Merging Gradual Typing

ANONYMOUS AUTHOR(S)*

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 $GTFL_\leqslant$ calculus; and show that $\lambda M^\star$ satisfies gradual 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.

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 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 $GTFL_\leqslant$ calculus, have been proposed [Bañados Schwerter et al. 2021; Garcia et al. 2016]. Similarly to Siek's work, the $GTFL_\leqslant$ 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 $\mathsf{GTFL}_{\lessgtr}$ calculus in $\lambda M^\star$.** Compared to the $\mathsf{GTFL}_{\lessgtr}$ 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 supplementary materials.[1]

---

[1]**Note to reviewers:** Due to the anonymous submission rules, reviewers need to install the implementation in their machine.

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 λM⋆ 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 ⋆ and consistency relations. The unknown type ⋆ 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 : ⋆ : Bool casting 1 to type Bool will result in *blame* (i.e. a runtime type error). However, if we have 1 : ⋆ : 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 : ⋆ : Bool there are two casts: a first cast from Int (the type of 1) to ⋆; and a second cast from ⋆ to Bool. We choose the use of this notation throughout the paper to be consistent with the notation in the λM and λM⋆ 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 $(1,, 2) :$ 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 True,, 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:

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

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

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

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 $\{l_2 : \text{Bool}\}$. Therefore, $(\{l_1 = 2\},, x).l_1$ would become $(\{l_1 = 2\},, \{l_2 = \text{True}\}).l_1$ 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

```
197    class A {
198      m() : number {return 5};
199      n() : number {return this.m() - 4;} }
200    interface C {n() : number}
201    type GConstructor<T = {}> = new (...args: any[]) ⇒ T;
202    function mkB<TBase extends GConstructor<C>>(Base: TBase) {
203      return class B extends Base {
204        m() :   string {return "hello";}  // If m exists in Base it will be overriden
205      };
206    }
207    const cl = mkB(A);    // Problem: Superclass already contains an m
208    const o = new cl;
209    console.log(o.n());
```
Fig. 1. Type unsoundness of first-class classes in TypeScript.

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 C#). In addition, TypeScript also supports first-class classes and dynamic inheritance[2]. 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

---

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

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; };
```

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]. The extended version 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, 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 ⋆ 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 λM because, to integrate gradual typing with the merge operator more easily, we need to modify some details of the semantics. In particular λM 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 λM 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 $\lambda M^\star$ calculus is a gradual version of the λM calculus, and adds the unknown type $\star$ to λM. 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, $(1 : \star ,, 2 : \star) : \text{Int}$ 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 $(\text{True} : \star ,, 1 : \star) : \text{Int}$. First, as both components of the merge operator are suitable for casting, we cast both components to Int: $\text{True} : \star \hookrightarrow_{\text{Int}} \text{err}$

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. 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, $(1 : \star,, 2 : \star) : \star \hookrightarrow_{\mathsf{Int}}$ 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.

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

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

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.

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

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

Compared to $\mathsf{GTFL}_{\lesssim}$, $\lambda M^\star$ has extensible records (via the merge operator), whereas $\mathsf{GTFL}_{\lesssim}$ only supports fixed size records. Thus, $\mathsf{GTFL}_{\lesssim}$ 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 $\mathsf{GTFL}_{\lesssim}$ and our work is that $\mathsf{GTFL}_{\lesssim}$ does *not* allow records with the same label to be present, even if these are in the dynamic parts of the rows: $\mathsf{GTFL}_{\lesssim}$ *statically* rejects records with repeated labels. This approach is possible to adopt in $\mathsf{GTFL}_{\lesssim}$ 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:

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

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 $\mathsf{GTFL}_{\lesssim}$. Thus we do not prove an operational correspondence result. The first reason for this is that $\lambda M^\star$ employs a lazy semantics, whereas $\mathsf{GTFL}_{\lesssim}$ uses an eager semantics for higher-order casts. The second reason is

that the original semantics of $\mathrm{GTFL}_{\leqslant}$ [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 $A <: B$ and program: let $x : B = \mathsf{new}\ A()$ in 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 $\mathrm{GTFL}_{\leqslant}$ fails to preserve expected modular type invariants. Let us consider program let $x : \{l_1 : \mathsf{Int}\} = \{l_1 = 5, l_2 = \mathsf{True}\}$ in 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: let $x : \{l_1 : \mathsf{Int}\} = \{l_1 = 5, l_2 = \mathsf{True}\} : \star$ in $(x : \star).l_2$. the original formulation of $\mathrm{GTFL}_{\leqslant}$ 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).

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

Fig. 2. The syntax of the λM calculus.

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.
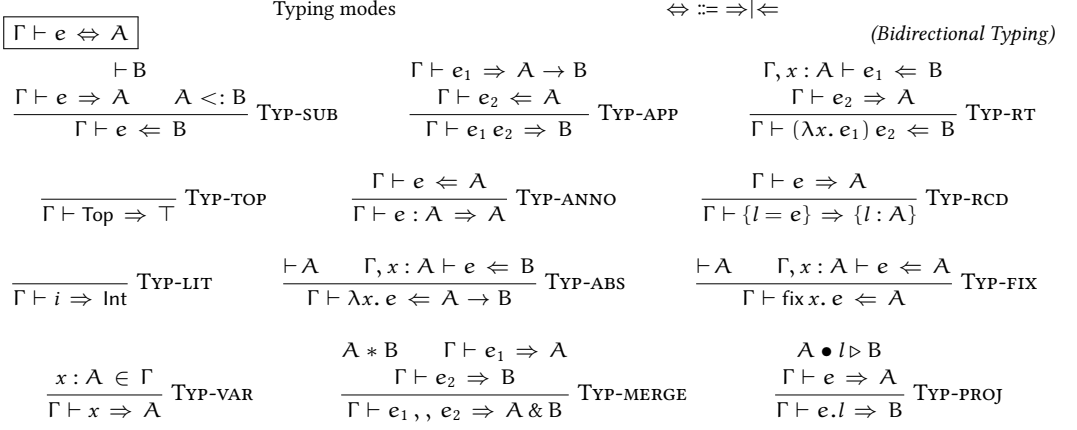
Meta-variable e ranges over expressions. Most expressions are typical: variables (x); integers (i); a canonical top value (Top); annotated expressions ($e : A$); applications ($e_1\, e_2$); lambda expressions ($\lambda x.\, e$) and fixpoints (fix $x.\, e$). The merge of expressions e_1 and e_2 is denoted 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.

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, though it includes annotated expressions, and merges. Also the left part of the application is not a value but an unannotated lambda.

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 : Int & 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

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

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

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

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

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

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

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

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). 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 $A \bullet l \rhd B$, 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

$$\boxed{A <: B} \hspace{10cm} \textit{(Subtyping)}$$

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

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

$$\boxed{A \sqcup B} \hspace{6cm} \textit{(Common Ordinary Super Types (COSTR))}$$

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

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

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

Fig. 4. Subtyping and the COSTR relation.

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).

$$\boxed{v \hookrightarrow_A v'} \hspace{10cm} \textit{(Casting)}$$

$$\frac{}{v \hookrightarrow_\top \mathsf{Top}} \text{ Cast-Top} \qquad \frac{}{i \hookrightarrow_{\mathsf{Int}} i} \text{ Cast-Lit} \qquad \frac{A \to B <: C \to D}{f : A \to B \hookrightarrow_{C \to D} f : A \to B : C \to D} \text{ Cast-Abs}$$

$$\frac{v \hookrightarrow_A v'}{\{l = v\} \hookrightarrow_{\{l:A\}} \{l = v'\}} \text{ Cast-Rcd} \qquad \frac{v_1 \hookrightarrow_{A^\circ} v_1'}{v_1 \,,, v_2 \hookrightarrow_{A^\circ} v_1'} \text{ Cast-MergeL}$$

$$\frac{v_2 \hookrightarrow_{A^\circ} v_2'}{v_1 \,,, v_2 \hookrightarrow_{A^\circ} v_2'} \text{ Cast-MergeR} \qquad \frac{v \hookrightarrow_A v_1 \qquad v \hookrightarrow_B v_2}{v \hookrightarrow_{A \,\&\, B} v_1 \,,, v_2} \text{ Cast-And}$$

$$\boxed{e \hookrightarrow e'} \hspace{10cm} \textit{(Small-step Semantics)}$$

$$\frac{e \hookrightarrow e'}{\mathsf{F}[e] \hookrightarrow \mathsf{F}[e']} \text{ Step-Eval} \qquad \frac{}{(f : A_1 \to A_2)\, e \hookrightarrow (f\,(e : A_1)) : A_2} \text{ Step-App}$$

$$\frac{}{(\lambda x.\, e)\, v \hookrightarrow e[x \mapsto v]} \text{ Step-Beta} \qquad \frac{}{(\mathsf{fix}\, x.\, e) : A \hookrightarrow e[x \mapsto (\mathsf{fix}\, x.\, e) : A] : A} \text{ Step-Fix}$$

$$\frac{v \hookrightarrow_A v' \qquad \mathsf{NotVal}\,(v : A)}{v : A \hookrightarrow v'} \text{ Step-AnnoV} \qquad \frac{\mathsf{ty}(v) \bullet l \rhd A \qquad v \hookrightarrow_{\{l:A\}} \{l = v'\}}{v.l \hookrightarrow v'} \text{ Step-Proj}$$

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

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 $\mathsf{NotVal}\,(v : A)$ which is defined as not a functional value: $\mathsf{NotVal}\, e \equiv e \neq (f : A \to B)$. 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 $\mathsf{ty}(v) = A$.*

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 $\cdot \vdash e \Leftrightarrow A$ then e is a value or $\exists e', e \hookrightarrow e'$.*

4 THE $\lambda M^\star$ CALCULUS : SYNTAX, TYPING AND SEMANTICS

This section introduces the $\lambda M^\star$ calculus, the gradual counterpart of λM. We prove determinism and type soundness. Section 5 presents the gradual typing criteria satisfied by $\lambda M^\star$.

4.1 Syntax

The syntax of $\lambda M^\star$ calculus is shown in Figure 6. Types extend the types of λM calculus with the unknown type $(\star)$. Because $\lambda M^\star$ 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

Types	$A, B, C ::= \text{Int} \mid \top \mid \bot \mid A \to B \mid A \mathbin{\&} B \mid \{l : A\} \mid \star$	
Expressions	$e ::= x \mid i \mid \text{Top} \mid \lambda x.\, e \mid \{l = e\} \mid e.l \mid e : A \mid e_1\, e_2 \mid e_1\,,, e_2 \mid \text{fix}\, x.\, e$	
Results	$r ::= e \mid \text{err}_*$	
Functionals	$f ::= \lambda x.\, e \mid f : A \to B$	
Ordinary values	$s ::= \text{Top} \mid i \mid f : A \to B \mid \{l = v\}$	
Ground values	$g ::= \text{Top} \mid i \mid \lambda x.\, e : \star \to \star \mid f : \star \to \star \mid \{l = g : \star\} \mid g : \star\,,, g : \star$	
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 $\lambda M^\star$ calculus.

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 $\star \mathbin{\&} \star$). 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 $\star \to \star$ 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 similar transformation for $\lambda M^\star$ terms, simply adding a $\star \to \star$ annotation in non-annotated lambdas, we can do better in $\lambda M^\star$ 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 $\lambda M^\star$ program by automatically annotating only one lambda abstraction: $((\lambda f.\lambda x.\, f\, x) : \star \to \star)(\lambda y.y)$. 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.

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

$$\frac{}{\mathsf{Int} \sim \mathsf{Int}} \text{ SIM-I} \qquad \frac{}{\top \sim \top} \text{ SIM-TOP} \qquad \frac{}{\bot \sim \bot} \text{ SIM-BOT} \qquad \frac{A \sim C \qquad B \sim D}{A \to B \sim C \to D} \text{ SIM-ARR}$$

$$\frac{}{\star \sim A} \text{ SIM-DYNL} \qquad \frac{}{A \sim \star} \text{ SIM-DYNR} \qquad \frac{A \sim C \qquad B \sim D}{A \,\&\, B \sim C \,\&\, D} \text{ SIM-MERGE} \qquad \frac{A_1 \sim A_2}{\{l : A_1\} \sim \{l : A_2\}} \text{ SIM-RCD}$$

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

$$\frac{\mathsf{Static}\ A}{A <: \top} \text{ S-TOP} \qquad \qquad \frac{}{\star <: \star} \text{ S-DYN}$$

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

$$\frac{}{\mathsf{Int} \lesssim \mathsf{Int}} \text{ CS-z} \qquad \frac{}{\star \lesssim A} \text{ CS-DYNL} \qquad \frac{}{A \lesssim \star} \text{ CS-DYNR} \qquad \frac{}{\bot \lesssim A} \text{ CS-BOT} \qquad \frac{}{A \lesssim \top} \text{ CS-TOP}$$

$$\frac{B_1 \lesssim A_1 \qquad A_2 \lesssim B_2}{A_1 \to A_2 \lesssim B_1 \to B_2} \text{ CS-ARR} \qquad \frac{A_1 \lesssim A_3}{A_1 \,\&\, A_2 \lesssim A_3} \text{ CS-ANDL} \qquad \frac{A_2 \lesssim A_3}{A_1 \,\&\, A_2 \lesssim A_3} \text{ CS-ANDR}$$

$$\frac{A_1 \lesssim A_2 \qquad A_1 \lesssim A_3}{A_1 \lesssim A_2 \,\&\, A_3} \text{ CS-AND} \qquad \frac{A_1 \lesssim A_2}{\{l : A_1\} \lesssim \{l : A_2\}} \text{ CS-RCD}$$

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

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

$$\frac{A_1 \sqsubseteq A_2 \qquad B_1 \sqsubseteq B_2}{A_1 \,\&\, B_1 \sqsubseteq A_2 \,\&\, B_2} \text{ TP-AND} \qquad \frac{A_1 \sqsubseteq A_2}{\{l : A_1\} \sqsubseteq \{l : A_2\}} \text{ TP-RCD}$$

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

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

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

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 $\star$. The remaining rules are defined inductively. The original AGT existential lifting of predicates 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.

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

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

with the labels $\Gamma \vdash e \Leftarrow B$ for Typ-cs.

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, $1 : \star : \text{Bool}$ and $\text{True} : \star : \text{Int}$ 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 $\star \rhd \star \to \star$, and $A_1 \to A_2 \rhd A_1 \to A_2$. For projections, we allow programs such as $((\{l_1 = 1\},, \text{True}) : \star).l_2$ and $((1,, \text{True}) : \star).l$. 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 ($A \bullet l \rhd B$) 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. $(1 : \star, \text{True})$ 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 $(1 : \star, \text{True})$ 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 $\text{ground}(A)$ function:

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

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, $\lambda x. x : \text{Int} \to \text{Int}$ cast to $\star$ returns $\lambda x. x : \text{Int} \to \text{Int} : \star \to \star : \star$ then, if the value is cast to $\text{Bool} \to \text{Bool}$, the $\star$ type can be dropped safely to obtain $\lambda x. x : \text{Int} \to \text{Int} : \star \to \star : \text{Bool} \to \text{Bool}$. 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 $(1,, \text{True})$ cast to $\star$. If we cast $(1,, \text{True})$ to $\star \& \star$, then by rule CAST-AND, we would yield casting from $(1,, \text{True})$ to $\star$ again, forming a cycle. Therefore, we cast v_1 and v_2 to $\star$ separately. For example, if $(1,, \lambda x. x : \text{Int} \to \text{Int})$ is cast to $\star$, the dynamic

$$\boxed{v \hookrightarrow_A r} \hspace{8cm} \text{(Casting)}$$

$$\frac{}{v \hookrightarrow_\top \mathsf{Top}} \text{ Cast-Top} \hspace{1.5cm} \frac{}{i \hookrightarrow_{\mathsf{Int}} i} \text{ Cast-Lit} \hspace{1.5cm} \frac{v \hookrightarrow_A v'}{\{l = v\} \hookrightarrow_{\{l:A\}} \{l = v'\}} \text{ Cast-Rcd}$$

$$\frac{A \to B \lesssim C \to D}{f : A \to B \hookrightarrow_{C \to D} f : A \to B : C \to D} \text{ Cast-Abs} \hspace{1cm} \frac{v_1 \hookrightarrow_{A^\circ} r_1 \quad v_2 \hookrightarrow_{A^\circ} r_2}{v_1 ,, v_2 \hookrightarrow_{A^\circ} r_1 \vee r_2} \text{ Cast-Merge}$$

$$\frac{v \hookrightarrow_A r_1 \quad v \hookrightarrow_B r_2}{v \hookrightarrow_{A \& B} r_1 \wedge r_2} \text{ Cast-And} \hspace{1.5cm} \frac{s \hookrightarrow_{\mathsf{ground}(\mathsf{ty}(s))} g}{s \hookrightarrow_\star g : \star} \text{ Cast-SD}$$

$$\frac{v_1 \hookrightarrow_\star v_1' \quad v_2 \hookrightarrow_\star v_2'}{v_1 ,, v_2 \hookrightarrow_\star (v_1' ,, v_2') : \star} \text{ Cast-Mergd} \hspace{1cm} \frac{}{g : \star \hookrightarrow_\star g : \star} \text{ Cast-DD} \hspace{1cm} \frac{g \hookrightarrow_{A^\circ} r}{g : \star \hookrightarrow_{A^\circ} r} \text{ Cast-Dyna}$$

$$\frac{\mathsf{ty}(v) \nleqslant A}{v \hookrightarrow_A \mathsf{err}_t} \text{ Cast-Err} \hspace{1cm} \frac{}{v \hookrightarrow_\bot \mathsf{err}_t} \text{ Cast-Bot} \hspace{1cm} \frac{v \hookrightarrow_A \mathsf{err}_*}{\{l = v\} \hookrightarrow_{\{l:A\}} \mathsf{err}_*} \text{ Cast-Rcdp}$$

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

Fig. 8. Casting for the $\lambda M^\star$ 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.

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* $r_1 = r_2$.

$$\boxed{e \hookrightarrow r} \hspace{6cm} \textit{(Small-step Semantics)}$$

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

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

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

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

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

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

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 $\cdot \vdash e \Leftrightarrow A$ 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:

$$(((1 : \star),, (\lambda x.\, (x : \text{Int}) : \star \to \star : \star)) : \star)\, (1,, \text{Top})$$

$\hookrightarrow^*$ {by rule STEP-DYN, rule STEP-EVAL, rule STEP-APP and rule STEP-ANNOV }

$$((\lambda x.\, (x : \text{Int}) : \star \to \star)\, (1,, \text{Top}) : \star) : \star$$

$\hookrightarrow^*$ {by rule STEP-EVAL, rule STEP-APP, rule STEP-BETA and rule STEP-ANNOV }

$$(1 : \star,, \text{Top} : \star) : \star : \text{Int} : \star : \star$$

$\hookrightarrow^*$ {by rule STEP-EVAL and rule STEP-ANNOV }

$$1 : \star$$

In this example, the lambda $(\lambda x.\, (x : \text{Int}) : \star \to \star)$ is extracted by casting $((1 : \star),, (\lambda x.\, (x : \text{Int}) : \star \to \star : \star))$ to $\star \to \star$. The argument $(1,, \text{Top})$ is cast with the function input type $\star$ to obtain $(1 : \star,, \text{Top} : \star) : \star$. 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 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.

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 and Γ, A are static. $\Gamma \vdash e \Leftrightarrow_m A$ if and only if $\Gamma \vdash e \Leftrightarrow A$.*

$$\boxed{e_1 \sqsubseteq e_2} \hspace{6cm} \textit{(Precision relation for expressions)}$$

$$\frac{}{e \sqsubseteq e}\ \text{EP-REFL} \hspace{2cm} \frac{e_1 \sqsubseteq e_2}{\lambda x.\, e_1 \sqsubseteq \lambda x.\, e_2}\ \text{EP-ABS} \hspace{2cm} \frac{e_1 \sqsubseteq e_2}{\text{fix}\ x.\, e_1 \sqsubseteq \text{fix}\ x.\, e_2}\ \text{EP-FIX}$$

$$\frac{e_1 \sqsubseteq e_1' \quad e_2 \sqsubseteq e_2'}{e_1\, e_2 \sqsubseteq e_1'\, e_2'}\ \text{EP-APP} \hspace{1.2cm} \frac{A \sqsubseteq B \quad e_1 \sqsubseteq e_2}{e_1 : A \sqsubseteq e_2 : B}\ \text{EP-ANNO} \hspace{1.2cm} \frac{e_1 \sqsubseteq e_1' \quad e_2 \sqsubseteq e_2'}{e_1 ,,\, e_2 \sqsubseteq e_1' ,,\, e_2'}\ \text{EP-MERGE}$$

$$\frac{e_1 \sqsubseteq e_2}{\{l = e_1\} \sqsubseteq \{l = e_2\}}\ \text{EP-RCD} \hspace{2cm} \frac{e_1 \sqsubseteq e_2}{e_1.l \sqsubseteq e_2.l}\ \text{EP-PROJ}$$

Fig. 10. Expression precision.

THEOREM 5.2 (EQUIVALENCE FOR FULLY-ANNOTATED TERMS (DYNAMIC)). *Suppose that e is fully annotated and Γ, A are static. If $\Gamma \vdash e \Leftrightarrow_m A$ then $e \hookrightarrow_m^* v \Longleftrightarrow e \hookrightarrow^* v$.*

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 GTFL$_\lessgtr$ *Calculus.* We proved that $\lambda M^\star$ can encode the type system of the GTFL$_\lessgtr$ calculus [Bañados Schwerter et al. 2021; Garcia et al. 2016]. In other words every well-typed expression in the GTFL$_\lessgtr$ 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 GTFL$_\lessgtr$. Thus we do not prove an operational correspondence result. An important difference in the semantics is that the original semantics of GTFL$_\lessgtr$ [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 GTFL$_\lessgtr$ are shown in the extended version of the paper. 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 $\bar{\Gamma} \vdash t : S \boxed{\leadsto e}$, which is shown in the extended version of the paper, has an elaboration step from GTFL$_\lessgtr$ expressions t to $\lambda M^\star$ expressions e in the gray portion of the judgement. This elaboration step is used to prove that $\lambda M^\star$ can encode well-typed programs of GTFL$_\lessgtr$. Theorem 5.4 shows that if a term t in GTFL$_\lessgtr$ 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 in the extended version of the paper.

THEOREM 5.4 (WELL-TYPED ENCODING OF GTFL$_\lessgtr$). *If $\bar{\Gamma} \vdash t : S \boxed{\leadsto e}$ 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

$$\mathcal{W}_k^{\Leftarrow}[\![A \sqsubseteq B]\!] = \{(w_1, w_2) \mid (w_1 : A, w_2 : B) \in \mathcal{E}_k^{\Rightarrow}[\![A \sqsubseteq B]\!]\}$$

$$\mathcal{V}_k^{\Rightarrow}[\![A_1 \to A_2 \sqsubseteq B_1 \to B_2]\!] = \{(v_1, v_2) \mid \forall j \leqslant k, \mathsf{ty}(v_1) = A_1 \to A_2, \mathsf{ty}(v_2) = B_1 \to B_2$$
$$(e_1, e_2) \in \mathcal{E}_j^{\Leftarrow}[\![A_1 \sqsubseteq B_1]\!] . (v_1 \ e_1, v_2 \ e_2) \in \mathcal{E}_j^{\Rightarrow}[\![A_2 \sqsubseteq B_2]\!]\}$$

$$\mathcal{V}_k^{\Rightarrow}[\![A_1 \& A_2 \sqsubseteq B_1 \& B_2]\!] = \{((v_{11},v_{12}),(v_{21},v_{22})) \mid (v_{11}, v_{21}) \in \mathcal{V}_{k-1}^{\Rightarrow}[\![A_1 \sqsubseteq B_1]\!]$$
$$\wedge (v_{12}, v_{22}) \in \mathcal{V}_{k-1}^{\Rightarrow}[\![A_2 \sqsubseteq B_2]\!]\}$$

$$\mathcal{V}_k^{\Rightarrow}[\![A \sqsubseteq \star]\!] = \{(s_1, g : \star) \mid \exists s_2 \in g : \star \wedge (s_1, s_2) \in \mathcal{V}_k^{\Rightarrow}[\![\mathsf{ty}(s_1) \sqsubseteq \mathsf{ty}(s_2)]\!]\}$$
$$\cup \{(v_1, v_2, (g : \star)) \mid (v_1, (g : \star)) \in \mathcal{V}_{k-1}^{\Rightarrow}[\![\mathsf{ty}(v_1) \sqsubseteq \star]\!]$$
$$\wedge (v_2, (g : \star)) \in \mathcal{V}_{k-1}^{\Rightarrow}[\![\mathsf{ty}(v_2) \sqsubseteq \star]\!]\}$$
$$\cup \{(g_1 : \star, g_2 : \star) \mid (g_1, g_2 : \star) \in \mathcal{V}_k^{\Rightarrow}[\![\mathsf{ty}(g_1) \sqsubseteq \star]\!]\}$$

$$\mathcal{R}_k^{\Leftrightarrow}[\![A \sqsubseteq B]\!] = \{(r_1, r_2) \mid (r_1 = \mathsf{err}_*) \vee (r_2 = \mathsf{err}_a)\}$$
$$\cup \{(w_1, w_2) \mid (w_1, w_2) \in \mathcal{W}_k^{\Leftrightarrow}[\![A \sqsubseteq B]\!]\}$$

$$\mathcal{E}_k^{\Leftrightarrow}[\![A \sqsubseteq B]\!] = \{(e_1, e_2) \mid \forall j < k, (e_1 \longmapsto_j r_1 \Rightarrow e_2 \longmapsto_* r_2$$
$$\wedge (r_1, r_2) \in \mathcal{R}_{k-j}^{\Leftrightarrow}[\![A \sqsubseteq B]\!])\}$$

$$\mathcal{G}[\![\Gamma_1 \sqsubseteq \Gamma_2]\!] = \{(\sigma_1, \sigma_2) \mid \forall k \geqslant 0, x \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2).$$
$$(\sigma_1(x), \sigma_2(x)) \in \mathcal{V}_k^{\Rightarrow}[\![\Gamma_1(x) \sqsubseteq \Gamma_2(x)]\!]\}$$

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

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

Fig. 11. Logical relation (excerpt).

(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 11. The interpretations of values and expressions are mutually defined using four category of sets: for *irreducible values* at check mode $\mathcal{W}_k^{\Leftarrow}[\![A \sqsubseteq B]\!]$, for values at infer mode $\mathcal{V}_k^{\Rightarrow}[\![A \sqsubseteq B]\!]$, for results at any mode $\mathcal{R}_k^{\Leftrightarrow}[\![A \sqsubseteq B]\!]$, and for expressions or computations at any mode $\mathcal{E}_k^{\Leftrightarrow}[\![A \sqsubseteq B]\!]$. 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 values at the same mode. A pair of expressions

(e_1, e_2) are related at k steps and some mode, if e_1 reduces in j steps to a result, then e_2 must reduce to a related result at $k - j$ steps and to 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 5.5 (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 5.6 (ASCRIPTION LEMMA). if $(v_1, v_2) \in \mathcal{V}_k^{\Rightarrow}[\![A' \sqsubseteq B']\!] \land A' \lesssim A \land B' \lesssim B \land A \sqsubseteq B$ then $(v_1 : A, v_2 : B) \in \mathcal{E}_k^{\Rightarrow}[\![A \sqsubseteq B]\!]$.

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

THEOREM 5.7 (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 $e_2 \longmapsto^* \text{err}_a$.
(2) $e_1 \Uparrow$ then $e_2 \Uparrow$ or $e_2 \longmapsto^* \text{err}_a$.
(3) $e_2 \longmapsto^* v_2$ then $e_1 \longmapsto^* v_1$ and $\vdash v_1 \sqsubseteq v_2 \Leftrightarrow A \sqsubseteq B$ or $e_1 \longmapsto^* \text{err}_*$.
(4) $e_2 \Uparrow$ then $e_1 \Uparrow$ or $e_1 \longmapsto^* \text{err}_*$.

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 $((((1,,\text{True}) : \text{Bool}),,2) : \text{Int}),,\Omega$ diverges, but less precise expression $((((1,,\text{True}) : \star),,2) : \text{Int}),,\Omega$ 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 $((1 : \star),,(2 : \star)) : \text{Int}$ reduces to an ambiguity error. For case (4), $(((1 : \star),,2) : \star),,\Omega$ diverges but the more precise expression $(((1 : \star),,2) : \text{Int}),,\Omega$ 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 ($\text{Ob}_{<:}^?$), which extended $\text{Ob}_{<:}$ [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 GTFL$_\lesssim$ by using the Abstracting Gradual Typing (AGT) methodology. 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 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 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 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 $\lambda M^\star$.

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 $\lambda M^\star$, 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 $\lambda M^\star$ is more fine-grained and it works at the level of expressions. Moreover, $\lambda M^\star$ 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 $\lambda M^{\star}$ 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 $\lambda M^{\star}$, that unifies two type-directed mechanisms: gradual typing and the merge operator. We prove that $\lambda M^{\star}$ is type sound, deterministic and satisfies the static gradual guarantee. $\lambda M^{\star}$ is expressive, and it can encode gradual rows and the GTFL$_{\leqslant}$ calculus using intersection types and the merge operator. In addition $\lambda M^{\star}$ 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 $\lambda M^{\star}$ 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 $\lambda M^{\star}$ and languages such as TypeScript. In particular $\lambda M^{\star}$ 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.

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*.

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*.

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.

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.

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.

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

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.

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*.

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.

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.

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.

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*.

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.

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*.

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.

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)*.

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.

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.

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

CONTENTS

A CP'S SOLUTION FOR THE TYPE UNSOUNDNESS PROBLEM

In this section, we want to emphasize that languages with the merge operator can deal with the type unsoundness problems in TypeScript discussed in Section 2.3. Let us look at the TypeScript example in CP: a programming language based on merges and disjointness. A piece of CP code that models a program similar to the one in TypeScript is:

