JFP, 64 pages, 2023. © Cambridge University Press 2023
doi:10.1017/xxxxx

A Type-Directed, Dictionary-Passing Translation of Method Overloading and Structural Subtyping in Featherweight Generic Go

MARTIN SULZMANN
Karlsruhe University of Applied Sciences, Germany
e-mail: martin.sulzmann@h-ka.de

STEFAN WEHR
Offenburg University of Applied Sciences, Germany
e-mail: stefan.wehr@hs-offenburg.de

Abstract

Featherweight Generic Go (FGG) is a minimal core calculus modeling the essential features of the programming language Go. It includes support for overloaded methods, interface types, structural subtyping and generics. The most straightforward semantic description of the dynamic behavior of FGG programs is to resolve method calls based on runtime type information of the receiver.

This article shows a different approach by defining a type-directed translation from FGG$^-$ to an untyped lambda-calculus. FGG$^-$ includes all features of FGG but type assertions. The translation of an FGG$^-$ program provides evidence for the availability of methods as additional dictionary parameters, similar to the dictionary-passing approach known from Haskell type classes. Then, method calls can be resolved by a simple lookup of the method definition in the dictionary.

Every program in the image of the translation has the same dynamic semantics as its source FGG$^-$ program. The proof of this result is based on a syntactic, step-indexed logical relation. The step-index ensures a well-founded definition of the relation in the presence of recursive interface types and recursive methods. Although being non-deterministic, the translation is coherent.

1 Introduction

Go (2022) is a statically typed programming language introduced by Google in 2009. It supports method overloading by allowing multiple declarations of the same method signature for different receivers. Receivers are structs, similar to structs in C. The language also supports interfaces; as in many object-oriented languages, an interface consists of a set of method signatures. But unlike in many object-oriented languages, subtyping in Go is structural not nominal.

Earlier work by Griesemer et al. (2020) introduces Featherweight Go (FG), a minimal core calculus that covers method overloading, structs, interfaces and structural subtyping. Their work specifies static typing rules and a dynamic semantics for FG based on runtime method lookup. However, the actual Go implementation appears to employ a different dynamic semantics. Quoting Griesemer and co-workers: "*Go is designed to enable efficient implementation. Structures are laid out in memory as a sequence of fields, while an interface is a pair of a pointer to an underlying structure and a pointer to a dictionary of methods.*"

In our own prior work (Sulzmann and Wehr, 2021, 2022), we formalize a type-directed dictionary-passing translation for FG and establish its semantic equivalence with FG's dynamic semantics. Griesemer and coworkers also introduce Featherweight Generic Go (FGG), an extension of FG with generics. In this work, we show how our translation approach can be extended to deal with generics. Our focus is on the integration of generics with method overloading and structural subtyping. Hence, we consider FGG$^-$, which is equivalent to FGG but does not support type assertions. Our contributions are as follows:

- We specify the translation of source FGG$^-$ programs to an untyped λ-calculus with recursive let-bindings, constructors and pattern matching. We employ a dictionary-passing translation scheme à la type classes (Hall et al., 1996) to statically resolve overloaded method calls. The translation is guided by the typing of the FGG$^-$ program. As the typing rules include a subsumption rule, the translation is inherently non-deterministic.
- We establish the semantic correctness of the dictionary-passing translation. The result relies on a syntactic, step-indexed logical relation to ensure well-foundedness of definitions in the presence of recursive interface types and recursive methods.
- We show that values produced by different translations of the same program are identical up to dictionaries embedded inside these values.
- We report on an implementation of the translation.

The upcoming Section 2 presents an overview of our translation by example. Section 3 gives a recap of the source language FGG$^-$, whereas Section 4 defines the target language and the translation itself. Next, Section 5 establishes the formal properties of the translation, rigorous proofs of our results can be found in the Appendix. Section 6 presents the implementation, Section 7 covers related work. Finally, Section 8 summarizes this work and points out directions for future work.

2 Overview

This section introduces Featherweight Generic Go (FGG Griesemer et al., 2020) and our type-directed dictionary-passing translation through a series of examples. FGG is a tiny model of Go that includes essential typing features such as method overloading, structs, interfaces, structural subtyping, and the extension with generics. Its original formulation also includes type assertions (dynamic type casts). As we omitted this feature from our translation, we use the name FGG$^-$ to refer to the source language of our translation. Except for the omission of type assertions, FGG$^-$ and FGG are equivalent.

An FGG$^-$ program consists of declarations for structs, interfaces, methods, and a main function. Function and method bodies only contain a single return statement, all expression are free from side effects. For the examples in this section, we extend FGG$^-$ with primitive

```
 1  type Num     struct { val int }
 2  type Format interface { format() string }
 3  type Pretty interface { format() string; pretty() string }
 4
 5  func (this Num) format() string { return intToString(this.val) }
 6  func (this Num) pretty() string { return this.format() }
 7
 8  func formatSome(x Format) string { return x.format() }
 9
10  func main() {
11    var s1 string = formatSome(Num{1})
12    var pr Pretty = Num{2}
13    var s2 string = formatSome(pr)
14  }
```

```
15  -- Field access for struct Num
16  val x = x
17
18  -- Method calls on interfaces
19  format_Format (x, f) = f x      -- call format on receiver of type Format
20  format_Pretty (x, (f,p)) = f x  -- call format on receiver of type Pretty
21  pretty_Pretty (x, (f,p)) = p x  -- call pretty on receiver of type Pretty
22
23  -- Method definitions (lines 5 and 6 in the FGG code)
24  format_Num this = intToString (val this)
25  pretty_Num this = format_Num this
26
27  -- Coercions
28  toFormat_Num x = (x,format_Num)                -- Num <: Format
29  toPretty_Num x = (x,(format_Num,pretty_Num))   -- Num <: Pretty
30  toFormat_Pretty (x,(f,p)) = (x,f)              -- Pretty <: Format
31
32  -- Function definitions (lines 8 and 10 in the FGG code)
33  formatSome x = format_Format x
34
35  main = let s1 = formatSome (toFormat_Num 1)
36             pr = toPretty_Num 2
37             s2 = formatSome (toFormat_Pretty pr)
```

Fig. 1. String-formatting and its translation

types for integers and strings, with an operator + for string concatenation and a builtin function `intToString`, with definitions of local variables, and with function definitions.

We will first consider FGG$^-$ without generics to highlight the idea behind our type-directed dictionary-passing translation scheme. Then, we show how the translation scheme can be adapted to deal with the addition of generics. All examples have been checked against our implementation[1] of the translation.

2.1 Starting Without Generics

The upper part of Figure 1 shows an (extended) FGG$^-$ program for formatting values as strings. The code does not use generics yet.

Structs in Go are similar to structs in C, a syntactic difference is the Go convention that field or variable names precede their types. Here, struct `Num` has a single field `val` of type `int`, so it simply acts as a wrapper for integers.

Interfaces in Go declare sets of method signatures sharing the same receiver where method names must be distinct and the receiver is left implicit. Interfaces are types and

[1] `https://github.com/skogsbaer/fgg-translate` and `http://doi.org/10.5281/zenodo.8147425`

describe all receivers that implement the methods declared by the interface. In our example, interface `Format` declares a method `format` for rendering its receiver as a string. The second interface `Pretty` also declares `format`, but adds a second method `pretty` with the intention to produce a visually more attractive output.

Methods and functions are introduced via the keyword **`func`**. A method can be distinguished from a function as the receiver argument in parenthesis precedes the method name. Methods can be overloaded on the receiver type. In lines 5 and 6, we find methods `format` and `pretty`, respectively, for receiver type `Num`. In the body of `format`, we assume a builtin function `intToString` for converting integers to strings. Lines 8 and 10 define two functions.

An interface only names a set of method signatures, its definition is not required for a method to be valid. For example, the methods in lines 5 and 6 could be defined without the interfaces in lines 2 and 3, or the methods could be placed before the interfaces.

However, interfaces and method definitions imply structural subtype relations. Interface `Format` contains a subset of the methods declared by interface `Pretty`. Hence, `Pretty` is a structural subtype of `Format`, written (1) `Pretty` <: `Format`. Line 5 defines method `format` for receiver type `Num`, we say that `Num` implements method `format`. Hence, `Num` is a structural subtype of `Format`, written (2) `Num` <: `Format`. Receiver `Num` also implements the `pretty` method, see line 6. Hence, we also find that (3) `Num` <: `Pretty`. Structural subtype relations play a crucial role when type checking programs.

For example, consider the function call `formatSome(Num{1})` in line 11. Here, `Num{1}` is a value of the `Num` struct with `val` set to `1`. From above we find that (2) `Num` <: `Format`. That is, `Num` implements the `Format` interface and therefore the function call type checks. Consider the variable declaration and assignment in line 12. Value `Num{2}` is assigned to a variable of interface type `Pretty`. Based on the subtype relation (3) `Num` <: `Pretty` the assignment type checks. Consider the function call `formatSome(pr)` in line 13, where `pr` has type `Pretty`. Based on the subtype relation (1) `Pretty` <: `Format` the function call type checks.

In Griesemer et al. (2020), the dynamic behavior of programs is explained via runtime lookup of methods, where based on the receiver's runtime type the appropriate method definition is selected. The Go (and FGG/FGG^{-}) conditions demand that for each method name and receiver type there can be at most one definition. This guarantees that method calls can be resolved unambiguously.

2.2 Type-Directed Translation

We explain the meaning of extended FGG^{-} programs by translation into an untyped λ-calculus with recursive top-level definitions, let-bindings, pattern matching, integers, strings, an operator ++ for string concatenation, and a builtin function `intToString`. We will use a Haskell-style notation.

Method definitions belonging to an interface are grouped together in a *dictionary* of methods. Thus, method calls can be turned into primitive function calls by simply looking up the appropriate method in the dictionary. Structural subtype relations are turned into *coercion* functions that transform, for example, a struct value into an interface value to make sure that the appropriate dictionaries are available. Where to insert dictionaries and coercions in the program is guided by the type checking rules. Hence, the translation is type-directed.

Our translation strategy can be summarized as follows:

Struct. An FGG$^-$ value at the type of a struct with n fields is represented by an n-tuple holding the values of the fields. We call such an n-tuple a *struct value*.

Interface. An FGG$^-$ value at the type of an interface is represented as a pair $(V, \mathcal{D})$, where V is a struct value and $\mathcal{D}$ is a *method dictionary*. Such a method dictionary is a tuple holding implementations of all interface methods for V, in order of declaration in the interface. We call the pair $(V, \mathcal{D})$ an *interface value*.

Coercion. A structural subtype relation $\tau <: \sigma$ implies a *coercion function* to transform the target representation of an FGG$^-$ value at type τ into a representation at type σ.

The lower part of Figure 1 gives the translation of our running example. In this overview section, we identify a 1-tuple with the single value it holds.

For each field name, we assume a helper function to access the field component, see line 16. Method calls on interface values lookup the respective method definition in the dictionary and apply it to the struct value embedded inside the interface value. See lines 19-21. Method definitions translate to plain functions, see lines 24-25. Recall that for each method name and receiver type there can be at most one definition. Hence, the generated function names are all distinct.

Structural subtype relations translate to coercions, see lines 28-30. For example, (2) `Num` $<:$ `Format` translates to the $\texttt{toFormat}_{\texttt{Num}}$ coercion. Input parameter `x` represents a target representation of a `Num` value. The output $\texttt{(x,format}_{\texttt{Num}}\texttt{)}$ is an interface value holding the receiver and the corresponding method definition. Coercion $\texttt{toPretty}_{\texttt{Num}}$ corresponds to (3) `Num` $<:$ `Pretty` and coercion $\texttt{toFormat}_{\texttt{Pretty}}$ to (1) `Pretty` $<:$ `Format`.

The translation of the main function, starting at line 35, is guided by the type checking of the source program. Each application of a structural subtype relation leads to the insertion of the corresponding coercion function in the target program. For example, the function call `formatSome(Num{1})` translates to $\texttt{formatSome (toFormat}_{\texttt{Num}}\texttt{ 1)}$ because typing of the source requires (2) `Num` $<:$ `Format`. The other coercions arise for similar reasons.

2.3 Adding Generics

We extend our running example by including pairs, see Figure 2. The struct type `Pair[T Any, U Any]` is *generic* in the type of the pair components, `T` and `U` are *type variables*. When introducing type variables we must also specify an upper *type bound* to constrain the set of concrete types that will replace type variables. The *bounded type parameter* `T Any` can therefore be interpreted as $\forall$`T.T` $<:$ `Any`. Upper bounds are always interface types. The upper bound `Any` is satisfied by any type because the set of methods that need to be implemented is empty.

To format pairs, we need to format the left and right component that are of generic types `T` and `U`. Hence, the method definition for `format` in line 4 states the type bound `Format` for type variables `T` and `U`. In general, bounds of type parameters for the receiver struct of a method declaration must be in a covariant subtype relation relative to the

```
 1  type Any interface {}
 2  type Pair[T Any, U Any] struct { left T; right U }
 3
 4  func (this Pair[T Format, U Format]) format() string {
 5    return "(" + this.left.format() + "," + this.right.format() + ")"
 6  }
 7
 8  func main2() {
 9    var p  Pair[Num, Num] = Pair[Num, Num]{ Num{1}, Num{2} }
10    var s1 string = p.format()
11    var s2 string = formatSome(p)
12  }
```

```
13  -- Field access for struct Pair
14  left (x, _) = x
15  right (_, x) = x
16
17  -- Method definition (line 4 in the FGG code)
18  format_Pair (toFormat_T, toFormat_U) this =
19    "(" ++ format_Format (toFormat_T (left this)) ++
20    "," ++ format_Format (toFormat_U (right this)) ++ ")"
21
22  -- Coercion Pair[T, U] <: Format (given T <: Format and U <: Format)
23  toFormat_Pair (toFormat_T,toFormat_U) p = (p,format_Pair (toFormat_T,toFormat_U))
24
25  -- Main function
26  main2 = let p = (1, 2)
27              s1 = format_Pair (toFormat_Num, toFormat_Num) p
28              s2 = formatSome (toFormat_Pair (toFormat_Num, toFormat_Num) p)
```

Fig. 2. String-formatting with generics (extending code from Figure 1)

bounds in the struct declaration. This is guaranteed in our case as we find $\texttt{Format} <: \texttt{Any}$. Importantly, the type bounds in line 4 imply the subtype relations (4) $\texttt{T} <: \texttt{Format}$ and (5) $\texttt{U} <: \texttt{Format}$. Thus, we can show that the method body type checks. For example, expression `this.left` is of type `T`. Based on (4), this expression is also of type `Format` and therefore the method call in line 5 `this.left.format()` type checks.

We consider type checking the main function. Instances for generic type variables must always be explicitly supplied. Hence, when constructing a pair that holds number values, see line 9, we find `Pair[Num, Num]`.

Consider the method call `p.format()` in line 10. The receiver struct `Pair[T Format, U Format]` of the method definition in line 4 matches `p`'s type `Pair[Num, Num]` by replacing `T` and `U` by `Num`. The type bounds in the receiver type are satisfied as we know from above that (2) $\texttt{Num} <: \texttt{Format}$. Hence, the method call type checks.

By generalizing the above argument we find that

$$(6) \qquad \{\texttt{T} <: \texttt{Format}, \texttt{U} <: \texttt{Format}\} \vdash \texttt{Pair[T, U]} <: \texttt{Format}.$$

That is, under the assumptions $\texttt{T} <: \texttt{Format}$ and $\texttt{U} <: \texttt{Format}$ we can derive that $\texttt{Pair[T, U]} <: \texttt{Format}$. In particular, we find that $\texttt{Pair[Num, Num]} <: \texttt{Format}$. Hence, the function call `formatSome(p)` in line 11 type checks.

Extending our type-directed translation scheme to deal with generics turns out to be fairly straightforward.

Bounded type parameter. A bounded type parameter `T Ifce` where `T` is a type variable and `Ifce` is an interface type becomes a coercion parameter $\texttt{toIfce}_{\texttt{T}}$. At instantiation sites, coercions need to be inserted.

The lower part of Figure 2 shows the translated program. Starting at line 18 we find the translation of the definition of method `format` for pairs. Each bounded type parameter `T Format` and `U Format` is turned into a coercion parameter $\texttt{toFormat}_{\texttt{T}}$ and $\texttt{toFormat}_{\texttt{U}}$. In the target, we use a curried function definition where coercion parameters are collected in a tuple.

A method call of `format` needs to supply concrete instances for these coercion parameters. See line 27 which is the translation of calling `format` on receiver type `Pair[Num,Num]`. Hence, we must pass as the first argument the tuple of coercions $(\texttt{toFormat}_{\texttt{Num}}, \texttt{toFormat}_{\texttt{Num}})$ to $\texttt{format}_{\texttt{Pair}}$.

Subtype relation (6) implies the (parameterized) coercion $\texttt{toFormat}_{\texttt{Pair}}$ in line 23. Given coercions $\texttt{toFormat}_{\texttt{T}}$ and $\texttt{toFormat}_{\texttt{U}}$ we can transform a pair `p` into an interface value for `Format`, where the method dictionary consists of the partially applied translated method definition $\texttt{format}_{\texttt{Pair}}$.

We make use of $\texttt{toFormat}_{\texttt{Pair}}$ in the translation of the function call `formatSome(p)`, see line 28. Based on the specific coercion $\texttt{toFormat}_{\texttt{Num}}$, the call $\texttt{toFormat}_{\texttt{Pair}}$ transforms the pair value `p` into the interface value $(\texttt{p}, \texttt{format}_{\texttt{Pair}} (\texttt{toFormat}_{\texttt{Num}},\texttt{toFormat}_{\texttt{Num}}))$. Then, we call `formatSome` on this interface value.

2.4 Bounded type parameters of methods

There may be bounded type parameters local to methods. Consider Figure 3 where we further extend our running example. Starting at line 1 we find a definition of method `formatSep` for pairs. This method takes an argument `s` that acts as a separator when formatting pairs. Argument `s` is of the generic type `S` constrained by the type bound `Format`. Type parameter `S` is local to the method and not connected to the receiver struct. Type arguments for `S` must also be explicitly specified in the program text, see method calls in lines 8 and 13.

In the translation, bounded type parameters of methods simply become additional coercion parameters. Consider the translation of `formatSep` defined on pairs starting at line 21. The translated method definition first expects the coercion parameters $(\texttt{toFormat}_{\texttt{T}}, \texttt{toFormat}_{\texttt{U}})$ that result from the bounded type parameters `T Format` and `U Format` of the receiver. Then, we find the receiver argument `this` followed by the coercion parameter $\texttt{toFormat}_{\texttt{S}}$ resulting from `S Format`, and finally the method argument `s`. The translation of the method body follows the scheme we have seen so far, see lines 22-24. When calling method `formatSep` on a pair we need to provide the appropriate coercions, see line 37.

