc{cast-dyna}$$

$$\frac{\mathsf{ty}(v) \not\lesssim A}{v \hookrightarrow_A \mathsf{err}_t}\ \textsc{cast-err} \qquad \frac{}{v \hookrightarrow_\bot \mathsf{err}_t}\ \textsc{cast-bot} \qquad \frac{v \hookrightarrow_A \mathsf{err}_*}{\{l = v\} \hookrightarrow_{\{l:A\}} \mathsf{err}_*}\ \textsc{cast-rcdp}$$

$v \vee v$	=	v	$v_1 \wedge v_2$	=	$v_1 ,, v_2$
$v_1 \vee v_2$	=	err_a where $v_1 \neq v_2$	$\mathsf{err}_a \wedge \mathsf{err}_a$	=	err_a
$\mathsf{err}_t \vee r = r$		$r \vee \mathsf{err}_t = r$	$r \wedge \mathsf{err}_t = \mathsf{err}_t$		$\mathsf{err}_t \wedge r = \mathsf{err}_t$
$\mathsf{err}_a \vee r = \mathsf{err}_a$		$r \vee \mathsf{err}_a = \mathsf{err}_a$	$v \wedge \mathsf{err}_a = \mathsf{err}_a$		$\mathsf{err}_a \wedge v = \mathsf{err}_a$

Fig. 8. Casting for the λM⋆ calculus.

annotated value $(1 : \star ,, (\lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \star \to \star : \star)) : \star$ is returned. Rule CAST-DD returns itself since the value being cast already has type $\star$. Finally, rule CAST-DYNA casts dynamic ground values to an ordinary type A. When $(1 : \star ,, \mathsf{True}) : \star$ is cast to Int, it results in 1.

Casting to error. Rule CAST-ERR raises a type error if the dynamic type of value v is not a consistent subtype of the cast type. Rule CAST-BOT raises a type error when a value is cast to $\bot$, to cover the case when a value v of unknown type is cast to $\bot$. Finally, rule CAST-RCDP propagates errors when a cast on the underlying value of a record fails.

4.5 Reduction

The reduction rules are shown in Figure 9. Rule STEP-EVAL, rule STEP-ANNOV, rule STEP-BETA, rule STEP-APP and rule STEP-PROJ are the same as λM. However note that NotVal e is extended to: NotVal $e \equiv e \neq (f : A \to B) \wedge e \neq g : \star$. In gradually typed lambda calculi, errors may be raised at run-time. Therefore, rule STEP-BLAME is designed to deal with that case. Rule STEP-ANNOP can deal with the case where casting fails. Rule STEP-PROJP shows that we need to consider the case of projecting a value with unknown types, and the projection fails. There are three rules related to beta reduction: rule STEP-BETA, rule STEP-APP and rule STEP-DYN. Compared to λM, rule STEP-DYN is new. Because the unknown type $\star$ can be matched with $\star \to \star$ in applications $((g : \star)\, e)$, $(g : \star)$ should be annotated with $\star \to \star$ (rule STEP-DYN). Then the lambda abstraction can be extracted via casting (rule STEP-ANNOV) or filter the ill-typed values, which are hidden by the type $\star$ (rule STEP-ANNOP). For example, both $(1 : \star)\, 2$ and $((1 ,, \lambda x.\, x : \star \to \star) : \star)\, 2$ are well typed. For the expression $(1 : \star)\, 2$, a type error is detected when rule STEP-DYN annotates $(1 : \star)$ with $\star \to \star$ and the cast fails via rule STEP-ANNOV. However the lambda value $(\lambda x.\, x : \star \to \star)$ is extracted for the second expression with a $\star \to \star$ annotation after using rule STEP-ANNOV.

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

$$\frac{e \hookrightarrow e'}{F[e] \hookrightarrow F[e']}\ \textsc{Step-eval} \qquad \frac{e \hookrightarrow \mathrm{err}_*}{F[e] \hookrightarrow \mathrm{err}_*}\ \textsc{Step-blame} \qquad \frac{}{(\lambda x.\, e)\, v \hookrightarrow e[x \mapsto v]}\ \textsc{Step-beta}$$

$$\frac{}{(g : \star)\, e \hookrightarrow ((g : \star) : \star \to \star)\, e}\ \textsc{Step-dyn} \qquad \frac{\mathrm{ty}(v) \bullet l \triangleright A \qquad v \hookrightarrow_{\{l:A\}} \{l = v'\}}{v.l \hookrightarrow v'}\ \textsc{Step-proj}$$

$$\frac{\mathrm{ty}(v) \bullet l \triangleright A \qquad v \hookrightarrow_{\{l:A\}} \mathrm{err}_*}{v.l \hookrightarrow \mathrm{err}_*}\ \textsc{Step-projp} \qquad \frac{v \hookrightarrow_A v' \qquad \mathrm{NotVal}\,(v : A)}{v : A \hookrightarrow v'}\ \textsc{Step-annov}$$

$$\frac{}{(f : A_1 \to A_2)\, e \hookrightarrow (f\,(e : A_1)) : A_2}\ \textsc{Step-app} \qquad \frac{}{(\lambda x.\, e) : \star \hookrightarrow (\lambda x.\, e) : \star \to \star : \star}\ \textsc{Step-abs}$$

$$\frac{v \hookrightarrow_A \mathrm{err}_*}{v : A \hookrightarrow \mathrm{err}_*}\ \textsc{Step-annop} \qquad \frac{}{(\mathrm{fix}\, x.\, e) : A \hookrightarrow e[x \mapsto (\mathrm{fix}\, x.\, e) : A] : A}\ \textsc{Step-fix}$$

Fig. 9. Semantics of $\lambda M^\star$.

Properties of Reduction. The $\lambda M^\star$ calculus is deterministic and type sound. Theorem 4.3 says that the dynamic semantics is deterministic. Furthermore, the $\lambda M^\star$ calculus preserves types (Theorem 4.4), and it has progress (Theorem 4.5).

Theorem 4.3 (Determinism). *If* $\cdot \vdash e \Leftrightarrow A$, $e \hookrightarrow r_1$ *and* $e \hookrightarrow r_2$ *then* $r_1 = r_2$.

Theorem 4.4 (Type Preservation). *If* $\cdot \vdash e \Leftrightarrow A$ *and* $e \hookrightarrow e'$ *then* $\cdot \vdash e' \Leftrightarrow A$.