```
type A = {m : Int; n : Int};

a = trait [this : A] ⇒ {m = 5; n = this.m - 4};
b (t : Trait<A>) = trait [this : A] inherits t ⇒ {m = "hello!"};

o : {n : Int; m : String} = new (b a);
toString (o.n) ++ " " ++ toString (o.m)
```

In this code, we first create a trait a with an integer field m and n, where n depends on m. Then we create a definition b that returns a (first-class) trait, which inherits from an unknown superclass of type A. Moreover b adds a method m of type **String**. Next, we create an object o using b with the superclass a. Here this raises the question: what happens when we call n? In CP, the code above returns 1 hello!. The method m in o returns the string "hello!" as expected. However, the method n *does not* use the definition of m in o; instead it uses the original definition in trait a and returns 5. In other words there is *no accidental overriding* here, which is what happens for example in TypeScript, and this would be a cause of *type unsoundness* (as we shown in Section 2.3). This is avoided with disjointness because a record is allowed to have multiple m as long as their types is disjoint. Thus, type unsafe overrides are avoided with this approach. In addition, we could even have a superclass that contains {m : **String**}, such as:

```
a1 = trait [this : A] inherits a ⇒ {m = "bye!"};
o : {n : Int; m : String} = new (b a1);
```

However, since b expects something with interface A there is an upcast that hides the {m : **String**} present in a1, and still allows the program to work as before. If instead we changed the A's in b to A & {m : **String**} then we would get a static type error complaining about disjointness. In that case the programmer would need to decide whether the method is meant to be overridden or not.

Note that the CP program above is slightly simplified as it assumes that the interface of the superclass is A. In contrast, the TypeScript program makes no such assumption. CP can also deal with this and model a program with similar assumptions to those of TypeScript via *disjoint polymorphism*. For completeness, we show that the version of the program:

```
type A = {m : Int; n : Int};
type B = {n : Int};

a = trait [this : A] ⇒ {m = 5; n = this.m - 4};
b (T * {m : String}) (t : Trait<T & B>)  =
    trait [this : T & B] inherits t ⇒ {m = "hello!"};

o : {n : Int; m : String} = new (b @({m : Int}) a);
toString (o.n) ++ " " ++ toString (o.m)
```

This program outputs the same as the previous one: 1 hello!. The key difference is that in b we express that the class that is returned can be inherited from any superclass that does not contain

{m : **String**} (this is done with the disjointness constraint ⊤ * {m : **String**}). We can find more details about first-class traits with disjoint polymorphism in Bi and Oliveira [2018]'s work.

In conclusion the semantics of casting, merges together with disjointness enables type-sound and expressive designs of languages with first-class classes that avoid the issues found in languages like TypeScript.

B 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
```

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].

C AN ALTERNATIVE SPECIFICATION FOR GRADUAL DISJOINTNESS

In this short section we provide an alternative specification for gradual disjointness that is equivalent to the existential lifting of predicates presented in Section 4.2. A nice aspect of this alternative definition is that it does not involve precision and/or existential types. The alternative formulation of disjointness is as follows:

Definition C.1 (Disjointness Specification). $A *_{\lceil s \rceil} B \equiv \forall C, \lceil A \rceil \lesssim C \land \lceil B \rceil \lesssim C \implies \top \lesssim C$

There are two changes compared to the definition of disjointness in the static language:

- **Replace subtyping by consistent subtyping:** In contrast to the definition of disjointness for the static language, which uses subtyping, we employ consistent subtyping instead.
- **Convert all dynamic types to** $\top$**:** We use the function $\rceil A \lceil$ to replace all occurrences of the unknown type $\star$ by $\top$. The definition of $\rceil A \lceil$ is straightforward and given next:

$\rceil A \lceil$	$=$	B
$\rceil \star \lceil$	$=$	$\top$
$\rceil (A \mathbin{\&} B) \lceil$	$=$	$\rceil A \lceil \mathbin{\&} \rceil B \lceil$
$\rceil (A \to B) \lceil$	$=$	$\rceil A \lceil \to \rceil B \lceil$
$\rceil (\{l : A\}) \lceil$	$=$	$\{l : \rceil A \lceil\}$
$\rceil \mathsf{Int} \lceil$	$=$	Int
$\rceil \top \lceil$	$=$	$\top$
$\rceil \bot \lceil$	$=$	$\bot$

From consistent subtyping, we know that the unknown type $\star$ is a consistent subtype of any type. If, in our definition, we used the gradual types A and B directly, then this would be problematic. The unknown type is a consistent subtype of all types. Therefore we would be able to always find a common super type between the unknown type and some other type. In short, if we did not use $\rceil A \lceil$, the unknown type would never be disjoint to any other type. Consequently, the alternative definition would not be equivalent to the existential lifting of predicates.

The key intuition as to why $\rceil A \lceil$ should be used is that we wish to *defer* ambiguity checks to execution time. That is, if we encounter the unknown type $\star$, then we have no static information available for deciding disjointness with some other type. Thus we would like to check for ambiguity at runtime instead. By replacing the unknown type $\star$ by $\top$ we achieve this, since $\top$ is disjoint to any other types.

Informally, we can see how this idea of $\rceil A \lceil$ relates to the existential lifting of predicates (Def. 4.2). In the existential lifting, we find a pair of static types A' and B' that are more precise than than the gradual types A and B being compared for disjointness. Clearly, an option that we have is to find types that have the same structure as A and B, but simply replace all the unknown types by $\top$, thus obtaining static types as a result. Thus, with $\rceil A \lceil$ we obtain a procedure to find a static type that is more precise than the original gradual type.

Equivalence to the existential lifting. We have proved the definition C.1 is equivalent to the existential lifting definition (definition 4.2) in Theorem C.2. In definition 4.2 we need to find the more precise static type, which we accomplish by using $\rceil A \lceil$.

THEOREM C.2. $A *_{\rceil s \lceil} B$ *iff* $A *_{\mathsf{spec}} B$

D   AUXILIARY RELATIONS

We show some relations which are not presented completely in the main paper.

D.1   Encoding the Static Semantics of the $\mathsf{GTFL}_{\lesssim}$ Calculus

Figure 12 presents the $\mathsf{GTFL}_{\lesssim}$ syntax and type system. The definition of translation for types and contexts are shown as follows.

1422 $\boxed{\text{Syntax}}$

GTypes $S ::= \text{Int} \mid S_1 \to S_2 \mid \{\overline{l : S}\} \mid \{\overline{l : S}, \star\} \mid \star$

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 $\bar{\Gamma} ::= \cdot \mid \bar{\Gamma}, x : S$

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

$$\frac{x : S \in \bar{\Gamma}}{\bar{\Gamma} \vdash x : S \boxed{\leadsto x}} \;\text{ATY-VAR} \qquad\qquad \frac{\bar{\Gamma}, x : S_1 \vdash t : S_2 \boxed{\leadsto e}}{\bar{\Gamma} \vdash \lambda x : S_1.\, t : S_1 \to S_2 \boxed{\leadsto \lambda x.\, e : |S_1| \to |S_2|}} \;\text{ATY-ABS}$$

$$\frac{}{\bar{\Gamma} \vdash i : \text{Int} \boxed{\leadsto i}} \;\text{ATY-I} \qquad \frac{S_3 \lesssim S_1 \qquad \bar{\Gamma} \vdash t_2 : S_3 \boxed{\leadsto e_2} \qquad \bar{\Gamma} \vdash t_1 : S_1 \to S_2 \boxed{\leadsto e_1}}{\bar{\Gamma} \vdash t_1\, t_2 : S_2 \boxed{\leadsto e_1\, e_2}} \;\text{ATY-APP}$$

$$\frac{\bar{\Gamma} \vdash t : S \boxed{\leadsto e}}{\bar{\Gamma} \vdash t.l : \widetilde{\text{proj}}(S, l) \boxed{\leadsto e.l}} \;\text{ATY-PRJ} \qquad \frac{S \lesssim S_1 \qquad \bar{\Gamma} \vdash t : S \boxed{\leadsto e}}{\bar{\Gamma} \vdash (t : S_1) : S_1 \boxed{\leadsto e : |S_1|}} \;\text{ATY-ASSERT}$$

$$\frac{\bar{\Gamma} \vdash t_i : S_i \boxed{\leadsto e_i}}{\bar{\Gamma} \vdash \{\overline{l_i = t_i}\} : \{\overline{l_i : S_i}\} \boxed{\leadsto \{l_1 = e_1\},,\, \ldots\, ,, \{l_n = e_n\}}} \;\text{ATY-REC}$$

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

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

$$\frac{}{\text{Int} \lesssim \text{Int}} \;\text{ACS-Z} \qquad \frac{}{\star \lesssim S} \;\text{ACS-DYNL} \qquad \frac{}{S \lesssim \star} \;\text{ACS-DYNR} \qquad \frac{S_3 \lesssim S_1 \qquad S_2 \lesssim S_4}{S_1 \to S_2 \lesssim S_3 \to S_4} \;\text{ACS-ARR}$$

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

Fig. 12. Type System of GTFL$_\lesssim$.

Definition D.1 (Type Translation). $|S|$ translates the types of GTFL$_\lesssim$ to the types of $\lambda M^\star$.

$$|\text{Int}| = \text{Int} \quad |\star| = \star \quad |(S_1 \to S_2)| = |S_1| \to |S_2|$$

$$|\{\overline{l_i : S_i}\}| = \{l_1 : |S_1|\} \,\&\, \ldots \,\&\, \{l_n : |S_n|\}$$

$$|\{\overline{l_i : S_i}, \star\}| = \{l_1 : |S_1|\} \,\&\, \ldots \,\&\, \{l_n : |S_n|\} \,\&\, \star$$

Definition D.2 (Context Translation). $|\bar{\Gamma}|$ translates the typing context of GTFL$_\lesssim$ to the typing context of $\lambda M^\star$.

$$|\cdot| = \cdot \qquad\qquad\qquad |\bar{\Gamma}, x : S| = |\bar{\Gamma}|, x : |S|$$

D.2 Type System

In figure 13 for λM and $\lambda M^\star$. The well-formed type $\vdash A$ of λM states that Int, $\top$, $\bot$, arrow type $A \to B$ with well-formed input type A and output type B, record type $\{l : A\}$ with well-formed field type A. Especially, intersection types $A \,\&\, B$ are with well-formed and disjoint types A and B.

$\boxed{\vdash A}$ *(Well-Formed Types for λM)*

$$\frac{}{\vdash \mathsf{Int}}\ \text{WF-INT} \qquad \frac{}{\vdash \top}\ \text{WF-TOP} \qquad \frac{}{\vdash \bot}\ \text{WF-BOT} \qquad \frac{\vdash A \quad \vdash B}{\vdash A \to B}\ \text{WF-ARR} \qquad \frac{\vdash A}{\vdash \{l : A\}}\ \text{WF-RCD}$$

$$\frac{\vdash A \quad \vdash B \quad A * B}{\vdash A \,\&\, B}\ \text{WF-AND}$$

$\boxed{A \bullet l \rhd B}$ *(Type Projection for λM)*

$$\frac{}{\{l : A\} \bullet l \rhd A}\ \text{GT-RCD} \qquad \frac{A \bullet l \rhd A_1 \quad \neg(l \in B)}{(A \,\&\, B) \bullet l \rhd A_1}\ \text{GT-ANDL} \qquad \frac{B \bullet l \rhd B_1 \quad \neg(l \in A)}{(A \,\&\, B) \bullet l \rhd B_1}\ \text{GT-ANDR}$$

$\boxed{\vdash A}$ *(Well-Formed Types for λM*)*

$$\frac{}{\vdash \mathsf{Int}}\ \text{WF-INT} \qquad \frac{}{\vdash \top}\ \text{WF-TOP} \qquad \frac{}{\vdash \bot}\ \text{WF-BOT} \qquad \frac{\vdash A \quad \vdash B}{\vdash A \to B}\ \text{WF-ARR} \qquad \frac{\vdash A}{\vdash \{l : A\}}\ \text{WF-RCD}$$

$$\frac{\vdash A \quad \vdash B \quad A * B}{\vdash A \,\&\, B}\ \text{WF-AND} \qquad\qquad \frac{}{\vdash \star}\ \text{WF-DYN}$$

$\boxed{A \bullet l \rhd B}$ *(Type Projection for λM*)*

$$\frac{}{\{l : A\} \bullet l \rhd A}\ \text{GT-RCD} \qquad \frac{A \bullet l \rhd A_1 \quad \neg(l \in B) \quad l \in A}{(A \,\&\, B) \bullet l \rhd A_1}\ \text{GT-ANDL}$$

$$\frac{B \bullet l \rhd B_1 \quad \neg(l \in A) \quad l \in B}{(A \,\&\, B) \bullet l \rhd B_1}\ \text{GT-ANDR} \qquad\qquad \frac{A <: \star \quad \neg(l \in A)}{A \bullet l \rhd \star}\ \text{GT-DYN}$$

$\boxed{l \in A}$ *(Label Presence)*

$$\frac{}{l \in \{l : A\}}\ \text{ITY-RCD} \qquad \frac{l \in A_1}{l \in A_1 \,\&\, A_2}\ \text{ITY-MERGEL} \qquad \frac{l \in A_2}{l \in A_1 \,\&\, A_2}\ \text{ITY-MERGER}$$

Fig. 13. Auxiliary relations for type systems.