From the method definition of `formatSep` for pairs and from the definition of interface `FormatSep`, we find that the following subtype relation holds:

$$(7) \quad \{\texttt{T} <: \texttt{Format}, \texttt{U} <: \texttt{Format}\} \vdash \texttt{Pair[T, U]} <: \texttt{FormatSep}.$$

```
1  func (this Pair[T Format, U Format]) formatSep[S Format](s S) string {
2    return this.left.format() + s.format() + this.right.format()
3  }
4
5  type FormatSep interface { formatSep[S Format](s S) string }
6
7  func formatSepSome(x FormatSep, s Format) string {
8      return x.formatSep[Format](s)
9  }
10
11 func main3 () {
12   var p Pair[Num, Num] = Pair[Num, Num]{ Num{1}, Num{2} }
13   var s1 string = p.formatSep[Num](Num{3})  // result: 132
14   var s2 string = formatSepSome(p,Num{4})  // result: 142
15 }
```

```
16 -- Method call on interface
17 -- call formatSep on receiver of type FormatSep
18 formatSep_FormatSep (x, f) = f x
19
20 -- Method definition (line 1 in the FGG code)
21 formatSep_Pair (toFormat_T, toFormat_U) this toFormat_S s =
22   format_Format (toFormat_T (left this)) ++
23   format_Format (toFormat_S s) ++
24   format_Format (toFormat_U (right this))
25
26 -- Coercions
27 toFormat_Format x = x   -- Format <: Format
28 toFormatSep_Pair (toFormat_T, toFormat_U) p =
29   -- Pair[T, U] <: FormatSep (given T <: Format and U <: Format)
30   (p, formatSep_Pair (toFormat_T, toFormat_U))
31
32 -- Function definitions (lines 7 and 11 in the FGG code)
33 formatSepSome (x, s) = (formatSep_FormatSep x) toFormat_Format s
34
35 main3 =
36   let p = (1,2)
37       s1 = formatSep_Pair (toFormat_Num, toFormat_Num) p toFormat_Num 3
38       s2 = formatSepSome
39              (toFormatSep_Pair (toFormat_Num, toFormat_Num) p,
40               toFormat_Num 4)
```

Fig. 3. Bounded type parameters of methods (extending code from Figure 2)

Subtype relation (7) implies the coercion $\texttt{toFormatSep}_{\texttt{Pair}}$ in line 28. Thus, the function call of `formatSepSome` from line 14 translates to the target code starting in line 38.

The point to note is that a coercion parameter corresponding to a bounded type parameter of a method is not part of the dictionary; it is only supplied at the call site of the method. Consider the call `x.formatSep[Format](s)` in line 8. In the translation (line 33), we first partially apply the respective dictionary entry on the receiver. This is done via the target expression $(\texttt{formatSep}_{\texttt{FormatSep}}\ \texttt{x})$. Type `Format` is a valid instantiation for type parameter `S` of `formatSep` because `Format` <: `Format` in FGG^{-}. In the translation, this corresponds to the (identity) coercion $\texttt{toFormat}_{\texttt{Format}}$. Hence, we supply the remaining arguments $\texttt{toFormat}_{\texttt{Format}}$ and `s`.

2.5 *Bounded type parameters of structs and interfaces*

Structs and interfaces may also carry bounded type parameters. In FGG^{-} and FGG, these type parameters do not have a meaning at runtime as their purpose is only to rule out more

```
 1  type FPair[T Format, U Format] struct { left T; right U }
 2  type Factory[T Format] interface { create() FPair[T, T] }
 3
 4  type MyFactory struct {}
 5  func (this MyFactory) create() FPair[Num, Num] {
 6    return FPair[Num, Num]{Num{1}, Num{2}}
 7  }
 8
 9  func doWork[T Format](factory Factory[T]) string {
10    var p FPair[T, T] = factory.create()
11    var t T = p.left
12    return t.format()
13  }
14
15  func main4() {
16    var s = doWork[Num](MyFactory{})
17  }
```

```
16  -- Field access for struct FPair
17  left_FPair (x, _) = x
18  right_FPair (_, x) = x
19
20  -- Method call on interface
21  create_Factory (x, f) = f x -- call create on receiver of type Factory
22
23  -- Method definition (line 5 in the source program)
24  create_MyFactory this = (1, 2)
25
26  -- Coercion
27  toFactory_MyFactory x = (x, create_MyFactory) -- MyFactory <: Factory[Num]
28
29  -- Function definition (line 9 in the source program)
30  doWork toFormat_T factory =
31    let p = create_Factory factory
32        t = left_FPair p
33    in format_Format (toFormat_T t)
34
35  main4 = doWork toFormat_Num (toFactory_MyFactory ())
```

Fig. 4. Bounded type parameters of structs and interfaces (extending code from Figure 1)

programs statically. Hence, in our translation approach they do not translate into additional dictionary parameters or coercions.

Let us explain with the example in Figure 4. Struct `FPair` (short for "formatted pairs") requires the type bound `Format` on its type parameters. The generic interface `Factory` defines a factory method returning formatted pairs. It requires a type bound `T Format` for the type `FPair[T, T]` in its method signature to be well-formed. The need for this type bound arises because FGG's type system does not allow to conclude from just an occurrence of `FPair[T, T]` that `T` is already a subtype of `Format`.

Struct `MyFactory` defines a concrete factory implementation for `FPair[Num, Num]`, function `doWork` accepts a generic `Factory[T]` for arbitrary `T`. Again, `doWork` requires a type bound `T Format` for type `Factory[T]` to be well-formed. The main function may then call `doWork` with a `MyFactory` value because `MyFactory` is a subtype of `Factory[Num]`.

The translated code (lower part of Figure 4) demonstrates that bounded type parameters of structs and interfaces have no representation at runtime, so the translation effectively ignores them. A struct value is still just a tuple with the fields of the struct (lines 17, 18), and an interface value just combines a struct value with a method dictionary (e.g.

line 27). Only bounded type parameters of receiver structs (Figure 2), methods (Figure 3) and functions (Figure 4) lead to additional coercion parameters.

An important point to note is that there is a difference between generic interfaces and interfaces with generic methods. Interface `Factory[T]` in Figure 4 is generic in `T`, a subtype of `Factory[U]` must provide an implementation of the `create` method for some fixed type `U`. In contrast, interface `FormatSep` from Figure 3 is not generic but contains a method `formatSep` that is generic in `S`. A subtype of `FormatSep` must provide an implementation of `formatSep` that is also generic in `S`.

2.6 Outlook

Next, Section 3 formalizes FGG$^-$ following the description by Griesemer et al. (2020). Then, we give the details of our type-directed translation scheme in Section 4 and establish that the meaning of FGG$^-$ programs is preserved in Section 5.

3 Featherweight Generic Go$^-$

Featherweight Go (FG, Griesemer et al., 2020) is a small subset of the full Go language (2022) supporting only essential features such as structs, interfaces, method overloading and structural subtyping. In the same article, the authors add generics to FG with the goal to scale the design to full Go. The resulting calculus is called Featherweight Generic Go (FGG). Since version 1.18, full Go includes generics as well, but with limited expressivity compared to the FGG proposal (see Section 7.1). For the translation presented in this article, we stick to the original FGG language with minor differences in presentation but excluding dynamic type assertions. We refer to this language as FGG$^-$.

The next two subsections introduce the syntax and the dynamic semantics of FGG$^-$. We defer the definition of its static semantics until Section 4.2, where we specify it as part of the type-directed dictionary-passing translation.

3.1 Syntax

Figure 5 introduces the syntax of FGG$^-$. We assume several countably infinite, pairwise disjoint sets for names, ranged over by $\mathcal{N}$ with some subscript (upper part of the figure). Meta variables t_S and u_S denote struct names, t_I and u_I interface names, α and β type variables, f field names, m method names, and x, y denote names for variables in expressions. Overbar notation $\overline{\mathfrak{s}}^n$ is a shorthand for the sequence $\mathfrak{s}_1 \ldots \mathfrak{s}_n$ where $\mathfrak{s}$ is some syntactic construct. In some places, commas separate the sequence items. If irrelevant, we omit the n and simply write $\overline{\mathfrak{s}}$. Using the index variable i under an overbar marks the parts that vary from sequence item to sequence item; for example, $\overline{\mathfrak{s}'\ \mathfrak{s}_i}^n$ abbreviates $\mathfrak{s}'\ \mathfrak{s}_1 \ldots \mathfrak{s}'\ \mathfrak{s}_n$ and $\overline{\mathfrak{s}_j}^q$ abbreviates $\mathfrak{s}_{j1}\ \ldots\ \mathfrak{s}_{jq}$.

The middle part of Figure 5 shows the syntax of types in FGG$^-$. A type name t, u is either a struct or interface name. Types τ, σ include types variables α and instantiated types $t[\overline{\tau}]$. For non-generic structs or interfaces, we often write just t instead of $t[]$. Struct types τ_S, σ_S and interface types τ_I, σ_I denote syntactic subsets of the full type syntax.

Struct name	$t_S, u_S \in \mathcal{N}_{struct}$	Field name	$f \in \mathcal{N}_{field}$
Interface name	$t_I, u_I \in \mathcal{N}_{iface}$	Method name	$m \in \mathcal{N}_{method}$
Type variable name	$\alpha, \beta \in \mathcal{N}_{tyvar}$	Variable name	$x, y \in \mathcal{N}_{var}$
Type name	$t, u ::= t_S \mid t_I$	Struct type	$\tau_S, \sigma_S ::= t_S[\overline{\tau}]$
Type	$\tau, \sigma ::= \alpha \mid t[\overline{\tau}]$	Interface type	$\tau_I, \sigma_I ::= t_I[\overline{\tau}]$

Expression	$e, g ::= x \mid e.m[\overline{\tau}](\overline{e}) \mid \tau_S\{\overline{e}\} \mid e.f$
Method signature	$R ::= m[\overline{\alpha\ \tau_I}](\overline{x\ \tau})\ \tau$
Declaration	$D ::= \textbf{type}\ t_S[\overline{\alpha\ \tau_I}]\ \textbf{struct}\ \{\overline{f\ \tau}\}$
	$\mid \textbf{type}\ t_I[\overline{\alpha\ \tau_I}]\ \textbf{interface}\ \{\overline{R}\}$
	$\mid \textbf{func}\ (x\ t_S[\overline{\alpha\ \tau_I}])\ R\ \{\textbf{return}\ e\}$
Program	$P ::= \overline{D}\ \textbf{func}\ \mathsf{main}()\{\_ = e\}$

Fig. 5. Syntax of FGG$^-$

The lower part of Figure 5 defines the syntax of FGG$^-$ expressions, declarations, and programs. Expressions, ranged over by e and g, include variables x, method calls, struct literals, and field selections. A method call $e.m[\overline{\tau}](\overline{e})$ invokes method m on receiver e with type arguments $\overline{\tau}$ and arguments $\overline{e}$. If m does not take type arguments, we often write just $e.m(\overline{e})$. A struct literals $\tau_S\{\overline{e}^n\}$ creates an instance of a struct with n fields, the arguments $\overline{e}^n$ become the values of the fields in order of appearance in the struct definition. A field selection $e.f$ projects the value of some struct field f from expression e.

A method signature $R ::= m[\overline{\alpha\ \tau_I}](\overline{x\ \tau})\ \tau$ consists of a name m, bounded type parameters α_i with interface type τ_{Ii} as upper bounds, parameters x_i of type τ_i, and return type τ. It binds $\overline{\alpha}$ and $\overline{x}$. The scope of a type variable α_i is $\overline{\tau}$, τ, and all upper bounds $\overline{\tau_I}$, so FGG$^-$ supports F-bounded quantification (Canning et al., 1989). For non-generic methods, we often write just $m(\overline{x_i\ \tau_i})\ \tau$.

A declaration D comes in three forms: a struct **type** $t_S[\overline{\alpha\ \tau_I}]$ **struct** $\{\overline{f\ \tau}\}$ with fields f_i of type τ_i; an interface **type** $t_I[\overline{\alpha\ \tau_I}]$ **interface** $\{\overline{R}\}$ with method signatures $\overline{R}$; or a method **func** $(x\ t_S[\overline{\alpha\ \tau_I}])\ R$ $\{$**return** $e\}$ providing an implementation of method R for struct t_S. All three forms bind the type variables $\overline{\alpha}$, a method implementation additionally binds the receiver parameter x. The scope of a type variable α_i includes all upper bounds $\overline{\tau_I}$, the body of the declaration enclosed in $\{\ldots\}$, and for method declarations also the signature R. We omit the $[\overline{\alpha\ \tau_I}]$ part completely if $\overline{\alpha\ \tau_I}$ is empty. Finally, a program P consists of a sequence of declarations together with a main function. Method and function bodies only contain a single expression. We follow the usual convention and identify syntactic constructs up to renaming of bound variables or type variables.

The syntax of FGG$^-$ as presented here differs slightly from its original form (Griesemer et al., 2020). The original article encloses type parameters in parenthesis, an additional **type** keyword starts a list of type parameters. Here, we follow the syntax of full Go and use square brackets without any keyword. Further, the original article prepends **package main** to each program, something we omit for succinctness. Finally, we reduce the number of syntactic meta-variables to improve readability.

Value	$v, u, w ::= \tau_S\{\overline{v}\}$
Evaluation context	$\mathcal{E} ::= \square \mid \tau_S\{\overline{v}, \mathcal{E}, \overline{e}\} \mid \mathcal{E}.f \mid \mathcal{E}.m[\overline{\tau}](\overline{e}) \mid v.m[\overline{\tau}](\overline{v}, \mathcal{E}, \overline{e})$
Value substitution	$\theta ::= \langle \overline{x \mapsto v} \rangle$
Type substitution	$\eta ::= \langle \overline{\alpha \mapsto \tau} \rangle$

$\boxed{e \longrightarrow e}$ *Reductions*

FG-CONTEXT
$$\frac{e \longrightarrow e'}{\mathcal{E}[e] \longrightarrow \mathcal{E}[e']}$$

FG-FIELD
$$\frac{\mathbf{type}\ t_S[\overline{\alpha\ \tau_I}]\ \mathbf{struct}\ \{\overline{f\ \sigma}^n\} \in \overline{D}}{t_S[\overline{\tau}]\{\overline{v}^n\}.f_i \longrightarrow v_i}$$

FG-CALL
$$\frac{v = t_S[\overline{\tau}]\{\overline{u}\} \qquad \mathbf{func}\ (x\ t_S[\overline{\alpha\ \tau_I}])\ m[\overline{\alpha'\ \tau'_I}](\overline{x\ \sigma})\ \sigma\ \{\mathbf{return}\ e\} \in \overline{D}}{v.m[\overline{\tau'}](\overline{v}) \longrightarrow \langle x \mapsto v, \overline{x \mapsto v}\rangle \langle \overline{\alpha \mapsto \tau}, \overline{\alpha' \mapsto \tau'} \rangle e}$$

Fig. 6. Dynamic semantics of FGG$^-$

3.2 Dynamic Semantics

Figure 6 defines a call-by-value dynamic semantics for FGG$^-$ using a small-step reduction semantics with evaluation contexts. The definition is largely taken from Griesemer et al. (2020).

We use v, u, w to denote values, where a value is a struct literal with all fields being values. A call-by-value evaluation context $\mathcal{E}$ is an expression with a hole $\square$ such that the hole marks the point where the next evaluation step should happen. We write $\mathcal{E}[e]$ to denote the replacement of the hole in $\mathcal{E}$ with expression e. A value substitution θ is a finite mapping $\langle \overline{x \mapsto v} \rangle$ from variables to values, whereas a type substitution η is a finite mapping $\langle \overline{\alpha \mapsto \tau} \rangle$ from type variables to types. The (type) variables in the domain of a substitution must be distinct. Substitution application, written in prefix notation as θe or ηe or $\eta\tau$, is defined in the usual, capture-avoiding way. When combining two sequences, we implicitly assume that both sequences have the same length. For example, combining variables $\overline{x}$ and values $\overline{v}$ to a substitution $\langle \overline{x \mapsto v} \rangle$ implicitly assumes that there are as many variables as values.

The reduction relation $e \longrightarrow e'$ denotes that expression e reduces to expression e'. To avoid clutter, the sequence of declarations $\overline{D}$ of the underlying program is implicitly available in the rules defining this reduction relation. Rule FG-CONTEXT applies a reduction step inside an expression. Rule FG-FIELD reduces a field selection $t_S[\overline{\tau}]\{\overline{v}\}.f_i$ by extracting value v_i corresponding to field f_i from the struct literal. Rule FG-CALL reduces a method call $t_S[\overline{\tau}]\{\overline{u}\}.m[\overline{\tau'}](\overline{v})$. It retrieves a method definition for m and t_S and substitutes type arguments, receiver, and value arguments in the method body.

Reduction in FGG$^-$ is deterministic (see Lemma A.1.1 in Appendix A.1 for a formal proof), assuming the following three restrictions:

FGG-UNIQUE-STRUCTS Each struct t_S is defined at most once in the program.

FGG-DISTINCT-FIELDS Each struct definition $\mathbf{type}\ t_S[\overline{\alpha\ \tau_I}]\ \mathbf{struct}\ \{\overline{f\ \tau}\}$ has distinct field names $\overline{f}$.

FGG-UNIQUE-METHOD-DEFS Each method definition **func** $(x\ t_S[\overline{\alpha\ \tau_I}])\ m[\overline{\alpha'\ \tau'_I}](\overline{x\ \sigma})\ \sigma\ \{\textbf{return}\ e\}$ is uniquely identified by struct name t_S and method name m.

The first two restrictions ensures that the value for a field in rule FG-FIELD is unambiguous. The third restriction avoids multiple matching method definitions in rule FG-CALL.

4 Type-directed translation

This section defines a type-directed, dictionary-passing translation from FGG$^-$ to an untyped λ-calculus extended with recursive let-bindings, constructors and pattern matching. We first introduce the target language, then specify the translation itself, and last but not least give some examples. Formal properties of the translation are deferred until Section 5.