Theorem 4.5 (Progress). *If* $\cdot \vdash e \Rightarrow A$ *then* e *is a value or* $\exists r, e \hookrightarrow r$.

Example. Finally, an example to demonstrate how reduction in $\lambda M^\star$ works is:

$$\begin{aligned}
&(((1 : \star)\,,\, (\lambda x.\, (x : \mathsf{Int}) : \star \to \star : \star)) : \star)\, (1\,,\, \mathsf{Top}) \\
&\hookrightarrow^* \{\text{by rule } \textsc{Step-dyn}, \text{ rule } \textsc{Step-eval}, \text{ rule } \textsc{Step-app} \text{ and rule } \textsc{Step-annov}\} \\
&((\lambda x.\, (x : \mathsf{Int}) : \star \to \star)\, (1\,,\, \mathsf{Top}) : \star) : \star \\
&\hookrightarrow^* \{\text{by rule } \textsc{Step-eval}, \text{ rule } \textsc{Step-app}, \text{ rule } \textsc{Step-beta} \text{ and rule } \textsc{Step-annov}\} \\
&(1 : \star\,,\, \mathsf{Top} : \star) : \star : \mathsf{Int} : \star : \star \\
&\hookrightarrow^* \{\text{by rule } \textsc{Step-eval} \text{ and rule } \textsc{Step-annov}\} \\
&1 : \star
\end{aligned}$$

In this example, the lambda $(\lambda x.\, (x : \mathsf{Int}) : \star \to \star)$ is extracted by casting $((1 : \star)\,,\, (\lambda x.\, (x : \mathsf{Int}) : \star \to \star : \star))$ to $\star \to \star$. The argument $(1\,,\, \mathsf{Top})$ is cast with the function input type $\star$ to obtain $(1 : \star\,,\, \mathsf{Top} : \star) : \star$. Then the argument is substituted into the function body and cast to Int. Finally the expected result $1 : \star$ is returned.

5 Gradual Typing Criteria and Encoding GTFL

In this section, we show that $\lambda M^\star$ satisfies gradual typing criteria, and can encode the static semantics of $\mathrm{GTFL}_{\lesssim}$ [Garcia et al. 2016], which is a gradual calculus with records and subtyping. As we have mentioned in Section 2.4, we need to employ a variant of the dynamic gradual guarantee.

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

$$\frac{}{e \sqsubseteq e}\ \text{EP-REFL} \qquad \frac{e_1 \sqsubseteq e_2}{\lambda x.\, e_1 \sqsubseteq \lambda x.\, e_2}\ \text{EP-ABS} \qquad \frac{e_1 \sqsubseteq e_2}{\mathrm{fix}\, x.\, e_1 \sqsubseteq \mathrm{fix}\, x.\, e_2}\ \text{EP-FIX}$$

$$\frac{e_1 \sqsubseteq e_1' \qquad e_2 \sqsubseteq e_2'}{e_1\, e_2 \sqsubseteq e_1'\, e_2'}\ \text{EP-APP} \qquad \frac{A \sqsubseteq B \qquad e_1 \sqsubseteq e_2}{e_1 : A \sqsubseteq e_2 : B}\ \text{EP-ANNO}$$

$$\frac{e_1 \sqsubseteq e_1' \qquad e_2 \sqsubseteq e_2'}{e_1 ,, e_2 \sqsubseteq e_1' ,, e_2'}\ \text{EP-MERGE} \qquad \frac{e_1 \sqsubseteq e_2}{\{l = e_1\} \sqsubseteq \{l = e_2\}}\ \text{EP-RCD} \qquad \frac{e_1 \sqsubseteq e_2}{e_1.l \sqsubseteq e_2.l}\ \text{EP-PROJ}$$

Fig. 10. Expression precision.

5.1 Conservative Extension, Static Gradual Guarantee and GTFL Encoding

Conservative Extension of the Static Discipline. We proved that if an expression is well-typed in λM then it is well-typed in $\lambda M^\star$, which is shown in Theorem 5.1. Theorem 5.2 shows that for any well-typed expressions, the dynamic semantics of λM can be encoded in $\lambda M^\star$. Note that a fully-annotated expression means that all subexpressions are static, which are expressions in λM. To be distinguishable, we use subscript m to represent typing or reduction from λM.

Theorem 5.1 (Equivalence for fully-annotated terms (static)). *Suppose that* e *is fully annotated,* Γ *is well-formed and* Γ, A *are static.* $\Gamma \vdash e \Leftrightarrow_m A$ *if and only if* $\Gamma \vdash e \Leftrightarrow A$.

Theorem 5.2 (Equivalence for fully-annotated terms (dynamic)). *Suppose that* e *is fully annotated,* Γ *is well-formed and* Γ, A *are static. If* $\Gamma \vdash e \Leftrightarrow_m A$ *then* $e \hookrightarrow_m^* v \iff e \hookrightarrow^* v$.

Static Gradual Guarantee. $\lambda M^\star$ comes with a static gradual guarantee [Siek et al. 2015b], defined in terms of precision relations for types and expressions. We have already shown the precision for types. The precision relation for expressions is shown in Figure 10. $e_1 \sqsubseteq e_2$ means that e_1 is more precise than e_2. Most of the rules are inductive and derived from the precision relation of types. Theorem 5.3 shows that the static criteria of the gradual guarantee holds for the $\lambda M^\star$ calculus: if e is more precise than e', e has type A then e' has type B, and type A is more precise than B.

Theorem 5.3 (Static Gradual Guarantee of the $\lambda M^\star$ Calculus). *If* $e \sqsubseteq e'$ *and* $\cdot \vdash e \Leftrightarrow A$ *then* $\exists B, \cdot \vdash e' \Leftrightarrow B$ *and* $A \sqsubseteq B$.