The well-formed type definition of λM* is extended with the unknown type $\star$. We have a relation $A \bullet l \rhd B$ for λM to extract the type of fields in record types. To avoid the ambiguity, we place a restriction to forbid multiple occurrences of the same label in types. This restriction is supported by a relation that checks if a label occurs in a type ($l \in A$), which is defined at the bottom of Figure 13. Type projection of λM* is an extension. If the type of the projected expression is a subtype of $\star$ (which we call a dynamic-like type), then the projection returns a result of type $\star$. If a dynamic-like type such as $\star \,\&\, \mathsf{Int}$ and $\star \,\&\, \star$ is allowed, label l may not be in type A. Because of the presence of the dynamic type, rule GT-ANDL and rule GT-ANDR should add the restriction about the label presence.

D.3 Call-by-Name

Figure 14 shows the call-by-name variant for λM*. The frame of argument for applications is omitted (($\lambda x.\, e$) $\square$). The Notable different rule is rule CBN-BETA. The argument e_2 does not need to be a value.

Frames $F ::= \square\, e \mid v\, ,, \square \mid \square\, ,, e \mid \{l = \square\} \mid \square.l \mid \square : A$

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

$$\frac{e \hookrightarrow e'}{F[e] \hookrightarrow F[e']}\ \text{CBN-EVAL} \qquad \frac{e \hookrightarrow \text{err}_*}{F[e] \hookrightarrow \text{err}_*}\ \text{CBN-BLAME} \qquad \frac{v \hookrightarrow_A \text{err}_*}{v : A \hookrightarrow \text{err}_*}\ \text{CBN-ANNOP}$$

$$\frac{}{(g : \star)\, e \hookrightarrow ((g : \star) : \star \to \star)\, e}\ \text{CBN-DYN} \qquad \frac{}{(\lambda x.\, e) : \star \hookrightarrow (\lambda x.\, e) : \star \to \star : \star}\ \text{CBN-ABS}$$

$$\frac{\text{ty}(v) \bullet l \triangleright A \qquad v \hookrightarrow_{\{l:A\}} \{l = v'\}}{v.l \hookrightarrow v'}\ \text{CBN-PROJ} \qquad \frac{\text{ty}(v) \bullet l \triangleright A \qquad v \hookrightarrow_{\{l:A\}} \text{err}_*}{v.l \hookrightarrow \text{err}_*}\ \text{CBN-PROJP}$$

$$\frac{v \hookrightarrow_A v' \qquad \text{NotVal}\,(v : A)}{v : A \hookrightarrow v'}\ \text{CBN-ANNOV} \qquad \frac{}{(f : A_1 \to A_2)\, e \hookrightarrow (f\, (e : A_1)) : A_2}\ \text{CBN-APP}$$

$$\frac{}{(\lambda x.\, e_1)\, e_2 \hookrightarrow e_1[x \mapsto e_2]}\ \text{CBN-BETA} \qquad \frac{}{(\text{fix}\, x.\, e) : A \hookrightarrow e[x \mapsto (\text{fix}\, x.\, e) : A] : A}\ \text{CBN-FIX}$$

Fig. 14. Call-by-Name Semantics of $\lambda M^\star$.

D.4 Encoding Dynamic Lambdas

The syntactic sugar for dynamic lambdas, we do not insert $\star \to \star$ every time for raw lambdas. We can do a simple optimization by exploiting bidirectional type-checking. Instead of blindly adding $\star \to \star$ annotations to lambdas without type annotations, we only need to add $\star \to \star$ to lambdas in inference positions. In essence, when raw lambdas are in checking positions, the types can be inferred from the contextual type information. The inserted function ($\lceil . \rceil_b$) is shown as follows and the boolean flag b indicates whether the dynamic function type should be inserted on raw lambdas or not:

$$\lceil x \rceil_b = x$$
$$\lceil i \rceil_b = i$$
$$\lceil \lambda x.\, e \rceil_{\text{true}} = \lambda x. \lceil e \rceil_{\text{false}} : \star$$
$$\lceil \lambda x.\, e \rceil_{\text{true}} = \lambda x. \lceil e \rceil_{\text{false}}$$
$$\lceil e_1\, e_2 \rceil_b = \lceil e_1 \rceil_{\text{true}}\, \lceil e_2 \rceil_{\text{false}}$$
$$\lceil e_1\, ,, e_2 \rceil_b = \lceil e_1 \rceil_{\text{true}}\, ,, \lceil e_2 \rceil_{\text{true}}$$
$$\lceil \{l = e\} \rceil_b = \{l : \lceil e \rceil_{\text{true}}\}$$
$$\lceil \text{fix}\, x.\, e \rceil_{\text{true}} = \text{fix}\, x. \lceil e \rceil_{\text{false}} : \star$$
$$\lceil \text{fix}\, x.\, e \rceil_{\text{false}} = \text{fix}\, x. \lceil e \rceil_{\text{false}}$$

For example, the dynamic expression $(\lambda x.\, \lambda y.\, x\, y)\, (\lambda x.\, x)\, 1$ can be inserted to be $((\lambda x.\, \lambda y.\, x\, y) : \star)\, (\lambda x.\, x)\, 1$. Only one unknown type $\star$ is inserted, despite the presence of 3 lambda expressions without annotations in the original expression.

E DYNAMIC GRADUAL GUARANTEE

Name	Reference
Ascription Lemma	Lemma E.4
Fundamental Property	Theorem E.11
Left Direction	Theorem E.16
Dynamic Gradual Guarantee	Theorem E.17

Table 1. Lemmas.

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

$$\boxed{s \in v}$$

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

Fig. 15. Logical relation.

In this section, we show the proof for dynamic gradual guarantee. The complete logic relation is shown in Figure 15. Note that the inferred mode of irreducible values $\mathcal{W}_k^{\Rightarrow}[\![A \sqsubseteq B]\!]$ is in the set of inferred mode of values $\mathcal{V}_k^{\Rightarrow}[\![A \sqsubseteq B]\!]$ and the checked mode of values $\mathcal{V}_k^{\Leftarrow}[\![A \sqsubseteq B]\!]$ is in the set of checked mode of $\mathcal{W}_k^{\Leftarrow}[\![A \sqsubseteq B]\!]$. Throughout the proof, we induction on the relation in Figure 16. Table 1 shows the overview of the important lemmas.

LEMMA E.1. *if* $(v_1, v_2) \in \mathcal{V}_k^{\Rightarrow}[\![A \sqsubseteq B]\!]$ *then* $\forall j \leqslant k, (v_1, v_2) \in \mathcal{V}_j^{\Rightarrow}[\![A \sqsubseteq B]\!]$

PROOF. by induction on (k, A, B) and case analysis on $(v_1, v_2) \in \mathcal{V}_k^{\Rightarrow}[\![A \sqsubseteq B]\!]$.

Case (i, Top). These cases are trivial.

$$(k, A, B) < (k', A', B') \Longleftrightarrow k < k' \vee (k = k' \vee (A, B) < (A', B'))$$

$$(k, A_1, A_2, B_1, B_2) < (k', A_1', A_2', B_1', B_2') \Longleftrightarrow k < k' \vee (k = k' \vee (A_1, A_2, B_1, B_2) < (A_1', A_2', B_1', B_2'))$$

$$(A_1, A_2, B_1, B_2) < (A_1', A_1', B_1', B_2') \Longleftrightarrow ((A_1, A_2) < (A_1', A_2') \wedge (B_1, B_2) < (B_1', B_2')) \vee$$

$$((A_1, A_2) < (A_1', A_2') \wedge (B_1, B_2) = (B_1', B_2')) \vee$$

$$((A_1, A_2) = (A_1', A_2') \wedge (B_1, B_2) < (B_1', B_2'))$$

$$(A, B) < (A', B') \Longleftrightarrow (A < A' \wedge B < B') \vee$$

$$(A < A' \wedge B = B') \vee$$

$$(A = A' \wedge B < B')$$

$$\frac{}{\mathsf{ground}(A) < \star}$$

Fig. 16. Induction.

Case (f).
we have
$(f_1, f_2) \in \mathcal{V}_k^{\Rightarrow}[\![A_1 \to A_2 \sqsubseteq B_1 \to B_2]\!]$
$\Rightarrow \forall j \leqslant k, (e_1, e_2) \in \mathcal{E}_j^{\Leftarrow}[\![A_1 \sqsubseteq B_1]\!].$
$(f_1 \ e_1, f_2 \ e_2) \in \mathcal{E}_j^{\Rightarrow}[\![A_2 \sqsubseteq B_2]\!]$
we want to prove
$\forall j' \leqslant k, (f_1, f_2) \in \mathcal{V}_{j'}^{\Rightarrow}[\![A_1 \to A_2 \sqsubseteq B_1 \to B_2]\!]$
$\Rightarrow \forall j'' \leqslant j', (e_1, e_2) \in \mathcal{E}_{j''}^{\Leftarrow}[\![A_1 \sqsubseteq B_1]\!].$
$(f_1 \ e_1, f_2 \ e_2) \in \mathcal{E}_{j''}^{\Rightarrow}[\![A_2 \sqsubseteq B_2]\!]$
since $j'' \leqslant j'$ and $j' \leqslant k$
then $j'' \leqslant k$, thus the result holds.

Case ($\{l = v\}$).
we want to prove
$j \leqslant k, (\{l = v_1\}, \{l = v_2\}) \in \mathcal{V}_j^{\Rightarrow}[\![\{l : A\} \sqsubseteq \{l : B\}]\!]$
$\Rightarrow (v_1, v_2) \in \mathcal{V}_{j-1}^{\Rightarrow}[\![A \sqsubseteq B]\!]$
we have
$(\{l = v_1\}, \{l = v_2\}) \in \mathcal{V}_k^{\Rightarrow}[\![\{l : A\} \sqsubseteq \{l : B\}]\!]$
$\Rightarrow (v_1, v_2) \in \mathcal{V}_{k-1}^{\Rightarrow}[\![A \sqsubseteq B]\!]$
by the induction hypothesis, we have
$\forall j' \leqslant k - 1, (v_1, v_2) \in \mathcal{V}_{j'}^{\Rightarrow}[\![A \sqsubseteq B]\!]$
since $j' \leqslant j - 1$
Thus the result holds.

Case (v_1, v_2).
we want to prove
$j \leqslant k, (v_1, v_2, v_3, v_4) \in \mathcal{V}_j^{\Rightarrow}[\![A_1 \& A_2 \sqsubseteq B_1 \& B_2]\!]$
$\Rightarrow$
$(v_1, v_3) \in \mathcal{V}_{j-1}^{\Rightarrow}[\![A_1 \sqsubseteq B_1]\!]$
$(v_2, v_4) \in \mathcal{V}_{j-1}^{\Rightarrow}[\![A_2 \sqsubseteq B_2]\!]$
we have
$(v_1, v_2, v_3, v_4) \in \mathcal{V}_k^{\Rightarrow}[\![A_1 \& A_2 \sqsubseteq B_1 \& B_2]\!]$

1667 $\Rightarrow$

1668 $(v_1, v_3) \in \mathcal{V}_{k-1}^{\Rightarrow}[\![A_1 \sqsubseteq B_1]\!]$

1669 $(v_2, v_4) \in \mathcal{V}_{k-1}^{\Rightarrow}[\![A_2 \sqsubseteq B_2]\!]$

1670 by the induction hypothesis, we have

1671 $\forall j' \leqslant k-1,$

1672 $(v_1, v_3) \in \mathcal{V}_{j'}^{\Rightarrow}[\![A_1 \sqsubseteq B_1]\!]$

1673 $(v_2, v_4) \in \mathcal{V}_{j'}^{\Rightarrow}[\![A_2 \sqsubseteq B_2]\!]$

1674 since $j' \leqslant j - 1$

1675 Thus the result holds.

1676

1677 *Case* $(s_1, g_2 : \star)$.

1678 we want to prove

1679 $j \leqslant k, (s_1, g_2 : \star) \in \mathcal{V}_j^{\Rightarrow}[\![A \sqsubseteq \star]\!]$

1680 $\Rightarrow s_2 \in g_2 : \star$

1681 $(s_1, s_2) \in \mathcal{V}_j^{\Rightarrow}[\![A \sqsubseteq B]\!]$

1682 we have

1683 $(s_1, g_2 : \star) \in \mathcal{V}_k^{\Rightarrow}[\![A \sqsubseteq \star]\!]$

1684 $\Rightarrow$

1685 $(s_1, s_2) \in \mathcal{V}_k^{\Rightarrow}[\![A \sqsubseteq B]\!]$

1686 by the induction hypothesis, we have

1687 $\forall j' \leqslant k,$

1688 $(s_1, s_2) \in \mathcal{V}_{j'}^{\Rightarrow}[\![A \sqsubseteq B]\!]$

1689 since $j' \leqslant j$

1690 Thus the result holds.

1691 *Case* $(v_1,, v_2, g_3 : \star)$.

1692 we want to prove

1693 $j \leqslant k, (v_1,, v_2, g_3 : \star) \in \mathcal{V}_j^{\Rightarrow}[\![A_1 \& A_2 \sqsubseteq \star]\!]$

1694 $\Rightarrow$

1695 $(v_1, g_3 : \star) \in \mathcal{V}_{j-1}^{\Rightarrow}[\![A_1 \sqsubseteq \star]\!]$

1696 $(v_2, g_3 : \star) \in \mathcal{V}_{j-1}^{\Rightarrow}[\![A_2 \sqsubseteq \star]\!]$

1697 we have

1698 $(v_1,, v_2, g_3 : \star) \in \mathcal{V}_k^{\Rightarrow}[\![A_1 \& A_2 \sqsubseteq \star]\!]$

1699 $\Rightarrow$

1700 $(v_1, v_3) \in \mathcal{V}_{k-1}^{\Rightarrow}[\![A_1 \sqsubseteq B_1]\!]$

1701 $(v_2, v_4) \in \mathcal{V}_{k-1}^{\Rightarrow}[\![A_2 \sqsubseteq B_2]\!]$

1702 by the induction hypothesis, we have

1703 $\forall j' \leqslant k-1,$

1704 $(v_1, g_3 : \star) \in \mathcal{V}_{j'}^{\Rightarrow}[\![A_1 \sqsubseteq \star]\!]$

1705 $(v_2, g_3 : \star) \in \mathcal{V}_{j'}^{\Rightarrow}[\![A_2 \sqsubseteq \star]\!]$

1706 since $j' \leqslant j - 1$

1707 Thus the result holds.

1708

1709 *Case* $(g_1 : \star, g_2 : \star)$.

1710 we want to prove

1711 $j \leqslant k, (g_1 : \star, g_2 : \star) \in \mathcal{V}_j^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

1712 $\Rightarrow$

1713 $(g_1, g_2 : \star) \in \mathcal{V}_j^{\Rightarrow}[\![A_1 \sqsubseteq \star]\!]$

1714 we have

1715

1716 $(g_1 : \star, g_2 : \star) \in \mathcal{V}_k^{\Rightarrow}\llbracket \star \sqsubseteq \star \rrbracket$

1717 $\Rightarrow$

1718 $(g_1, g_2 : \star) \in \mathcal{V}_k^{\Rightarrow}\llbracket A_1 \sqsubseteq \star \rrbracket$

1719   by the induction hypothesis, we have

1720 $\forall j' \leqslant k,$

1721 $(g_1, g_2 : \star) \in \mathcal{V}_{j'}^{\Rightarrow}\llbracket A_1 \sqsubseteq \star \rrbracket$

1722   since $j' \leqslant j$

1723   Thus the result holds.

1724

1725

                                                                    □

1726   LEMMA E.2. $e : \star \longmapsto^* g : \star$ *iff* $e : \star : \star \longmapsto^* g : \star$

1727   PROOF.

1728   we have $e : \star \longmapsto^* g : \star$

1729   then we have $e \longmapsto^* v'$

1730   then we have $v' : \star \longmapsto^* g : \star$

1731   then we have $e : \star : \star \longmapsto^* v' : \star : \star$

1732   then $v' : \star : \star \longmapsto^* g : \star : \star$

1733   then $g : \star : \star \longmapsto^* g : \star$

1734   so $e : \star : \star \longmapsto^* g : \star$

1735   Thus the result holds.   □

1736

1737   LEMMA E.3. $(r_1, r_2) \in \mathcal{R}_k^{\Rightarrow}\llbracket A \sqsubseteq B \rrbracket$ *and* $(r_3, r_4) \in \mathcal{R}_k^{\Rightarrow}\llbracket A \sqsubseteq B \rrbracket$ *then* $(r_1 \vee r_3, r_2 \vee r_4) \in$

1738 $\mathcal{R}_k^{\Rightarrow}\llbracket A \sqsubseteq B \rrbracket$

1739   PROOF.

1740

1741   *Case* (err_a). $r_1 = err_a \vee r_3 = err_a \vee r_2 = err_a \vee r_4 = err_a$ the result holds.

1742   *Case* (err_t). $r_1 = err_t \wedge r_3 = err_t$ the result holds.

1743

1744   *Case* (err_t, v). if $r_1 = err_t \wedge r_3 = v_3$ and $r_2 = err_t \wedge r_4 = v_4$

1745   we want to prove

1746 $(r_1 \vee r_3, r_2 \vee r_4) \in \mathcal{R}_k^{\Rightarrow}\llbracket A \sqsubseteq B \rrbracket$

1747 $\Rightarrow (v_3, v_4) \in \mathcal{R}_k^{\Rightarrow}\llbracket A \sqsubseteq B \rrbracket$

1748   the result holds.

1749   *Case* (err_t, v). if $r_1 = v_1 \wedge r_3 = v_1$ and $r_2 = v_2 \wedge r_4 = v_2$

1750   we want to prove

1751 $(r_1 \vee r_3, r_2 \vee r_4) \in \mathcal{R}_k^{\Rightarrow}\llbracket A \sqsubseteq B \rrbracket$

1752 $\Rightarrow (v_1, v_2) \in \mathcal{R}_k^{\Rightarrow}\llbracket A \sqsubseteq B \rrbracket$

1753   the result holds.

1754

1755   *Case* (err_t, v). if $r_1 = v_1 \wedge r_3 = v_2 \wedge v_1 \neq v_2$ or $r_2 = v_3 \wedge r_4 = v_4 \wedge v_3 \neq v_4$

1756   the result holds.

1757

                                                                      □

1758

1759   LEMMA E.4 (ASCRIPTION LEMMA). *if* $(v_1, v_2) \in \mathcal{V}_k^{\Rightarrow}\llbracket A' \sqsubseteq B' \rrbracket \wedge A' \lesssim A \wedge B' \lesssim B \wedge A \sqsubseteq B$

1760   *then* $(v_1 : A, v_2 : B) \in \mathcal{E}_k^{\Rightarrow}\llbracket A \sqsubseteq B \rrbracket$

1761   PROOF. We do the induction on (k, A', B', A, B) and case analysis on $(v_1, v_2) \in \mathcal{V}_k^{\Rightarrow}\llbracket A' \sqsubseteq B' \rrbracket$.

1762   Note that throughout the proof, if the $v_2 : B$ reduces to err_t, then by Lemma E.12, $v_1 : A$ reduces

1763   to err_*, thus the result holds. Furthermore, if $v_2 : B$ reduces to err_a, the result holds directly.

1764

1765   *Case* $(\mathsf{Top}, \mathsf{Top}, \top, \top)$. We have

1766 $(\mathsf{Top}, \mathsf{Top}) \in \mathcal{V}_k^{\rightrightarrows}[\![\top \sqsubseteq \top]\!]$

1767   we want to prove

1768 $(\mathsf{Top} : \top, \mathsf{Top} : \top) \in \mathcal{E}_k^{\rightrightarrows}[\![\top \sqsubseteq \top]\!]$

1769 $\Rightarrow (\mathsf{Top}, \mathsf{Top}) \in \mathcal{V}_{k-1}^{\rightrightarrows}[\![\top \sqsubseteq \top]\!]$

1770   by Lemma E.1, the result holds.

1771

1772   *Case* $(\mathsf{Top}, \mathsf{Top}, \top, \star)$. We have

1773 $(\mathsf{Top}, \mathsf{Top}) \in \mathcal{V}_k^{\rightrightarrows}[\![\top \sqsubseteq \top]\!]$

1774   we want to prove

1775 $(\mathsf{Top} : \top, \mathsf{Top} : \star) \in \mathcal{E}_k^{\rightrightarrows}[\![\top \sqsubseteq \star]\!]$

1776 $\Rightarrow (\mathsf{Top}, \mathsf{Top} : \star) \in \mathcal{V}_{k-1}^{\rightrightarrows}[\![\top \sqsubseteq \star]\!]$

1777   by the definition of related values at $\top \sqsubseteq \star$,

1778 $\Rightarrow (\mathsf{Top}, \mathsf{Top}) \in \mathcal{V}_{k-1}^{\rightrightarrows}[\![\top \sqsubseteq \top]\!]$

1779   by Lemma E.1, the result holds.