Variable	$X, Y \in \mathcal{V}_{Var}$	Pattern clause	$Cls ::= Pat \rightarrow E$
Constructor	$K \in \mathcal{V}_{Con}$	Pattern	$Pat ::= K\ \overline{X}$
Expression	$E, G ::= X \mid K \mid E\ E \mid \lambda X.E \mid \textbf{case}\ E\ \textbf{of}\ \overline{Cls}$	Program	$Prog ::= \textbf{let}\ \overline{X = V}\ \textbf{in}\ E$

Value	$V, U, W ::= K\ \overline{V} \mid \lambda X.E$
Evaluation context	$\mathcal{R} ::= \square \mid \textbf{case}\ \mathcal{R}\ \textbf{of}\ \overline{Cls} \mid \mathcal{R}\ E \mid V\ \mathcal{R}$
Substitution	$\rho, \mu ::= \langle \overline{X \mapsto V} \rangle$

$\boxed{E \longrightarrow_\mu E'}$ *TL expression reductions*

TL-CONTEXT
$$\frac{E \longrightarrow_\mu E'}{\mathcal{R}[E] \longrightarrow_\mu \mathcal{R}[E']}$$

TL-LAMBDA
$$(\lambda X.E)\ V \longrightarrow_\mu \langle X \mapsto V \rangle E$$

TL-CASE
$$\frac{K\ \overline{X}^n \rightarrow E \in \overline{Cls}}{\textbf{case}\ K\ \overline{V}^n\ \textbf{of}\ \overline{Cls} \longrightarrow_\mu \langle \overline{X \mapsto V}^n \rangle E}$$

TL-METHOD
$$X \longrightarrow_\mu \mu(X)$$

$\boxed{Prog \longrightarrow Prog'}$ *TL reductions*

TL-PROG
$$\frac{\mu = \langle \overline{X \mapsto V} \rangle \qquad E \longrightarrow_\mu E'}{\textbf{let}\ \overline{X = V}\ \textbf{in}\ E \longrightarrow \textbf{let}\ \overline{X = V}\ \textbf{in}\ E'}$$

Fig. 7. Target language (TL)

4.1 Target Language

Figure 7 defines the syntax and the call-by-value dynamic semantics of the target language (TL). We use uppercase letters for constructs of the target language. Variables X, Y and constructors K are drawn from countably infinite, pairwise disjoint sets $\mathcal{V}_{Var}$ and $\mathcal{V}_{Con}$, respectively. Expressions, ranged over by E and G, include variables X, constructors K, function applications $E\,E'$, λ-abstractions $\lambda X.E$, and pattern matching via case-expressions $\textsf{\textbf{case}}\ E\ \textsf{\textbf{of}}\ \overline{Pat \longrightarrow E}$. Patterns Pat have the form $K\,\overline{X}$, they do not nest. We assume that all constructors in $\overline{Pat}$ are distinct. To avoid some parentheses, we use the conventions that application binds to the left and that the body of a λ extends to the right as far as possible.

A program $\textsf{\textbf{let}}\ \overline{X = V}\ \textsf{\textbf{in}}\ E$ consists of a sequence of (mutually recursive) definitions and a (main) expression, where we assume that the variables $\overline{X}$ are distinct. In the translation from FGG$^-$, the values $\overline{V}$ are always functions resulting as translations of FGG$^-$ methods. We identify expressions, pattern clauses and programs up to renaming of bound variables. Variables are bound by λ expressions, patterns, and let-bindings of programs.

Some syntactic sugar simplifies the construction of patterns, expressions and programs. (a) We use nested patterns to abbreviate nested case-expressions. (b) We assume data constructors for tuples up to some fixed but arbitrary size. The syntax $(\overline{E}^n)$ constructs an n-tuple when used as an expression, and $(\overline{Pat}^n)$ deconstructs it when used in a pattern context. (c) We use patterns in λ-expressions; that is, the notation $\lambda Pat.E$ stands for $\lambda X.\textsf{\textbf{case}}\ X\ \textsf{\textbf{of}}\ Pat \longrightarrow E$ where X is fresh.

Target values V, U, W are either λ-expressions or constructors applied to values. A constructor value $K\,\overline{V}^n$ is short for $(\dots(K\,V_1)\dots)\,V_n$. A call-by-value evaluation context $\mathcal{R}$ is an expression with a hole $\square$ such that the hole marks the point where the next evaluation step should happen. We write $\mathcal{R}[E]$ to denote the replacement of the hole in $\mathcal{R}$ with expression E.

A substitution ρ, μ is a finite mapping $\langle\overline{X \mapsto V}\rangle$ from variables to values. The variables $\overline{X}$ in the domain must be distinct. Substitution application, written in prefix notation ρE, is defined in the usual, capture-avoiding way. We use two different meta variables μ and ρ for substitutions in the target language with the convention that the domain of μ contains only top-level variables bound by **let**. As top-level variables result from translating FGG$^-$ methods, we sometimes call μ a *method substitution*.

The reduction semantics for the target language is defined by two relations: $E \longrightarrow_\mu E'$ reduces expression E to E' under method substitution μ, and $Prog \longrightarrow Prog'$ reduces $Prog$ to $Prog'$. The definition of the latter simply forms a method substitution μ from the top-level bindings of $Prog$ and then reduces the main expression of $Prog$ under μ (rule TL-PROG). We defer the substitution of top-level–bound variables because they might be recursive.

The definition of the reduction relation for expressions extends over four rules. Rule TL-CONTEXT uses evaluation context $\mathcal{R}$ to reduce inside an expression, rule TL-LAMBDA reduces function application in the usual way. Pattern matching in rule TL-CASE assumes that the scrutinee is a constructor value $K\,\overline{V}^n$; the lookup of a pattern clause matching K yields at most one result as we assume that clauses have distinct constructors. During a sequence of reduction steps, a variable bound by **let** at the top-level might become a redex, as only λ-bound variables are substituted right away. Thus, rule TL-METHOD finds the value for the variable in the method substitution μ.

4.2 Translation

Before we dive into the technical details, we summarize our translation strategy.

Struct. An FGG$^-$ value of some struct type is represented in the TL as a *struct value*; that is, a tuple $(\overline{V}^n)$ where n is the number of fields and V_i represents the i-th field of the struct.

Interface. An FGG$^-$ value of some interface type is represented in the TL as an *interface value*; that is a pair $(V, \mathcal{D})$, where V is a struct value realizing the interface and $\mathcal{D}$ is a *dictionary*.

Dictionary. A dictionary $\mathcal{D}$ for an interface with methods $\overline{R}^n$ is a tuple $(\overline{V}^n)$ such that V_i is a *dictionary entry* for method R_i.

Dictionary entry. A dictionary entry for a method with signature $R = m[\overline{\alpha\ \tau_I}](\overline{x\ \sigma})\sigma$ is a function accepting a triple: (1) receiver, (2) tuple with coercions corresponding to the bounded type parameters $\overline{\alpha\ \tau_I}$ of the method, (3) tuple for parameters $\overline{x}$.

Coercion. A structural subtype relation $\tau <: \sigma$ implies a *coercion function* to transform the target representation of an FGG$^-$ value at type τ into a representation at type σ.

Bounded type parameter. A bounded type parameter $\alpha\ \tau_I$ becomes a coercion parameter X_α transforming the type supplied for α to its bound τ_I. At instantiation sites, coercions need to be inserted.

Method declaration. A method declaration **func** $(x\ t_S[\overline{\alpha\ \tau_I}])\ m[\overline{\alpha'\ \tau'_I}](\overline{x\ \sigma})\ \sigma\ \{$**return** $e\}$ is represented as a top-level function X_{m,t_S} accepting a quadruple: (1) tuple with coercions corresponding to the bounded type parameters $\overline{\alpha\ \tau_I}$ of the receiver, (2) receiver x, (3) tuple with coercions corresponding to bounded type parameters $\overline{\alpha'\ \tau'_I}$ of the method, (4) tuple for parameters $\overline{x}$.

In essence, the above is a more detailed description of the translation scheme motivated in Section 2. The only difference is that dictionary entries and translations of methods are now represented as uncurried functions. For example, instead of the curried representation in Figure 3

```
formatSepPair (toFormatT, toFormatU) this toFormatS x = ...

toFormatSepPair (toFormatT, toFormatU) p =
   (p, formatSepPair (toFormatT, toFormatU))
```

our actual translation scheme uses uncurried functions, as in the following code:

```
-- translation of method
formatSepPair ((toFormatT, toFormatU), this, toFormatS, x) = ...

toFormatSepPair (toFormatT, toFormatU) p =
   (p, \(this,locals,arg) ->     -- dictionary entry
          formatSepPair ((toFormatT, toFormatU),locals,arg))
```

Using an uncurried representation instead of a curried representation is just a matter taste. As we have carried out the semantic equivalence proof initially based on the uncurried representation, we stick to it from now on.

4.2.1 Conventions and Notations

The translation relies on three total, injective functions with pairwise disjoint ranges for mapping FGG$^-$ names to TL variables. The first function $\mathcal{N}_{var} \to \mathcal{V}_{Var}$ translates a FGG$^-$ variable x to a TL variable X. To avoid clutter, we do not spell out the translation function explicitly but use the abbreviation that a lowercase x always translates into its uppercase counterpart X. The second function $\mathcal{N}_{tyvar} \to \mathcal{V}_{Var}$ translates an FGG$^-$ type variable α into a TL variable, abbreviated X_α. The third function $\mathcal{N}_{method} \times \mathcal{N}_{struct} \to \mathcal{V}_{Var}$ gives us the TL variable X_{m,t_S} representing the translation of a method m for struct t_S. Here is a summary of the shorthand notations for name translation functions, where $\mathsf{methodName}(R)$ denotes the name part of method signature R.

$$x \rightsquigarrow X \qquad\qquad \alpha \rightsquigarrow X_\alpha \qquad\qquad \frac{m = \mathsf{methodName}(R)}{\textbf{func } (x\ t_S[\overline{\alpha\ \tau_I}])\ R\ \{\textbf{return } e\} \rightsquigarrow X_{m,t_S}}$$

The notation for translating names slightly differs from the approach used in the examples of Section 2. For instance, the coercion $\texttt{toFormat}_{\mathtt{T}}$ from Figure 3 is now named $X_{\mathtt{T}}$ and method $\texttt{formatSep}_{\texttt{Pair}}$ becomes $X_{\texttt{formatSep},\texttt{Pair}}$. The notation of the formal translation stresses that $X_{\mathtt{T}}$ and $X_{\texttt{formatSep},\texttt{Pair}}$ are variables of the target language.

An FGG$^-$ type environment Δ is a mapping $\{\overline{\alpha : \tau_I}\}$ from type variables α_i to their upper bounds τ_{Ii}. An FGG$^-$ value environment Γ is a mapping $\{\overline{x : \tau}\}$ from FGG$^-$ variables x_i to their types τ_i. An environment may contain at most one binding for a type variable or variable. We write $\emptyset$ for the empty environment, $\mathsf{dom}(\cdot)$ for the domain of an environment, and $\cup$ for the disjoint union of two environments. The notation $\mathsf{distinct}(\overline{\mathfrak{s}})$ asserts that $\overline{\mathfrak{s}}$ is a sequence of disjoint items. We let $[n]$ denote the set $\{1, \ldots, n\}$.

In the following, we assume that the declarations $\overline{D}$ of the FGG$^-$ program being translated are implicitly available in all rules. This avoids the need for threading the declarations through all translation rules.

4.2.2 Auxiliary Judgments

Figure 8 defines some auxiliary judgments. The judgment $\Delta \vdash_{\mathsf{subst}} \overline{\alpha\ \tau_I} \mapsto \overline{\sigma} : \eta \rightsquigarrow V$, defined by rule TYPE-INST-CHECKED, constructs a type substitution $\eta = \langle \overline{\alpha \mapsto \sigma} \rangle$ and checks that the $\overline{\sigma}$ conform to their upper bounds $\overline{\tau_I}$ under type environment Δ. In the tuple $(\overline{V}^n)$ of λ-abstractions each V_i coerces the actual type argument to its upper bound. The relevant premise for checking upper bounds is $\Delta \vdash_{\mathsf{coerce}} \sigma_i <: \eta\tau_{Ii} \rightsquigarrow V_i$, which asserts that σ_i is a structural subtype of $\eta\tau_{Ii}$ giving raise to a coercion function V_i. The judgment will be defined and explained in the next subsection.

The lower part of Figure 8 defines two judgments for looking up methods defined for a struct or interface type. Judgment $\langle R, V \rangle \in \mathsf{methods}(\Delta, \tau_S)$ states that method signature R is available for struct type τ_S under type environment Δ, see rule METHODS-STRUCT. The value V is a tuple of coercion functions resulting from checking the bounds of the receiver's type parameters. Judgment $\mathsf{methods}(\tau_I) = \{\overline{R}\}$ states that the set of method signatures available for interface type τ_I is $\{\overline{R}\}$, see rule METHODS-IFACE. As stated before, this rule forms the substitution $\langle \overline{\alpha \mapsto \sigma} \rangle$ by implicitly assuming that $\overline{\alpha}$ and $\overline{\sigma}$ have the same length.

$$\boxed{\Delta \vdash_{\mathsf{subst}} \overline{\alpha\ \tau_I} \mapsto \overline{\sigma} : \eta \rightsquigarrow V} \qquad \textit{Instantiation of bounded type parameters}$$

TYPE-INST-CHECKED

$$\frac{\eta = \langle \overline{\alpha \mapsto \sigma}^n \rangle \qquad \Delta \vdash_{\mathsf{coerce}} \sigma_i <: \eta \tau_{Ii} \rightsquigarrow V_i \qquad (\forall i \in [n])}{\Delta \vdash_{\mathsf{subst}} \overline{\alpha\ \tau_I}^n \mapsto \overline{\sigma}^n : \eta \rightsquigarrow (\overline{V}^n)}$$

$$\boxed{\langle R, V \rangle \in \mathsf{methods}(\Delta, \tau_S) \qquad \mathsf{methods}(\tau_I) = \{\overline{R}\}} \qquad \textit{Method access}$$

METHODS-STRUCT

$$\frac{\textbf{func}\ (x\ t_S[\overline{\alpha\ \tau_I}])\ R\ \{\textbf{return}\ e\} \in \overline{D} \qquad \Delta \vdash_{\mathsf{subst}} \overline{\alpha\ \tau_I} \mapsto \overline{\sigma} : \eta \rightsquigarrow V}{\langle \eta R, V \rangle \in \mathsf{methods}(\Delta, t_S[\overline{\sigma}])}$$

METHODS-IFACE

$$\frac{\textbf{type}\ t_I[\overline{\alpha\ \tau_I}]\ \textbf{interface}\ \{\overline{R}\} \in \overline{D} \qquad \eta = \langle \overline{\alpha \mapsto \sigma} \rangle}{\mathsf{methods}(t_I[\overline{\sigma}]) = \{\eta \overline{R}\}}$$

Fig. 8. Auxiliary judgments for the translation

$$\boxed{\Delta \vdash_{\mathsf{coerce}} \tau <: \sigma \rightsquigarrow V} \qquad \textit{Translation of structural subtyping}$$

COERCE-TYVAR

$$\frac{Y \text{ fresh} \qquad (\alpha : \sigma_I) \in \Delta \qquad \Delta \vdash_{\mathsf{coerce}} \sigma_I <: \tau \rightsquigarrow V}{\Delta \vdash_{\mathsf{coerce}} \alpha <: \tau \rightsquigarrow \lambda Y . V\ (X_\alpha\ Y)}$$

COERCE-STRUCT-IFACE

$$\frac{\begin{array}{c} X, \overline{Y}^3 \text{ fresh} \\ \textbf{type}\ t_I[\overline{\alpha\ \tau_I}]\ \textbf{interface}\ \{\overline{R}^n\} \in \overline{D} \qquad \eta = \langle \overline{\alpha \mapsto \tau} \rangle \qquad \langle \eta R_i, V_i \rangle \in \mathsf{methods}(\Delta, \tau_S) \\ m_i = \mathsf{methodName}(R_i) \qquad U_i = \lambda(\overline{Y}^3) . X_{m_i, t_S}\ (V_i, \overline{Y}^3) \qquad (\forall i \in [n]) \end{array}}{\Delta \vdash_{\mathsf{coerce}} \tau_S <: t_I[\overline{\tau}] \rightsquigarrow \lambda X . (X, (\overline{U}^n))}$$

COERCE-IFACE-IFACE

$$\frac{\begin{array}{c} Y, \overline{X}^n \text{ fresh} \qquad \pi : [q] \to [n] \text{ total} \qquad \textbf{type}\ t_I[\overline{\alpha\ \tau_I}]\ \textbf{interface}\ \{\overline{R}^n\} \in \overline{D} \\ \textbf{type}\ u_I[\overline{\beta\ \sigma_I}]\ \textbf{interface}\ \{\overline{R'}^q\} \in \overline{D} \qquad \langle \overline{\beta \mapsto \sigma} \rangle R'_i = \langle \overline{\alpha \mapsto \tau} \rangle R_{\pi(i)} \qquad (\forall i \in [q]) \end{array}}{\Delta \vdash_{\mathsf{coerce}} t_I[\overline{\tau}] <: u_I[\overline{\sigma}] \rightsquigarrow \lambda(Y, (\overline{X}^n)) . (Y, (X_{\pi(1)}, \ldots, X_{\pi(q)}))}$$

Fig. 9. Translation of structural subtyping

4.2.3 Translation of Structural Subtyping

Figure 9 defines the relation $\Delta \vdash_{\mathsf{coerce}} \tau <: \sigma \rightsquigarrow V$ for asserting that τ is a structural subtype of σ, yielding a coercion function V to convert the target representations of τ to σ.

Rule COERCE-TYVAR covers the case of a type variable α. The premise states that type bound $(\alpha : \sigma_I)$ exists in the environment. By convention, X_α is the name of the corresponding coercion function. We further find that $\Delta \vdash_{\mathsf{coerce}} \sigma_I <: \sigma \rightsquigarrow V$. Hence, we obtain the coercion function for $\alpha <: \sigma$ by composition of coercion functions V and X_α.