Encoding the Static Semantics of the $\mathrm{GTFL}_{\lesssim}$ *Calculus.* We proved that $\lambda M^\star$ can encode the type system of the $\mathrm{GTFL}_{\lesssim}$ calculus [Bañados Schwerter et al. 2021; Garcia et al. 2016]. In other words every well-typed expression in the $\mathrm{GTFL}_{\lesssim}$ calculus can be translated into a well-typed expression in the $\lambda M^\star$ calculus. The dynamic (lazy) semantics of $\lambda M^\star$ does *not* preserve the (eager) semantics of $\mathrm{GTFL}_{\lesssim}$. Thus we do not prove an operational correspondence result. An important difference in the semantics is that the original semantics of $\mathrm{GTFL}_{\lesssim}$ [Garcia et al. 2016] fails to preserve some expected modular type invariants. However, as we discussed in Section 2, the $\lambda M^\star$ calculus is capable of smoothly dealing with the problem of modular type-based invariants.

The syntax and type system of $\mathrm{GTFL}_{\lesssim}$ are shown in the Figure 11. Its expressions are standard and the interesting part are the (gradual) types. Not only we have an unknown type $\star$, but also we have gradual rows ($\{\overline{l : S}, \star\}$), which represent rows with statically unknown extra fields. In addition, the syntactic sugar $\{\overline{l : S}, *\}$ is used to represent either a normal multi-field record type ($\{\overline{l : S}\}$) or gradual row types ($\{\overline{l : S}, \star\}$). The judgment $\bar{\Gamma} \vdash t : S \rightsquigarrow e$ has an elaboration step from $\mathrm{GTFL}_{\lesssim}$ expressions t to $\lambda M^\star$ expressions e in the gray portion of the judgement. This elaboration

Syntax

GTypes	$S ::= \mathsf{Int} \mid S_1 \to S_2 \mid \{\overline{l : S}\} \mid \{\overline{l : S}, \star\} \mid \star$
Expressions	$t ::= x \mid i \mid \lambda x : S.\, t \mid \{\overline{l = t}\} \mid t.l \mid t : S \mid t_1\ t_2$
Term contexts	$\bar{\Gamma} ::= \cdot \mid \bar{\Gamma}, x : S$

$\boxed{\bar{\Gamma} \vdash t : S \rightsquigarrow e}$ *(Typing)*

$$\frac{x : S \in \bar{\Gamma}}{\bar{\Gamma} \vdash x : S \rightsquigarrow x}\ \textsc{Aty-var} \qquad \frac{\bar{\Gamma}, x : S_1 \vdash t : S_2 \rightsquigarrow e}{\bar{\Gamma} \vdash \lambda x : S_1.\, t : S_1 \to S_2 \rightsquigarrow \lambda x.\, e : |S_1| \to |S_2|}\ \textsc{Aty-abs}$$

$$\frac{}{\bar{\Gamma} \vdash i : \mathsf{Int} \rightsquigarrow i}\ \textsc{Aty-i} \qquad \frac{S_3 \lesssim S_1 \qquad \bar{\Gamma} \vdash t_2 : S_3 \rightsquigarrow e_2 \qquad \bar{\Gamma} \vdash t_1 : S_1 \to S_2 \rightsquigarrow e_1}{\bar{\Gamma} \vdash t_1\ t_2 : S_2 \rightsquigarrow e_1\ e_2}\ \textsc{Aty-app}$$

$$\frac{\bar{\Gamma} \vdash t : S \rightsquigarrow e}{\bar{\Gamma} \vdash t.l : \widetilde{\mathsf{proj}}(S, l) \rightsquigarrow e.l}\ \textsc{Aty-prj} \qquad \frac{S \lesssim S_1 \qquad \bar{\Gamma} \vdash t : S \rightsquigarrow e}{\bar{\Gamma} \vdash (t : S_1) : S_1 \rightsquigarrow e : |S_1|}\ \textsc{Aty-assert}$$

$$\frac{\overline{\bar{\Gamma} \vdash t_i : S_i \rightsquigarrow e_i}}{\bar{\Gamma} \vdash \{\overline{l_i = t_i}\} : \{\overline{l_i : S_i}\} \rightsquigarrow \{l_1 = e_1\}, , \ldots , , \{l_n = e_n\}}\ \textsc{Aty-rec}$$

$$\begin{array}{rcl@{\qquad}rcl} \widetilde{\mathsf{proj}}(\{l : S, \overline{l_i : S_i}, *\}, l) & = & S & \widetilde{\mathsf{proj}}(\star, l) & = & \star \\ \widetilde{\mathsf{proj}}(\{\overline{l_i : S_i}, \star\}, l) & = & \star \text{ if } l \notin \{\overline{l_i}\} & \widetilde{\mathsf{proj}}(S, l) & = & \text{undef. otherwise} \end{array}$$

$\boxed{S_1 \lesssim S_2}$ *(Consistent Subtyping)*

$$\frac{}{\mathsf{Int} \lesssim \mathsf{Int}}\ \text{ACS-z} \qquad \frac{}{\star \lesssim S}\ \text{ACS-dynl} \qquad \frac{}{S \lesssim \star}\ \text{ACS-dynr} \qquad \frac{S_3 \lesssim S_1 \qquad S_2 \lesssim S_4}{S_1 \to S_2 \lesssim S_3 \to S_4}\ \text{ACS-arr}$$

$$\frac{\overline{S_{i1} \lesssim S_{i2}}}{\{\overline{l_i : S_{i1}}, \overline{l_j : S_j}, \star\} \lesssim \{\overline{l_i : S_{i2}}, \overline{l_k : S_k}, *\}}\ \text{ACS-rcdr} \qquad \frac{\overline{S_{i1} \lesssim S_{i2}}}{\{\overline{l_i : S_{i1}}, \overline{l_j : S_j}\} \lesssim \{\overline{l_i : S_{i2}}, *\}}\ \text{ACS-rcdl}$$

Fig. 11. Type System of $\text{GTFL}_{\lesssim}$.

step is used to prove that $\lambda M^{\star}$ can encode well-typed programs of $\text{GTFL}_{\lesssim}$. Theorem 5.6 shows that if a term t in $\text{GTFL}_{\lesssim}$ calculus is well-formed with type S and context $\bar{\Gamma}$ and t elaborates to $\lambda M^{\star}$ expression e then e infers the type $|S|$ and context $|\bar{\Gamma}|$. The definition of translation for types and contexts are shown as follows.