1780   *Case* $(\mathsf{Top}, \mathsf{Top}, \star, \star)$. We have

1781 $(\mathsf{Top}, \mathsf{Top}) \in \mathcal{V}_k^{\rightrightarrows}[\![\top \sqsubseteq \top]\!]$

1782   we want to prove

1783 $(\mathsf{Top} : \star, \mathsf{Top} : \star) \in \mathcal{V}_k^{\rightrightarrows}[\![\star \sqsubseteq \star]\!]$

1784   by the definition of related values at $\star \sqsubseteq \star$,

1785 $\Rightarrow (\mathsf{Top}, \mathsf{Top} : \star) \in \mathcal{V}_k^{\rightrightarrows}[\![\top \sqsubseteq \star]\!]$

1786   by the definition of related values at $\top \sqsubseteq \star$,

1787 $\Rightarrow (\mathsf{Top}, \mathsf{Top}) \in \mathcal{V}_k^{\rightrightarrows}[\![\top \sqsubseteq \top]\!]$

1788   by Lemma E.1, the result holds.

1789

1790   *Case* $(\mathsf{i}, \mathsf{i}, \mathsf{Int}, \mathsf{Int})$. We have

1791 $(\mathsf{i}, \mathsf{i}) \in \mathcal{V}_k^{\rightrightarrows}[\![\mathsf{Int} \sqsubseteq \mathsf{Int}]\!]$

1792   we want to prove

1793 $(\mathsf{i} : \mathsf{Int}, \mathsf{i} : \mathsf{Int}) \in \mathcal{E}_k^{\rightrightarrows}[\![\mathsf{Int} \sqsubseteq \mathsf{Int}]\!]$

1794   by the definition of related values at $\mathsf{Int} \sqsubseteq \star$,

1795 $\Rightarrow (\mathsf{i}, \mathsf{i}) \in \mathcal{V}_{k-1}^{\rightrightarrows}[\![\mathsf{Int} \sqsubseteq \mathsf{Int}]\!]$

1796   by Lemma E.1, the result holds.

1797   *Case* $(\mathsf{i}, \mathsf{i}, \mathsf{Int}, \star)$. We have

1798 $(\mathsf{i}, \mathsf{i}) \in \mathcal{V}_k^{\rightrightarrows}[\![\mathsf{Int} \sqsubseteq \mathsf{Int}]\!]$

1799   we want to prove

1800 $(\mathsf{i} : \mathsf{Int}, \mathsf{i} : \star) \in \mathcal{E}_k^{\rightrightarrows}[\![\mathsf{Int} \sqsubseteq \star]\!]$

1801 $\Rightarrow (\mathsf{i}, \mathsf{i} : \star) \in \mathcal{V}_{k-1}^{\rightrightarrows}[\![\mathsf{Int} \sqsubseteq \star]\!]$

1802   by the definition of related values at $\mathsf{Int} \sqsubseteq \star$,

1803 $\Rightarrow (\mathsf{i}, \mathsf{i}) \in \mathcal{V}_{k-1}^{\rightrightarrows}[\![\mathsf{Int} \sqsubseteq \mathsf{Int}]\!]$

1804   by Lemma E.1, the result holds.

1805

1806   *Case* $(\mathsf{i}, \mathsf{i}, \star, \star)$. We have

1807 $(\mathsf{i}, \mathsf{i}) \in \mathcal{V}_k^{\rightrightarrows}[\![\mathsf{Int} \sqsubseteq \mathsf{Int}]\!]$

1808   we want to prove

1809 $(\mathsf{i} : \star, \mathsf{i} : \star) \in \mathcal{V}_k^{\rightrightarrows}[\![\star \sqsubseteq \star]\!]$

1810   by the definition of related values at $\star \sqsubseteq \star$,

1811 $\Rightarrow (\mathsf{i}, \mathsf{i} : \star) \in \mathcal{V}_k^{\rightrightarrows}[\![\mathsf{Int} \sqsubseteq \star]\!]$

1812   by the definition of related values at $\mathsf{Int} \sqsubseteq \star$,

1813

1814 $\Rightarrow (i, i) \in \mathcal{V}_k^{\Rightarrow} [\![\text{Int} \sqsubseteq \text{Int}]\!]$

1815   by Lemma E.1, the result holds.

1816

1817   *Case* $(f_1, f_2, A_1 \rightarrow B_1, A_2 \rightarrow B_2)$. We have

1818 $(f_1, f_2) \in \mathcal{V}_k^{\Rightarrow} [\![A_1' \rightarrow A_2' \sqsubseteq B_1' \rightarrow B_2']\!]$

1819 $\Rightarrow \forall j' \leqslant k, (e_1', e_2') \in \mathcal{E}_{j'}^{\Leftarrow} [\![A_1' \sqsubseteq B_1']\!].$

1820 $(f_1 \; e_1', f_2 \; e_2') \in \mathcal{E}_{j'}^{\Rightarrow} [\![A_2' \sqsubseteq B_2']\!]$

1821   we want to prove

1822 $(f_1 : A_1 \rightarrow A_2, f_2 : B_1 \rightarrow B_2) \in \mathcal{E}_k^{\Rightarrow} [\![A_1 \rightarrow A_2 \sqsubseteq B_1 \rightarrow B_2]\!]$

1823 $\Rightarrow \forall j \leqslant k, (e_1, e_2) \in \mathcal{E}_j^{\Leftarrow} [\![A_1 \sqsubseteq B_1]\!].$

1824 $((f_1 : A_1 \rightarrow A_2) \; e_1, (f_2 : B_1 \rightarrow B_2) \; e_2) \in \mathcal{E}_j^{\Rightarrow} [\![A_2 \sqsubseteq B_2]\!]$

1825 $\Rightarrow ((f_1 \; (e_1 : A_1)) : A_2, (f_2 \; (e_2 : B_1)) : B_2) \in \mathcal{E}_{j-1}^{\Rightarrow} [\![A_2 \sqsubseteq B_2]\!]$

1826   if $(e_1 : A_1)$ reduce to an err_*, the result vacuously holds.

1827   Let us assume it reduces to a value v_1 in $j_1 \geqslant 0$ steps

1828 $\Rightarrow ((f_1 \; v_1) : A_2, (f_2 \; v_2) : B_2) \in \mathcal{E}_{j-1-j_1}^{\Rightarrow} [\![A_2 \sqsubseteq B_2]\!]$, for some v_2

1829   if $((f_1 \; v_1))$ reduce to an err_*, the result vacuously holds.

1830   Let us assume it reduces to a value v_3 in $j_2 \geqslant 0$

1831 $\Rightarrow (v_3 : A_2, v_4 : B_2) \in \mathcal{E}_{j-1-j_1-j_2}^{\Rightarrow} [\![A_2 \sqsubseteq B_2]\!]$

1832   since

1833 $(e_1, e_2) \in \mathcal{E}_j^{\Leftarrow} [\![A_1 \sqsubseteq B_1]\!]$

1834   by the definition of related values at check mode

1835 $\Rightarrow (e_1 : A_1, e_2 : B_1) \in \mathcal{E}_j^{\Rightarrow} [\![A_1 \sqsubseteq B_1]\!]$

1836 $\Rightarrow (v_1, v_2) \in \mathcal{E}_{j-j_1}^{\Rightarrow} [\![A_1 \sqsubseteq B_1]\!]$

1837   by lemma E.1

1838 $\Rightarrow (v_1, v_2) \in \mathcal{E}_{j-1-j_1}^{\Rightarrow} [\![A_1 \sqsubseteq B_1]\!]$

1839   by induction hypothesis,

1840 $\Rightarrow (v_1 : A_1', v_2 : B_1') \in \mathcal{E}_{j-1-j_1}^{\Rightarrow} [\![A_1' \sqsubseteq B_1']\!]$

1841   by the definition of related values at check mode

1842   then $(v_1, v_2) \in \mathcal{W}_{j-1-j_1}^{\Leftarrow} [\![A_1' \sqsubseteq B_1']\!]$

1843   since

1844 $\Rightarrow \forall j' \leqslant k, (e_1', e_2') \in \mathcal{E}_{j'}^{\Leftarrow} [\![A_1' \sqsubseteq B_1']\!].$

1845 $(f_1 \; e_1', f_2 \; e_2') \in \mathcal{E}_{j'}^{\Rightarrow} [\![A_2' \sqsubseteq B_2']\!]$

1846   let $e_1' = v_1, e_2' = v_2, j' = j - 1 - j_1$

1847   then

1848 $(f_1 \; v_1, f_2 \; v_2) \in \mathcal{E}_{j-1-j_1}^{\Rightarrow} [\![A_2' \sqsubseteq B_2']\!]$

1849   then

1850 $(v_3, v_4) \in \mathcal{V}_{j-1-j_1-j_2}^{\Rightarrow} [\![A_2' \sqsubseteq B_2']\!]$

1851   by induction hypothesis,

1852 $(v_3 : A_2, v_4 : B_2) \in \mathcal{E}_{k-1-j_1-j_2}^{\Rightarrow} [\![A_2 \sqsubseteq B_2]\!]$

1853   Thus the result holds.

1854

1855   *Case* $(f_1, f_2, A_1 \rightarrow A_2, \star)$.

1856   if the type of $f_2 = \star \rightarrow \star$

1857   We have

1858 $(f_1, f_2) \in \mathcal{V}_k^{\Rightarrow} [\![A_1' \rightarrow A_2' \sqsubseteq \star \rightarrow \star]\!]$

1859   we want to prove

1860 $(f_1 : A_1 \rightarrow A_2, f_2 : \star) \in \mathcal{V}_k^{\Rightarrow} [\![A_1 \rightarrow A_2 \sqsubseteq \star]\!]$

1861   by the definition of related values at $A_1 \rightarrow A_2 \sqsubseteq \star$

1862

1863 $\Rightarrow (f_1 : A_1 \to A_2, f_2) \in \mathcal{V}_k^{\Rightarrow}[\![A_1 \to A_2 \sqsubseteq \star \to \star]\!]$

1864 $\Rightarrow \forall j \leqslant k,$

1865 $(e_1, e_2) \in \mathcal{E}_j^{\Leftarrow}[\![A_1 \sqsubseteq \star]\!].$

1866 $((f_1 : A_1 \to A_2)\ e_1, f_2\ e_2) \in \mathcal{E}_j^{\Rightarrow}[\![A_2 \sqsubseteq \star]\!]$

1867 It must be the case that $f_2 = f_2' : \star \to \star$, for some f_2', by definition of ground functions.

1868 $\Rightarrow \forall j \leqslant k,$

1869 $(e_1, e_2) \in \mathcal{E}_j^{\Leftarrow}[\![A_1 \sqsubseteq \star]\!].$

1870 $((f_1 : A_1 \to A_2)\ e_1, (f_2' : \star \to \star)\ e_2) \in \mathcal{E}_j^{\Rightarrow}[\![A_2 \sqsubseteq \star]\!]$

1871 $\Rightarrow ((f_1 : A_1 \to A_2)\ e_1, (f_2'\ (e_2 : \star)) : \star) \in \mathcal{E}_j^{\Rightarrow}[\![A_2 \sqsubseteq \star]\!]$

1872 by induction hypothesis on B',

1873 $(f_1 : A_1 \to A_2, f_2 : \star \to \star) \in \mathcal{V}_k^{\Rightarrow}[\![A_1 \to A_2 \sqsubseteq \star \to \star]\!]$

1874 $\Rightarrow \forall j' \leqslant k,$

1875 $(e_1', e_2') \in \mathcal{E}_{j'}^{\Leftarrow}[\![A_1 \sqsubseteq \star]\!].$

1876 $((f_1 : A_1 \to A_2)\ e_1', (f_2 : \star \to \star)\ e_2') \in \mathcal{E}_{j'}^{\Rightarrow}[\![A_2 \sqsubseteq \star]\!]$

1877

1878 $\Rightarrow ((f_1 : A_1 \to A_2)\ e_1', (f_2\ (e_2' : \star)) : \star) \in \mathcal{E}_{j'}^{\Rightarrow}[\![A_2 \sqsubseteq \star]\!]$

1879 $\Rightarrow ((f_1 : A_1 \to A_2)\ e_1', ((f_2' : \star \to \star)\ (e_2' : \star)) : \star) \in \mathcal{E}_{j'}^{\Rightarrow}[\![A_2 \sqsubseteq \star]\!]$

1880 $\Rightarrow ((f_1 : A_1 \to A_2)\ e_1', (f_2'\ (e_2' : \star : \star)) : \star : \star) \in \mathcal{E}_{j'}^{\Rightarrow}[\![A_2 \sqsubseteq \star]\!]$

1881 let $e_1' = e_1, e_2' = e_2, j' = j$

1882 $\Rightarrow ((f_1 : A_1 \to A_2)\ e_1, (f_2'\ (e_2 : \star : \star)) : \star : \star) \in \mathcal{E}_j^{\Rightarrow}[\![A_2 \sqsubseteq \star]\!]$

1883 by Lemma E.2,

1884 $(f_2'\ (e_2 : \star : \star)) : \star : \star$ reduce to the same result as $(f_2'\ (e_2 : \star)) : \star$

1885 Thus the result holds.

1886 if the type of $f_2 \neq \star \to \star$

1887 We have

1888 $(f_1, f_2) \in \mathcal{V}_k^{\Rightarrow}[\![A_1' \to A_2' \sqsubseteq B_1' \to B_2']\!]\ (B_1' \to B_2' \neq \star \to \star)$

1889 we want to prove

1890 $(f_1 : A_1 \to A_2, f_2 : \star \to \star : \star) \in \mathcal{V}_k^{\Rightarrow}[\![A_1 \to A_2 \sqsubseteq \star]\!]$

1891 by the definition of related values at $A_1 \to A_2 \sqsubseteq \star$

1892 $\Rightarrow (f_1 : A_1 \to A_2, f_2 : \star \to \star) \in \mathcal{V}_k^{\Rightarrow}[\![A_1 \to A_2 \sqsubseteq \star \to \star]\!]$

1893 by induction hypothesis on B,

1894 $(f_1 : A_1 \to A_2, f_2 : \star \to \star) \in \mathcal{V}_k^{\Rightarrow}[\![A_1 \to A_2 \sqsubseteq \star \to \star]\!]$

1895 Thus the result holds.

1896

1897 *Case* $(f_1, f_2, \star, \star)$. We have

1898 $(f_1, f_2) \in \mathcal{V}_k^{\Rightarrow}[\![A_1' \to A_2' \sqsubseteq B_1' \to B_2']\!]\ (A_1' \to A_2' \neq \star \to \star, B_1' \to B_2' \neq \star \to \star)$

1899 we want to prove

1900 $(f_1 : \star, f_2 : \star) \in \mathcal{V}_k^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

1901 $f_1 : \star$ takes one step to be $f_1 : \star \to \star : \star$

1902 $\Rightarrow (f_1 : \star \to \star : \star, f_2 : \star \to \star : \star) \in \mathcal{V}_{k-1}^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

1903 by the definition of related values at $\star \sqsubseteq \star$

1904 $\Rightarrow (f_1 : \star \to \star, f_2 : \star \to \star : \star) \in \mathcal{V}_{k-1}^{\Rightarrow}[\![\star \to \star \sqsubseteq \star]\!]$

1905 by the definition of related values at $\star \to \star \sqsubseteq \star$

1906 $\Rightarrow (f_1 : \star \to \star, f_2 : \star \to \star) \in \mathcal{V}_{k-1}^{\Rightarrow}[\![\star \to \star \sqsubseteq \star \to \star]\!]$

1907 by induction hypothesis on A and B,

1908 $(f_1 : \star \to \star, f_2 : \star \to \star) \in \mathcal{V}_k^{\Rightarrow}[\![\star \to \star \sqsubseteq \star \to \star]\!]$

1909 by Lemma E.1, the result holds.

1910

1911

Case $(f_1, f_2, \star, \star)$. We have

$(f_1, f_2) \in \mathcal{V}_k^{\Rightarrow}[\![A_1' \to A_2' \sqsubseteq \star \to \star]\!] \ (A_1' \to A_2' \neq \star \to \star)$

we want to prove

$(f_1 : \star, f_2 : \star) \in \mathcal{V}_k^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

$f_1 : \star$ takes one step to be $f_1 : \star \to \star : \star$

$\Rightarrow (f_1 : \star \to \star : \star, f_2 : \star) \in \mathcal{V}_{k-1}^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

by the definition of related values at $\star \sqsubseteq \star$

$\Rightarrow (f_1 : \star \to \star, f_2 : \star) \in \mathcal{V}_{k-1}^{\Rightarrow}[\![\star \to \star \sqsubseteq \star]\!]$

$\Rightarrow (f_1 : \star \to \star, f_2) \in \mathcal{V}_{k-1}^{\Rightarrow}[\![\star \to \star \sqsubseteq \star \to \star]\!]$

$\Rightarrow j \leqslant k - 1, (e_1, e_2) \in \mathcal{E}_j^{\Leftarrow}[\![\star \sqsubseteq \star]\!].$

$((f_1 : \star \to \star) \ e_1, f_2 \ e_2) \in \mathcal{E}_j^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

$\Rightarrow ((f_1 : \star \to \star) \ e_1, (f_2' \ (e_2 : \star)) : \star) \in \mathcal{E}_j^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

by induction hypothesis on A and B,

$(f_1 : \star \to \star, f_2 : \star \to \star) \in \mathcal{V}_k^{\Rightarrow}[\![\star \to \star \sqsubseteq \star \to \star]\!]$

$\Rightarrow j' \leqslant k, (e_1', e_2') \in \mathcal{E}_{j'}^{\Leftarrow}[\![\star \sqsubseteq \star]\!].$

$((f_1 : \star \to \star) \ e_1', (f_2 : \star \to \star) \ e_2') \in \mathcal{E}_{j'}^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

let $e_1' = e_1, e_2' = e_2, j' = j$

$\Rightarrow ((f_1 : \star \to \star) \ e_1, (f_2 : \star \to \star) \ e_2) \in \mathcal{E}_j^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

$\Rightarrow ((f_1 : \star \to \star) \ e_1, (f_2 \ (e_2 : \star)) : \star) \in \mathcal{E}_j^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

$\Rightarrow ((f_1 : \star \to \star) \ e_1, (f_2' \ (e_2 : \star : \star)) : \star : \star) \in \mathcal{E}_j^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

by Lemma E.2

$(f_2' \ (e_2 : \star)) : \star$ reduces to the same result as $(f_2' \ (e_2 : \star : \star)) : \star : \star$

Thus the result holds.

Case $(f_1, f_2, \star, \star)$. We have

$(f_1, f_2) \in \mathcal{V}_k^{\Rightarrow}[\![\star \to \star \sqsubseteq \star \to \star]\!]$

we want to prove

$(f_1 : \star, f_2 : \star) \in \mathcal{V}_k^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

by the definition of values related at $\star \sqsubseteq \star$

$\Rightarrow (f_1, f_2 : \star) \in \mathcal{V}_k^{\Rightarrow}[\![\star \to \star \sqsubseteq \star]\!]$

by the definition of values related at $\star \to \star \sqsubseteq \star$

$\Rightarrow (f_1, f_2) \in \mathcal{V}_k^{\Rightarrow}[\![\star \to \star \sqsubseteq \star \to \star]\!]$

Thus the result holds.

Case $(\{l = v_1\}, \{l = v_2\}, \{l : A\}, \{l : B\})$. We have

$(\{l = v_1\}, \{l = v_2\}) \in \mathcal{V}_k^{\Rightarrow}[\![\{l : A'\} \sqsubseteq \{l : B'\}]\!]$

$\Rightarrow (v_1, v_2) \in \mathcal{V}_{k-1}^{\Rightarrow}[\![A' \sqsubseteq B']\!]$

we want to prove

$(\{l = v_1\} : \{l : A\}, \{l = v_2\} : \{l : B\}) \in \mathcal{E}_k^{\Rightarrow}[\![\{l : A\} \sqsubseteq \{l : B\}]\!]$

if $(v_1 : A)$ reduce to err_*, the result holds.

lets assume it reduces to v_1' in j steps.

$\Rightarrow (\{l = v_1'\}, \{l = v_2'\}) \in \mathcal{V}_{k-1-j}^{\Rightarrow}[\![\{l : A\} \sqsubseteq \{l : B\}]\!]$

by the definition of related values at $\{l : A\} \sqsubseteq \{l : B\}$

$\Rightarrow (v_1', v_2') \in \mathcal{V}_{k-2-j}^{\Rightarrow}[\![A \sqsubseteq B]\!]$

by induction hypothesis,

$(v_1 : A, v_2 : B) \in \mathcal{E}_{k-1}^{\Rightarrow}[\![A \sqsubseteq B]\!]$

$\Rightarrow (v_1', v_2') \in \mathcal{V}_{k-1-j}^{\Rightarrow}[\![A \sqsubseteq B]\!]$

by Lemma E.1, the result holds.

1961   *Case* $(\{l = v_1\}, \{l = v_1\}, \{l : A\}, \star)$. We have

1962 $(\{l = v_1\}, \{l = v_2\}) \in \mathcal{V}_k^{\Rightarrow}[\![\{l : A'\} \sqsubseteq \{l : B'\}]\!]$

1963   by the definition of related values at $\{l : A'\} \sqsubseteq \{l : B'\}$

1964 $\Rightarrow (v_1, v_2) \in \mathcal{V}_{k-1}^{\Rightarrow}[\![A' \sqsubseteq B']\!]$

1965   we want to prove

1966 $(\{l = v_1\} : \{l : A\}, \{l = v_2\} : \star) \in \mathcal{E}_k^{\Rightarrow}[\![\{l : A\} \sqsubseteq \star]\!]$

1967   by the definition of reduction

1968 $\Rightarrow (\{l = (v_1 : A)\}, \{l = (v_2 : \star)\} : \star) \in \mathcal{E}_k^{\Rightarrow}[\![\{l : A\} \sqsubseteq \star]\!]$

1969   if $(v_1 : A)$ reduce to err_*, the result holds

1970   lets assume it reduces to v_1' in j steps.