Rule COERCE-STRUCT-IFACE covers structs. The premise $\langle \eta R_i, V_i \rangle \in \mathsf{methods}(\Delta, \tau_S)$ asserts that each method with name $\mathsf{methodName}(R_i)$ of interface t_I is defined for τ_S. Value V_i

$\boxed{\langle\Delta,\Gamma\rangle \vdash_{\mathsf{exp}} e : \tau \rightsquigarrow E}$ *Translating expressions*

VAR

$$\frac{(x:\tau)\in\Gamma}{\langle\Delta,\Gamma\rangle \vdash_{\mathsf{exp}} x:\tau \rightsquigarrow X}$$

STRUCT

$$\frac{\begin{array}{c}\Delta \vdash_{\mathsf{ok}} t_S[\overline{\tau}] \qquad \textbf{type } t_S[\overline{\alpha\ \tau_I}]\ \textbf{struct}\ \{\overline{f\ \sigma}^n\} \in \overline{D} \\ \langle\Delta,\Gamma\rangle \vdash_{\mathsf{exp}} e_i : \langle\overline{\alpha\mapsto\tau}\rangle\sigma_i \rightsquigarrow E_i \quad (\forall\, i\in[n])\end{array}}{\langle\Delta,\Gamma\rangle \vdash_{\mathsf{exp}} t_S[\overline{\tau}]\{\overline{e}^n\} : t_S[\overline{\tau}] \rightsquigarrow (\overline{E}^n)}$$

ACCESS

$$\frac{\langle\Delta,\Gamma\rangle \vdash_{\mathsf{exp}} e : t_S[\overline{\tau}] \rightsquigarrow E \qquad \textbf{type } t_S[\overline{\alpha\ \tau_I}]\ \textbf{struct}\ \{\overline{f\ \sigma}^n\} \in \overline{D}}{\langle\Delta,\Gamma\rangle \vdash_{\mathsf{exp}} e.f_i : \langle\overline{\alpha\mapsto\tau}\rangle\sigma_i \rightsquigarrow \textbf{case } E \textbf{ of } (\overline{X}^n) \to X_i}$$

CALL-STRUCT

$$\frac{\begin{array}{c}\langle\Delta,\Gamma\rangle \vdash_{\mathsf{exp}} e : t_S[\overline{\tau}] \rightsquigarrow E \qquad \langle m[\overline{\alpha'\ \tau'_I}](\overline{x\ \sigma}^n)\sigma, V\rangle \in \mathsf{methods}(\Delta, t_S[\overline{\tau}]) \\ \Delta \vdash_{\mathsf{subst}} \overline{\alpha'\ \tau'_I} \mapsto \overline{\tau'} : \eta \rightsquigarrow V' \qquad \langle\Delta,\Gamma\rangle \vdash_{\mathsf{exp}} e_i : \eta\sigma_i \rightsquigarrow E_i \quad (\forall\, i\in[n])\end{array}}{\langle\Delta,\Gamma\rangle \vdash_{\mathsf{exp}} e.m[\overline{\tau'}](\overline{e}^n) : \eta\sigma \rightsquigarrow X_{m,t_S}\ (V, E, V', (\overline{E}^n))}$$

CALL-IFACE

$$\frac{\begin{array}{lll} & \langle\Delta,\Gamma\rangle \vdash_{\mathsf{exp}} e : \tau_I \rightsquigarrow E & \\ \mathsf{methods}(\tau_I) = \overline{R}^q & R_j = m[\overline{\alpha'\ \tau'_I}](\overline{x\ \sigma}^n)\sigma \quad (\text{for some } j\in[q]) & \\ \Delta \vdash_{\mathsf{subst}} \overline{\alpha'\ \tau'_I} \mapsto \overline{\tau'} : \eta \rightsquigarrow V' & \langle\Delta,\Gamma\rangle \vdash_{\mathsf{exp}} e_i : \eta\sigma_i \rightsquigarrow E_i \quad (\forall\, i\in[n]) & Y, \overline{X}^q \text{ fresh}\end{array}}{\langle\Delta,\Gamma\rangle \vdash_{\mathsf{exp}} e.m[\overline{\tau'}](\overline{e}^n) : \eta\sigma \rightsquigarrow \textbf{case } E \textbf{ of } (Y, (\overline{X}^q)) \to X_j\,(Y, V', (\overline{E}^n))}$$

SUB

$$\frac{\langle\Delta,\Gamma\rangle \vdash_{\mathsf{exp}} e : \tau \rightsquigarrow E \qquad \Delta \vdash_{\mathsf{coerce}} \tau <: \sigma \rightsquigarrow V}{\langle\Delta,\Gamma\rangle \vdash_{\mathsf{exp}} e : \sigma \rightsquigarrow V\ E}$$

Fig. 10. Translation of expressions

is a tuple with coercion parameters corresponding to the bounds of the receiver's type parameters. Thus, $U_i = \lambda(\overline{Y}^3)\,.\,X_{m_i,t_S}\ (V_i, \overline{Y}^3)$ is the dictionary entry for the i-th method: a function accepting receiver Y_1, coercion parameters Y_2 corresponding to bounded type parameters of the method, and the argument tuple Y_3. As written earlier, dictionary entries and top-level functions X_{m_i,t_S} are uncurried. Thus, we need to deconstruct the argument triple $(\overline{Y}^3)$ and construct a new quadruple $(V, \overline{Y}^3)$ for calling X_{m_i,t_S}.

Rule COERCE-IFACE-IFACE covers structural subtyping between interface types $t_I[\overline{\tau}]$ and $u_I[\overline{\sigma}]$. In this case, t_I must declare all methods of u_I, so we can build a dictionary for u_I from the methods in the dictionary for t_I. Thus, the premise of the rule requires the total function π to be chosen in such a way that the i-th method of u_I has the same signature as the $\pi(i)$-th method of t_I. The translation uses pattern matching to deconstruct the dictionary of t_I as $(\overline{X}^n)$. Then the i-th method in the dictionary of u_I is $X_{\pi(i)}$, so we construct the wanted dictionary as $(X_{\pi(1)}, \ldots, X_{\pi(q)})$.

4.2.4 Translation of Expressions

Figure 10 defines the typing and translation relation for expressions. The judgment $\langle\Delta,\Gamma\rangle \vdash_{\mathsf{exp}} e : \tau \rightsquigarrow E$ states that under type environment Δ and value environment Γ the FGG$^-$ expression e has type τ and translates to TL expression E.

Rule VAR retrieves the type of FGG$^-$ variable x from the environment and translates x to its TL counterpart X. The context makes variable X available, see the translation of method definitions in Section 4.2.6. Rule STRUCT type checks and translates a struct literal $t_S[\overline{\tau}](\overline{e})$. Premise $\Delta \vdash_{\text{ok}} t_S[\overline{\tau}]$ checks that type $t_S[\overline{\tau}]$ is well-formed; the definition of the judgment $\Delta \vdash_{\text{ok}} \tau$ is given in Figure 11 and will be explained in the next subsection. Each argument e_i translates to E_i, so the result is $(\overline{E}^n)$. Rule ACCESS deals with field access $e.f_i$, where expression e must have struct type $t_S[\overline{\tau}]$ such that t_S defines field f_i. Thus, e translates to a tuple E, from which we extract the i-th component via pattern matching.

Rule CALL-STRUCT handles a method call $e.m[\overline{\tau'}](\overline{e})$, where receiver e has struct type $t_S[\overline{\tau}]$ and translates to E. The V in the premise corresponds to a tuple of coercion functions that result from checking the bounds of the receiver's type parameters, whereas V' is a tuple of coercion functions for the bounds of the type parameters of the method. Argument e_i translates to E_i. According to our translation strategy, a method declaration for m and t_S is represented as a top-level function X_{m,t_S} accepting a quadruple: coercions for the receiver's type parameters, receiver, coercions for the bounded type parameters local to the method, and method arguments. Thus, the result of the translation is $X_{m,t_S}\ (V, E, V', (\overline{E}))$.

Rule CALL-IFACE handles a method call $e.m[\overline{\tau'}](\overline{e})$, where receiver e has interface type τ_I and translates to E. Similar to CALL-STRUCT, V' is a tuple of coercion functions that result from checking the bounds of the type parameters local to the method. Expressions E_i are the translation of the arguments e_i. Following our translation strategy, receiver E is a pair where the first component is a struct value and the second component is a dictionary for the interface. Thus, we use pattern matching to extract the struct as Y and the wanted method as X_j. This X_j is a function accepting a triple: receiver, coercions for bounded type parameters of the method, and method arguments. Hence, the translation result is $X_j\ (Y, V', (\overline{E}))$. The difference to rule CALL-STRUCT is that there is no need to supply coercions for the bounded type parameters of the receiver. These coercions have already been supplied when building the dictionary, see rule COERCE-STRUCT-IFACE of Figure 9.

The last rule SUB is a subtyping rule allowing an expression e with translation E at type τ to be assigned some (structural) supertype σ. Premise $\Delta \vdash_{\text{coerce}} \tau <: \sigma \rightsquigarrow V$ serves two purposes: it ensures that σ is a supertype of τ and it yields a coercion function V from τ to σ. The translation of e at type σ is then $V\ E$. In Griesemer et al. (2020), the subtype check is included for each form of expression. For clarity, we choose to have a separate subtyping rule as in our translation scheme each subtyping relation implies a coercion function.

4.2.5 Well-formedness

Figure 11 defines several well-formedness judgments. The judgments $\Delta \vdash_{\text{ok}} \tau$ and $\Delta \vdash_{\text{ok}} \overline{\tau}$ assert that a single type and multiple types, respectively, are well-formed under type environment Δ. A type variable is well-formed if it is contained in Δ (rule OK-TYVAR). A named type $t[\overline{\tau}]$ is well-formed if its type arguments $\overline{\tau}$ are well-formed and if they are subtypes of the upper bounds in the definition of t. The latter is checked by the premise $\Delta \vdash_{\text{subst}} \overline{\alpha\ \tau_I} \mapsto \overline{\tau} : \eta \rightsquigarrow V$ of rule OK-TYNAMED, thereby ignoring the type substitution η and the coercion functions V. We have already seen in Section 2.5 that these coercions V are not represented in the translated program because type bounds of structs and interfaces have no operational meaning.

$$\boxed{\Delta \vdash_{\mathsf{ok}} \tau \qquad \Delta \vdash_{\mathsf{ok}} \overline{\tau}} \qquad \textit{Well-formedness of types}$$

$$\frac{(\alpha : \tau_I) \in \Delta}{\Delta \vdash_{\mathsf{ok}} \alpha}\ \textsc{ok-tyvar} \qquad \frac{\Delta \vdash_{\mathsf{ok}} \overline{\tau} \qquad \textbf{type}\ t[\overline{\alpha\ \tau_I}] \ldots \in \overline{D} \qquad \Delta \vdash_{\mathsf{subst}} \overline{\alpha\ \tau_I} \mapsto \overline{\tau} : \eta \rightsquigarrow V}{\Delta \vdash_{\mathsf{ok}} t[\overline{\tau}]}\ \textsc{ok-tynamed} \qquad \frac{\Delta \vdash_{\mathsf{ok}} \tau_i \quad (\forall i \in [n])}{\Delta \vdash_{\mathsf{ok}} \overline{\tau}^n}\ \textsc{ok-many-ty}$$

$$\boxed{\Delta \vdash_{\mathsf{ok}} \overline{\alpha\ \tau_I} \qquad \Delta \vdash_{\mathsf{ok}} R} \qquad \textit{Well-formedness of type parameters and method signatures}$$

$$\frac{\mathsf{dom}(\Delta) \cap \{\overline{\alpha}\} = \emptyset \qquad \mathsf{distinct}(\overline{\alpha}) \qquad \Delta \cup \{\overline{\alpha : \tau_I}\} \vdash_{\mathsf{ok}} \overline{\tau_I}}{\Delta \vdash_{\mathsf{ok}} \overline{\alpha\ \tau_I}}\ \textsc{ok-bounded-typarams} \qquad \frac{\Delta \vdash_{\mathsf{ok}} \overline{\alpha\ \tau_I} \qquad \mathsf{distinct}(\overline{x}) \qquad \Delta \cup \{\overline{\alpha : \tau_I}\} \vdash_{\mathsf{ok}} \overline{\sigma}\sigma}{\Delta \vdash_{\mathsf{ok}} m[\overline{\alpha\ \tau_I}](\overline{x\ \sigma})\sigma}\ \textsc{ok-msig}$$

$$\boxed{\vdash_{\mathsf{ok}} D} \qquad \textit{Well-formedness of declarations}$$

$$\frac{t_S \text{ defined once in } \overline{D} \qquad \emptyset \vdash_{\mathsf{ok}} \overline{\alpha\ \tau_I} \qquad \{\overline{\alpha : \tau_I}\} \vdash_{\mathsf{ok}} \overline{\sigma} \qquad \mathsf{distinct}(\overline{f})}{\vdash_{\mathsf{ok}} \textbf{type}\ t_S[\overline{\alpha\ \tau_I}]\ \textbf{struct}\ \{\overline{f\ \sigma}\}}\ \textsc{ok-struct}$$

$$\frac{t_I \text{ defined once in } \overline{D} \qquad \emptyset \vdash_{\mathsf{ok}} \overline{\alpha\ \tau_I} \qquad (\forall i \in [n])\ \{\overline{\alpha : \tau_I}\} \vdash_{\mathsf{ok}} R_i \qquad \mathsf{distinct}(\overline{\mathsf{methodName}(R_i)})}{\vdash_{\mathsf{ok}} \textbf{type}\ t_I[\overline{\alpha\ \tau_I}]\ \textbf{interface}\ \{\overline{R}^n\}}\ \textsc{ok-iface}$$

$$\frac{\begin{array}{c}\overline{D} \text{ contains one } \textbf{func}\text{-declaration for } t_S \text{ and } \mathsf{methodName}(R) \\ \emptyset \vdash_{\mathsf{ok}} \overline{\alpha\ \tau_I} \qquad \{\overline{\alpha : \tau_I}\} \vdash_{\mathsf{ok}} R \\ (\textbf{type}\ t_S[\overline{\alpha\ \tau'_I}^n]\ \textbf{struct} \ldots) \in \overline{D} \qquad \mathsf{methods}(\tau'_{Ii}) \subseteq \mathsf{methods}(\tau_{Ii}) \quad (\forall i \in [n])\end{array}}{\vdash_{\mathsf{ok}} \textbf{func}\ (x\ t_S[\overline{\alpha\ \tau_I}^n])R\ \{\textbf{return}\ e\}}\ \textsc{ok-method}$$

Fig. 11. Well-formedness

Judgment $\Delta \vdash_{\mathsf{ok}} \overline{\alpha\ \tau_I}$ asserts that bounded type parameters $\overline{\alpha\ \tau_I}$ are well-formed under type environment Δ (rule OK-BOUNDED-TYPARAMS). Judgment $\Delta \vdash_{\mathsf{ok}} R$ ensures that a method signature is well-formed (rule OK-MSIG). To form the combined environment $\Delta \cup \{\overline{\alpha : \tau_I}\}$ in the premise requires disjointness of the type variables in $\mathsf{dom}(\Delta)$ and $\overline{\alpha}$. This can always be achieved by α-renaming the type variables bound by R.

Judgment $\vdash_{\mathsf{ok}} D$ validates declaration D. A struct declaration is well-formed if it is defined only once (restriction FGG-UNIQUE-STRUCTS in Section 3.2), if all field names are distinct (restriction FGG-DISTINCT-FIELDS), and if the field types are well-formed. An interface declaration is well-formed if it is defined only once, if all its method signatures are well-formed, and if all methods have distinct names.

A method declaration for t_S and m is well-formed if there is no other declaration for t_S and m (restriction FGG-UNIQUE-METHOD-DEFS), if the method signature is well-formed, and if each bound τ_{Ii} of the method declaration is a structural subtype of the corresponding bound τ'_{Ii} in the declaration of t_S. In FGG$^-$, this boils down to checking that the methods of τ'_{Ii} are a subset of the methods of τ_{Ii}. The well-formedness conditions for method declarations

$$\boxed{\vdash_{\mathsf{meth}} \textbf{func}\ (x\ t_S[\overline{\alpha\ \tau_I}])\ R\ \{\textbf{return}\ e\} \rightsquigarrow X = V}\qquad \textit{Translating method declarations}$$

$$\textsc{method}\quad \frac{\begin{array}{c} \Delta = \{\overline{\alpha\ \tau_I}, \overline{\beta\ \sigma_I}\} \qquad \Gamma = \{x : t_S[\overline{\alpha}], \overline{x : \sigma}\} \\ x \notin \{\overline{x}\} \qquad \langle \Delta, \Gamma \rangle \vdash_{\mathsf{exp}} e : \sigma \rightsquigarrow E \qquad V = \lambda((\overline{X_{\alpha_i}}), X, (\overline{X_{\beta_i}}), (\overline{X})).E \end{array}}{\vdash_{\mathsf{meth}} \textbf{func}\ (x\ t_S[\overline{\alpha\ \tau_I}])\ m[\overline{\beta\ \sigma_I}](\overline{x\ \sigma})\ \sigma\ \{\textbf{return}\ e\} \rightsquigarrow X_{m,t_S} = V}$$

$$\boxed{\vdash_{\mathsf{prog}} P \rightsquigarrow \mathit{Prog}}\qquad \textit{Translating programs}$$

$$\textsc{prog}\quad \frac{\begin{array}{c} \overline{D} \text{ implicitly available in all subderivations} \qquad \langle \emptyset, \emptyset \rangle \vdash_{\mathsf{exp}} e : \tau \rightsquigarrow E \\ \vdash_{\mathsf{ok}} D_i \qquad (\text{for all } D_i \in \overline{D}) \\ \vdash_{\mathsf{meth}} D_i \rightsquigarrow X_i = V_i \qquad (\text{for all } D_i = \textbf{func}\ \ldots \in \overline{D}) \end{array}}{\vdash_{\mathsf{prog}} \overline{D}\ \textbf{func}\ \mathsf{main}()\{\_ = e\} \rightsquigarrow \textbf{let}\ \overline{X_i = V_i}\ \textbf{in}\ E}$$

Fig. 12. Translation of methods and programs

do not type check the method body. We will deal with this in the upcoming translation rule for methods.

4.2.6 Translation of Methods and Programs

Figure 12 defines the translation for method declarations and programs. Rule METHOD deals with method declarations $\textbf{func}\ (x\ t_S[\overline{\alpha\ \tau_I}])\ m[\overline{\beta\ \sigma_I}](\overline{x\ \sigma})\ \sigma\ \{\textbf{return}\ e\}$. The translation of such a declaration is the binding $X_{m,t_S} = V$. According to our translation strategy, V must be a function accepting a quadruple: coercions $(\overline{X_{\alpha_i}})$ for the bounded type parameters of the receiver, receiver X corresponding to x, coercions $(\overline{X_{\beta_i}})$ for the bounded type parameters local to the method, and finally method arguments $\overline{X}$ corresponding to $\overline{x}$. Binding all these variables with a λ makes them available in the translated body E.

Judgment $\vdash_{\mathsf{prog}} P \rightsquigarrow \mathit{Prog}$ denotes the translation of an FGG$^-$ program P to a TL program Prog. Rule PROG type checks the main expression e under empty environments against some type τ to get its translation E. Next, the rule requires all struct or interface declarations to be well-formed. Finally, it translates each method declaration to a binding $X_i = V_i$. The resulting TL program is then $\textbf{let}\ \overline{X_i = V_i}\ \textbf{in}\ E$.

4.3 Example

We now give an example of the translation. The FGG$^-$ code in the top part of Figure 13 defines equality for numbers Num and for generic boxes $\mathit{Box}[\alpha\ \mathit{Any}]$. Interface Any defines no methods, it serves as an upper bound for otherwise unrestricted type variables. We take the liberty to assume a basic type int and an operator == for equality. Interface $\mathit{Eq}[\alpha]$ requires a method eq for comparing the receiver with a value of type α. We provide implementations of eq for Num and $\mathit{Box}[\alpha]$. Comparing the content of a box requires the F-bound $\mathit{Eq}[\alpha]$ (Canning et al., 1989). The main function compares two boxes for equality.