Definition 5.4 (Type Translation). $|S|$ translates the types of $\text{GTFL}_{\lesssim}$ to the types of $\lambda M^{\star}$.

$$\begin{aligned} |\mathsf{Int}| &= \mathsf{Int} \\ |\star| &= \star \\ |(S_1 \to S_2)| &= |S_1| \to |S_2| \\ |\{\overline{l_i : S_i}\}| &= \{l_1 : |S_1|\} \,\&\, \ldots \,\&\, \{l_n : |S_n|\} \\ |\{\overline{l_i : S_i}, \star\}| &= \{l_1 : |S_1|\} \,\&\, \ldots \,\&\, \{l_n : |S_n|\} \,\&\, \star \end{aligned}$$

$$
\begin{aligned}
\mathcal{W}^{\Leftarrow}_{k}[\![A \sqsubseteq B]\!] &= \{(w_1, w_2) \mid (w_1 : A, w_2 : B) \in \mathcal{E}^{\Rightarrow}_{k}[\![A \sqsubseteq B]\!]\} \\
\mathcal{V}^{\Rightarrow}_{k}[\![\top \sqsubseteq \top]\!] &= \{(\mathsf{Top}, \mathsf{Top})\} \\
\mathcal{V}^{\Rightarrow}_{k}[\![\mathsf{Int} \sqsubseteq \mathsf{Int}]\!] &= \{(i, i)\} \\
\mathcal{V}^{\Rightarrow}_{k}[\![A_1 \to A_2 \sqsubseteq B_1 \to B_2]\!] &= \{(v_1, v_2) \mid \forall j \leqslant k, \mathrm{ty}(v_1) = A_1 \to A_2, \mathrm{ty}(v_2) = B_1 \to B_2 \\
&\qquad (e_1, e_2) \in \mathcal{E}^{\Leftarrow}_{j}[\![A_1 \sqsubseteq B_1]\!]. (v_1\ e_1, v_2\ e_2) \in \mathcal{E}^{\Rightarrow}_{j}[\![A_2 \sqsubseteq B_2]\!]\} \\
\mathcal{V}^{\Rightarrow}_{k}[\![\{l : A\} \sqsubseteq \{l : B\}]\!] &= \{(\{l = v_1\}, \{l = v_2\}) \mid (v_1, v_2) \in \mathcal{V}^{\Rightarrow}_{k-1}[\![A \sqsubseteq B]\!]\} \\
\mathcal{V}^{\Rightarrow}_{k}[\![A_1 \& A_2 \sqsubseteq B_1 \& B_2]\!] &= \{((v_{11}, , v_{12}), (v_{21}, , v_{22})) \mid (v_{11}, v_{21}) \in \mathcal{V}^{\Rightarrow}_{k-1}[\![A_1 \sqsubseteq B_1]\!] \\
&\qquad \wedge (v_{12}, v_{22}) \in \mathcal{V}^{\Rightarrow}_{k-1}[\![A_2 \sqsubseteq B_2]\!]\} \\
\mathcal{V}^{\Rightarrow}_{k}[\![A \sqsubseteq \star]\!] &= \{(s_1, g : \star) \mid \exists s_2 \in g : \star \wedge (s_1, s_2) \in \mathcal{V}^{\Rightarrow}_{k}[\![\mathrm{ty}(s_1) \sqsubseteq \mathrm{ty}(s_2)]\!]\} \\
&\qquad \cup \{(v_1, , v_2, (g : \star)) \mid (v_1, (g : \star)) \in \mathcal{V}^{\Rightarrow}_{k-1}[\![\mathrm{ty}(v_1) \sqsubseteq \star]\!] \\
&\qquad \wedge (v_2, (g : \star)) \in \mathcal{V}^{\Rightarrow}_{k-1}[\![\mathrm{ty}(v_2) \sqsubseteq \star]\!]\} \\
&\qquad \cup \{(g_1 : \star, g_2 : \star) \mid (g_1, g_2 : \star) \in \mathcal{V}^{\Rightarrow}_{k}[\![\mathrm{ty}(g_1) \sqsubseteq \star]\!]\} \\
\mathcal{R}^{\Leftrightarrow}_{k}[\![A \sqsubseteq B]\!] &= \{(r_1, r_2) \mid (r_1 = \mathsf{err}_*) \vee (r_2 = \mathsf{err}_a)\} \\
&\qquad \cup \{(w_1, w_2) \mid (w_1, w_2) \in \mathcal{W}^{\Leftrightarrow}_{k}[\![A \sqsubseteq B]\!]\} \\
\mathcal{E}^{\Leftrightarrow}_{k}[\![A \sqsubseteq B]\!] &= \{(e_1, e_2) \mid \forall j < k, (e_1 \longmapsto_j r_1 \Rightarrow e_2 \longmapsto_* r_2 \\
&\qquad \wedge (r_1, r_2) \in \mathcal{R}^{\Leftrightarrow}_{k-j}[\![A \sqsubseteq B]\!])\} \\
\mathcal{G}[\![\Gamma_1 \sqsubseteq \Gamma_2]\!] &= \{(\sigma_1, \sigma_2) \mid \forall k \geqslant 0, x \in \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2). \\
&\qquad (\sigma_1(x), \sigma_2(x)) \in \mathcal{V}^{\Rightarrow}_{k}[\![\Gamma_1(x) \sqsubseteq \Gamma_2(x)]\!]\}
\end{aligned}
$$

$$
\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \sqsubseteq e_2 \ \Leftrightarrow\ A \sqsubseteq B \iff \forall k \geqslant 0, (\sigma_1, \sigma_2) \in \mathcal{G}[\![\Gamma_1 \sqsubseteq \Gamma_2]\!]. (\sigma_1(e_1), \sigma_2(e_2)) \in \mathcal{E}^{\Leftrightarrow}_{k}[\![A \sqsubseteq B]\!]
$$

$\boxed{s \in v}$

$$
\frac{}{s \in s} \qquad \frac{}{\mathsf{Top} \in s} \qquad \frac{s \in g}{s \in g : \star} \qquad \frac{s \in v_1}{s \in v_1, , v_2} \qquad \frac{s \in v_2}{s \in v_1, , v_2}
$$

Fig. 12. Logical relation.

Definition 5.5 (Context Translation). $|\bar{\Gamma}|$ translates the typing context of GTFL$_{\lesssim}$ to the typing context of $\lambda M^{\star}$.

$$
\begin{aligned}
|\cdot| &= \cdot \\
|\bar{\Gamma}, x : S| &= |\bar{\Gamma}|, x : |S|
\end{aligned}
$$

Theorem 5.6 (Well-typed Encoding of GTFL$_{\lesssim}$). *If* $\bar{\Gamma} \vdash t : S \rightsquigarrow e$ *then* $|\bar{\Gamma}| \vdash e \Rightarrow |S|$.