1971 $\Rightarrow (\{l = v_1'\}, \{l = v_2'\} : \star) \in \mathcal{V}_{k-1-j}^{\Rightarrow}[\![\{l : A\} \sqsubseteq \star]\!]$

1972 $\Rightarrow (\{l = v_1'\}, \{l = v_2'\}) \in \mathcal{V}_{k-1-j}^{\Rightarrow}[\![\{l : A\} \sqsubseteq \{l : \star\}]\!]$

1973 $\Rightarrow (v_1', v_2') \in \mathcal{V}_{k-2-j}^{\Rightarrow}[\![A \sqsubseteq \star]\!]$

1974   by induction hypothesis on step number,

1975 $(v_1 : A, v_2 : \star) \in \mathcal{E}_{k-1}^{\Rightarrow}[\![A \sqsubseteq \star]\!]$

1976 $\Rightarrow (r_1, r_2) \in \mathcal{R}_{k-1-j}^{\Rightarrow}[\![A \sqsubseteq \star]\!]$

1977   Thus the result holds.

1978

1979   *Case* $(\{l = v_1\}, \{l = v_1\}, \star, \star)$. We have

1980 $(\{l = g_1 : \star\}, \{l = g_2 : \star\}) \in \mathcal{V}_k^{\Rightarrow}[\![\{l : \star\} \sqsubseteq \{l : \star\}]\!]$

1981   we want to prove

1982 $(\{l = g_1 : \star\} : \star, \{l = g_2 : \star\} : \star) \in \mathcal{V}_k^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

1983 $\Rightarrow (\{l = g_1 : \star\} : \star, \{l = g_2 : \star\} : \star) \in \mathcal{V}_k^{\Rightarrow}[\![\{l : \star\} \sqsubseteq \star]\!]$

1984 $\Rightarrow (\{l = g_1 : \star\}, \{l = g_2 : \star\}) \in \mathcal{V}_k^{\Rightarrow}[\![\{l : \star\} \sqsubseteq \{l : \star\}]\!]$

1985   Thus the result holds.

1986

1987   *Case* $(\{l = v_1\}, \{l = v_1\}, \star, \star)$. We have

1988 $(\{l = v_1\}, \{l = v_2\}) \in \mathcal{V}_k^{\Rightarrow}[\![\{l : A'\} \sqsubseteq \{l : B'\}]\!]$

1989 $\Rightarrow (v_1, v_2) \in \mathcal{V}_{k-1}^{\Rightarrow}[\![A' \sqsubseteq B']\!]$

1990   we want to prove

1991 $(\{l = v_1\} : \star, \{l = v_2\} : \star) \in \mathcal{E}_k^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

1992   by the definition of reduction

1993 $\Rightarrow (\{l = (v_1 : \star)\} : \star, \{l = (v_2 : \star)\} : \star) \in \mathcal{E}_k^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

1994   if $(v_1 : \star)$ reduce to v_1' in j steps

1995 $\Rightarrow (\{l = v_1'\} : \star, \{l = v_2'\} : \star) \in \mathcal{V}_{k-1-j}^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

1996 $\Rightarrow (\{l = v_1'\}, \{l = v_2'\}) \in \mathcal{V}_{k-1-j}^{\Rightarrow}[\![\{l : \star\} \sqsubseteq \{l : \star\}]\!]$

1997 $\Rightarrow (v_1', v_2') \in \mathcal{V}_{k-2-j}^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

1998   by induction hypothesis on step number,

1999 $(v_1 : \star, v_2 : \star) \in \mathcal{E}_{k-1}^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

2000 $\Rightarrow (v_1', v_2') \in \mathcal{V}_{k-1-j}^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

2001   Thus the result holds.

2002

2003   *Case* $(v_{11}, , v_{12}, , v_{21}, , v_{22}, A^\circ, B^\circ)$. we have

2004 $(v_{11}, , v_{12}, v_{21}, , v_{22}) \in \mathcal{V}_k^{\Rightarrow}[\![A_1' \& A_2' \sqsubseteq B_1' \& B_2']\!]$

2005   by the definition of related values at $A_1 \& A_2' \sqsubseteq B_1' \& B_2'$

2006 $\Rightarrow$

2007 $(v_{11}, v_{21}) \in \mathcal{V}_{k-1}^{\Rightarrow}[\![A_1' \sqsubseteq B_1']\!]$

2008 $(v_{12}, v_{22}) \in \mathcal{V}_{k-1}^{\Rightarrow}[\![A_2' \sqsubseteq B_2']\!]$

2009

we want to prove

$((v_{11},,v_{12}) : A^\circ, (v_{21},,v_{22}) : B^\circ) \in \mathcal{E}_k^{\Rightarrow} \llbracket A^\circ \sqsubseteq B^\circ \rrbracket$

if $v_{11} : A^\circ \longmapsto^{j_1} r_{11}$, $v_{12} : A^\circ \longmapsto^{j_2} r_{12}$

and $v_{21} : B^\circ \longmapsto^* r_{21}$, $v_{22} : B^\circ \longmapsto^* r_{22}$

j is the max number between j_1 and j_2

$\Rightarrow (r_{11} \vee r_{12}, r_{21} \vee r_{22}) \in \mathcal{R}_{k-j-1}^{\Rightarrow} \llbracket A^\circ \sqsubseteq B^\circ \rrbracket$

by induction hypothesis on step number,

$(v_{11} : A^\circ, v_{21} : B^\circ) \in \mathcal{E}_{k-1}^{\Rightarrow} \llbracket A^\circ \sqsubseteq B^\circ \rrbracket$

$\Rightarrow (r_{11}, r_{21}) \in \mathcal{R}_{k-1-j_1}^{\Rightarrow} \llbracket A^\circ \sqsubseteq B^\circ \rrbracket$

$(v_{12} : A^\circ, v_{22} : B^\circ) \in \mathcal{E}_{k-1}^{\Rightarrow} \llbracket A^\circ \sqsubseteq B^\circ \rrbracket$

$\Rightarrow (r_{21}, r_{22}) \in \mathcal{R}_{k-1-j_2}^{\Rightarrow} \llbracket A^\circ \sqsubseteq B^\circ \rrbracket$

by Lemma E.3 and Lemma E.1

$\Rightarrow (r_{11} \vee r_{12}, r_{21} \vee r_{22}) \in \mathcal{R}_{k-j-1}^{\Rightarrow} \llbracket A^\circ \sqsubseteq B^\circ \rrbracket$

Thus the result holds.

Case $(v_{11},,v_{12}, v_{21},,v_{22}, A^\circ, \star)$. we have

$(v_{11},,v_{12}, v_{21},,v_{22}) \in \mathcal{V}_k^{\Rightarrow} \llbracket A_1' \& A_2' \sqsubseteq B_1' \& B_2' \rrbracket$

by the definition of related values at $A_1 \& A_2' \sqsubseteq B_1' \& B_2'$

$\Rightarrow$

$(v_{11}, v_{21}) \in \mathcal{V}_{k-1}^{\Rightarrow} \llbracket A_1' \sqsubseteq B_1' \rrbracket$

$(v_{12}, v_{22}) \in \mathcal{V}_{k-1}^{\Rightarrow} \llbracket A_2' \sqsubseteq B_2' \rrbracket$

we want to prove

$((v_{11},,v_{12}) : A^\circ, (v_{21},,v_{22}) : \star) \in \mathcal{E}_k^{\Rightarrow} \llbracket A^\circ \sqsubseteq \star \rrbracket$

if $v_{11} : A^\circ \longmapsto^j r_{11}$, $v_{12} : A^\circ \longmapsto^j r_{12}$

and $v_{21} : \star \longmapsto^* v_{21}'$, $v_{22} : \star \longmapsto^* v_{22}'$

$\Rightarrow (r_{11} \vee r_{12}, (v_{21}',,v_{22}') : \star) \in \mathcal{R}_{k-j-1}^{\Rightarrow} \llbracket A^\circ \sqsubseteq \star \rrbracket$

if $r_{11} \vee r_{12} = \text{err}_*$, the result holds

lets assume $r_{11} \vee r_{12} = s_1$, then we need to prove

$\Rightarrow s_2 \in (v_{21}',,v_{22}') : \star$,

$(s_1, s_2) \in \mathcal{V}_{k-j-1}^{\Rightarrow} \llbracket A^\circ \sqsubseteq B \rrbracket$

by induction hypothesis on step number,

$(v_{11} : A^\circ, v_{21} : \star) \in \mathcal{E}_{k-1}^{\Rightarrow} \llbracket A^\circ \sqsubseteq \star \rrbracket$

$\Rightarrow (r_{11}, v_{21}') \in \mathcal{R}_{k-1-j}^{\Rightarrow} \llbracket A^\circ \sqsubseteq \star \rrbracket$

$(v_{12} : A^\circ, v_{22} : \star) \in \mathcal{E}_{k-1}^{\Rightarrow} \llbracket A^\circ \sqsubseteq \star \rrbracket$

$\Rightarrow (r_{12}, v_{22}') \in \mathcal{R}_{k-1-j}^{\Rightarrow} \llbracket A^\circ \sqsubseteq \star \rrbracket$

if $s_2 \in v_{21}'$

then $s_2 \in (v_{21}',,v_{22}') : \star$

if $s_2 \in v_{22}'$

then $s_2 \in (v_{21}',,v_{22}') : \star$

since $r_{11} \vee r_{12} = s_1$, we want to prove

$\Rightarrow (s_1, (v_{21}',,v_{22}') : \star) \in \mathcal{V}_{k-j-1}^{\Rightarrow} \llbracket A^\circ \sqsubseteq \star \rrbracket$

Thus the result holds.

Case $(v_{11},,v_{12}, v_{21},,v_{22}, \star, \star)$. we have

$(v_{11},,v_{12}, v_{21},,v_{22}) \in \mathcal{V}_k^{\Rightarrow} \llbracket A_1' \& A_2' \sqsubseteq B_1' \& B_2' \rrbracket$

by the definition of related values at $A_1 \& A_2' \sqsubseteq B_1' \& B_2'$

$\Rightarrow$

2059 $(v_{11}, v_{21}) \in \mathcal{V}^{\Rightarrow}_{k-1}[\![A'_1 \sqsubseteq B'_1]\!]$
2060 $(v_{12}, v_{22}) \in \mathcal{V}^{\Rightarrow}_{k-1}[\![A'_2 \sqsubseteq B'_2]\!]$
2061 we want to prove
2062 $((v_{11},,v_{12}):\star, (v_{21},,v_{22}):\star) \in \mathcal{E}^{\Rightarrow}_k[\![A^\circ \sqsubseteq \star]\!]$
2063 if $v_{11} : \star \longmapsto^{j_1} v'_{11}, v_{12} : \star \longmapsto^{j_2} v'_{12}$
2064 and $v_{21} : \star \longmapsto^* v'_{21}, v_{22} : \star \longmapsto^* v'_{22}$
2065 $\Rightarrow ((v'_{11},,v'_{12}):\star, (v'_{21},,v'_{22}):\star) \in \mathcal{R}^{\Rightarrow}_{k-j_1-j_2-1}[\![\star \sqsubseteq \star]\!]$
2066 by induction hypothesis on step number,
2067 $(v_{11}:\star, v_{21}:\star) \in \mathcal{E}^{\Rightarrow}_{k-1}[\![\star \sqsubseteq \star]\!]$
2068 $\Rightarrow (v'_{11}, v'_{21}) \in \mathcal{V}^{\Rightarrow}_{k-1-j_1}[\![\star \sqsubseteq \star]\!]$
2069 $(v_{12}:\star, v_{22}:\star) \in \mathcal{E}^{\Rightarrow}_{k-1}[\![\star \sqsubseteq \star]\!]$
2070 $\Rightarrow (v'_{21}, v'_{22}) \in \mathcal{V}^{\Rightarrow}_{k-1-j_2}[\![\star \sqsubseteq \star]\!]$
2071 by the definition of related values at $\star \& \star \sqsubseteq \star \& \star$
2072 $\Rightarrow (v'_{11},,v'_{21},,v'_{21},,v'_{22}) \in \mathcal{V}^{\Rightarrow}_{k-1-j_1-j_2}[\![\star \& \star \sqsubseteq \star \& \star]\!]$
2073 by induction hypothesis on step number,
2074 $((v'_{11},,v'_{21}):\star, (v'_{21},,v'_{22}):\star) \in \mathcal{V}^{\Rightarrow}_{k-1-j_1-j_2}[\![\star \sqsubseteq \star]\!]$
2075 Thus the result holds.
2076
2077 *Case* $(s, g : \star, A^\circ, B^\circ)$. we have
2078 $(s_1, g : \star) \in \mathcal{V}^{\Rightarrow}_k[\![A' \sqsubseteq \star]\!]$
2079 by the definition of related values at $A' \sqsubseteq \star$
2080 $\Rightarrow \exists s_2 \in g : \star,$
2081 $(s_1, s_2) \in \mathcal{V}^{\Rightarrow}_k[\![A' \sqsubseteq B']\!]$
2082 we want to prove
2083 $(s_1 : A^\circ, (g : \star) : B^\circ) \in \mathcal{V}^{\Rightarrow}_k[\![A^\circ \sqsubseteq B^\circ]\!]$
2084 lets assume $(g : \star) : B^\circ \longmapsto s'_2$
2085 then it means that $s_2 : B^\circ \longmapsto s'_2$
2086 then we need to prove
2087 $(s_1 : A^\circ, s_2 : B^\circ) \in \mathcal{E}^{\Rightarrow}_k[\![A^\circ \sqsubseteq B^\circ]\!]$
2088 by induction hypothesis on B',
2089 $(s_1 : A^\circ, s_2 : B^\circ) \in \mathcal{E}^{\Rightarrow}_k[\![A^\circ \sqsubseteq B^\circ]\!]$
2090 thus the result holds.
2091
2092 *Case* $(s, g : \star, A^\circ, \star)$. we have
2093 $(s_1, g : \star) \in \mathcal{V}^{\Rightarrow}_k[\![A' \sqsubseteq \star]\!]$
2094 by the definition of related values at $A' \sqsubseteq \star$
2095 $\Rightarrow \exists s_2 \in g : \star,$
2096 $(s_1, s_2) \in \mathcal{V}^{\Rightarrow}_k[\![A' \sqsubseteq B']\!]$
2097 we want to prove
2098 $(s_1 : A^\circ, (g : \star) : \star) \in \mathcal{E}^{\Rightarrow}_k[\![A^\circ \sqsubseteq \star]\!]$
2099 if $s_1 : A^\circ$ reduce to err_*, then the result holds
2100 lets assume it reduces to s'_1
2101 $\Rightarrow (s'_1, g : \star) \in \mathcal{V}^{\Rightarrow}_{k-1}[\![A^\circ \sqsubseteq \star]\!]$
2102 by the definition of related values at $A^\circ \sqsubseteq \star$
2103 $\Rightarrow \exists s'_2 \in g : \star,$
2104 $(s'_1, s'_2) \in \mathcal{V}^{\Rightarrow}_{k-1}[\![A^\circ \sqsubseteq B^\circ]\!]$
2105 by induction hypothesis on B',
2106 $(s_1 : A^\circ, s_2 : \star) \in \mathcal{E}^{\Rightarrow}_k[\![A^\circ \sqsubseteq \star]\!]$
2107

2108 $\Rightarrow (s_1', s_2 : \star) \in \mathcal{V}^{\Rightarrow}_{k-1}[\![A^\circ \sqsubseteq \star]\!]$
2109 by the definition of related values at $A^\circ \sqsubseteq \star$
2110 $\Rightarrow (s_1', s_2) \in \mathcal{V}^{\Rightarrow}_{k-1}[\![A^\circ \sqsubseteq B']\!]$
2111 let $s_2' = s_2$, thus the result holds.
2112
2113 *Case* $(s, g : \star, \star, \star)$. we have
2114 $(s_1, g : \star) \in \mathcal{V}^{\Rightarrow}_{k}[\![A' \sqsubseteq \star]\!]$
2115 by the definition of related values at $A' \sqsubseteq \star$
2116 $\Rightarrow \exists s_2 \in g : \star,$
2117 $(s_1, s_2) \in \mathcal{V}^{\Rightarrow}_{k}[\![A' \sqsubseteq B']\!]$
2118 we want to prove
2119 $(s_1 : \star, (g : \star) : \star) \in \mathcal{E}^{\Rightarrow}_{k}[\![\star \sqsubseteq \star]\!]$
2120 $\Rightarrow (s_1 : \star, g : \star) \in \mathcal{E}^{\Rightarrow}_{k}[\![\star \sqsubseteq \star]\!]$
2121 let us assume $s_1 : \star$ take j steps to $s_1' : \star$
2122 $\Rightarrow (s_1' : \star, g : \star) \in \mathcal{V}^{\Rightarrow}_{k-j-1}[\![\star \sqsubseteq \star]\!]$
2123 by the definition of related values at $\star \sqsubseteq \star$
2124 $\Rightarrow (s_1', g : \star) \in \mathcal{V}^{\Rightarrow}_{k-j-1}[\![A' \sqsubseteq \star]\!]$
2125 by the definition of related values at $A' \sqsubseteq \star$
2126 $\Rightarrow \exists s_2' \in g : \star,$
2127 $(s_1', s_2') \in \mathcal{V}^{\Rightarrow}_{k-j-1}[\![A' \sqsubseteq B']\!]$
2128 by induction hypothesis on A' and B',
2129 $(s_1 : \star, s_2 : \star) \in \mathcal{E}^{\Rightarrow}_{k}[\![\star \sqsubseteq \star]\!]$
2130 $\Rightarrow (s_1' : \star, s_2 : \star) \in \mathcal{V}^{\Rightarrow}_{k-j-1}[\![\star \sqsubseteq \star]\!]$
2131 by the definition of related values at $\star \sqsubseteq \star$
2132 $\Rightarrow (s_1', s_2 : \star) \in \mathcal{V}^{\Rightarrow}_{k-j-1}[\![A' \sqsubseteq \star]\!]$
2133 by the definition of related values at $A' \sqsubseteq \star$
2134 $\Rightarrow (s_1', s_2) \in \mathcal{V}^{\Rightarrow}_{k-j-1}[\![A' \sqsubseteq B']\!]$
2135 let $s_2' = s_2$
2136 thus the result holds.
2137
2138 *Case* $((v_1,,v_2), g : \star, A^\circ, B^\circ)$. we have
2139 $(v_1,,v_2, g : \star) \in \mathcal{V}^{\Rightarrow}_{k}[\![A'\&B' \sqsubseteq \star]\!]$
2140 by the definition of related values at $A'\&B' \sqsubseteq \star$
2141 $\Rightarrow$
2142 $(v_1, g : \star) \in \mathcal{V}^{\Rightarrow}_{k-1}[\![A' \sqsubseteq \star]\!]$
2143 $(v_2, g : \star) \in \mathcal{V}^{\Rightarrow}_{k-1}[\![B' \sqsubseteq \star]\!]$
2144 we want to prove
2145 $((v_1,,v_2) : A^\circ, (g : \star) : B^\circ) \in \mathcal{E}^{\Rightarrow}_{k}[\![A^\circ \sqsubseteq B^\circ]\!]$
2146 if $v_1 : A^\circ \longmapsto^{j_1} r_1, v_2 : A^\circ \longmapsto^{j_2} r_2$ and $g : B^\circ \longmapsto^* r_3$
2147 $\Rightarrow (r_1 \vee r_2, r_3) \in \mathcal{R}^{\Rightarrow}_{k-1-j}[\![A^\circ \sqsubseteq B^\circ]\!]$ j is the max number between j_1 and j_2
2148 by induction hypothesis on step number,
2149 $(v_1 : A^\circ, (g : \star) : B^\circ) \in \mathcal{E}^{\Rightarrow}_{k-1}[\![A^\circ \sqsubseteq B^\circ]\!]$
2150 $(v_2 : A^\circ, (g : \star) : B^\circ) \in \mathcal{E}^{\Rightarrow}_{k-1}[\![A^\circ \sqsubseteq B^\circ]\!]$
2151 $\Rightarrow$
2152 $(r_1, r_3) \in \mathcal{R}^{\Rightarrow}_{k-1-j_1}[\![A^\circ \sqsubseteq B^\circ]\!]$
2153 $(r_2, r_3) \in \mathcal{R}^{\Rightarrow}_{k-1-j_2}[\![A^\circ \sqsubseteq B^\circ]\!]$
2154 by Lemma E.3, the result holds.
2155
2156

Case $((v_1,,v_2), g : \star, A^\circ, \star)$. we have

$(v_1,,v_2, g : \star) \in \mathcal{V}_k^\Rightarrow[\![A'\&B' \sqsubseteq \star]\!]$

by the definition of related values at $A'\&B' \sqsubseteq \star$

$\Rightarrow$

$(v_1, g : \star) \in \mathcal{V}_{k-1}^\Rightarrow[\![A' \sqsubseteq \star]\!]$

$(v_2, g : \star) \in \mathcal{V}_{k-1}^\Rightarrow[\![B' \sqsubseteq \star]\!]$

we want to prove

$((v_1,,v_2) : A^\circ, (g : \star) : \star) \in \mathcal{E}_k^\Rightarrow[\![A^\circ \sqsubseteq \star]\!]$

if $v_1 \longmapsto^{j_1} r_1, v_2 \longmapsto^{j_2} r_2$, j is the max number between j_1 and j_2

$(r_1 \vee r_2, (g : \star)) \in \mathcal{R}_{k-1-j}^\Rightarrow[\![A^\circ \sqsubseteq \star]\!]$

by induction hypothesis on step number,

$(v_1 : A^\circ, g : \star : \star) \in \mathcal{E}_{k-1}^\Rightarrow[\![A^\circ \sqsubseteq \star]\!]$

$(v_2 : A^\circ, g : \star : \star) \in \mathcal{E}_{k-1}^\Rightarrow[\![A^\circ \sqsubseteq \star]\!]$

$\Rightarrow$

$(r_1, g : \star) \in \mathcal{R}_{k-1-j_1}^\Rightarrow[\![A^\circ \sqsubseteq \star]\!]$

$(r_2, g : \star) \in \mathcal{R}_{k-1-j_2}^\Rightarrow[\![A^\circ \sqsubseteq \star]\!]$

let us assume $r_1 \vee r_2 = s_1$

by Lemma E.1, then we have

$(s_1, g : \star) \in \mathcal{V}_{k-1-j}^\Rightarrow[\![A^\circ \sqsubseteq \star]\!]$

by the definition of related values at $A^\circ \sqsubseteq \star$

$\Rightarrow \exists s_2 \in g : \star,$

$(s_1, s_2) \in \mathcal{V}_{k-1-j}^\Rightarrow[\![A^\circ \sqsubseteq \star]\!]$

and we need to prove

$\exists s_2 \in g : \star,$

$(s_1, s_2) \in \mathcal{V}_{k-1-j}^\Rightarrow[\![A^\circ \sqsubseteq \star]\!]$

Thus the result holds.