The middle part of the figure shows the translation of the FGG$^-$ code, using abbreviations in the bottom part. Variable $X_{eq,Num}$ holds the translation of the declaration of eq for Num; it simply compares E_2 (translation of $\mathit{this.val}$) with E_3 (translation of $\mathit{that.val}$).

```
type Any interface {}
type Num struct { val int }
type Box[α Any] struct { content α }
type Eq[α Any] interface { eq(that α) bool }
func (this Num) eq(that Num) bool { return this.val == that.val }
func (this Box[α Eq[α]]) eq(that Box[α]) bool { return this.content.eq(that.content) }
func main(){ _ = Box[Num]{Num{1}}.eq(Box[Num]{Num{2}}) }
```

$$
\begin{aligned}
&\textbf{let } X_{eq,Num} = \lambda((), \mathit{This}, (), (\mathit{That})) \,.\, E_2 == E_3 \\
&\quad\;\; X_{eq,Box} = \lambda((X_\alpha), \mathit{This}, (), (\mathit{That})) \,.\, E_1 \\
&\textbf{in } X_{eq,Box}\ ((V_3), ((1)), (), ((2)))
\end{aligned}
$$

-- translated body of *eq* for *Box*

$$E_1 = \textbf{case } V_1\ E_2 \textbf{ of } (Y, (X_1)) \rightarrow X_1\ (Y, (), (E_3))$$

-- selectors for field *content* of *Box*

$$E_2 = \textbf{case } \mathit{This} \textbf{ of } (X_1) \rightarrow X_1$$

$$E_3 = \textbf{case } \mathit{That} \textbf{ of } (X_1) \rightarrow X_1$$

-- coercion $\alpha <: Eq[\alpha]$

$$V_1 = \lambda Y \,.\, V_2\ (X_\alpha\ Y)$$

-- identity coercion $Eq[\alpha] <: Eq[\alpha]$

$$V_2 = \lambda(Y, (X)) \,.\, (Y, (X))$$

-- coercion $Num <: Eq[Num]$

$$V_3 = \lambda X \,.\, (X, (\lambda(\overline{Y}^3) \,.\, X_{eq,Num}\ ((), \overline{Y}^3)))$$

Fig. 13. Example: FGG$^-$ code (top) and its translation (middle) with abbreviations (bottom)

Remember that the translation of a method declaration takes a quadruple with coercions for the bounded type parameters of the receiver, the receiver itself, coercions for the bounded type parameters of the method, and the method arguments. Here, $()$ is a tuple of size zero, corresponding to the non-existing type parameters, (That) denotes a tuple of size one, corresponding to the single argument *that*.

The translation of *eq* for *Box* is more involved. Figure 14 shows its derivation. We omit "obvious" premises and some trivial details from the derivation trees. Rule CALL-IFACE translates the body of the method. It coerces the receiver to the interface type $Eq[\alpha]$ and then extracts the method to be called via pattern matching, see E_1. The construction of the coercion is done via $\Delta \vdash_{\text{coerce}} \alpha <: Eq[\alpha] \rightsquigarrow V_1$, see subderivation ①. Coercion V_1 is slightly more complicated then necessary because the translation does not optimize the identity coercion V_2. Inside of V_1, we use X_α. This variables denotes a coercion from α to the representation of $Eq[\alpha]$; it is bound by the λ-expression in the definition of $X_{eq,Box}$.

The translation of the main expression calls $X_{eq,Box}$ with appropriate arguments, see Figure 15 for the derivation. The values $((1))$ and $((2))$ are nested tuples of size one, representing numbers wrapped in *Num* and *Box* structs. The method call of *eq* is translated by rule CALL-STRUCT, relying on rule METHODS-STRUCT to instantiate the type variable α to *Num*, as witnessed by the coercion V_3.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\langle \Delta, \Gamma\rangle \vdash_{\mathsf{exp}} \mathit{this.content} : \alpha \rightsquigarrow E_2}\,\textsc{access}\quad \textcircled{1}}{\langle \Delta, \Gamma\rangle \vdash_{\mathsf{exp}} \mathit{this.content} : \mathit{Eq}[\alpha] \rightsquigarrow V_1\; E_2}\,\textsc{sub} \qquad \dfrac{\dots}{\langle \Delta, \Gamma\rangle \vdash_{\mathsf{exp}} \mathit{that.content} : \alpha \rightsquigarrow E_3}\,\textsc{access} \\ \mathsf{methods}(\mathit{Eq}[\alpha]) = \mathit{eq}(\mathit{that}\ \alpha)\mathit{bool}}{\langle \Delta, \Gamma\rangle \vdash_{\mathsf{exp}} \mathit{this.content.eq}(\mathit{that.content}) : \mathit{bool} \rightsquigarrow E_1}\,\textsc{call-iface} \\ \Delta = \{\alpha : \mathit{Eq}[\alpha]\} \qquad \Gamma = \{\mathit{this} : \mathit{Box}[\alpha], \mathit{that} : \mathit{Box}[\alpha]\}}{\begin{array}{l}\vdash_{\mathsf{meth}} \textbf{func}\ (\mathit{this}\ \mathit{Box}[\alpha\ \mathit{Eq}[\alpha]])\ \mathit{eq}(\mathit{that}\ \mathit{Box}[\alpha])\ \mathit{bool}\ \{ \\ \quad \textbf{return}\ \mathit{this.content.eq}(\mathit{that.content}) \\ \}\rightsquigarrow X_{\mathit{eq},\mathit{Box}} = \lambda(\,(X_\alpha)\,,\mathit{This},\,()\,,\,(\mathit{That})\,)\,.\,E_1\end{array}}\,\textsc{method}$$

Subderivation $\textcircled{1}$

$$\dfrac{\dfrac{}{\Delta \vdash_{\mathsf{coerce}} \mathit{Eq}[\alpha] <: \mathit{Eq}[\alpha] \rightsquigarrow V_2}\,\textsc{coerce-iface-iface} \qquad (\alpha : \mathit{Eq}[\alpha]) \in \Delta}{\Delta \vdash_{\mathsf{coerce}} \alpha <: \mathit{Eq}[\alpha] \rightsquigarrow V_1}\,\textsc{coerce-tyvar}$$

Fig. 14. Example: translation of the method declaration for *Box* and *eq*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\langle \mathit{eq}(\mathit{That}\ \mathit{Num})\ \mathit{bool},\ (\,)\,\rangle \in \mathsf{methods}(\emptyset, \mathit{Num})}\,\textsc{methods-struct}}{\emptyset \vdash_{\mathsf{coerce}} \mathit{Num} <: \mathit{Eq}[\mathit{Num}] \rightsquigarrow V_3}\,\textsc{coerce-struct-iface}}{\emptyset \vdash_{\mathsf{subst}} \alpha\ \mathit{Eq}[\alpha] \mapsto \mathit{Num} : \langle \alpha \mapsto \mathit{Num}\rangle \rightsquigarrow (\,V_3\,)}\,\textsc{type-inst-checked} \\ \textbf{func}\ (\mathit{this}\ \mathit{Box}[\alpha\ \mathit{Eq}[\alpha]])\ \mathit{eq}(\mathit{that}\ \mathit{Box}[\alpha])\ \mathit{bool}\ \dots \in \overline{D}}{\langle \mathit{eq}(\mathit{That}\ \mathit{Box}[\mathit{Num}])\ \mathit{bool},\ (\,V_3\,)\,\rangle \in \mathsf{methods}(\emptyset, \mathit{Box}[\mathit{Num}])}\,\textsc{methods-struct} \\ \langle \emptyset, \emptyset\rangle \vdash_{\mathsf{exp}} \mathit{Box}[\mathit{Num}]\{\mathit{Num}\{1\}\} : \mathit{Box}[\mathit{Num}] \rightsquigarrow (\,(\,1\,)\,) \\ \langle \emptyset, \emptyset\rangle \vdash_{\mathsf{exp}} \mathit{Box}[\mathit{Num}]\{\mathit{Num}\{2\}\} : \mathit{Box}[\mathit{Num}] \rightsquigarrow (\,(\,2\,)\,)}{\begin{array}{l}\langle \emptyset, \emptyset\rangle \vdash_{\mathsf{exp}} \mathit{Box}[\mathit{Num}]\{\mathit{Num}\{1\}\}.\mathit{eq}(\mathit{Box}[\mathit{Num}]\{\mathit{Num}\{2\}\}) : \mathit{bool} \\ \quad \rightsquigarrow X_{\mathit{eq},\mathit{Box}}\ (\,(\,V_3\,)\,,\,(\,(\,1\,)\,)\,,\,(\,)\,,\,(\,(\,2\,)\,)\,)\end{array}}\,\textsc{call-struct}$$

Fig. 15. Example: translation of the main function

5 Formal Properties

In this section, we establish that the type-directed translation from Section 4.2 preserves the static and dynamic semantics of FGG^{-} programs. The translation as formalized is non-deterministic: for the same source program we may derive syntactically different target programs. Thus, we further show that all target programs resulting from the same source program behave equivalently. Detailed proofs for all lemmas and theorems are given in the appendix.

5.1 Preservation of Static Semantics

It is straightforward to verify that the type system originally defined for FGG is equivalent to the type system induced by the type-directed translation presented in Section 4.2, provided the FGG program does not contain type assertions. In the following, we write $\Delta \vdash_{\mathsf{G}} \tau <: \sigma$ for FGG's subtyping relation, $\Delta; \Gamma \vdash_{\mathsf{G}} e : \tau$ for its typing relation on expressions, and $\vdash_{\mathsf{G}} P\ \mathsf{ok}$ for the FGG typing relation on programs. These three relations were specified by Griesemer et al. (2020). The original article on FGG also includes support for dynamic type assertions, something we do not consider for our translation. Hence, we assume that FGG expressions do not contain type assertions.

Lemma 5.1.1 (FGG typing equivalence)**.** *Typing in FGG is equivalent to the type system induced by the translation, provided there are no type assertions.*

(*a*) *If* $\Delta \vdash_{\mathsf{G}} \tau <: \sigma$ *then either* $\Delta \vdash_{\mathsf{coerce}} \tau <: \sigma \rightsquigarrow V$ *for some* V *or* $\sigma = \tau$ *and* τ *is not an interface type.*
(*b*) *If* $\Delta \vdash_{\mathsf{coerce}} \tau <: \sigma \rightsquigarrow V$ *then* $\Delta \vdash_{\mathsf{G}} \tau <: \sigma$.
(*c*) *If* $\Delta; \Gamma \vdash_{\mathsf{G}} e : \tau$ *then* $\langle \Delta, \Gamma \rangle \vdash_{\mathsf{exp}} e : \tau \rightsquigarrow E$ *for some* E.
(*d*) *If* $\langle \Delta, \Gamma \rangle \vdash_{\mathsf{exp}} e : \tau \rightsquigarrow E$ *then* $\Delta; \Gamma \vdash_{\mathsf{G}} e : \tau'$ *for some* τ' *and* $\Delta \vdash_{\mathsf{G}} \tau' <: \tau$.
(*e*) $\vdash_{\mathsf{G}} P\ \mathsf{ok}$ *iff* $\vdash_{\mathsf{prog}} P \rightsquigarrow Prog$.

Claims (a) and (b) state that structural subtyping in FGG is equivalent to the relation from Figure 9, except that the latter is not reflexive for type variables and struct types. Claims (c) and (d) establish that expression typing in FGG and our expression typing from Figure 10 are equivalent modulo subtyping. The exposition in Griesemer et al. (2020) includes a subtyping check for each form of expression whereas we choose to have a separate subtyping rule. Hence, the type computed by the original rules for FGG might be a subtype of the type deduced by our system.

FGG enjoys type soundness (see Theorem 4.3 and 4.4 of Griesemer et al. 2020). The reduction rules for FGG and FGG$^-$ are obviously equivalent. Thus, Lemma 5.1.1 gives the following type soundness result for our type system:

Corollary 5.1.2. *Assume* $\langle \emptyset, \emptyset \rangle \vdash_{\mathsf{exp}} e : \tau \rightsquigarrow E$ *for some* e*,* τ*, and* E*. Then either* e *reduces to some value of type* τ *or* e *diverges.*

5.2 Preservation of Dynamic Semantics

This section proves that evaluating a well-typed FGG$^-$ program yields the same behavior as evaluating one of its translations. Thereby, we consider all possible outcomes of evaluation: reduction to a value or divergence. Further, we show that different translations of the same program have equivalent behavior.

The proof of semantic equivalence is enabled by a syntactic, step-indexed logical relation that relates an FGG$^-$ expression and a TL expression at some FGG$^-$ type. We write $e \longrightarrow^k e'$ if e reduces to e' in *exactly* $k \in \mathbb{N}$ steps, where $\mathbb{N}$ denotes the natural numbers including zero. By convention, we write $e \longrightarrow^0 e'$ to denote $e = e'$. The notation $e \longrightarrow^* e'$

$\boxed{e \approx E \in [\![\tau]\!]_k}$ *Expressions*

EQUIV-EXP

$$\frac{\begin{array}{c}(\forall k' < k, v \,.\, e \longrightarrow^{k'} v \implies \exists V. E \longrightarrow_{\mu}^{*} V \wedge v \equiv V \in [\![\tau]\!]_{k-k'}) \\ (\forall k' < k, e' \,.\, e \longrightarrow^{k'} e' \wedge \mathsf{diverge}(e') \implies \mathsf{diverge}(E))\end{array}}{e \approx E \in [\![\tau]\!]_k}$$

$\boxed{v \equiv V \in [\![\tau]\!]_k}$ *Values*

EQUIV-STRUCT

$$\frac{\textbf{type}\ t_S[\overline{\alpha\ \tau_I}]\ \textbf{struct}\ \{\overline{f\ \sigma}^n\} \in \overline{D} \qquad \forall i \in [n]. v_i \equiv V_i \in [\![\langle\overline{\alpha \mapsto \tau}\rangle \sigma_i]\!]_k}{t_S[\overline{\tau}]\{\overline{v}^n\} \equiv (\overline{V}^n) \in [\![t_S[\overline{\tau}]]\!]_k}$$

EQUIV-IFACE

$$\frac{\begin{array}{c}\exists \sigma_S . \forall k_1 < k . v \equiv U \in [\![\sigma_S]\!]_{k_1} \qquad \mathsf{methods}(\tau_I) = \{\overline{R}^n\} \\ \forall i \in [n], k_2 < k \,.\, \mathsf{methodLookup}(\mathsf{methodName}(R_i), \sigma_S) \approx V_i \in [\![R_i]\!]_{k_2}\end{array}}{v \equiv (U, (\overline{V}^n)) \in [\![\tau_I]\!]_k}$$

$\boxed{\mathsf{methodLookup}(m, \tau_S) = \langle x, \tau_S, R, e\rangle}$ *Method lookup*

METHOD-LOOKUP

$$\frac{\textbf{func}\ (x\ t_S[\overline{\alpha\ \tau_I}])\ R\ \{\textbf{return}\ e\} \in \overline{D} \qquad m = \mathsf{methodName}(R) \qquad \eta = \langle \overline{\alpha \mapsto \tau}\rangle}{\mathsf{methodLookup}(m, t_S[\overline{\tau}]) = \langle x, t_S[\overline{\tau}], \eta R, \eta e\rangle}$$

$\boxed{\langle x, \tau_S, R, e\rangle \approx (\lambda X . E) \in [\![R]\!]_k}$ *Method dictionary entries*

EQUIV-METHOD-DICT-ENTRY

$$\frac{\begin{array}{c}\forall k' \le k, \overline{\tau}^p, W, v, V, \overline{v}^n, \overline{V}^n . \\ (\eta = \langle \overline{\alpha \mapsto \tau}^p\rangle \wedge \overline{\tau}^p \approx W \in [\![\overline{\alpha\ \tau_I}^p]\!]_{k'} \wedge v \approx V \in [\![\tau_S]\!]_{k'} \wedge (\forall i \in [n]. v_i \approx V_i \in [\![\eta\sigma_i]\!]_{k'})) \\ \implies \langle x \mapsto v, \overline{x \mapsto v}^n\rangle \eta e \approx (\lambda X . E)\ (V, W, (\overline{V}^n)) \in [\![\eta\sigma]\!]_{k'}\end{array}}{\langle x, \tau_S, m[\overline{\alpha\ \tau_I}^p](\overline{x\ \sigma}^n)\ \sigma, e\rangle \approx (\lambda X . E) \in [\![m[\overline{\alpha\ \tau_I}^p](\overline{x\ \sigma}^n)\ \sigma]\!]_k}$$

$\boxed{\overline{\sigma} \approx V \in [\![\overline{\alpha\ \tau_I}]\!]_k}$ *Bounded type parameters*

EQUIV-BOUNDED-TYPARAMS

$$\frac{\eta = \langle \overline{\alpha \mapsto \sigma}^n\rangle \qquad \forall k' \le k, i \in [n], u_i, U_i \,.\, u_i \approx U_i \in [\![\sigma_i]\!]_{k'} \implies u_i \approx V_i\ U_i \in [\![\eta\tau_{Ii}]\!]_{k'}}{\overline{\sigma}^n \approx (\overline{V}^n) \in [\![\overline{\alpha\ \tau_I}^n]\!]_k}$$

Fig. 16. Relating FGG$^-$ to TL expressions

states that $e \longrightarrow^k e'$ for some $k \in \mathbb{N}$. We write $\mathsf{diverge}(e)$ to denote that e does not terminate; that is, for all $k \in \mathbb{N}$ there exists some e' with $e \longrightarrow^k e'$. The same definitions apply analogously to reductions in the target language.

5.2.1 The Logical Relation

The definition of the logical relation spreads over two figures 16 and 17. In these figures, we assume that the declarations $\overline{D}$ of the FGG$^-$ program being translated are implicitly available in all rules. Also, we assume that an arbitrary but fixed method substitution μ

is implicitly available to all rules. This μ is used in the reduction rules of the target language to resolve let-bound variables (i.e. translations of methods). In our main theorem (Theorem 5.2.6), we will then require that μ results from translating the methods in $\overline{D}$.

We now explain the logical relation on expressions, see Figure 16. The relation $e \approx E \in [\![\tau]\!]_k$ denotes that FGG$^-$ expression e and TL expression E are equivalent at type τ for at most k reduction steps. We call k the *step index*. Rule EQUIV-EXP has two implications as its premises. The first states that if e reduces to a value v in $k' < k$ steps, then E reduces to some value V in an arbitrary number of steps and v is equivalent to V at type τ for the remaining $k - k'$ steps. The second premise is for diverging expressions: if e reduces in less than k steps to some expression e' and e' diverges, then E diverges as well.

The relation $v \equiv V \in [\![\tau]\!]_k$ defines equivalence of FGG$^-$ value v and TL value V at type τ with step index k. Rule EQUIV-STRUCT handles the case where τ is a struct type. Then v must be a value of this struct type and V must be a struct value such that all field values of v and V are equivalent. Rule EQUIV-IFACE deals with the case that τ is an interface type. Hence, V must be an interface value $(U, (\overline{V}))$ with two requirements. First, v and U are equivalent for all step indices $k_1 < k$ at some struct type σ_S. Second, $(\overline{V})$ must be an appropriate dictionary for the methods of the interface with receiver type σ_S. To check this requirement, rule METHOD-LOOKUP defines the auxiliary $\mathsf{methodLookup}(m_i, \sigma_S)$ to retrieve a quadruple $\langle x, \sigma_S, R, e\rangle$ from the declaration of m_i for σ_S. This quadruple has to be equivalent to dictionary entry V_i for all step indices $k_2 < k$ at the signature of the method.