5.2 Dynamic Gradual Guarantee

Section 2 illustrates that the standard formulation of the DGG and determinism (Theorem 4.3) are incompatible. In this section we present a relaxed notion of the DGG that states that reduction is monotone with respect to imprecision, but modulo ambiguity errors. Instead of syntactic precision (defined in Figure 10), we use a semantic notion of precision [New et al. 2020]. To motivate this choice, consider (syntactically) related expressions $(1, , \mathsf{True}) : \mathsf{Int} \sqsubseteq (1, , \mathsf{True}) : \star$. As the first expression reduces to 1, according to the DGG, the second expression should reduce to a related value. But it reduces to $(1, , \mathsf{True}) : \star$ which is not related by the syntactic relation.

To address this, we define a semantic notion of precision using a step-indexed logical relation, shown in Figure 12. The interpretations of values and expressions are mutually defined using four category of sets: for *irreducible values* at check mode $\mathcal{W}^{\Leftarrow}_{k}[\![A \sqsubseteq B]\!]$, for values at infer mode $\mathcal{V}^{\Rightarrow}_{k}[\![A \sqsubseteq B]\!]$, for results at any mode $\mathcal{R}^{\Leftrightarrow}_{k}[\![A \sqsubseteq B]\!]$, and for expressions or computations at any mode $\mathcal{E}^{\Leftrightarrow}_{k}[\![A \sqsubseteq B]\!]$. Each category is indexed by the step index k, the mode $\Leftrightarrow$, and a pair of types.

An irreducible value w represents an irreducible expression that can be typed by the check mode, and can be a value v or a raw lambda $\lambda x.e$. A pair of irreducible values are related at $A \sqsubseteq B$ and check mode, if the respective ascriptions to A and B yield related computations at infer mode.

Two values are related at the same base types, if the values are the same. Two values are related at two function types, if their application to related expressions yield related computations. Note that we use expressions as arguments (instead of values) in order to simplify the proofs. Two merge values are related at two intersection types, if the first (resp. second) components of the merges are related at the first (resp. second) components of the types.

The most complicated case is when the least precise type of the relation is $\star$. For this, we consider three sub-cases. First, ordinary value s_1 and dynamic value $g : \star$ are related if there exists a related ordinary value s_2 that can be projected from $g : \star$, denoted as $s_2 \in g : \star$. The $s_2 \in g : \star$ relation is defined at the bottom of the Figure, and checks whether s_2 is a subcomponent of $g : \star$. For example, $1 \in (1, , \mathsf{True})$ holds, and $\mathsf{Top} \in v$ holds for any v. Going back to the first example in this section, 1 is now more precise than $(1,,\mathsf{True}) : \star$ at $\mathsf{Int} \sqsubseteq \star$, because $(1, 1)$ is related at $\mathsf{Int} \sqsubseteq \mathsf{Int}$. Second, a merge and a dynamic value are related if each component of the merge is related to the dynamic value. For example, $1 : \mathsf{Int}\&\top$ is more precise than $1 : \star$. Program $1 : \mathsf{Int}\&\top$ reduces to $1,,\mathsf{Top}$ while $1 : \star$ is a value. They are related because 1 is related to $1 : \star$, and Top with $1 : \star$ (because $\mathsf{Top} \in 1 : \star$). Third, two dynamic values are related if the underlying first ground value is related to the second dynamic value at the underlying ground type and $\star$. Although some cases do not reduce the index k, the relation is well-founded because each recursive occurrence will eventually lower the index.

Two results are related to some mode if either (1) the first result is an error, (2) the second result is an ambiguity error, or (3) the results are related irreducible values at the same mode. Note that the relation for irreducible values at infer mode $\mathcal{W}^{\rightrightarrows}_k[\![A \sqsubseteq B]\!]$ is defined as the relation for values at the same mode $\mathcal{V}^{\rightrightarrows}_k[\![A \sqsubseteq B]\!]$. A pair of expressions (e_1, e_2) are related at k steps and some mode $\Leftrightarrow$ if, when e_1 reduces to a result in j steps, e_2 must reduce to a related result at $k - j$ steps within the same mode. A pair of type environments are related if every variable maps to a related value at infer mode.

Finally, we use notation $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \sqsubseteq e_2 \Leftrightarrow A \sqsubseteq B$ to denote that expression e_1 is semantically more precise than e_2 under related type environments $\Gamma_1 \sqsubseteq \Gamma_2$ at related types $A \sqsubseteq B$ and some mode $\Leftrightarrow$, if the expressions, closed under related value environments, are related expressions for any number of steps k. For simplicity, if contexts are empty we use notation $\vdash e_1 \sqsubseteq e_2 \Leftrightarrow A \sqsubseteq B$.

Armed with the logical relation and semantic precision, we can establish the fundamental property that states that well-typed expressions related by the syntactic precision relation are related by the semantic precision relation.

Theorem 5.7 (Fundamental Property).

(1) *if* $\Gamma_1 \vdash e_1 \Rightarrow A, \Gamma_2 \vdash e_2 \Rightarrow B$ *and* $e_1 \sqsubseteq e_2$ *then* $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \sqsubseteq e_2 \Rightarrow A \sqsubseteq B$.
(2) *if* $\Gamma_1 \vdash e_1 \Leftarrow A, \Gamma_2 \vdash e_2 \Leftarrow B$ *and* $e_1 \sqsubseteq e_2$ *then* $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \sqsubseteq e_2 \Leftarrow A \sqsubseteq B$.

The key lemma to prove this theorem is the ascription lemma, which states that the ascriptions of related values to related types, yield related expressions.

Lemma 5.8 (Ascription Lemma). *if* $(v_1, v_2) \in \mathcal{V}^{\rightrightarrows}_k[\![A' \sqsubseteq B']\!] \wedge A' \lesssim A \wedge B' \lesssim B \wedge A \sqsubseteq B$ *then* $(v_1 : A, v_2 : B) \in \mathcal{E}^{\rightrightarrows}_k[\![A \sqsubseteq B]\!]$.