Case $((v_1,,v_2), g : \star, \star, \star)$. we have

$(v_1,,v_2, g : \star) \in \mathcal{V}_k^\Rightarrow[\![A'\&B' \sqsubseteq \star]\!]$

by the definition of related values at $A'\&B' \sqsubseteq \star$

$\Rightarrow$

$(v_1, g : \star) \in \mathcal{V}_{k-1}^\Rightarrow[\![A' \sqsubseteq \star]\!]$

$(v_2, g : \star) \in \mathcal{V}_{k-1}^\Rightarrow[\![B' \sqsubseteq \star]\!]$

we want to prove

$((v_1,,v_2) : \star, (g : \star) : \star) \in \mathcal{E}_k^\Rightarrow[\![\star \sqsubseteq \star]\!]$

if $v_1 : \star \longmapsto^{j_1} v_1', v_2 : \star \longmapsto^{j_2} v_2'$

$\Rightarrow ((v_1',,v_2') : \star, (g : \star)) \in \mathcal{V}_{k-1-j_1-j_2}^\Rightarrow[\![\star \sqsubseteq \star]\!]$

by the definition of related values at $\star \sqsubseteq \star$

$\Rightarrow ((v_1',,v_2'), (g : \star)) \in \mathcal{V}_{k-1-j_1-j_2}^\Rightarrow[\![\star\&\star \sqsubseteq \star]\!]$

by the definition of related values at $\star\&\star \sqsubseteq \star$

$\Rightarrow$

$(v_1', (g : \star)) \in \mathcal{V}_{k-2-j_1-j_2}^\Rightarrow[\![\star \sqsubseteq \star]\!]$

$(v_2', (g : \star)) \in \mathcal{V}_{k-2-j_1-j_2}^\Rightarrow[\![\star \sqsubseteq \star]\!]$

by induction hypothesis on step numbers,

$(v_1 : \star, g : \star : \star) \in \mathcal{E}_{k-1}^\Rightarrow[\![\star \sqsubseteq \star]\!]$

$(v_2 : \star, g : \star : \star) \in \mathcal{E}_{k-1}^\Rightarrow[\![\star \sqsubseteq \star]\!]$

$\Rightarrow$

2206 $(v_1', (g : \star)) \in \mathcal{V}_{k-1-j_1}^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

2207 $(v_2', (g : \star)) \in \mathcal{V}_{k-1-j_2}^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

2208   by the definition of related values at $\star \& \star \sqsubseteq \star$ and Lemma E.1, the result holds.

2209

2210   *Case* $(g_1 : \star, g_2 : \star, A^\circ, B^\circ)$. we have

2211 $(g_1 : \star, g_2 : \star) \in \mathcal{V}_k^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

2212   by the definition of related values at $\star \sqsubseteq \star$

2213 $(g_1, g_2 : \star) \in \mathcal{V}_k^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

2214   we want to prove

2215 $(g_1 : \star : A^\circ, g_2 : \star : B^\circ) \in \mathcal{E}_k^{\Rightarrow}[\![A^\circ \sqsubseteq B^\circ]\!]$

2216   let us assume it reduce to s_1 in j steps

2217 $\Rightarrow (s_1, s_2) \in \mathcal{V}_{k-j-1}^{\Rightarrow}[\![A^\circ \sqsubseteq B^\circ]\!]$

2218   by induction hypothesis on A',

2219 $(g_1 : A^\circ, g_2 : \star : B^\circ) \in \mathcal{E}_k^{\Rightarrow}[\![A^\circ \sqsubseteq B^\circ]\!]$

2220 $\Rightarrow (s_1, s_2) \in \mathcal{V}_{k-j}^{\Rightarrow}[\![A^\circ \sqsubseteq B^\circ]\!]$

2221   by Lemma E.1, the result holds.

2222

2223   *Case* $(g_1 : \star, g_2 : \star, A^\circ, \star)$. we have

2224 $(g_1 : \star, g_2 : \star) \in \mathcal{V}_k^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

2225   by definition of value related at $\star \sqsubseteq \star$

2226 $\Rightarrow (g_1, g_2 : \star) \in \mathcal{V}_k^{\Rightarrow}[\![A' \sqsubseteq \star]\!]$

2227   we want to prove

2228 $(g_1 : \star : A^\circ, g_2 : \star : \star) \in \mathcal{E}_k^{\Rightarrow}[\![A^\circ \sqsubseteq \star]\!]$

2229   if $g_1 : \star : A^\circ$ reduce to err_*, the result holds

2230   let us assume it reduces to s_1 in j steps

2231 $\Rightarrow (s_1, g_2 : \star) \in \mathcal{V}_{k-j-1}^{\Rightarrow}[\![A^\circ \sqsubseteq \star]\!]$

2232   by induction hypothesis on A',

2233 $(g_1 : A^\circ, g_2 : \star : \star) \in \mathcal{E}_k^{\Rightarrow}[\![A^\circ \sqsubseteq \star]\!]$

2234 $\Rightarrow (s_1, g_2 : \star) \in \mathcal{V}_{k-j}^{\Rightarrow}[\![A^\circ \sqsubseteq \star]\!]$

2235   by Lemma E.1, the result holds.

2236   *Case* $(g_1 : \star, g_2 : \star, \star, \star)$. we have

2237 $(g_1 : \star, g_2 : \star) \in \mathcal{V}_k^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

2238   we want to prove

2239 $(g_1 : \star : \star, g_2 : \star : \star) \in \mathcal{E}_k^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

2240 $\Rightarrow (g_1 : \star, g_2 : \star) \in \mathcal{V}_{k-1}^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

2241   by Lemma E.1, the result holds.

2242

2243   *Case* $(v_1, v_2, A_1 \& A_2, B_1 \& B_2)$. We have

2244 $(v_1, v_2) \in \mathcal{V}_k^{\Rightarrow}[\![A' \sqsubseteq B']\!]$

2245   we want to prove

2246 $(v_1 : A_1 \& A_2, v_2 : B_1 \& B_2) \in \mathcal{E}_k^{\Rightarrow}[\![A_1 \& A_2 \sqsubseteq B_1 \& B_2]\!]$

2247   if $v_1 : A_1 \& A_2$ reduces to err_*, then result holds.

2248   let us assume $v_1 : A_1$ reduces to v_{11} in j_1 steps

2249 $v_1 : A_2$ reduces to v_{12} in j_2 steps

2250 $v_2 : B_1$ reduces to v_{21}

2251 $v_2 : B_2$ reduces to v_{22}

2252 $\Rightarrow ((v_{11}, v_{12}), (v_{21}, v_{22})) \in \mathcal{V}_{k-1-j_1-j_2}^{\Rightarrow}[\![A_1 \& A_2 \sqsubseteq B_1 \& B_2]\!]$

2253   by definition of related values at $A_1 \& A_2 \sqsubseteq B_1 \& B_2$

2254

2255 $\Rightarrow (v_{11}, v_{21}) \in \mathcal{V}_{k-2-j_1-j_2}^{\Rightarrow}[\![A_1 \sqsubseteq B_1]\!]$

2256 $\Rightarrow (v_{12}, v_{22}) \in \mathcal{V}_{k-2-j_1-j_2}^{\Rightarrow}[\![A_2 \sqsubseteq B_2]\!]$

2257 by Lemma E.1,

2258 $(v_1, v_2) \in \mathcal{V}_{k-1}^{\Rightarrow}[\![A' \sqsubseteq B']\!]$

2259 by induction hypothesis on step number,

2260 $(v_1 : A_1, v_2 : B_1) \in \mathcal{E}_{k-1}^{\Rightarrow}[\![A_1 \sqsubseteq B_1]\!]$

2261 $\Rightarrow (v_{11}, v_{21}) \in \mathcal{V}_{k-1-j_1}^{\Rightarrow}[\![A_1 \sqsubseteq B_1]\!]$

2262 $(v_1 : A_2, v_2 : B_2) \in \mathcal{E}_{k-1}^{\Rightarrow}[\![A_2 \sqsubseteq B_2]\!]$

2263 $\Rightarrow (v_{12}, v_{22}) \in \mathcal{V}_{k-1-j_2}^{\Rightarrow}[\![A_2 \sqsubseteq B_2]\!]$

2264 by definition of related values at $A_1\&A_2 \sqsubseteq B_1\&B_2$ and Lemma E.1, the result holds.

2265

2266 *Case* $(v_1, v_2, A\&B, \star)$. We have

2267 $(v_1, v_2) \in \mathcal{V}_k^{\Rightarrow}[\![A' \sqsubseteq B']\!]$

2268 we want to prove

2269 $(v_1 : A_1\&A_2, v_2 : \star) \in \mathcal{E}_k^{\Rightarrow}[\![A_1\&A_2 \sqsubseteq \star]\!]$

2270 if $v_1 : A_1\&A_2$ reduces to err_*, then result holds.

2271 let us assume $v_1 : A_1$ reduces to v_{11} in j_1 steps

2272 $v_1 : A_2$ reduces to v_{12} in j_2 steps

2273 $v_2 : \star$ reduces to $g : \star$

2274 $\Rightarrow ((v_{11}, v_{12}), (g : \star)) \in \mathcal{V}_{k-1-j_1-j_2}^{\Rightarrow}[\![A_1\&A_2 \sqsubseteq \star]\!]$

2275 by the definition of related values at $A_1\&A_2 \sqsubseteq \star$

2276 $\Rightarrow (v_{11}, g : \star) \in \mathcal{V}_{k-2-j_1-j_2}^{\Rightarrow}[\![A_1 \sqsubseteq \star]\!]$

2277 $\Rightarrow (v_{12}, g : \star) \in \mathcal{V}_{k-2-j_1-j_2}^{\Rightarrow}[\![A_2 \sqsubseteq \star]\!]$

2278 by Lemma E.1,

2279 $(v_1, v_2) \in \mathcal{V}_{k-1}^{\Rightarrow}[\![A' \sqsubseteq B']\!]$

2280 by induction hypothesis on step number,

2281 $(v_1 : A_1, v_2 : \star) \in \mathcal{E}_{k-1}^{\Rightarrow}[\![A_1 \sqsubseteq \star]\!]$

2282 $\Rightarrow (v_{11}, g : \star) \in \mathcal{V}_{k-1-j_1}^{\Rightarrow}[\![A_1 \sqsubseteq \star]\!]$

2283 $(v_1 : A_2, v_2 : \star) \in \mathcal{E}_{k-1}^{\Rightarrow}[\![A_2 \sqsubseteq \star]\!]$

2284 $\Rightarrow (v_{12}, g : \star) \in \mathcal{V}_{k-1-j_2}^{\Rightarrow}[\![A_2 \sqsubseteq \star]\!]$

2285 by definition of related values at $A_1\&A_2 \sqsubseteq \star$ and Lemma E.1, the result holds.

2286

2287 *Case* $(v_1, v_2, \top, \top)$. We have

2288 $(v_1, v_2) \in \mathcal{V}_k^{\Rightarrow}[\![A \sqsubseteq B]\!]$

2289 we want to prove

2290 $(v_1 : \top, v_2 : \top) \in \mathcal{E}_k^{\Rightarrow}[\![\top \sqsubseteq \top]\!]$

2291 $\Rightarrow (\text{Top}, \text{Top}) \in \mathcal{V}_{k-1}^{\Rightarrow}[\![\top \sqsubseteq \top]\!]$

2292 by the definition of related values at $\top \sqsubseteq \top$, the result holds.

2293

2294 *Case* $(v_1, v_2, \top, \star)$. We have

2295 $(v_1, v_2) \in \mathcal{V}_k^{\Rightarrow}[\![A \sqsubseteq B]\!]$

2296 we want to prove

2297 $(v_1 : \top, v_2 : \star) \in \mathcal{E}_k^{\Rightarrow}[\![\top \sqsubseteq \star]\!]$

2298 $\Rightarrow (\text{Top}, g : \star) \in \mathcal{V}_{k-1}^{\Rightarrow}[\![\top \sqsubseteq \star]\!]$

2299 by the definition of related values at $\top \sqsubseteq \star$ and $\top \sqsubseteq \top$, the result holds.

2300   □

2301

2302   LEMMA E.5. $\sigma[x/v]e = \sigma(e)[x/v]$

2303

PROOF. The proof follows by the induction on the size of σ. □

LEMMA E.6 (COMPATIBILITY(MERGE)). *if* $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \sqsubseteq e_3 \Rightarrow A_1 \sqsubseteq B_1$ *and* $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_2 \sqsubseteq e_4 \Rightarrow A_2 \sqsubseteq B_2$ *then* $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1,,e_2 \sqsubseteq e_3,,e_4 \Rightarrow A_1 \& A_2 \sqsubseteq B_1 \& B_2$.

PROOF. we want to prove
$\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1,,e_2 \sqsubseteq e_3,,e_4 \Rightarrow A_1 \& A_2 \sqsubseteq B_1 \& B_3$
$\Rightarrow \forall k \geqslant 0, (\sigma_1, \sigma_2) \in \mathcal{G}[\![\Gamma_1 \sqsubseteq \Gamma_2]\!].$
$(\sigma(e_1,,e_2), \sigma(e_3,,e_4)) \in \mathcal{E}_k^{\Rightarrow}[\![A_1 \& A_2 \sqsubseteq B_1 \& B_2]\!]$
$\Rightarrow (\sigma(e_1),,\sigma(e_2), \sigma(e_3),,\sigma(e_4)) \in \mathcal{E}_k^{\Rightarrow}[\![A_1 \& A_2 \sqsubseteq B_1 \& B_2]\!]$
$\Rightarrow (v_1,,v_2, v_3,,v_4) \in \mathcal{V}_{k-n}^{\Rightarrow}[\![A_1 \& A_2 \sqsubseteq B_1 \& B_2]\!]$
$\Rightarrow$
$(v_1, v_3) \in \mathcal{V}_{k-n-1}^{\Rightarrow}[\![A_1 \sqsubseteq B_1]\!]$
$(v_2, v_4) \in \mathcal{V}_{k-n-1}^{\Rightarrow}[\![A_2 \sqsubseteq B_2]\!]$
we have
$\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \sqsubseteq e_3 \Rightarrow A_1 \sqsubseteq B_1$
$\Rightarrow \forall k \geqslant 0, (\sigma_1, \sigma_2) \in \mathcal{G}[\![\Gamma_1 \sqsubseteq \Gamma_2]\!].$
$(\sigma_1(e_1), \sigma_2(e_3)) \in \mathcal{E}_k^{\Rightarrow}[\![A_1 \sqsubseteq B_1]\!]$
$\Rightarrow$
$(v_1, v_3) \in \mathcal{V}_{k-n}^{\Rightarrow}[\![A_1 \sqsubseteq B_1]\!]$
$\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_2 \sqsubseteq e_4 \Rightarrow A_2 \sqsubseteq B_2$
$\Rightarrow \forall k \geqslant 0, (\sigma_1, \sigma_2) \in \mathcal{G}[\![\Gamma_1 \sqsubseteq \Gamma_2]\!].$
$(\sigma_1(e_2), \sigma_2(e_4)) \in \mathcal{E}_k^{\Rightarrow}[\![A_2 \sqsubseteq B_2]\!]$
$\Rightarrow$
$(v_2, v_4) \in \mathcal{V}_{k-n}^{\Rightarrow}[\![A_2 \sqsubseteq B_2]\!]$
Thus the result holds. □

LEMMA E.7 (COMPATIBILITY(RCD)). *if* $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \sqsubseteq e_2 \Rightarrow A \sqsubseteq B$ *then* $\Gamma_1 \sqsubseteq \Gamma_2 \vdash \{l = e_1\} \sqsubseteq \{l = e_2\} \Rightarrow \{l : A\} \sqsubseteq \{l : B\}$.

PROOF. we want to prove
$\Gamma_1 \sqsubseteq \Gamma_2 \vdash \{l = e_1\} \sqsubseteq \{l = e_2\} \Rightarrow \{l : A\} \sqsubseteq \{l : B\}$
$\Rightarrow \forall k \geqslant 0, (\sigma_1, \sigma_2) \in \mathcal{G}[\![\Gamma_1 \sqsubseteq \Gamma_2]\!].$
$(\sigma_1(\{l = e_2\}), \sigma_2(\{l = e_2\})) \in \mathcal{E}_k^{\Rightarrow}[\![\{l : A\} \sqsubseteq \{l : B\}]\!]$
$\Rightarrow (\{l = \sigma_1(e_2)\}, \{l = \sigma_2(e_2)\}) \in \mathcal{E}_k^{\Rightarrow}[\![\{l : A\} \sqsubseteq \{l : B\}]\!]$
$\Rightarrow (\{l = v_1\}, \{l = v_2\}) \in \mathcal{V}_{k-n}^{\Rightarrow}[\![\{l : A\} \sqsubseteq \{l : B\}]\!]$
$\Rightarrow (v_1, v_2) \in \mathcal{V}_{k-n-1}^{\Rightarrow}[\![A \sqsubseteq B]\!]$
we have
$\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \sqsubseteq e_2 \Rightarrow A \sqsubseteq B$
$\Rightarrow \forall k \geqslant 0, (\sigma_1, \sigma_2) \in \mathcal{G}[\![\Gamma_1 \sqsubseteq \Gamma_2]\!].$
$(\sigma_1(e_1), \sigma_2(e_2)) \in \mathcal{E}_k^{\Rightarrow}[\![A \sqsubseteq B]\!]$
$\Rightarrow (v_1, v_2) \in \mathcal{V}_{k-n}^{\Rightarrow}[\![A \sqsubseteq B]\!]$
Thus the result holds. □

LEMMA E.8 (COMPATIBILITY(ANNO)). *if* $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \sqsubseteq e_2 \Leftarrow B_1 \sqsubseteq B_2$ *then* $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 : B_1 \sqsubseteq e_2 : B_2 \Rightarrow B_1 \sqsubseteq B_2$.

PROOF. we want to prove
$\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 : B_1 \sqsubseteq e_2 : B_2 \Rightarrow B_1 \sqsubseteq B_2$
$\Rightarrow \forall k \geqslant 0, (\sigma_1, \sigma_2) \in \mathcal{G}[\![\Gamma_1 \sqsubseteq \Gamma_2]\!].$

2353 $(\sigma_1(e_1 : B_1), \sigma_2(e_2 : B_2)) \in \mathcal{E}_k^{\Rightarrow}[\![B_1 \sqsubseteq B_2]\!]$

2354 $\Rightarrow (\sigma_1(e_1) : B_1, \sigma_2(e_2) : B_2) \in \mathcal{E}_k^{\Rightarrow}[\![B_1 \sqsubseteq B_2]\!]$

2355 we have

2356 $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \sqsubseteq e_2 \ \Leftarrow \ B_1 \sqsubseteq B_2$

2357 $\Rightarrow \forall k \geqslant 0, (\sigma_1, \sigma_2) \in \mathcal{G}[\![\Gamma_1 \sqsubseteq \Gamma_2]\!].$

2358 $(\sigma_1(e_1), \sigma_2(e_2)) \in \mathcal{E}_k^{\Leftarrow}[\![B_1 \sqsubseteq B_2]\!]$

2359 $\Rightarrow (r_1, r_2) \in \mathcal{R}_{k-n}^{\Leftarrow}[\![B_1 \sqsubseteq B_2]\!]$

2360 if $r_1 = \text{err}_* \vee r_2 = \text{err}_a$ then the result holds

2361 if $r_1 = w_1 \vee r_2 = w_2$

2362 $\Rightarrow (w_1, w_2) \in \mathcal{W}_{k-n}^{\Leftarrow}[\![B_1 \sqsubseteq B_2]\!]$

2363 $\Rightarrow (w_1 : B_2, w_2 : B_2) \in \mathcal{E}_{k-n}^{\Rightarrow}[\![B_1 \sqsubseteq B_2]\!]$

2364 then we need to prove

2365 $(\sigma_1(e_1) : B_1, \sigma_2(e_2) : B_2) \in \mathcal{E}_k^{\Rightarrow}[\![B_1 \sqsubseteq B_2]\!]$

2366 $\Rightarrow (w_1 : B_1, w_2 : B_2) \in \mathcal{E}_{k-n}^{\Rightarrow}[\![B_1 \sqsubseteq B_2]\!]$

2367 Thus the result holds.   □

2368

2369 LEMMA E.9 (COMPATIBILITY(APP)). *if* $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \sqsubseteq e_3 \ \Rightarrow \ A \sqsubseteq B, \Gamma_1 \sqsubseteq \Gamma_2 \vdash e_2 \sqsubseteq e_4 \ \Leftarrow \ A_1 \sqsubseteq$

2370 $B_1, A \rhd A_1 \to A_2$ *and* $B \rhd B_1 \to B_2$ *then* $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \ e_2 \sqsubseteq e_3 \ e_4 \ \Rightarrow \ A_2 \sqsubseteq B_2$.