A dictionary entry is always a function value. We write $\langle x, \tau_S, R, e\rangle \approx (\lambda X\,.\,E) \in [\![R]\!]_k$ to denote equivalence between a quadruple for a method declaration and some dictionary entry $\lambda X\,.\,E$. Rule EQUIV-METHOD-DICT-ENTRY defines this equivalence such that method body e and $\lambda X\,.\,E$ take related arguments to related outputs. Thus, the premise of the rule requires for all step indices $k' \leq k$, all related type parameters $\overline{\tau}$ and W, all related receiver values v and V, and all related arguments $\overline{v}$ and $\overline{V}$ that e and $\lambda X\,.\,E$ yield related results when applied to the respective arguments.

The judgment $\overline{\sigma} \approx V \in [\![\overline{\alpha\ \tau_I}]\!]_k$ denotes equivalence between concrete type arguments $\overline{\sigma}$ and their TL realization V when checking the bounds of type parameters $\overline{\alpha\ \tau_I}$. The definition in rule EQUIV-BOUNDED-TYPARAMS relies on our translation strategy that bounded type parameters are represented by coercions.

Having explained all judgments from Figure 16, we verify that the recursive definitions of $e \approx E \in [\![\tau]\!]_k$ and $v \equiv V \in [\![\tau]\!]_k$ are well-founded. Often, logical relations are defined by induction on the structure of types. In our case, this approach does not work because interface types in FGG$^-$ might be recursive, see our previous work (Sulzmann and Wehr, 2022) for an example. Thus, we use the step index as part of a decreasing measure $\mathcal{M}$. Writing $|V|$ for the size of some target value V, we define $\mathcal{M}(e \approx E \in [\![\tau]\!]_k) = (k, 1, 0)$ and $\mathcal{M}(v \equiv V \in [\![\tau]\!]_k) = (k, 0, |V|)$. In EQUIV-EXP, either k decreases or stays constant but the second component of $\mathcal{M}$ decreases. In EQUIV-STRUCT, k and the second component stay constant but $|V|$ decreases, and in EQUIV-IFACE together with EQUIV-METHOD-DICT-ENTRY and EQUIV-BOUNDED-TYPARAMS step index k decreases. Note that EQUIV-METHOD-DICT-ENTRY and EQUIV-BOUNDED-TYPARAMS only require $k' \leq k$. This is ok because we already have $k_2 < k$ in EQUIV-IFACE.

Figure 17 extends the logical relation to whole programs. Judgment $\eta \approx \rho \in [\![\Delta]\!]_k$ denotes how a FGG$^-$ type substitution η intended to substitute the type variables from Δ is

$\boxed{\eta \approx \rho \in [\![\Delta]\!]_k \qquad \theta \approx \rho \in [\![\Gamma]\!]_k}$ *Substitutions*

EQUIV-TY-SUBST

$$\frac{\overline{\eta\alpha_i \approx (\rho X_{\alpha_i}) \in [\![\alpha\ \tau]\!]_k}}{\eta \approx \rho \in [\![\{\overline{\alpha : \tau}\}]\!]_k}$$

EQUIV-VAL-SUBST

$$\frac{\forall (x : \tau) \in \Gamma.\ \theta(x) \approx \rho(X) \in [\![\tau]\!]_k}{\theta \approx \rho \in [\![\Gamma]\!]_k}$$

$\boxed{\textsf{\textbf{func}}\ (x\ t_S[\overline{\alpha\ \tau_I}])\ R\ \{\textsf{\textbf{return}}\ e\} \approx_k X}$ *Method declarations*

EQUIV-METHOD-DECL

$$\frac{\begin{array}{c}\forall k' < k, \overline{\tau}^p, \overline{\tau'}^q, \overline{W}^p, \overline{W'}^q, v, V, \overline{v}^n, \overline{V}^n.\\ \eta = \langle \overline{\alpha \mapsto \tau}^p, \overline{\alpha' \mapsto \tau'}^q \rangle \wedge \overline{\tau}^p \overline{\tau'}^q \approx (\overline{W}^p, \overline{W'}^q) \in [\![\overline{\alpha\ \tau_I}^p, \overline{\alpha'\ \tau'_I}^q]\!]_{k'} \wedge \\ v \approx V \in [\![t_S[\eta\overline{\alpha}^p]]\!]_{k'} \wedge (\forall i \in [n].v_i \approx V_i \in [\![\eta\sigma_i]\!]_{k'}) \implies \\ \langle x \mapsto v, \overline{x \mapsto v}^n \rangle \eta e \approx X\ ((\overline{W}^p), V, (\overline{W'}^q), (\overline{V}^n)) \in [\![\eta\sigma]\!]_{k'}\end{array}}{\textsf{\textbf{func}}\ (x\ t_S[\overline{\alpha\ \tau_I}^p])\ m[\overline{\alpha'\ \tau'_I}^q](\overline{x\ \sigma}^n)\ \sigma\ \{\textsf{\textbf{return}}\ e\} \approx_k X}$$

$\boxed{\overline{D} \approx_k \mu}$ *Programs*

EQUIV-DECLS

$$\frac{\begin{array}{c}\overline{D}, \mu \text{ are implicitly available in all subderivations}\\ \forall D_i \in \overline{D}.D_i = \textsf{\textbf{func}}\ (x\ t_S[\overline{\alpha\ \tau}])\ mM\ \{\textsf{\textbf{return}}\ e\} \implies D_i \approx_k X_{m,t_S}\end{array}}{\overline{D} \approx_k \mu}$$

Fig. 17. Relating FGG$^-$ to TL substitutions and declarations

related to a TL substitution ρ. The definition in rule EQUIV-TY-SUBST falls back to equivalence of type parameters. Judgment $\theta \approx \rho \in [\![\Gamma]\!]_k$ similarly relates a FGG$^-$ value substitution θ intended for value environment Γ with a TL substitution ρ. See rule EQUIV-VAL-SUBST.

Judgment **func** $(x\ t_S[\overline{\alpha\ \tau_I}])\ R$ {**return** $e\} \approx_k X$ states equivalence of a function declaration with a TL variable X. Rule EQUIV-METHOD-DECL takes an approach similar as in rule EQUIV-METHOD-DICT-ENTRY: method body e and variable X must yield related outputs when applied to related arguments. Thus, for all related type arguments $\overline{\tau}$, $\overline{\tau'}$ and $(\overline{W}, \overline{W'})$, all related receiver values v and V, and all related arguments $\overline{v}$ and $\overline{V}$, the expression e and variable X must be related when applied to the appropriate arguments. However, different than in EQUIV-METHOD-DICT-ENTRY, we only requires this to hold for all $k' < k$.

Judgment $\overline{D} \approx_k \mu$ defines equivalence between FGG$^-$ declarations $\overline{D}$ and TL method substitution μ. The definition in rule EQUIV-DECLS is straightforward: each method declaration for some method m and struct t_S must be equivalent to variable X_{m,t_S}.

5.2.2 Equivalence Between Source and Translation

To establish the desired result of semantic equivalence between a source program and one of its translations, we implicitly make the following assumptions about the globally available declarations $\overline{D}$ and method substitution μ.

Assumption 5.2.1. We assume that the globally available declarations $\overline{D}$ are well-formed; that is, $\vdash_{\mathsf{ok}} D_i$ for all $D_i \in \overline{D}$ and $\vdash_{\mathsf{meth}} D'_i \rightsquigarrow X_i = V_i$ for some X_i and V_i and all $D'_i =$

$\textsf{func} \ldots \in \overline{D}$. Further, we assume that the globally available method substitution μ has only variables of the form X_{m,t_S} in its domain.

Several basic properties hold for our logical relation. For example, monotonicity gives us that with $e \approx E \in [\![\tau]\!]_k$ and $k' \leq k$ we also have $e \approx E \in [\![\tau]\!]_{k'}$. Another property is how target and source reductions preserve equivalence:

Lemma 5.2.2 (Target reductions preserve equivalence). *If $e \approx E \in [\![\tau]\!]_k$ and $E_2 \longrightarrow^* E$ then $e \approx E_2 \in [\![\tau]\!]_k$.*

Lemma 5.2.3 (Source reductions preserve equivalence). *If $e \approx E \in [\![\tau]\!]_k$ and $e_2 \longrightarrow e$ then $e_2 \approx E \in [\![\tau]\!]_{k+1}$.*

The lemmas for monotonicity and several other properties are stated in Appendix A.3, together with all proofs. We can then establish that an FGG$^-$ expression e is semantically equivalent to its translation E.

Lemma 5.2.4 (Expression equivalence). *Assume $\overline{D} \approx_k \mu$ and $\eta \approx \rho \in [\![\Delta]\!]_k$ and $\theta \approx \rho \in [\![\eta\Gamma]\!]_k$. If $\langle\Delta, \Gamma\rangle \vdash_{\mathsf{exp}} e : \tau \rightsquigarrow E$ then $\theta\eta e \approx \rho E \in [\![\eta\tau]\!]_k$.*

The proof is by induction on the derivation of $\langle\Delta, \Gamma\rangle \vdash_{\mathsf{exp}} e : \tau \rightsquigarrow E$, see Appendix A.3.2.1 (page 50) for the full proof. We next establish semantic equivalence for method declarations.

Lemma 5.2.5 (Method equivalence). *Let $\overline{D}$ and μ such that for each $D = \textsf{func}\ (x\ t_S[\overline{\alpha\ \tau_I}])\ R\ \{\textsf{return}\ e\} \in \overline{D}$ with $m = \mathsf{methodName}(R)$ we have $\vdash_{\mathsf{meth}} D \rightsquigarrow X_{m,t_S} = V$ and $\mu(X_{m,t_S}) = V$ for some V. Then $\overline{D} \approx_k \mu$ for any k.*

The proof of this lemma is by induction on k, see Appendix A.3.2.2 (page 59) for the full proof. Finally, the following theorem states our desired result: semantic equivalence between an FGG$^-$ program and its translation.

Theorem 5.2.6 (Program equivalence). *Let $\vdash_{\mathsf{prog}} \overline{D}\ \textsf{func}\ \mathsf{main}()\{\_ = e\} \rightsquigarrow \textsf{let}\ \overline{X_i = V_i}\ \textsf{in}\ E$ with e having type τ. Let $\mu = \langle\overline{X_i \mapsto V_i}\rangle$. Then both of the following holds:*

1. *If $e \longrightarrow^* v$ for some value v then there exists a target language value V such that $E \longrightarrow^*_\mu V$ and $v \equiv V \in [\![\tau]\!]_k$ for any k.*
2. *If e diverges then so does E.*

The "with e having type τ" part means that the last rule in the derivation of the program translation has $\langle\emptyset, \emptyset\rangle \vdash_{\mathsf{exp}} e : \tau \rightsquigarrow E$ as a premise. Obviously, $\overline{D}$ and μ meet the requirements of Assumption 5.2.1. The theorem then follows from Lemma 5.2.4 and Lemma 5.2.5. See Appendix A.3.2.3 (page 61) for the full proof.

$\boxed{\textsf{erase}(\tau, V) = V \quad \textsf{erase}(V) = V}$ *Erasure of dictionaries*

$$\textsf{erase}(\tau_S, V) = \textsf{erase}(V) \qquad \textsf{erase}(K\ \overline{V}) = K\ \overline{\textsf{erase}(V_i)}$$
$$\textsf{erase}(\tau_I,\ (V, U)) = \textsf{erase}(V) \qquad \textsf{erase}(\lambda X\,.\,E) = \mathbf{K}_\lambda$$

Fig. 18. Erasure of dictionaries

5.2.3 Equivalence Between Different Translations

Our translation is non-deterministic because different translations of the same expression may contain distinct sequences of applications of the subsumption rule SUB. Recall the example from Figure 1. There are (at least) two different ways to translate expression `Num{1}` at type `Format`.

1. Use rules COERCE-STRUCT-IFACE and SUB to go directly from `Num` to supertype `Format`. The translation is then $\texttt{toFormat}_{\texttt{Num}}$ `1`.
2. First use COERCE-STRUCT-IFACE and SUB to go from `Num` to `Pretty`, then use COERCE-IFACE-IFACE and SUB to go from `Pretty` to `Format`. The translation is then $\texttt{toFormat}_{\texttt{Pretty}}$ $(\texttt{toPretty}_{\texttt{Num}}\texttt{1})$.

Each choice leads to a syntactically distinct target expression. In general, evaluating the target expressions might lead to syntactically different target values because target values might contain dictionaries (i.e. tuple of λ-expressions), and different translations might produce syntactically different dictionaries.

Another source of non-determinism is that rule PROG for typing programs is allowed to choose the type τ of the main expression. For example, instead of typing `Num{1}` at type `Format`, another translation might pick type `Pretty` or `Num` for the main expression. The choice for the type of the main expression might also lead to syntactically different target expressions.

To summarize, different translations of the same source program might lead to syntactically different target language programs, and the syntactic differences might persist in the final values after evaluation. But a slightly weaker property holds: if we remove all dictionaries in the final values, then the results are syntactically equal.

Figure 18 defines a function erase that removes all dictionaries from a target-language value. The function comes in two variations:

- $\textsf{erase}(\tau, V)$ removes all dictionaries from value V when viewed at type τ. Its duty is to remove the topmost dictionary if τ is an interface type. In this case, the function is partial but this is not an issue: a value viewed at an interface type is always a pair of values.
- $\textsf{erase}(V)$ removes all dictionaries from V by replacing the λ-expressions with a fixed, otherwise unused, nullary constructor $\mathbf{K}_\lambda$. This definition relies on then fact that the translation only produces λ-expressions for dictionary entries. We replace λ-expressions with a dedicated constructor $\mathbf{K}_\lambda$ instead of the nullary tuple `()` so as to avoid confusion between an erased λ and a struct value without fields.

The following theorem states that evaluating the outcomes of two translations of the same source program yields values that are identical up to removal of dictionaries (or both diverge). This holds even if the two translations assign different types to the main expression of the source program. That is, there are no semantic ambiguities and we can establish that our translation is coherent (Reynolds, 1991).

Theorem 5.2.7 (Coherence). *Let* $P = \overline{D}$ $\textsf{func}$ $\textsf{main}()\{\_ = e\}$. *Assume* $\vdash_{\textsf{prog}} P \rightsquigarrow$ $\textsf{let}$ $\overline{X_i = V_i}$ $\textsf{in}$ E *with* e *having type* τ *and* $\vdash_{\textsf{prog}} P \rightsquigarrow$ $\textsf{let}$ $\overline{X'_i = V'_i}$ $\textsf{in}$ E' *with* e *having type* τ'. *Define* $\mu = \langle \overline{X_i \mapsto V_i} \rangle$ *and* $\mu' = \langle \overline{X'_i \mapsto V'_i} \rangle$. *Then both of the following holds:*

1. *If* $E \longrightarrow^*_\mu V$ *for some* V, *then* $E' \longrightarrow^*_{\mu'} V'$ *for some* V' *with* $\textsf{erase}(\tau, V) = \textsf{erase}(\tau', V')$.
2. *If* E *diverges then so does* E'.

See Appendix A.3.3.1 (page 63) for the proof.

5.3 *Getting the step index right*

The logical relation in Figures 16 and 17 requires at some places the step index in the premise to be strictly smaller than in the conclusion ($<$), other places require only less-than-or-equal ($\leq$). In EQUIV-EXP, we have $<$ to keep the definition of the LR well-founded. The $<$ in rule EQUIV-METHOD-DECL is required for the inductive argument in the proof of Lemma 5.2.5. Rule EQUIV-IFACE also has $<$, but rule EQUIV-METHOD-DICT-ENTRY only requires $\leq$. For well-foundedness, it is crucial that one of these two rules decreases the step index. However, equally important is that the step index is not forced to decrease more than once, so we need $<$ in one rule and $\leq$ in the other. If both rules had $<$, then the proof of Lemma 5.2.4 would not go through for case CALL-IFACE.

Consider the following example in the context of Figure 13:

$$\begin{array}{rcl}
w_1 = \mathit{Num}\{1\} \text{ at type } \mathit{Eq}[\mathit{Num}] & \rightsquigarrow & W_1 = \texttt{((1),(}U\texttt{))} \\
 & & \quad \text{where } U = \lambda(\overline{Y}^3)\,.\,X_{eq,Num}\ \texttt{((),}\overline{Y}^3\texttt{)} \\
w_2 = \mathit{Num}\{2\} \text{ at type } \mathit{Num} & \rightsquigarrow & W_2 = \texttt{((2))} \\
w_1.eq(w_2) & \rightsquigarrow & E = \textbf{case } W_1 \textbf{ of } \texttt{(}Y\texttt{,(}X_1\texttt{))} \rightarrow X_1\ \texttt{(}Y\texttt{,(),(}W_2\texttt{))}
\end{array}$$

For values w_1 and w_2, we may assume (1) $w_1 \equiv W_1 \in [\![\mathit{Eq}[\mathit{Num}]]\!]_k$ and $w_2 \approx W_2 \in [\![\mathit{Num}]\!]_k$ for some k. To verify that the translation yields related expressions, we must show

$$w_1.eq(w_2) \approx E \in [\![\mathit{bool}]\!]_k \tag{2}$$

From (1), via inversion of rule EQUIV-IFACE, we can derive

$$\textsf{methodLookup}(eq, \mathit{Num}) \approx U \in [\![eq(\mathit{that}\ \mathit{Num})\ \mathit{bool}]\!]_{k-1} \tag{3}$$

because the premise of the rule requires this to hold for all $k_2 < k$. Let e be the body of the method declaration of *eq* for *Num*. Inverting rule EQUIV-METHOD-DICT-ENTRY for (3) yields

$$\langle \mathit{this} \mapsto w_1, \mathit{that} \mapsto w_2 \rangle e \approx U\ \texttt{((1),(),((2)))} \in [\![\mathit{bool}]\!]_{k'} \tag{4}$$

for $k' = k - 1$ because rule EQUIV-METHOD-DICT-ENTRY has $\leq$ in its premise. Also, we have $w_1.eq(w_2) \longrightarrow^1 \langle \mathit{this} \mapsto w_1, \mathit{that} \mapsto w_2 \rangle e$ and $E \longrightarrow^* U\ \texttt{((1),(),((2)))}$. Thus, with (4), Lemma 5.2.2, and Lemma 5.2.3 we get $w_1.eq(w_2) \approx E \in [\![bool]\!]_{k'+1}$. For $k' = k - 1$, this is exactly (2), as required. But if rule EQUIV-METHOD-DICT-ENTRY required $<$ in its premise, then (4) would only hold for $k' = k - 2$ and we could not derive (2).