Finally, based on the fundamental property, we can establish the DGG modulo ambiguity errors. We use $e \Uparrow$ to denote that e diverges.

Theorem 5.9 (Dynamic Gradual Guarantee). *Suppose that* $\vdash e_1 \Leftrightarrow A, \vdash e_2 \Leftrightarrow B$ *and* $e_1 \sqsubseteq e_2$.
(1) $e_1 \longmapsto^* v_1$ *then* $((e_2 \longmapsto^* v_2$ *and* $\vdash v_1 \sqsubseteq v_2 \Leftrightarrow A \sqsubseteq B)$ *or* $e_2 \longmapsto^* \mathsf{err}_a)$.

(2) $e_1 \Uparrow$ *then* $e_2 \Uparrow$ *or* $e_2 \longmapsto^* \mathsf{err}_a$.
(3) $e_2 \longmapsto^* v_2$ *then* $((e_1 \longmapsto^* v_1$ *and* $\vdash v_1 \sqsubseteq v_2 \Leftrightarrow A \sqsubseteq B)$ *or* $e_1 \longmapsto^* \mathsf{err}_*)$.
(4) $e_2 \Uparrow$ *then* $e_1 \Uparrow$ *or* $e_1 \longmapsto^* \mathsf{err}_*$.

Cases (1) and (2) are similar to the original DGG [Siek et al. 2015b] except for the fact that the less precise expression can reduce to an ambiguity error. Case (2) may be counter-intuitive, in particular when e_2 reduces to err_a, so we provide a simple example that illustrates this case. Let $\Omega = ((\lambda x.x\ x)\ (\lambda x.x\ x))$. Expression $((((1,,\mathsf{True}) : \mathsf{Bool}),,2) : \mathsf{Int}),,\Omega$ diverges, but less precise expression $((((1,,\mathsf{True}) : \star),,2) : \mathsf{Int}),,\Omega$ raises an ambiguity error. Cases (3) and (4) also include instances where the most precise expression can reduce to an ambiguity error. To illustrate case (3), consider value $((1 : \star),,2) : \star$. The more precise expression $((1 : \star),,2) : \mathsf{Int}$ reduces to an ambiguity error. For case (4), $(((1 : \star),,2) : \star),,\Omega$ diverges but the more precise expression $(((1 : \star),,2) : \mathsf{Int}),,\Omega$ reduces to an ambiguity error.

6 Related Work

In Section 2 we already discussed the most closely related work. Thus here we will only briefly summarize key points and discuss other closely related work.

Gradual Objects. Siek and Taha [2007] designed a calculus ($\mathrm{Ob}^{?}_{<:}$), which extended $\mathrm{Ob}_{<:}$ [Abadi and Cardelli 1996] with the unknown type $\star$. Although the unknown type $\star$ is powerful and general for gradual typing, there is a significant loss of information with record types and subtyping. To solve this, Garcia et al. [2016] proposed a new kind of gradual type called *gradual row*. A gradual row type ($\{\overline{l_i : S_i}, \star\}$) has extra (statically) unknown fields in the record type. With gradual rows, Garcia et al. [2016] defined a calculus with records and subtyping named $\mathrm{GTFL}_{\lesssim}$ by using the Abstracting Gradual Typing (AGT) methodology. $\mathrm{GTFL}_{\lesssim}$ represents gradual typing derivations as intrinsically typed terms to give dynamic semantics directly instead of elaborating to an intermediate language.

As Bañados Schwerter et al. [2021] point out $\mathrm{GTFL}_{\lesssim}$ fails to enforce modular invariants, which are expected from the static type discipline. They address the problems by refining the underlying theory of AGT dynamic checking, and have also designed their calculus to be space efficient. Since we employ a conventional lazy semantics, $\lambda M^{\star}$ is not space efficient. Sekiyama and Igarashi [2019] generalize gradual row types to variant types and row polymorphism [Wand 1994]. Compared to $\mathrm{GTFL}_{\lesssim}$, their records are extensible. However, they drop subtyping, in favour of row polymorphism. As we have shown, $\lambda M^{\star}$ can encode the static semantics of $\mathrm{GTFL}_{\lesssim}$ and gradual rows using single field record types, intersection types and the unknown type $\star$. Furthermore, with $\lambda M^{\star}$, records are extensible, by employing the merge operator as record concatenation, and subtyping is supported. Thus, not only $\lambda M^{\star}$ can encode multiple inheritance, but it can encode dynamic inheritance and first-class traits/classes as well. Because AGT gradual rows have fixed size, there is no concatenation and ambiguity is statically rejected (records with repeated labels are not allowed). Thus, it is not possible to encode dynamic inheritance and first-class classes. Finally, casting in $\lambda M^{\star}$ preserves the modular invariants expected from the static type discipline naturally.

Based on their earlier work in Nom [Muehlboeck and Tate 2017], Muehlboeck and Tate [2021] present MonNom: a gradual language supporting seamless transition between untyped structural and typed nominal paradigms. They propose a novel approach to transitioning from untyped structural objects to nominal objects. Precision between types is restricted as any type is more precise than itself or the unknown type, disallowing precision between different partially untyped types. Precision between expressions is complex, as it enables the correlation of untyped structural code with nominal code. The authors provide proof for both gradual guarantees and type safety. Unlike $\lambda M^{\star}$, MonNom does not support dynamic inheritance and first-class traits/classes. Instead

MonNom's focus is on improving the performance of more conventional (gradual) OOP languages. In the future we hope to learn from Nom and MonNon to improve the performance of λM⋆.

First-Class Classes and Traits. Many dynamically-typed languages support first-class classes/traits, including Racket [Flatt et al. 2006] or JavaScript. To type first-class classes, Takikawa et al. [2012] extended Racket with a gradual type system, called TFCC, for first-class classes. TFCC consists of two parts: an untyped portion and a typed portion of the language. The interactions between the two portions are mediated by contracts. Row polymorphism [Wand 1994] is used to type mixins. Compared to λM⋆, TFCC mixes typed and untyped modules instead of allowing fine-grained gradual typing at the level of expressions. TFCC can have fully typed modules, dynamically typed modules and these modules can interoperate with each other. However TFCC cannot have a module that mixes static and dynamically typed expressions. In contrast the form of gradual typing in λM⋆ is more fine-grained and it works at the level of expressions. Moreover, λM⋆ is a lower level and simpler language, since it is basically a lambda calculus extended with merges and single label records. So, high-level constructs like classes/traits are encodable in terms of simpler, more atomic constructs. In contrast, TFCC is significantly more complex, as it has a built-in notion of classes, and requires both a form of row polymorphism and subtyping for modelling first-class classes.