2371 PROOF.

2372 we want to prove

2373 $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \ e_2 \sqsubseteq e_3 \ e_4 \ \Rightarrow \ A_2 \sqsubseteq B_2$

2374 $\Rightarrow \forall k \geqslant 0, (\sigma_1, \sigma_2) \in \mathcal{G}[\![\Gamma_1 \sqsubseteq \Gamma_2]\!].$

2375 $(\sigma_1(e_1 \ e_2), \sigma_2(e_3 \ e_4)) \in \mathcal{E}_k^{\Rightarrow}[\![A_2 \sqsubseteq \star]\!]$

2376 $\Rightarrow (\sigma_1(e_1) \ \sigma_2(e_2), \sigma_2(e_3) \ \sigma_2(e_4)) \in \mathcal{E}_k^{\Rightarrow}[\![A_2 \sqsubseteq \star]\!]$

2377

2378 if $A = A_1 \to A_2$ and $B = \star$

2379 we have

2380 $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \sqsubseteq e_3 \ \Rightarrow \ A_1 \to A_2 \sqsubseteq \star$

2381 $\Rightarrow \forall k \geqslant 0, (\sigma_1, \sigma_2) \in \mathcal{G}[\![\Gamma_1 \sqsubseteq \Gamma_2]\!].$

2382 $(\sigma_1(e_1), \sigma_2(e_3)) \in \mathcal{E}_k^{\Rightarrow}[\![A_1 \to A_2 \sqsubseteq \star]\!]$

2383 if $\sigma_1(e_1) \longmapsto^* \text{err}_* \vee \sigma_2(e_3) \longmapsto^* \text{err}_a$, then the result holds

2384 if $\sigma_1(e_1) \longmapsto^* f_1 \wedge \sigma_2(e_3) \longmapsto^* g_3 : \star$

2385 $\Rightarrow (f_1, g_3 : \star) \in \mathcal{V}_{k-n_1}^{\Rightarrow}[\![A_1 \to A_2 \sqsubseteq \star]\!]$

2386 $\Rightarrow f_2 \in g_3 : \star$

2387 $(f_1, f_2) \in \mathcal{V}_{k-n_1}^{\Rightarrow}[\![A_1 \to A_2 \sqsubseteq \star \to \star]\!]$

2388 $\Rightarrow \forall j' \leqslant k, (e_2', e_4') \in \mathcal{E}_{j'}^{\Leftarrow}[\![A_1 \sqsubseteq \star]\!],$

2389 $(f_1 \ e_2', f_2 \ e_4') \in \mathcal{E}_{j'}^{\Rightarrow}[\![A_2 \sqsubseteq \star]\!]$

2390 we also have

2391 $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_2 \sqsubseteq e_4 \ \Leftarrow \ A_1 \sqsubseteq \star$

2392 $\Rightarrow (\sigma_1(e_2), \sigma_2(e_4)) \in \mathcal{E}_k^{\Leftarrow}[\![A_1 \sqsubseteq \star]\!]$

2393 then we want to prove

2394 $\Rightarrow (f_1 \ \sigma_1(e_2), (f_2 : \star \to \star) \ \sigma_2(e_4)) \in \mathcal{E}_k^{\Rightarrow}[\![A_2 \sqsubseteq \star]\!]$

2395 $\Rightarrow (f_1 \ \sigma_1(e_2), (f_2 \ (\sigma_2(e_4) : \star)) : \star) \in \mathcal{E}_k^{\Rightarrow}[\![A_2 \sqsubseteq \star]\!]$

2396 let $e_2' = \sigma_1(e_2), e_4' = \sigma_2(e_4), j' = k$, we have

2397 $(f_1 \ \sigma_1(e_2), f_2 \ \sigma_2(e_4)) \in \mathcal{E}_k^{\Rightarrow}[\![A_2 \sqsubseteq \star]\!]$

2398 by Lemma E.2,

2399 $f_2 \ \sigma_2(e_4) \longmapsto^* r$ and $(f_2 \ (\sigma_2(e_4) : \star)) : \star \longmapsto^* r$

2400

2401

2402   thus the result holds.

2403

2404   if $A = A_1 \rightarrow A_2$ and $B = B_1 \rightarrow B_2$

2405   we have

2406 $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \sqsubseteq e_3 \Rightarrow A_1 \rightarrow A_2 \sqsubseteq B_1 \rightarrow B_2$

2407 $\Rightarrow \forall k \geqslant 0, (\sigma_1, \sigma_2) \in \mathcal{G}[\![\Gamma_1 \sqsubseteq \Gamma_2]\!].$

2408 $(\sigma_1(e_1), \sigma_2(e_3)) \in \mathcal{E}_k^{\Rightarrow}[\![A_1 \rightarrow A_2 \sqsubseteq B_1 \rightarrow B_2]\!]$

2409   if $\sigma_1(e_1) \longmapsto^* \mathrm{err}_* \vee \sigma_2(e_3) \longmapsto^* \mathrm{err}_a$, then the result holds

2410   if $\sigma_1(e_1) \longmapsto^* f_1 \wedge \sigma_2(e_3) \longmapsto^* f_3$

2411 $\Rightarrow (f_1, f_3) \in \mathcal{V}_{k-n_1}^{\Rightarrow}[\![A_1 \rightarrow A_2 \sqsubseteq B_1 \rightarrow B_2]\!]$

2412 $\Rightarrow \forall j' \leqslant k, (e_2', e_4') \in \mathcal{E}_{j'}^{\Leftarrow}[\![A_1 \sqsubseteq B_1]\!],$

2413 $(f_1\ e_2', f_3\ e_4') \in \mathcal{E}_{j'}^{\Rightarrow}[\![A_2 \sqsubseteq B_2]\!]$

2414   we also have

2415 $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_2 \sqsubseteq e_4 \Leftarrow A_1 \sqsubseteq B_1$

2416 $\Rightarrow (\sigma_1(e_2), \sigma_2(e_4)) \in \mathcal{E}_k^{\Leftarrow}[\![A_1 \sqsubseteq B_1]\!]$

2417   then we want to prove

2418 $\Rightarrow (f_1\ \sigma_1(e_2), f_3\ \sigma_2(e_4)) \in \mathcal{E}_k^{\Rightarrow}[\![A_2 \sqsubseteq B_2]\!]$

2419   let $e_2' = \sigma_1(e_2), e_4' = \sigma_2(e_4), j' = k$, we have

2420 $(f_1\ \sigma_1(e_2), f_3\ \sigma_2(e_4)) \in \mathcal{E}_k^{\Rightarrow}[\![A_2 \sqsubseteq B_2]\!]$

2421   thus the result holds.

2422

2423   if $A = \star$ and $B = \star$

2424   we have

2425 $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \sqsubseteq e_3 \Rightarrow \star \sqsubseteq \star$

2426 $\Rightarrow \forall k \geqslant 0, (\sigma_1, \sigma_2) \in \mathcal{G}[\![\Gamma_1 \sqsubseteq \Gamma_2]\!].$

2427 $(\sigma_1(e_1), \sigma_2(e_3)) \in \mathcal{E}_k^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

2428   if $\sigma_1(e_1) \longmapsto^* \mathrm{err}_* \vee \sigma_2(e_3) \longmapsto^* \mathrm{err}_a$, then the result holds

2429   if $\sigma_1(e_1) \longmapsto^* g_1 : \star \wedge \sigma_2(e_3) \longmapsto^* g_3 : \star$

2430 $\Rightarrow (g_1 : \star, g_3 : \star) \in \mathcal{V}_{k-n_1}^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

2431   we also have

2432 $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_2 \sqsubseteq e_4 \Leftarrow \star \sqsubseteq \star$

2433 $\Rightarrow (\sigma_1(e_2), \sigma_2(e_4)) \in \mathcal{E}_k^{\Leftarrow}[\![\star \sqsubseteq \star]\!]$

2434   then we want to prove

2435 $\Rightarrow ((g_1 : \star : \star \rightarrow \star)\ \sigma_1(e_2), (g_3 : \star : \star \rightarrow \star)\ \sigma_2(e_4)) \in \mathcal{E}_{k-n_1}^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

2436 $\Rightarrow (f_1\ \sigma_1(e_2), f_3\ \sigma_2(e_4)) \in \mathcal{E}_{k-n_1-n_2}^{\Rightarrow}[\![\star \sqsubseteq \star]\!]$

2437   by Lemma E.4, we have

2438 $\Rightarrow (g_1 : \star : \star \rightarrow \star, g_3 : \star : \star \rightarrow \star) \in \mathcal{E}_{k-n_1}^{\Rightarrow}[\![\star \rightarrow \star \sqsubseteq \star \rightarrow \star]\!]$

2439 $\Rightarrow (f_1, f_3) \in \mathcal{V}_{k-n_1-n_2}^{\Rightarrow}[\![\star \rightarrow \star \sqsubseteq \star \rightarrow \star]\!]$

2440   thus the result holds.

2441

2442   □

2443   LEMMA E.10 (COMPATIBILITY(PROJ)). *if* $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \sqsubseteq e_2 \Rightarrow A \sqsubseteq B, A \rhd \{l : A_1\}$ *and*

2444 $B \rhd \{l : B_1\}$ *then* $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1.l \sqsubseteq e_2.l \Rightarrow A_1 \sqsubseteq B_1$.

2445

2446   PROOF. we want to prove

2447 $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1.l \sqsubseteq e_2.l \Rightarrow A_1 \sqsubseteq B_1$

2448 $\Rightarrow \forall k \geqslant 0, (\sigma_1, \sigma_2) \in \mathcal{G}[\![\Gamma_1 \sqsubseteq \Gamma_2]\!].$

2449 $(\sigma_1(e_1.l), \sigma_2(e_2.l)) \in \mathcal{E}_k^{\Rightarrow}[\![A_1 \sqsubseteq B_1]\!]$

2450

2451 $\Rightarrow (\sigma_1(e_1).l, \sigma_2(e_2).l) \in \mathcal{E}_k^{\Rightarrow}[\![A_1 \sqsubseteq B_1]\!]$

2452 we have

2453 $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \sqsubseteq e_2 \Rightarrow A \sqsubseteq B$

2454 $\Rightarrow \forall k \geqslant 0, (\sigma_1, \sigma_2) \in \mathcal{G}[\![\Gamma_1 \sqsubseteq \Gamma_2]\!].$

2455 $(\sigma_1(e_1), \sigma_2(e_2)) \in \mathcal{E}_k^{\Rightarrow}[\![A \sqsubseteq B]\!]$

2456 if $\sigma_1(e_1) \longmapsto^* err_* \vee \sigma_2(e_2) \longmapsto^* err_a$, then the result holds

2457 if $\sigma_1(e_1) \longmapsto^* v_1 \wedge \sigma_2(e_2) \longmapsto^* v_2$

2458 $\Rightarrow (v_1, v_2) \in \mathcal{V}_{k-n_1}^{\Rightarrow}[\![A \sqsubseteq B]\!]$

2459 then we want to prove

2460 $(v_1.l, v_2.l) \in \mathcal{E}_{k-n_1}^{\Rightarrow}[\![A_1 \sqsubseteq B_1]\!]$

2461 $\Rightarrow (v_1 : \{l : A_1\}, v_2 : \{l : B_1\}) \in \mathcal{E}_{k-n_1}^{\Rightarrow}[\![\{l : A_1\} \sqsubseteq \{l : B_1\}]\!]$

2462 by Lemma E.4

2463 $(v_1 : \{l : A_1\}, v_2 : \{l : B_1\}) \in \mathcal{E}_{k-n_1}^{\Rightarrow}[\![\{l : A_1\} \sqsubseteq \{l : B_1\}]\!]$

2464 thus the result holds.

2465

2466   □

2467

2468   THEOREM E.11 (FUNDAMENTAL PROPERTY).

2469

2470 • *if* $\Gamma_1 \vdash e_1 \Rightarrow A, \Gamma_2 \vdash e_2 \Rightarrow B$ *and* $e_1 \sqsubseteq e_2$ *then* $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \sqsubseteq e_2 \Rightarrow A \sqsubseteq B$.

2471 • *if* $\Gamma_1 \vdash e_1 \Leftarrow A, \Gamma_2 \vdash e_2 \Leftarrow B$ *and* $e_1 \sqsubseteq e_2$ *then* $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \sqsubseteq e_2 \Leftarrow A \sqsubseteq B$.

2472

2473   PROOF. We do the induction on the typing derivation and case analysis on $e_1 \sqsubseteq e_2$.

2474 *Case* (typ-cs). we have

2475 $\dfrac{\Gamma_1 \vdash e_1 \Rightarrow A_1 \qquad A_1 \lesssim B_1}{\Gamma_1 \vdash e_1 \Leftarrow B_1}$

2476

2477 and $e_1 \sqsubseteq e_2$

2478 then e_2 has the following typing rule

2479 $\dfrac{\Gamma_2 \vdash e_2 \Rightarrow A_2 \qquad A_2 \lesssim B_2}{\Gamma_2 \vdash e_2 \Leftarrow B_2}$

2480

2481 we want to prove

2482 $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \sqsubseteq e_2 \Leftarrow B_1 \sqsubseteq B_2$

2483 $\Rightarrow \forall k \geqslant 0, (\sigma_1, \sigma_2) \in \mathcal{G}[\![\Gamma_1 \sqsubseteq \Gamma_2]\!].$

2484 $(\sigma(e_1), \sigma(e_2)) \in \mathcal{E}_k^{\Leftarrow}[\![B_1 \sqsubseteq B_2]\!]$

2485 $\Rightarrow (r_1, r_2) \in \mathcal{R}_{k-n}^{\Leftarrow}[\![B_1 \sqsubseteq B_2]\!]$

2486 by induction hypothesis, we have

2487 $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \sqsubseteq e_2 \Rightarrow A_1 \sqsubseteq A_2$

2488 $\Rightarrow \forall k \geqslant 0, (\sigma_1, \sigma_2) \in \mathcal{G}[\![\Gamma_1 \sqsubseteq \Gamma_2]\!].$

2489 $(\sigma(e_1), \sigma(e_2)) \in \mathcal{E}_k^{\Rightarrow}[\![A_1 \sqsubseteq A_2]\!]$

2490 $\Rightarrow (r_1, r_2) \in \mathcal{R}_{k-n}^{\Rightarrow}[\![A_1 \sqsubseteq A_2]\!]$

2491 if $r_1 = err_* \vee r_2 = err_a$, the result holds

2492 if $r_1 = v_1, r_2 = v_2$

2493 $\Rightarrow (v_1, v_2) \in \mathcal{V}_{k-n}^{\Rightarrow}[\![A_1 \sqsubseteq A_2]\!]$

2494 then we need to prove

2495 $(v_1, v_2) \in \mathcal{W}_{k-n}^{\Leftarrow}[\![B_1 \sqsubseteq B_2]\!]$

2496 $\Rightarrow (v_1 : B_1, v_2 : B_2) \in \mathcal{E}_{k-n}^{\Rightarrow}[\![B_1 \sqsubseteq B_2]\!]$

2497 the proof follows by Lemma E.4.

2498

2499

Case (typ-rt). we have

$$\frac{\Gamma_1, x : A_1 \vdash e_1 \Leftarrow A_2 \qquad \Gamma_1 \vdash e_3 \Rightarrow A_1}{\Gamma_1 \vdash (\lambda x.e_1)\ e_3 \Leftarrow A_2}$$

and $(\lambda x.e_1)\ e_3 \sqsubseteq (\lambda x.e_2)\ e_4$

then $(\lambda x.e_2)\ e_4$ has the following typing rule

$$\frac{\Gamma_2, x : B_1 \vdash e_2 \Leftarrow B_2 \qquad \Gamma_2 \vdash e_4 \Rightarrow B_1}{\Gamma_2 \vdash (\lambda x.e_2)\ e_4 \Leftarrow B_2}$$

we want to prove

$\Gamma_1 \sqsubseteq \Gamma_2 \vdash (\lambda x.e_1)\ e_3 \sqsubseteq (\lambda x.e_2)\ e_4 \ \Leftarrow\ A_2 \sqsubseteq B_2$
$\Rightarrow \forall k \geqslant 0, (\sigma_1, \sigma_2) \in \mathcal{G}[\![\Gamma_1 \sqsubseteq \Gamma_2]\!].$
$(\sigma((\lambda x.e_1)\ e_3), \sigma((\lambda x.e_2)\ e_4)) \in \mathcal{E}_k^{\Leftarrow}[\![A_2 \sqsubseteq B_2]\!]$
$\Rightarrow ((\lambda x.\sigma_1(e_1))\ \sigma_2(e_3), (\lambda x.\sigma_1(e_2))\ \sigma_2(e_4)) \in \mathcal{E}_k^{\Leftarrow}[\![A_2 \sqsubseteq B_2]\!]$

by the induction hypothesis, we have

$\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_3 \sqsubseteq e_4 \ \Rightarrow\ A_1 \sqsubseteq B_1$
$\Rightarrow \forall k \geqslant 0, (\sigma_1, \sigma_2) \in \mathcal{G}[\![\Gamma_1 \sqsubseteq \Gamma_2]\!].$
$(\sigma_1(e_3), \sigma_2(e_4)) \in \mathcal{E}_k^{\Rightarrow}[\![A_1 \sqsubseteq B_1]\!]$

if $\sigma_1(e_3) \longmapsto^* \mathsf{err}_* \vee \sigma_2(e_4) \longmapsto^* \mathsf{err}_a$, then the result holds

if $\sigma_1(e_3) \longmapsto^* v_3 \wedge \sigma_2(e_4) \longmapsto^* v_4$
then $(v_3, v_4) \in \mathcal{V}_{k-n}^{\Rightarrow}[\![A_1 \sqsubseteq B_1]\!]$

by the induction hypothesis, we also have

$\Gamma_1, x : A_1 \sqsubseteq \Gamma_2, x : B_1 \vdash e_1 \sqsubseteq e_2 \ \Leftarrow\ A_2 \sqsubseteq B_2$
$\Rightarrow \forall k \geqslant 0, (\sigma_1[x/v_2], \sigma_2[x/v_4]) \in \mathcal{G}[\![\Gamma_1, x : A_1 \sqsubseteq \Gamma_2, x : B_1]\!].$
$(\sigma_1[x/v_2](e_1), \sigma_2[x/v_4](e_3)) \in \mathcal{E}_k^{\Leftarrow}[\![A_2 \sqsubseteq B_2]\!]$

by Lemma E.5

$\Rightarrow (\sigma_1(e_1)[x/v_3], \sigma_2(e_2)[x/v_4]) \in \mathcal{E}_k^{\Leftarrow}[\![A_2 \sqsubseteq B_2]\!]$

we need to prove

$((\lambda x.\sigma_1(e_1))\ \sigma_1(e_3), (\lambda x.\sigma_2(e_2))\ \sigma_2(e_4)) \in \mathcal{E}_k^{\Leftarrow}[\![A_2 \sqsubseteq B_2]\!]$
$\Rightarrow ((\lambda x.\sigma_1(e_1))\ v_3, (\lambda x.\sigma_1(e_2))\ v_4) \in \mathcal{E}_{k-n}^{\Leftarrow}[\![A_2 \sqsubseteq B_2]\!]$
$\Rightarrow (\sigma_1(e_1)[x/v_3], \sigma_2(e_2)[x/v_4]) \in \mathcal{E}_{k-n-1}^{\Leftarrow}[\![A_2 \sqsubseteq B_2]\!]$

Thus the result holds.