Whether we have $<$ in EQUIV-IFACE and $\leq$ in EQUIV-METHOD-DICT-ENTRY or vice versa is a matter of taste. In our previous work at MPC (Sulzmann and Wehr, 2022), we established a dictionary-passing translation for Featherweight Go without generics. The situation is slightly different there. With generics, we need two rules with respect to methods: EQUIV-METHOD-DECL for method declarations and EQUIV-METHOD-DICT-ENTRY for dictionary entries where the coercions for the bounds of the receiver's type parameters have already been supplied. Without generics, there are no type parameters, so a single rule suffices (rule RED-REL-METHOD in MPC). So in the article at MPC, we use $<$ for rule RED-REL-METHOD and $\leq$ for rule RED-REL-IFACE, the pendant to rule EQUIV-IFACE of the current article.

6 Implementation

We provide an implementation of the translation[2] written in Haskell (2022). All examples in this article were checked against the implementation. Competitive runtime performance of the translated code was not our goal. Hence, we took a convenient route and used Racket (2022) as the target language. The implementation features all language concepts from Section 3, as well as type assertions, generic functions, and several base types (integers, characters, strings, and booleans).

Generic functions and base types are straightforward to support. Implementing the typing and translation rules from Figure 10 requires some care because the presence of subsumption rule SUB renders the translation non-deterministic (see Section 5.2.3). We solved this problem by "inlining" the subsumption step when checking the arguments of a method call against the parameter types (rules CALL-STRUCT and CALL-IFACE) and when checking the field values of a struct against the declared field types (rule STRUCT). On formal grounds, this is justified as Featherweight Go (2020) inlines the subsumption step in similar ways and typing in FGG$^-$ is equivalent the type system induced by our translation rules (Lemma 5.1.1). The realization of type assertions (dynamic type casts) uses type tags (Ohori and Ueno, 2021). At runtime, a type assertion $e.(\tau)$ checks compatibility between e's type tag and the type tag corresponding to τ.

Our implementation comes with a large test suite and contains in total 181 tests, covering all main features of the source language. See Figure 19 for a summary. We wrote 25 new tests and included all 148 tests and examples from the OOPSLA 2020 implementation[3]. Moreover, we also included 8 examples from the OOPSLA 2022 implementation[4]. (The implementation for OOPSLA 2022 builds on the one for OOPSLA 2020, adding 12 new test cases. We ignored 4 test cases because they use concurrency features not supported by

[2] `https://github.com/skogsbaer/fgg-translate` and `http://doi.org/10.5281/zenodo.8147425`
[3] Griesemer et al. (2020), `https://github.com/rhu1/fgg`
[4] Ellis et al. (2022), `https://github.com/sfzhu93/fgg2go`

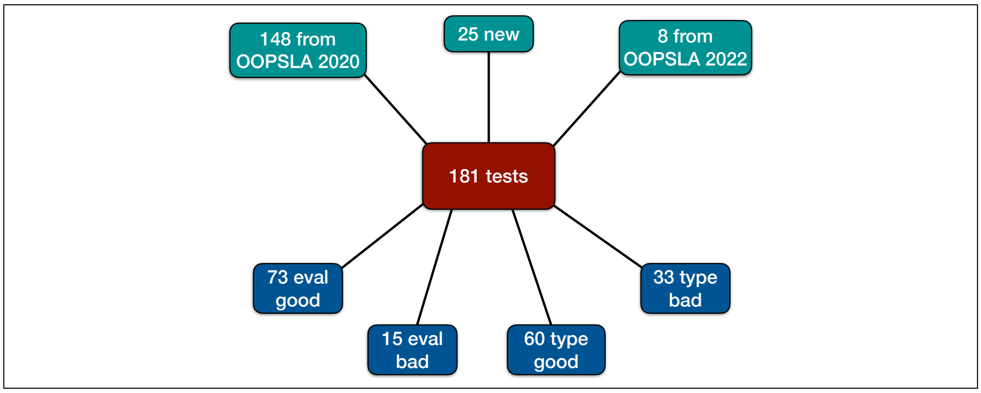

Fig. 19. Summary of the test suite for the implementation

our implementation.) Most tests from OOPSLA 2020/2022 could be integrated in our test suite without changes, for some we had to perform minor syntactic adjustments.

Each test is a source file in one of the following categories:

1. **eval good**, 73 tests: The test type checks and evaluates successfully. For the test to succeed, the result of evaluation must match the expected result. We arrived at the expected result by inspecting the program and (if applicable) comparing it against the run of the OOPSLA 2020/2022 implementation.
2. **eval bad**, 15 tests: The test type checks successfully but fails at runtime. For the test to succeed, the error message must match an expected string. We determined the expected string in similar way as for the eval good category.
3. **type good**, 60 tests: The test type checks successfully but is not executed because it has no interesting operational behavior.
4. **type bad**, 33 tests: The test fails to type check. For the test to succeed, the error message must match an expected string.

Some of the tests behave differently under our implementation when compared with the original implementations for OOPSLA 2020/2022:

- The OOPSLA 2020 implementation compiles generics by monomorphization; that is, generic code is specialized for all type arguments appearing in the program. But monomorphization cannot deal with all programs, so their type checker rejects several programs based on some syntactic condition (see Section 7.1 for details). Our implementation type checks these programs successfully.
- The OOPSLA 2020 implementation statically rejects type assertions $e.(\tau)$ where the type of e is a struct type, even though evaluation might succeed at runtime. Our implementation is more liberal and only rejects type assertions statically that are guaranteed to fail at runtime.
- The OOPSLA 2020 implementation rejects recursive definitions of structs. For simplicity, we omitted this check from our implementation.

- The OOPSLA 2020 implementation runs several tests only for a fixed number of reduction steps because these tests would diverge otherwise. Our implementation only type checks such tests.

7 Related Work

The related work section covers generics in Go, type classes in Haskell, logical relations, and a summary of our own prior work.[5] At the end, we give an overview of the existing translations with source language Featherweight Generic Go.

7.1 Generics in Go

The results of this work rest on the definition of Featherweight Generic Go (FGG) provided by Griesemer and colleagues (2020). FGG is a minimal core calculus modeling the essential features of the programming language Go (2022). It includes support for overloaded methods, interface types, structural subtyping, generics, and type assertions. Our formalization of FGG ignores dynamic type assertions but otherwise sticks to the original definition of FGG, apart from some minor cosmetic changes in presentation. We prove that the type system implied by our translation is equivalent to the original type system of FGG, and that translated programs behave the same way as under the original dynamic semantics.

The original dynamic semantics of FGG uses runtime method lookup, in the same way as we did in Section 3. The authors define an alternative semantics via monomorphisation; that is, they specialize generic code for all type arguments appearing in the program. This alternative semantics is equivalent to the one based on runtime method lookup, but there exist type-correct FGG programs that cannot be monomorphized. For instance, polymorphic recursion leads to infinitely many type instantiations, so programs with a polymorphic-recursive method cannot be monomorphized.[6] Further, monomorphization often leads to a blowup in code size. In contrast, our translation handles all type-correct FGG programs, and instantiations of generic code with different type arguments do not increase the code size. However, we expect that monomorphized code will offer better performance than code generated by our dictionary-passing translation, because method dictionaries imply several indirections not present in monomorphized programs.

The current implementation of generics (Taylor and Griesemer, 2021) in Go versions 1.18, 1.19, and 1.20 (2022) differs significantly from the formalization in FGG. For example, full Go requires a method declaration for a generic struct to have exactly the same type bounds as the struct. In FGG, bounds of the receiver struct in a method declaration might be stricter than the bounds in the corresponding struct declaration. In Figure 2, we used this feature to implement formatting for the fully generic `Pair` type, provided the type parameters can be formatted as well. Go cannot express this scenario without falling back to dynamic type assertions.

[5] Several points in Sections 7.1, 7.2, and 7.3 were already included in own prior work (Sulzmann and Wehr, 2021, 2022).

[6] See Griesemer et al. (2020), Figure 10 for an example.

Ellis et al. (2022) formalize a dictionary-passing translation from a restricted subset of FGG to FG. The restriction for FGG is the same as previously explained for full Go: a method declaration must have the same type bounds as its receiver struct. The translation utilizes this restriction to translate an FGG struct together with all its methods into a single FG struct (dictionary). This approach would scale to full Go even with separate compilation because a struct and all its methods must be part of the same package. Further, the translation of Ellis and coworkers replaces all types in method signatures with the top-type `Any`, relying on dynamic type assertions to enable type checking of the resulting FG program. The authors provide a working implementation and a benchmark suite to compare their translation against several other approaches, including the current implementation of generics in full Go. Our translation targets an extended λ-calculus and does not restrict the type bounds of the receiver struct in a method declaration. We also provide an implementation but no evaluation of its performance.

Method dictionaries bear some resemblance to virtual method tables (vtables) used to implement virtual method dispatch in object-oriented languages (Driesen and Hölzle, 1996). The main difference between vtables and dictionaries is that there is a fixed connection between an object and its vtable (via the class of the object), whereas the connection between a value and a dictionary may change at runtime, depending on the type the value is used at. Dictionaries allow access to a method at a fixed offset, whereas vtables in the presence of multiple inheritance require a more sophisticated lookup algorithm (Alpern et al., 2001).

Generics in class-based languages such as Java (Bracha et al., 1998; Igarashi et al., 2001), and C♯ (Kennedy and Syme, 2001; Emir et al., 2006) do not require a dictionary-passing translation because all methods are part of the virtual method table of an object. In Go, however, methods are not necessarily attached to the receiver struct, so additional evidence in form of dictionaries must be passed for such methods. Further, subtyping in Java and C♯ is nominal, whereas Go has structural subtyping.

A possible optimization to the dictionary-passing translation is selective code specialization (Dean et al., 1995). With this approach, the dictionary-passing translation generates code that runs for all type arguments. In addition, specialized code is generated for frequently used combinations of type arguments. This approach allows to trade code size against runtime performance. The GHC compiler for Haskell supports a `SPECIALIZE` pragma (GHC User's Guide, 2022, § 6.20.11.) that allows developers to specialize a polymorphic function to a particular type. The specialization also supports type-class dictionaries.

7.2 Type Classes in Haskell

The dictionary-passing translation is well-studied in the context of Haskell type classes (Wadler and Blott, 1989; Hall et al., 1996). A type-class constraint translates to an extra function parameter, constraint resolution provides a dictionary with the methods of the type class for this parameter. In FGG, structural subtyping relations imply coercions and bounded type parameters translate to coercion parameters. An interface value pairs a struct value with a dictionary for the methods of the interface. Thus, interface values can be viewed as representations of existential types (Mitchell and Plotkin, 1988; Läufer, 1996; Thiemann and Wehr, 2008).

Another important property in the type class context is coherence. Bottu and coworkers (2019) make use of logical relations to state equivalence among distinct target terms resulting from the same source type class program. In the setting of FGG-, we first establish semantic equivalence among source and target programs, see Theorem 5.2.6. From this property, we can derive the coherence property (Theorem 5.2.7) almost for free. We believe it is worthwhile to establish a property similar to this theorem for type classes. We could employ a simple denotational semantics for source type class programs similar as Thatte (1994) or Morris (2014), which would then be related to target programs obtained via the dictionary-passing translation.

Section 2.5 demonstrated that type bounds on generic structs and interfaces have no operational meaning. This situation is similar to contexts of data type definitions in Haskell 2010 (Marlow, 2010). A data type such as

```
data Eq a => Set a = NilSet | ConsSet a (Set a)
```

may require the context `Eq a`. However, an occurrence of type `Set a` does not imply that `Eq a` holds but always requires the constraint to be justified elsewhere. The GHC manual states that "this is widely considered a misfeature" (GHC Team, 2021, Section 6.4.2).

7.3 Logical Relations

Logical relations have a long tradition of proving properties of typed programming languages. Such properties include termination (Tait, 1967; Statman, 1985), type safety (Skorstengaard, 2019), and program equivalence (Pierce, 2004, Chapters 6, 7). A logical relation (LR) is often defined inductively, indexed by type. If its definition is based on an operational semantics, the LR is called syntactic (Pitts, 1998; Crary and Harper, 2007). With recursive types, a step-index (Appel and McAllester, 2001; Ahmed, 2006) provides a decreasing measure to keep the definition well-founded. See Mitchell (1996, Chapter 8) and Skorstengaard (2019) for introductions to the topic.

LRs are often used to relate two terms of the same language. For our translation, the two terms are from different languages, related at a type from the source language. Benton and Hur (2009) prove correctness of compiler transformations. They use a step-indexed LR to relate a denotational semantics of the λ-calculus with recursion to configurations of a SECD-machine. Hur and Dreyer (2011) build on this idea to show equivalence between an expressive source language (polymorphic λ-calculus with references, existentials, and recursive types) and assembly language. Their biorthogonal, step-indexed Kripke LR does not directly relate the two languages but relies on abstract language specifications.

Our setting is different in that we consider a source language with support for overloading. Besides structured data and functions, we need to cover recursive interface values. This leads to some challenges to get the step index right (Sulzmann and Wehr, 2022).

Simulation or bisimulation (see e.g. Sumii and Pierce 2007) is another common technique for showing program equivalences. In our setting, using this technique amounts to proving that reduction and translation commutes: if source term e reduces to e' and translates to target term E, then e' translates to E' such that E reduces to E'' (potentially in several steps) with $E' = E''$. One challenge is that the two target terms E' and E'' are not necessarily syntactically equal but only semantically. In our setting, this might be the case if E' and E'' contain coercions for structural subtyping. Even if such coercions behave the same, their syntax might be different. With LR, we abstract away certain details of single step reductions, as we only compare values, not intermediate results. A downside of the LR is that getting the step index right is sometimes not trivial.

Paraskevopoulou and Grover (2021) combine simulation and an untyped, step-indexed LR (Acar et al., 2008) to relate the translation of a reduced expression (the E' from the preceding paragraph) with the reduction result of the translated expression (the E''). They use this technique to prove correctness of CPS transformations using small-step and big-step operational semantics. Resource invariants connect the number of steps a term and its translation might take, allowing them to prove that divergence and asymptotic runtime is preserved by the transformation. Our LR does not support resource invariants but includes a case for divergence directly.

7.4 Prior Work

Our own work published at APLAS (Sulzmann and Wehr, 2021) and MPC (Sulzmann and Wehr, 2022) laid the foundations for the dictionary-passing translation and its correctness proof of the present article. For the APLAS paper, we defined a dictionary-passing translation for Featherweight Go (FG, Griesemer et al., 2020), the non-generic variant of FGG. That translation is similar in spirit to the translation presented here, it supports type assertions but not generics. The APLAS paper includes a proof for the semantic equivalence between the source FG program and its translation. The result is, however, somewhat limited as semantic equivalence only holds for terminating programs whose translation is also known to terminate.

In the MPC paper, we addressed the aforementioned limitation by extending the proof of semantic equivalence to all possible outcomes of an FG program: termination, panic (failure of a dynamic type assertion), and divergence. The proof uses a logical relation similar to the one used here, but without support for generics. We have already shown more differences in Section 5.3.

7.5 Summary of Translations

The diagram in Figure 20 summarizes the existing translations by Griesemer et al. (OOPSLA 2020), by Ellis et al. (OOPSLA 2022), from our MPC 2022 paper (Sulzmann and Wehr, 2022), and from the article at hand. The three resulting target language programs P_{TL}, P'_{TL}, and P''_{TL} are semantically equivalent because all translations preserve the dynamic semantics. Each translation with $P_{\mathrm{FGG}^\star}$ as its source has different restrictions. OOPSLA 2022 requires the receiver struct of some method declaration to have exactly the same type bounds as the struct declaration itself. OOPSLA 2020 requires P_{FGG} to be monomorphizable, checked by a simple syntactic condition. The translation of this article does not support type assertions.

$$\begin{tikzcd}
 & P'_{\mathrm{FG}} \arrow[r, "\text{MPC 2022}"] & P'_{\mathrm{TL}} \\
P_{\mathrm{FGG}^\star} \arrow[ur, "\text{OOPSLA 2022}"] \arrow[r, "\text{OOPSLA 2020}"] \arrow[drr, "\text{this article}"'] & P_{\mathrm{FG}} \arrow[r, "\text{MPC 2022}"] & P_{\mathrm{TL}} \\
 & & P''_{\mathrm{TL}}
\end{tikzcd}$$

Fig. 20. Summary of translations. Arrows represent translations, P_ℓ is a program in language ℓ. Program $P_{\mathrm{FGG}^\star}$ is subject to certain restrictions, depending on the translation being performed.

8 Conclusion and Future Work

This article defined a type-directed dictionary-passing translation from Featherweight Generic Go (FGG) without type assertions to an extension of the untyped λ-calculus. The translation represents a value at the type of an interface as a combination of a concrete struct value with a dictionary for all methods of the interface. Bounded type parameters become extra function arguments in the target. These extra arguments are coercions from the instantiation of a type variable to its upper bound.

Every program in the image of the translation has the same dynamic semantics as its source program. Different translations of the same source program may result in syntactically different but equivalent target programs. The proof of semantic equivalence is based on a syntactic, step-indexed logical relation. The step-index ensures a well-founded definition of the relation in the presence of recursive interface types and recursive methods. We also reported on an implementation of the translation.

In this article, we relied on FGG as defined by Griesemer and coworkers (2020), without reconsidering design decisions. But our translation raises several questions with respect to the design of generics in FGG and more generally also in Go. For example, the translation clearly shows that type bounds in structs and interfaces have no operational meaning. Should we eliminate these type bounds? Or should we give them a meaning inspired by Haskell's type class mechanism? Further, a method declaration in full Go must reuse the type bounds of its struct and must be defined in the same package as the struct. Clearly, this limits extensibility and flexibility. Can we provide a more flexible design to solve the expression problem (Wadler, 1998) in Go, without resorting to unsafe type assertions? We would like to use the insights gained through this article to answer these and similar questions in future work.

A somewhat related point is performance. As explained earlier, generics in Go are compiled by monomorphization. This gives the best possible performance because the resulting code is specialized for each type argument. However, not all programs can be monomorphized and the increase in code size is often considered problematic. This raises another interesting question for future work. Could selective monomorphization or specialization offer a viable trade-off between performance, code size, and the ability to compile Go programs which are not monomorphizable?

A statically-typed target language typically offers more room for compiler optimization (Harper and Morrisett, 1995). Thus, another interesting direction for future work is a translation to a typed backend, for example System F (Girard, 1972; Reynolds, 1974).

The work presented here does not include type assertions (dynamic type casts), although FGG supports them. We omitted type assertions from our theory for two reasons: Firstly, type assertions are largely orthogonal to the dictionary-passing translation, so their inclusion would obscure the working of the translation. Secondly, type assertions would require some extra design choices to consider. In our implementation we construct the check for a type assertion at the same place as in the source program, relying on dynamic type-tag–passing for gathering all information necessary. But other approaches are possible. For example, one could construct downcast coercions at call-sites and pass these coercions around. The second option could make the treatment of type assertions more lightweight, but would require significant research in this direction.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable feedback and constructive suggestions, which greatly improved the quality of this paper.

Declaration of interests

The authors report no conflict of interest.