Some statically typed calculi support first-class classes, but do not support gradual typing. Tagged objects are used to type first-class classes by Lee et al. [2015]. SEDEL was proposed by Bi and Oliveira [2018] to type first-class traits. The type system of SEDEL is based on disjoint intersection types [Oliveira et al. 2016] and disjoint polymorphism [Alpuim et al. 2017]. In SEDEL traits are elaborated into a target calculus with the merge operator and disjoint intersection types. The later CP language [Zhang et al. 2021] also adopts a similar approach to typed-first class traits.

Gradual Typing with Intersection Types. Castagna and Lanvin [2017] developed a gradual typing system with union and intersection types using set-theoretic types. They show how to lift definitions, such as subtyping, from non-gradual types to gradually typed ones. There are two main parts: a gradually-typed language with its type system, and a cast calculus. The dynamic semantics is given in the cast calculus. In later work, Castagna et al. [2019] improved the work of Castagna and Lanvin [2017] with a blame calculus style dynamic semantics and blame labels. An important difference to this line of work is that λM⋆ includes a merge operator, which brings significant complications, such as the issue of ambiguity or type safety in the presence of subtyping in merges.

7 Conclusion and Future Work

This paper presented a calculus, called λM⋆, that unifies two type-directed mechanisms: gradual typing and the merge operator. We prove that λM⋆ is type sound, deterministic and satisfies the gradual guarantee. λM⋆ is expressive, and it can encode gradual rows and the $\text{GTFL}_{\lesssim}$ calculus using intersection types and the merge operator. In addition λM⋆ has extensible records via the merge operator and it can encode first-class classes/traits and dynamic inheritance following an existing encoding by Bi and Oliveira [2018]. This brings λM⋆ closer to dynamically typed languages, such as JavaScript, which are common targets for practical implementations of gradual typing.

There are still several important gaps between λM⋆ and languages such as TypeScript. In particular λM⋆ in purely functional, and omits imperative features like *references* [Toro and Éric Tanter 2020], as well as other common features such as *polymorphism* [Ahmed et al. 2011]. References will require some further study. Although there is already some work integrating references and polymorphism in a TDOS with gradual typing [Ye and Oliveira 2023], merges have not been considered. An issue that is important to study is related to the notion of object identity, which most OOP languages rely on. In our work, due to our coercive semantics, we essentially create proxy objects around existing objects. However proxy objects may have a different object

identity. We expect to be able to address this issue by building on existing research on *transparent proxies* [Keil et al. 2015], which aims at addressing such concerns with proxy objects.

On the more theoretical side, it will be interesting to study a general framework for type-directed mechanisms and look at integrating other type-directed mechanisms such as type classes [Wadler and Blott 1989] or implicits [Oliveira et al. 2010].

An important question that we have not touched in this paper is performance. There are at least three points that deserve further study in the future. Firstly, we are interested in exploring a variant of $\lambda M^{\star}$ with either threesomes [Siek and Wadler 2009], or an eager semantics for higher-order casts. Both of these can help avoid the accumulation of type annotations, which are known to cause time and space inefficiencies [Herman et al. 2010]. Secondly, in its current form, all applications in the TDOS are *flexible*. Since applications in the TDOS model the semantics of a *source* gradual language, they allow mismatched (but consistent) types between arguments and functions, thus requiring casting. To address this performance drawback, a possible solution is to introduce *strict* forms of applications, alongside flexible applications, where the type of the argument must exactly match the expected input type of the function. In this way casting can be avoided for strict applications. This would be somewhat analogous to optimization techniques used in OOP languages, where some dynamically dispatched method calls can be optimized to statically dispatched method calls. Adding strict applications could be complemented with further optimizations that removes unnecessary annotations such as in $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int})\,(1 : \mathsf{Int})$. Finally, runtime ambiguity detection is costly. It is possible to avoid runtime ambiguity detection by forbidding merges with unknown types. But this would be quite restrictive. A better solution would be to detect static merges (merges without components with unknown types) and employ an optimized version of casting without ambiguity detection. This should be possible because, for static merges, all ambiguity can be statically detected.

Acknowledgments

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 project number 17209821.

Data-Availability Statement

The artifact that supports the paper is available on Zenodo [Ye et al. 2024].

References

Martín Abadi and Luca Cardelli. 1996. A Theory of Objects. In *Monographs in Computer Science*.

Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. 2011. Blame for All. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 201–214. https://doi.org/10.1145/1926385.1926409

João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. 2017. Disjoint Polymorphism. In *ESOP*. https://doi.org/10.1007/978-3-662-54434-1_1

Felipe Bañados Schwerter, Alison M. Clark, Khurram A. Jafery, and Ronald Garcia. 2021. Abstracting Gradual Typing Moving Forward: Precise and Space-Efficient. *Proc. ACM Program. Lang.* 5, POPL, Article 61 (jan 2021), 28 pages. https://doi.org/10.1145/3434342

Xuan Bi and Bruno C. d. S. Oliveira. 2018. Typed First-Class Traits. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2018.9

Gilad Bracha and William Cook. 1990. Mixin-Based Inheritance. In *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications* (Ottawa, Canada) *(OOPSLA/ECOOP '90)*. Association for Computing Machinery, 303–311. https://doi.org/10.1145/97946.97982

Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. 1999. Comparing Object Encodings. *Inf. Comput.* 155, 1-2 (1999), 108–133.

Luca Cardelli. 1988. A Semantics of Multiple Inheritance. *Inf. Comput.* 76, 2/3 (1988), 138–164.

Luca Cardelli and John C. Mitchell. 1991. Operations on records. *Mathematical Structures in Computer Science* 1, 1 (1991), 3–48.