Case (typ-abs). we have

$$\frac{\Gamma_1, x : A_1 \vdash e_1 \Leftarrow A_2 \qquad A \triangleright A_1 \rightarrow A_2}{\Gamma_1 \vdash \lambda x.e_1 \Leftarrow A}$$

and $\lambda x.e_1 \sqsubseteq \lambda x.e_2$

then $\lambda x.e_2$ has the following typing rule

$$\frac{\Gamma_2, x : B_1 \vdash e_2 \Leftarrow B_2 \qquad B \triangleright B_1 \rightarrow B_2}{\Gamma_2 \vdash \lambda x.e_2 \Leftarrow B}$$

we want to prove

$\Gamma_1 \sqsubseteq \Gamma_2 \vdash \lambda x.e_1 \sqsubseteq \lambda x.e_2 \ \Leftarrow\ A \sqsubseteq B$
$\Rightarrow \forall k \geqslant 0, (\sigma_1, \sigma_2) \in \mathcal{G}[\![\Gamma_1 \sqsubseteq \Gamma_2]\!].$
$(\lambda x.\sigma_1(e_1), \lambda x.\sigma_2(e_2)) \in \mathcal{E}_k^{\Leftarrow}[\![A \sqsubseteq B]\!]$
$\Rightarrow ((\lambda x.\sigma_1(e_1)) : A, (\lambda x.\sigma_2(e_2)) : B) \in \mathcal{E}_k^{\Rightarrow}[\![A \sqsubseteq B]\!]$
$\Rightarrow ((\lambda x.\sigma_1(e_1)) : A_1 \rightarrow A_2, (\lambda x.\sigma_2(e_2)) : B_1 \rightarrow B_2) \in \mathcal{V}_k^{\Rightarrow}[\![A_1 \rightarrow A_2 \sqsubseteq B_1 \rightarrow B_2]\!]$
$\Rightarrow \forall j \leqslant k, (e_3, e_4) \in \mathcal{E}_j^{\Leftarrow}[\![A_1 \sqsubseteq B_1]\!].$
$(((\lambda x.\sigma_1(e_1)) : A_1 \rightarrow A_2)\ e_3, ((\lambda x.\sigma_2(e_2)) : B_1 \rightarrow B_2)\ e_4) \in \mathcal{E}_j^{\Rightarrow}[\![A_2 \sqsubseteq B_2]\!]$
$\Rightarrow (((\lambda x.\sigma_1(e_1))\ (e_3 : A_1)) : A_2, ((\lambda x.\sigma_2(e_2))\ (e_4 : B_1)) : B_2) \in \mathcal{E}_{j-1}^{\Rightarrow}[\![A_2 \sqsubseteq B_2]\!]$

2549 $\Rightarrow (((\lambda x.\sigma_1(e_1))\ v_3) : A_2, ((\lambda x.\sigma_2(e_2))\ v_4) : B_2) \in \mathcal{E}^{\Rightarrow}_{j-1-n_1}[\![A_2 \sqsubseteq B_2]\!]$

2550 $\Rightarrow ((\sigma_1(e_1)[x/v_3]) : A_2, (\sigma_2(e_2)[x/v_4]) : B_2) \in \mathcal{E}^{\Rightarrow}_{j-2-n_1}[\![A_2 \sqsubseteq B_2]\!]$

2551   by induction hypothesis, we have

2552 $\Gamma_1, x : A_1 \sqsubseteq \Gamma_2, x : A_2 \vdash e_1 \sqsubseteq e_2 \ \Leftarrow\ A_2 \sqsubseteq B_2$

2553 $\Rightarrow \forall k \geqslant 0, (\sigma_1[x/v_3], \sigma_2[x/v_4]) \in \mathcal{G}[\![\Gamma_1, x : A_1 \sqsubseteq \Gamma_2, x : B_2]\!].$

2554 $(\sigma_1[x/v_3](e_1), \sigma_2[x/v_4](e_2)) \in \mathcal{E}^{\Leftarrow}_k[\![A_2 \sqsubseteq B_2]\!]$

2555   The proof follows by Lemma E.5 and Lemma E.4.

2556

2557   *Case* (typ-anno). we have

2558 $\dfrac{\Gamma_1 \vdash e_1 \Leftarrow B_1}{\Gamma_1 \vdash e_1 : B_1 \Rightarrow B_1}$

2559

2560   and $e_1 : B_1 \sqsubseteq e_2 : B_2$

2561   then $e_2 : B_2$ has the following typing rule

2562 $\dfrac{\Gamma_2 \vdash e_2 \Leftarrow B_2}{\Gamma_2 \vdash e_2 : B_2 \Rightarrow B_2}$

2563

2564   we want to prove

2565 $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 : B_1 \sqsubseteq e_2 : B_2 \ \Rightarrow\ B_1 \sqsubseteq B_2$

2566   by induction hypothesis, we have

2567 $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \sqsubseteq e_2 \ \Leftarrow\ B_1 \sqsubseteq B_2$

2568   The proof follows by Lemma E.8.

2569   *Case* (typ-merge). we have

2570 $\Gamma_1 \vdash e_1 \Rightarrow A_1$

2571 $\dfrac{\Gamma_1 \vdash e_2 \Rightarrow A_2}{\Gamma_1 \vdash e_1,,e_2 \Rightarrow A_1 \& A_2}$

2572

2573   and $e_1,,e_2 \sqsubseteq e_3,,e_4$

2574   then $e_3,,e_4$ has the following typing rule

2575 $\Gamma_2 \vdash e_3 \Rightarrow B_1$

2576 $\dfrac{\Gamma_2 \vdash e_4 \Rightarrow B_2}{\Gamma_2 \vdash e_3,,e_4 \Rightarrow B_1 \& B_2}$

2577

2578   we want to prove

2579 $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1,,e_2 \sqsubseteq e_3,,e_4 \ \Rightarrow\ A_1 \& A_2 \sqsubseteq B_1 \& B_2$

2580   by induction hypothesis, we have

2581 $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \sqsubseteq e_3 \ \Rightarrow\ A_1 \sqsubseteq B_1$

2582 $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_2 \sqsubseteq e_4 \ \Rightarrow\ A_2 \sqsubseteq B_2$

2583   The proof follows by Lemma E.6.

2584   *Case* (typ-rcd). we have

2585 $\dfrac{\Gamma_1 \vdash e_1 \Rightarrow A}{\Gamma_1 \vdash \{l = e_1\} \Rightarrow \{l : A\}}$

2586

2587   and $\{l = e_1\} \sqsubseteq \{l = e_2\}$

2588   then $e_2 : B_2$ has the following typing rule

2589 $\dfrac{\Gamma_2 \vdash e_2 \Rightarrow B}{\Gamma_2 \vdash \{l = e_2\} \Rightarrow \{l : B\}}$

2590

2591   we want to prove

2592 $\Gamma_1 \sqsubseteq \Gamma_2 \vdash \{l = e_1\} \sqsubseteq \{l = e_2\} \ \Rightarrow\ \{l : A\} \sqsubseteq \{l : B\}$

2593   by induction hypothesis, we have

2594 $\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \sqsubseteq e_2 \ \Rightarrow\ A \sqsubseteq B$

2595   The proof follows by Lemma E.7.

2596

2597

Case (typ-app). we have
$$\frac{\Gamma_1 \vdash e_1 \Rightarrow A \qquad A \rhd A_1 \rightarrow A_2 \qquad \Gamma_1 \vdash e_2 \Rightarrow A_1}{\Gamma_1 \vdash e_1 \; e_2 \Rightarrow A_2}$$
and $e_1 \; e_2 \sqsubseteq e_3 \; e_4$
then $e_3 \; e_4$ has the following typing rule
$$\frac{\Gamma_2 \vdash e_3 \Rightarrow B \qquad B \rhd B_1 \rightarrow B_2 \qquad \Gamma_2 \vdash e_4 \Rightarrow B_1}{\Gamma_2 \vdash e_3 \; e_4 \Rightarrow B_2}$$
we want to prove
$\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \; e_2 \sqsubseteq e_3 \; e_4 \; \Rightarrow \; A_2 \sqsubseteq B_2$
by induction hypothesis, we have
$\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \sqsubseteq e_3 \; \Rightarrow \; A \sqsubseteq B$
$\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_2 \sqsubseteq e_4 \; \Leftarrow \; A_1 \sqsubseteq B_1$
The proof follows by Lemma E.9.

Case (typ-proj). we have
$$\frac{\Gamma_1 \vdash e_1 \Rightarrow A \qquad A \rhd \{l : A_1\}}{\Gamma_1 \vdash e_1.l \Rightarrow A_1}$$
and $e_1.l \sqsubseteq e_2.l$
then $e_2.l$ has the following typing rule
$$\frac{\Gamma_2 \vdash e_2 \Rightarrow B \qquad B \rhd \{l : B_1\}}{\Gamma_2 \vdash e_2.l \Rightarrow B_1}$$
we want to prove
$\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1.l \sqsubseteq e_2.l \; \Rightarrow \; A_1 \sqsubseteq B_1$
by induction hypothesis, we have
$\Gamma_1 \sqsubseteq \Gamma_2 \vdash e_1 \sqsubseteq e_2 \; \Rightarrow \; A \sqsubseteq B$
The proof follows by Lemma E.10.

Case (typ-lit). This case is trivial.

Case (typ-top). This case is trivial.

Case (typ-var). we have
$$\frac{x : A \in \Gamma_1}{\Gamma_1 \vdash x \Rightarrow A}$$
and $x \sqsubseteq x$
then x has the following typing rule
$$\frac{x : B \in \Gamma_2}{\Gamma_2 \vdash x \Rightarrow B}$$
we want to prove
$\Gamma_1 \sqsubseteq \Gamma_2 \vdash x \sqsubseteq x \; \Rightarrow \; A \sqsubseteq B$
$\Rightarrow \forall k \geqslant 0, (\sigma_1, \sigma_2) \in \mathcal{G}[\![\Gamma_1 \sqsubseteq \Gamma_2]\!].$
$(\sigma_1(x), \sigma_2(x)) \in \mathcal{E}_k^{\Rightarrow}[\![A \sqsubseteq B]\!]$
$\Rightarrow (\sigma_1(x), \sigma_2(x)) \in \mathcal{V}_k^{\Rightarrow}[\![A \sqsubseteq B]\!]$
since
$(\sigma_1(x), \sigma_2(x)) \in \mathcal{V}_k^{\Rightarrow}[\![\Gamma_1(x) \sqsubseteq \Gamma_2(x)]\!]$
$\Rightarrow (\sigma_1(x), \sigma_2(x)) \in \mathcal{V}_k^{\Rightarrow}[\![A \sqsubseteq B]\!]$
Thus the result holds.

$\square$

LEMMA E.12. *if* $\vdash v_1 \Leftarrow A, \vdash v_2 \Leftarrow B, v_1 \sqsubseteq v_2, A \sqsubseteq B$ *and* $v_2 \longmapsto_A \mathrm{err}_t$ *then* $v_1 \longmapsto_B \mathrm{err}_*$.

PROOF. By induction on the casting. The proof follows by induction hypothesis and the coq lemma named "precise_csub". □

THEOREM E.13. *if* $\vdash e_1 \Leftrightarrow A, \vdash e_2 \Leftrightarrow B, e_1 \sqsubseteq e_2$ *and* $e_2 \longmapsto \text{err}_t$ *then* $e_1 \longmapsto^* \text{err}_*$.

PROOF. By induction on the precision. The proof follows by induction hypothesis and Lemma E.12. □

THEOREM E.14. *if* $\vdash e_1 \Leftrightarrow A, \vdash e_2 \Leftrightarrow B, e_1 \sqsubseteq e_2$ *and* $e_2 \longmapsto^* \text{err}_t$ *then* $e_1 \longmapsto^* \text{err}_*$.

PROOF. The proof follows by Theorem E.13. □

THEOREM E.15. *if* $\vdash e_1 \Leftrightarrow A, \vdash e_2 \Leftrightarrow B, e_1 \sqsubseteq e_2$ *and* $e_2 \longmapsto_* v_2$ *then* $e_1 \longmapsto_* v_1$ *or* $e_1 \longmapsto_* \text{err}_*$.

PROOF. by induction on the precision $e_1 \sqsubseteq e_2$.

Case $(i, x, \text{Top}, \lambda x.e)$. these cases are trivial.

Case (e_1, e_2).
we have
$$\frac{e_1 \sqsubseteq e_3 \qquad e_2 \sqsubseteq e_4}{e_1, e_2 \sqsubseteq e_3, e_4}$$
then the result follows by the induction hypothesis.

Case $(e : A)$.
we have
$$\frac{e_1 \sqsubseteq e_2}{e_1 : A \sqsubseteq e_2 : B}$$
then the result follows by the induction hypothesis and the casting progress lemma.

Case $(\{l = e\})$.
we have
$$\frac{e_1 \sqsubseteq e_2}{\{l = e_1\} \sqsubseteq \{l = e_2\}}$$
then the result follows by the induction hypothesis.

Case $(e.l)$.
we have
$$\frac{e_1 \sqsubseteq e_2}{e_1.l \sqsubseteq e_2.l}$$
then the result follows by the induction hypothesis and the casting progress lemma.

Case $((\lambda x.e_1)\ e_2)$.
we have
$$\frac{\lambda x.e_1 \sqsubseteq \lambda x.e_3 \qquad e_2 \sqsubseteq e_4}{(\lambda x.e_1)\ e_2 \sqsubseteq (\lambda x.e_3)\ e_4}$$
the result follows by Theorem E.11 and the induction hypothesis.

Case $(e_1\ e_2)$.
we have
$$\frac{e_1 \sqsubseteq e_3 \qquad e_2 \sqsubseteq e_4}{e_1\ e_2 \sqsubseteq e_3\ e_4}$$
we have $e_3\ e_4 \longmapsto^* v_2$ then $e_3 \longmapsto^* g_3 : \star \vee e_3 \longmapsto^* f_3$
+ if $e_3 \longmapsto^* g_3 : \star$
by induction hypothesis,
$e_1 \longmapsto^* \text{err}_*$ or $e_1 \longmapsto^* g_1 : \star$ or $e_1 \longmapsto^* f_1$
* if $e_1 \longmapsto^* \text{err}_*$, then the result holds

2696   * if $e_1 \longmapsto^* g_1 : \star$, then
2697   by Theorem E.11, we have
2698 $(g_1 : \star, g_3 : \star) \in \mathcal{V}^{\Rightarrow}_{n_1}[\![\star \sqsubseteq \star]\!]$
2699   by Lemma E.4
2700 $(g_1 : \star : \star \to \star, g_3 : \star : \star \to \star) \in \mathcal{E}^{\Rightarrow}_{n_1}[\![\star \to \star \sqsubseteq \star \to \star]\!]$
2701 $\Rightarrow (f_1, f_2) \in \mathcal{V}^{\Rightarrow}_{n_2}[\![\star \to \star \sqsubseteq \star \to \star]\!]$
2702   by Theorem E.11, we have
2703 $(e_2, e_4) \in \mathcal{E}^{\Leftarrow}_{n_2}[\![\star \sqsubseteq \star]\!]$
2704   we need to prove
2705 $e_1 \, e_2 \longmapsto^* v_1 \vee e_1 \, e_2 \longmapsto^* \text{err}_*$
2706   we have
2707 $e_1 \, e_2 \longmapsto^* (g_1 : \star) \, e_2$
2708   then we need to prove
2709 $(g_1 : \star) \, e_2 \longmapsto^* (g_1 : \star : \star \to \star) \, e_2$
2710 $\Rightarrow (g_1 : \star : \star \to \star) \, e_2 \longmapsto^* f_1 \, e_2$
2711 $\Rightarrow f_1 \, e_2 \longmapsto^* r_1$
2712   from $(f_1, f_2) \in \mathcal{V}^{\Rightarrow}_{n_2}[\![\star \to \star \sqsubseteq \star \to \star]\!]$
2713   we know
2714 $(f_1 \, e_2, f_2 \, e_4) \in \mathcal{E}^{\Rightarrow}_{n_2}[\![\star \sqsubseteq \star]\!]$
2715   because $e_3 \, e_4 \longmapsto^* f_2 \, e_4$
2716 $f_2 \, e_4 \longmapsto^* v_2$
2717   thus the result holds
2718   * if $e_1 \longmapsto^* f_1$ then
2719   by Theorem E.11, we have
2720 $(f_1, g_3 : \star) \in \mathcal{V}^{\Rightarrow}_{n_1}[\![A_1 \to A_2 \sqsubseteq \star]\!]$
2721 $(e_2, e_4) \in \mathcal{E}^{\Leftarrow}_{n_1}[\![A_1 \sqsubseteq \star]\!]$
2722   from $(f_1, g_3 : \star) \in \mathcal{V}^{\Rightarrow}_{n_1}[\![A_1 \to A_2 \sqsubseteq \star]\!]$
2723 $\Rightarrow f_2 \in g_3 : \star.$
2724 $(f_1, f_2) \in \mathcal{V}^{\Rightarrow}_{n_1}[\![A_1 \to A_2 \sqsubseteq \star \to \star]\!]$
2725   we need to prove
2726 $e_1 \, e_2 \longmapsto^* (g_1 : \star) \, e_2$
2727 $\Rightarrow (g_1 : \star) \, e_2 \longmapsto^* (f_1 : \star \to \star) \, e_2$
2728 $\Rightarrow (f_1 : \star \to \star) \, e_2 \longmapsto^* (v_1 \vee \text{err}_*)$
2729   because $(f_1, f_2) \in \mathcal{V}^{\Rightarrow}_{n_1}[\![A_1 \to A_2 \sqsubseteq \star \to \star]\!]$
2730 $\Rightarrow (f_1 \, e_2, f_2 \, e_4) \in \mathcal{V}^{\Rightarrow}_{n_1}[\![A_2 \sqsubseteq \star]\!]$
2731   because $e_3 \, e_4 \longmapsto^* f_2 \, e_4$
2732 $f_2 \, e_4 \longmapsto^* v_2$
2733   from Lemma E.2, we also know
2734   if $f_1 \, e_2 \longmapsto^* v_1$
2735   then $(f_1 : \star \to \star) \, e_2 \longmapsto^* v_1$
2736   Thus the result holds.
2737   + if $e_3 \longmapsto^* f_2$
2738   then $e_1 \longmapsto^* f_1$
2739   by Theorem E.11, we have
2740 $(f_1, f_2) \in \mathcal{V}^{\Rightarrow}_{n_1}[\![A_1 \to A_2 \sqsubseteq B_1 \to B_2]\!]$
2741 $(e_2, e_4) \in \mathcal{E}^{\Leftarrow}_{n_1}[\![A_1 \sqsubseteq B_1]\!]$
2742   from $(f_1, f_2) \in \mathcal{V}^{\Rightarrow}_{n_1}[\![A_1 \to A_2 \sqsubseteq B_1 \to B_2]\!]$
2743
2744

2745 $\Rightarrow (f_1\ e_2, f_2\ e_4) \in \mathcal{E}^{\Rightarrow}_{n_1}[\![A_2 \sqsubseteq B_2]\!]$

2746   we need to prove

2747 $e_1\ e_2 \longmapsto^* f_1\ e_2$

2748 $\Rightarrow f_1\ e_2 \longmapsto^* (v_1 \vee \text{err}_*)$

2749   we know that

2750 $e_3\ e_4 \longmapsto^* f_2\ e_4$

2751 $f_2\ e_4 \longmapsto^* v_2$

2752   Thus the result holds.

2753

2754   □

2755   THEOREM E.16 (LEFT DIRECTION).   *Suppose* $\vdash e_1 \Leftrightarrow A, \vdash e_2 \Leftrightarrow B, e_1 \sqsubseteq e_2$.

2756 (1) $e_2 \longmapsto_* v_2$ *then* $e_1 \longmapsto_* v_1$ *or* $e_1 \longmapsto_* \text{err}_*$

2757 (2) $e_2 \longmapsto_* \text{err}_t$ *then* $e_1 \longmapsto_* \text{err}_*$

2758

2759   PROOF. The proof follows by Theorem E.15 and Theorem E.14.   □

2760   THEOREM E.17 (DYNAMIC GRADUAL GUARANTEE). *Suppose* $\vdash e_1 \Leftrightarrow A, \vdash e_2 \Leftrightarrow B$ *and* $e_1 \sqsubseteq e_2$.

2761 (1) $e_1 \longmapsto^* v_1$ *then* $e_2 \longmapsto^* v_2$ *and* $\vdash v_1 \sqsubseteq v_2 \Leftrightarrow A \sqsubseteq B$ *or* $e_2 \longmapsto^* \text{err}_a$

2762 (2) $e_1 \Uparrow$ *then* $e_2 \Uparrow$ *or* $e_2 \longmapsto^* \text{err}_a$

2763 (3) $e_2 \longmapsto^* v_2$ *then* $e_1 \longmapsto^* v_1$ *and* $\vdash v_1 \sqsubseteq v_2 \Leftrightarrow A \sqsubseteq B$ *or* $e_1 \longmapsto^* \text{err}_*$

2764 (4) $e_2 \Uparrow$ *then* $e_1 \Uparrow$ *or* $e_1 \longmapsto^* \text{err}_*$

2765

2766   PROOF.

2767

2768 (1) The proof follows by Theorem E.11.

2769 (2) we know that $e_2 \longmapsto^* r_2$ or $e_2 \Uparrow$, if $r_2 = \text{err}_a$, or $e_2 \Uparrow$, the result holds.

2770   if $e_2 \longmapsto^* r_2$ and $r_2 \neq \text{err}_a$, by Theorem E.16, $e_1 \longmapsto^* r_1$, this is contradictory. Thus the result

2771   holds.

2772 (3) The result follows by Theorem E.16 and (1).

2773 (4) we know that $e_1 \longmapsto^* r_1$ or $e_1 \Uparrow$, if $r_1 = \text{err}_*$, or $e_1 \Uparrow$, the result holds.

2774   if $e_1 \longmapsto^* v_1$ by (1), $e_2 \longmapsto^* v_2$ or $e_2 \longmapsto^* \text{err}_a$, this is contradictory. Thus the result holds.

2775

2776   □

2777

2778

2779

2780

2781

2782

2783

2784

2785

2786

2787

2788

2789

2790

2791

2792

2793