Giuseppe Castagna and Victor Lanvin. 2017. Gradual typing with union and intersection types. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 1–28. https://doi.org/10.1145/3110285

Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. 2019. Gradual typing: a new perspective. *Proceedings of the ACM on Programming Languages* 3 (2019), 1 – 32. https://doi.org/10.1145/3290329

William R. Cook and Jens Palsberg. 1989. A denotational semantics of inheritance and its correctness. In *Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/74878.74922

Rowan Davies and Frank Pfenning. 2000. Intersection types and computational effects. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, 198–208. https://doi.org/10.1145/351240.351259

Jana Dunfield. 2014. Elaborating intersection and union types. *Journal of Functional Programming (JFP)* 24, 2-3 (2014), 133–165. https://doi.org/10.1145/2398856.2364534

Erik Ernst. 2000. gbeta-a language with virtual attributes, Block Structure, and Propagating, Dynamic Inheritance. *DAIMI Report Series* 29, 549 (2000).

Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. 2006. Scheme with Classes, Mixins, and Traits. In *APLAS*. https://doi.org/10.1007/11924661_17

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. *SIGPLAN Not.* 51, 1 (jan 2016), 429–442. https://doi.org/10.1145/2914770.2837670

Ben Greenman. 2023. GTP Benchmarks for Gradual Typing Performance. In *REP*. ACM, 102–114. https://doi.org/10.1145/3589806.3600034

Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. 2019. How to evaluate the performance of gradual type systems. *Journal of Functional Programming* 29 (2019), 45. https://doi.org/10.1017/S0956796818000217

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *Higher-Order and Symbolic Computation* 23, 2 (2010), 167. https://doi.org/10.1007/s10990-011-9066-z

Xuejing Huang and Bruno C. d. S. Oliveira. 2020. A Type-Directed Operational Semantics For a Calculus with a Merge Operator. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 26:1–26:32. https://doi.org/10.4230/LIPIcs.ECOOP.2020.26

Xuejing Huang, Jinxu Zhao, and Bruno C. d. S. Oliveira. 2021. Taming the Merge Operator. *Journal of Functional Programming* 31 (2021), e28. https://doi.org/10.1017/S0956796821000186

Matthias Keil, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, and Peter Thiemann. 2015. Transparent Object Proxies in JavaScript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic (LIPIcs, Vol. 37)*, John Tang Boyland (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 149–173. https://doi.org/10.4230/LIPIcs.ECOOP.2015.149

Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. 2019. Toward Efficient Gradual Typing for Structural Types via Coercions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 517–532. https://doi.org/10.1145/3314221.3314627

Joseph Lee, Jonathan Aldrich, Troy Shaw, and Alex Potanin. 2015. A Theory of Tagged Objects. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2015.174

Fabian Muehlboeck and Ross Tate. 2017. Sound Gradual Typing is Nominally Alive and Well. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 56 (oct 2017), 30 pages. https://doi.org/10.1145/3133880

Fabian Muehlboeck and Ross Tate. 2021. Transitioning from structural to nominal code with efficient gradual typing. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–29. https://doi.org/10.1145/3485504

Max S. New, Dustin Jamner, and Amal Ahmed. 2020. Graduality and parametricity: together again for the first time. *Proc. ACM Program. Lang.* 4, POPL (2020), 46:1–46:32. https://doi.org/10.1145/3371114

Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. 2010. Type Classes as Objects and Implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. 341–360. https://doi.org/10.1145/1932682.1869489

Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. 2016. Disjoint Intersection Types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 364–377. https://doi.org/10.1145/2951913.2951945

John C Reynolds. 1997. Design of the programming language Forsythe. In *ALGOL-like languages*. 173–233.

Taro Sekiyama and Atsushi Igarashi. 2019. Gradual Typing for Extensibility by Rows. *ArXiv* abs/1910.08480 (2019).

Jeremy Siek, Peter Thiemann, and Philip Wadler. 2015a. Blame and Coercion: Together Again for the First Time. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 425–435. https://doi.org/10.1145/2737924.2737968

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

Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *ECOOP*. https://doi.org/10.1007/978-3-540-73589-2_2

Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. 2015b. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

Jeremy G. Siek and Philip Wadler. 2009. Threesomes, with and without Blame. In *Proceedings for the 1st Workshop on Script to Program Evolution* (Genova, Italy) *(STOP '09)*. Association for Computing Machinery, New York, NY, USA, 34–46. https://doi.org/10.1145/1570506.1570511

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is Sound Gradual Typing Dead? *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 456–468. https://doi.org/10.1145/2837614.2837630

Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Gradual Typing for First-Class Classes. *SIGPLAN Not.* 47, 10 (oct 2012), 793–810. https://doi.org/10.1145/2398857.2384674

Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage migration: from scripts to programs. https://doi.org/10.1145/1176617.1176755

Matías Toro and Éric Tanter. 2020. Abstracting gradual references. *Science of Computer Programming* 197 (2020), 102496. https://doi.org/10.1016/j.scico.2020.102496

P. Wadler and S. Blott. 1989. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. https://doi.org/10.1145/75277.75283

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009* (York, UK) *(ESOP '09)*. Springer-Verlag, Berlin, Heidelberg, 1–16. https://doi.org/10.1007/978-3-642-00590-9_1

Mitchell Wand. 1989. Type Inference for Record Concatenation and Multiple Inheritance. In *Symposium on Logic in Computer Science (LICS)*.

Mitchell Wand. 1994. Type inference for objects with instance variables and inheritance.

Ningning Xie, Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. Consistent Subtyping for All. *ACM Trans. Program. Lang. Syst.* 42, 1, Article 2 (nov 2019), 79 pages. https://doi.org/10.1145/3310339

Wenjia Ye and Bruno C. d. S. Oliveira. 2023. Pragmatic Gradual Polymorphism with References. In *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 13990)*, Thomas Wies (Ed.). Springer, 140–167. https://doi.org/10.1007/978-3-031-30044-8_6

Wenjia Ye, Bruno C. d. S. Oliveira, and Xuejing Huang. 2021. Type-Directed Operational Semantics for Gradual Typing. In *35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2021.12

Wenjia Ye, Bruno C. d. S. Oliveira, and Matías Toro. 2024. *Merging Gradual Typing (Artifact)*. https://doi.org/10.5281/zenodo.13378311

Weixin Zhang, Yaozhu Sun, and Bruno C. d. S. Oliveira. 2021. Compositional Programming. *ACM Trans. Program. Lang. Syst.* 43, 3, Article 9 (sep 2021), 61 pages. https://doi.org/10.1145/3460228

Received 2024-04-04; accepted 2024-08-18