Supplementary material

For supplementary material for this article, please visit `https://github.com/skogsbaer/fgg-translate` and `http://doi.org/10.5281/zenodo.8147425`.

References

Acar, U. A., Ahmed, A. & Blume, M. (2008) Imperative self-adjusting computation. Proc. of POPL 2008. ACM.

Ahmed, A. (2006) Step-indexed syntactic logical relations for recursive and quantified types. Proc. of ESOP 2006. Springer-Verlag.

Alpern, B., Cocchi, A., Fink, S. J., Grove, D. & Lieber, D. (2001) Efficient implementation of Java interfaces: Invokeinterface considered harmless. Proc. of OOPSLA 2001. ACM.

Appel, A. W. & McAllester, D. A. (2001) An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.* **23**(5).

Benton, N. & Hur, C. (2009) Biorthogonality, step-indexing and compiler correctness. Proc. of ICFP 2009. ACM.

Bottu, G.-J., Xie, N., Marntirosian, K. & Schrijvers, T. (2019) Coherence of type class resolution. *Proc. ACM Program. Lang.* **3**(ICFP).

Bracha, G., Odersky, M., Stoutamire, D. & Wadler, P. (1998) Making the future safe for the past: Adding genericity to the java programming language. *SIGPLAN Not.* **33**(10), 183–200.

Canning, P., Cook, W., Hill, W., Olthoff, W. & Mitchell, J. C. (1989) F-bounded polymorphism for object-oriented programming. Proc. of FPCA 1989. ACM.

Crary, K. & Harper, R. (2007) Syntactic logical relations for polymorphic and recursive types. *Electron. Notes Theor. Comput. Sci.* **172**.

Dean, J., Chambers, C. & Grove, D. (1995) Selective specialization for object-oriented languages. Proc. of PLDI 1995. ACM.

Driesen, K. & Hölzle, U. (1996) The direct cost of virtual function calls in C++. Proc. of OOPSLA 1996. ACM.

Ellis, S., Zhu, S., Yoshida, N. & Song, L. (2022) Generic go to go: dictionary-passing, monomorphisation, and hybrid. *Proc. ACM Program. Lang.* **6**(OOPSLA2).

Emir, B., Kennedy, A., Russo, C. V. & Yu, D. (2006) Variance and generalized constraints for c$^{\#}$ generics. ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings. Springer.

GHC Team. (2021) *GHC User's Guide*.

GHC User's Guide. (2022) https://downloads.haskell.org/ghc/9.4.1/docs/users_guide/index.html.

Girard, J. (1972) *Interpretation fonctionnelle et elimination des coupures de l'arithmetique d'ordre superieur*. Thèse d'état. Université Paris 7.

Go Programming Language. (2022) https://golang.org.

Griesemer, R., Hu, R., Kokke, W., Lange, J., Taylor, I. L., Toninho, B., Wadler, P. & Yoshida, N. (2020) Featherweight Go. *Proc. ACM Program. Lang.* **4**(OOPSLA).

Hall, C. V., Hammond, K., Peyton Jones, S. L. & Wadler, P. L. (1996) Type classes in Haskell. *ACM Trans. Program. Lang. Syst.* **18**(2), 109–138.

Harper, R. & Morrisett, J. G. (1995) Compiling polymorphism using intensional type analysis. Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995. ACM Press.

Haskell Programming Language. (2022) https://www.haskell.org.

Hur, C. & Dreyer, D. (2011) A Kripke logical relation between ML and Assembly. Proc. of POPL 2011. ACM.

Igarashi, A., Pierce, B. C. & Wadler, P. (2001) Featherweight java: a minimal core calculus for java and GJ. *ACM Trans. Program. Lang. Syst.* **23**(3).

Kennedy, A. & Syme, D. (2001) Design and implementation of generics for the .net common language runtime. Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001. ACM.

Läufer, K. (1996) Type classes with existential types. *Journal of Functional Programming*. **6**(3).

Marlow, S. (2010) Haskell 2010 language report. https://www.haskell.org/onlinereport/haskell2010/.

Mitchell, J. C. (1996) *Foundations for programming languages*. Foundation of computing series. MIT Press.

Mitchell, J. C. & Plotkin, G. D. (1988) Abstract types have existential type. *ACM Trans. Program. Lang. Syst.* **10**(3).

Morris, J. G. (2014) A simple semantics for Haskell overloading. Proc. of Haskell 2014. ACM.

Ohori, A. & Ueno, K. (2021) A compilation method for dynamic typing in ML. Programming Languages and Systems - 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17-18, 2021, Proceedings. Springer.

Paraskevopoulou, Z. & Grover, A. (2021) Compiling with continuations, correctly. *Proc. ACM Program. Lang.* **5**(OOPSLA).

Pierce, B. (2004) *Advanced Topics in Types and Programming Languages*. The MIT Press.

Pitts, A. M. (1998) Existential types: Logical relations and operational equivalence. Proc. of ICALP 1998. Springer.

Racket Programming Lanugage. (2022) https://racket-lang.org.

Reynolds, J. C. (1974) Towards a theory of type structure. Programming Symposium, Proceedings Colloque sur la Programmation. Springer-Verlag. pp. 408–423.

Reynolds, J. C. (1991) The coherence of languages with intersection types. Theoretical Aspects of Computer Software, International Conference TACS '91, Sendai, Japan, September 24-27, 1991, Proceedings. Springer.

Skorstengaard, L. (2019) An introduction to logical relations. http://arxiv.org/abs/1907.11133.

Statman, R. (1985) Logical relations and the typed lambda-calculus. *Inf. Control.* **65**(2/3).

Sulzmann, M. & Wehr, S. (2021) A dictionary-passing translation of Featherweight Go. Proc. of APLAS 2021. Springer.

Sulzmann, M. & Wehr, S. (2022) Semantic preservation for a type directed translation scheme of Featherweight Go. Proc. of MPC 2022. Springer.

Sumii, E. & Pierce, B. C. (2007) A bisimulation for type abstraction and recursion. *J. ACM*. **54**(5).

Tait, W. W. (1967) Intensional interpretations of functionals of finite type I. *J. Symb. Log.* **32**(2), 198–212.

Taylor, I. L. & Griesemer, R. (2021) Type parameters proposal. https://go.googlesource.com/proposal/+/refs/heads/master/design/43651-type-parameters.md.

Thatte, S. R. (1994) Semantics of type classes revisited. Proc. of LISP 1994. ACM.

Thiemann, P. & Wehr, S. (2008) Interface types for Haskell. Proc. of APLAS 2008. Springer.

Wadler, P. (1998) The expression problem. Posted on the Java Genericity mailing list. http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.

Wadler, P. & Blott, S. (1989) How to make ad-hoc polymorphism less ad hoc. Proc. of POPL 1989. ACM.

A Proofs

A.1 Deterministic Evaluation in FGG$^-$ and TL

Lemma A.1.1 (Deterministic evaluation in FGG$^-$)**.** *If $e \longrightarrow e'$ and $e \longrightarrow e''$ then $e' = e''$. If $E \longrightarrow E'$ and $E \longrightarrow E''$ then $E' = E''$.*

Proof. We first state and prove three sublemmas:

(a) If $e = \mathcal{E}_1[\mathcal{E}_2[e']]$ then there exists $\mathcal{E}_3$ with $e = \mathcal{E}_3[e']$. The proof is by induction on $\mathcal{E}_1$.

(b) If $e \longrightarrow e'$ then there exists a derivation of $e \longrightarrow e'$ that ends with at most one consecutive application of rule FG-CONTEXT. The proof is by induction on the derivation of $e \longrightarrow e'$. From the IH, we know that this derivation ends with at most *two* consecutive applications of rule FG-CONTEXT. If there are two such consecutive applications, (a) allow us to merge the two evaluation contexts involved, so that we need only one consecutive application of FG-CONTEXT.

(c) We call an FGG$^-$ expression *directly reducible* if it reduces but not by rule FG-CONTEXT. If e_1 and e_2 are now directly reducible and $\mathcal{E}_1[e_1] = \mathcal{E}_2[e_2]$ then $\mathcal{E}_1 = \mathcal{E}_2$ and $e_1 = e_2$. For the proof, we first note that $\mathcal{E}_1 = \square$ iff $\mathcal{E}_2 = \square$. This holds because directly reducible expressions have no inner redexes. The rest of the proof is then a straightforward induction on $\mathcal{E}_1$.

Now assume $e \longrightarrow e'$ and $e \longrightarrow e''$. By (b) we may assume that both derivations ends with at most one consecutive application of rule FG-CONTEXT. It is easy to see (as values do not reduce) that both derivations must end with the same rule. If this rule is FG-FIELD, then $e' = e''$ by restrictions FGG-UNIQUE-STRUCTS and FGG-DISTINCT-FIELDS. If this rule is FG-CALL, then $e' = e''$ by FGG-UNIQUE-METHOD-DEFS. If the rule is FG-CONTEXT, we have the following situation with $R_1 \neq$ FG-CONTEXT and $R_2 \neq$ FG-CONTEXT:

$$\text{FG-CONTEXT}\ \dfrac{R_1\ \dfrac{}{g_1 \longrightarrow g_1'}}{\underbrace{\mathcal{E}_1[g_1]}_{=e} \longrightarrow \underbrace{\mathcal{E}_1[g_1']}_{=e'}} \qquad\qquad \dfrac{\dfrac{}{g_2 \longrightarrow g_2'}\ R_2}{\underbrace{\mathcal{E}_2[g_2]}_{=e} \longrightarrow \underbrace{\mathcal{E}_2[g_2']}_{=e''}}\ \text{FG-CONTEXT}$$

As neither R_1 nor R_2 are FG-CONTEXT, we know that g_1 and g_2 are directly reducible. Thus, with $\mathcal{E}_1[g_1] = \mathcal{E}_2[g_2]$ and (c) we get $\mathcal{E}_1 = \mathcal{E}_2$ and $g_1 = g_2$. With R_1 and R_2 not being FG-CONTEXT, we have $g_1' = g_2'$, so $e' = e''$ as required. ■

Lemma A.1.2 (Deterministic evaluation in TL)**.** *If $E \longrightarrow_\mu E'$ and $E \longrightarrow_\mu E''$ then $E' = E''$. Further, if $E \longrightarrow E'$ and $E \longrightarrow E''$ then $E' = E''$.*

Proof. We first prove the first implication of the lemma

$$\forall E, E', E'', \mu \,.\, E \longrightarrow_\mu E' \wedge E \longrightarrow_\mu E'' \implies E' = E'' \tag{1}$$

There are three sublemmas, analogously to the proof of Lemma A.1.1.

(a) If $E = \mathcal{R}_1[\mathcal{R}_2[E']]$ then there exists $\mathcal{R}_3$ with $E = \mathcal{R}_3[E']$.

(b) If $E \longrightarrow_\mu E'$ then there exists a derivation of $E \longrightarrow_\mu E'$ that ends with at most one consecutive application of rule TL-CONTEXT.

(c) We call a target-language expression *directly reducible* if it reduces but not by rule TL-CONTEXT. If E_1 and E_2 are now directly reducible and $\mathcal{R}_1[E_1] = \mathcal{R}_2[E_2]$ then $\mathcal{R}_1 = \mathcal{R}_2$ and $E_1 = E_2$.

The proofs of these lemmas are similar to the proofs of the sublemmas in Lemma A.1.1. Then (1) follows with reasoning similar to the proof of Lemma A.1.1. If the derivations of $E \longrightarrow_\mu E'$ and $E \longrightarrow_\mu E''$ both end with rule TL-CASE, then our assumption that the constructors of a case-expression are distinct ensures determinacy.

The second claim of the lemma ($E \longrightarrow E'$ and $E \longrightarrow E''$ imply $E' = E''$) then follows directly from (1). Our assumption that the variables of a top-level let-binding are distinct ensures that the substitution μ built from the top-level let-bindings is well-defined. ■

A.2 Preservation of Static Semantics

Proof of Lemma 5.1.1. We prove (a) and (b) by case distinctions on the last rule of the given derivations; (c) and (d) follow by induction on the derivations, using (a) and (b). Claim (e) then follows by examining the typing rules, using (c) and (d). ■

A.3 Preservation of Dynamic Semantics

Convention A.3.1. We omit μ from reductions in the target language, writing $E \longrightarrow E'$ instead of $E \longrightarrow_\mu E'$.

Definition A.3.2. We make use of some extra metavariables and notations.

- Φ, Ψ denote formal type parameters $\overline{\alpha\ \tau_I}$.
- $\hat{\Phi}$ denotes the type variables of Φ; that is, if $\Phi = \overline{\alpha\ \tau_I}$ then $\hat{\Phi} = \overline{\alpha}$.

- ϕ, ψ denote actual type arguments $\overline{\tau}$.
- $M ::= [\Phi](\overline{x\ \tau})\ \tau$ denotes the type-part of a method signature R.
- L denotes a type literal **struct** $\{\overline{f\ \tau}\}$ or **interface** $\{\overline{R}\}$.
- $\Phi \mapsto \phi : \eta$ create a type substitution η form type parameters Φ and arguments ϕ. It is defined like this: $\overline{\alpha\ \tau}^n \mapsto \overline{\sigma}^n : \langle\overline{\alpha_i \mapsto \sigma_i}^n\rangle$

A.3.1 The Logical Relation

Lemma A.3.3 (Monotonicity for expressions)**.** *Assume $k' \leq k$. If $e \approx E \in [\![\tau]\!]_k$ then $e \approx E \in [\![\tau]\!]_{k'}$. If $v \equiv V \in [\![\tau]\!]_k$ then $v \equiv V \in [\![\tau]\!]_{k'}$.*

Proof. We proceed by induction on (k, s) where s is the combined size of v, V.
Case distinction on the last rule used in the two derivations.

- *Case* rule EQUIV-EXP: We label the two implications in the premise of the rule as (a) and (b).
  (a) Assume $k'' < k'$ and $e \longrightarrow^{k''} u$ for some value u. From $e \approx E \in [\![\tau]\!]_k$
  $$\exists U\,.\,E \longrightarrow^* U$$
  $$u \equiv U \in [\![\tau]\!]_{k-k''} \tag{1}$$
  If $k = k'$ then $u \equiv U \in [\![\tau]\!]_{k'-k''}$. Otherwise, $k' - k'' < k - k''$, so the IH (induction hypothesis) applied to (1) also yields $u \equiv U \in [\![\tau]\!]_{k'-k''}$. This proves implication (a).
  (b) Assume $k'' < k'$ and $e \longrightarrow^{k''} e'$ and $\mathsf{diverge}(e')$. Then we get with $e \approx E \in [\![\tau]\!]_k$ and $k' \leq k$ that $\mathsf{diverge}(E)$.
- *Case* rule EQUIV-STRUCT: Follows from IH.
- *Case* rule EQUIV-IFACE: Obvious.

End case distinction. ■

Lemma A.3.4 (Monotonicity for method dictionaries)**.** *If $\langle x, \tau_S, R, e\rangle \approx V \in [\![R']\!]_k$ and $k' \leq k$ then $\langle x, \tau_S, R, e\rangle \approx V \in [\![R']\!]_{k'}$.*

Proof. Obvious. ■

Lemma A.3.5 (Monotonicity for type parameters)**.** *If $\phi \approx V \in [\![\Phi]\!]_k$ and $k' \leq k$ then $\phi \approx V \in [\![\Phi]\!]_{k'}$.*

Proof. Obvious. ■

Lemma A.3.6 (Monotonicity for method declarations)**.** *Assume declaration $D =$ **func** $(x\ t_S[\Phi])\ R\ \{$**return** $e\}$ and $k' \leq k$. If $D \approx_k X$ then $D \approx_{k'} X$.*

Proof. Obvious. ■

Lemma A.3.7 (Monotonicity for programs)**.** *If $\overline{D} \approx_k \mu$ and $k' \leq k$ then $\overline{D} \approx_{k'} \mu$.*

Proof. Follows from Lemma A.3.6. ■

A.3.2 Equivalence Between Source and Translation

Proof of Lemma 5.2.2. Straightforward.

Proof of Lemma 5.2.3. We label the two implications in the premise of rule EQUIV-EXP with (a) and (b).

(a) Assume $k' < k+1$ and $e_2 \longrightarrow^{k'} v$. Then by Lemma A.1.1 $e_2 \longrightarrow e \longrightarrow^{k'-1} v$. Noting that $k'-1 < k$ we get with the assumption $e \approx E \in [\![\tau]\!]_k$

$$\exists V \,.\, E \longrightarrow^{*} V \wedge v \equiv V \in [\![\tau]\!]_{k+1-k'}$$

But this is exactly what is needed to prove implication (a) for $e_2 \approx E \in [\![\tau]\!]_{k+1}$.

(b) Assume $k' < k+1$ and $e_2 \longrightarrow^{k'} e'$ and $\mathsf{diverge}(e')$. Then by Lemma A.1.1 $e_2 \longrightarrow e \longrightarrow^{k'-1} e'$. Noting that $k'-1 < k$ we get with the assumption $e \approx E \in [\![\tau]\!]_k$ that $\mathsf{diverge}(E)$. This proves implication (b). ■

Lemma A.3.8 (Expression equivalence implies value equivalence). *If $k \geq 1$ and $v \approx V \in [\![\tau]\!]_k$ then $v \equiv V \in [\![\tau]\!]_k$.*

Proof. From the first implication of rule EQUIV-EXP we get for $k' = 0 < k$ and with $v \longrightarrow^{0} v$ that $\exists V' \,.\, V \longrightarrow^{*} V' \wedge v \equiv V' \in [\![\tau]\!]_k$. But V is already a value, so $V' = V$. ■

Lemma A.3.9 (Value equivalence implies expression equivalence). *If $v \equiv V \in [\![\tau]\!]_k$ then $v \approx V \in [\![\tau]\!]_k$ for any k.*

Proof. We have $v \longrightarrow^{0} v$, so we get the first implication of rule EQUIV-EXP by setting $E = V$ and by assumption $v \equiv V \in [\![\tau]\!]_k$. The second implication holds vacuously because values do not diverge. ■

Lemma A.3.10. *Assume* $\textbf{func}\ (x\ t_S[\Phi])\ mM\ \{\textbf{return}\ e\} \approx_k X$. *Then the following holds:*

$$\forall k' < k, \phi, W \,.\, \phi \approx W \in [\![\Phi]\!]_{k'} \implies \Phi \mapsto \phi : \eta \wedge$$
$$\langle x, t_S[\phi], \eta m M, \eta e\rangle \approx \lambda(Y_1, Y_2, Y_3)\,.\,X\ (W, Y_1, Y_2, Y_3) \in [\![\eta m M]\!]_{k'}$$

Proof. Let $M = [\Phi'](\overline{x_i\ \tau_i}^{n})\ \tau$ and assume for any k', ϕ, W

$$k' < k \tag{1}$$

$$\phi \approx W \in [\![\Phi]\!]_{k'} \tag{2}$$

Obviously

$$\Phi \mapsto \phi : \eta \tag{3}$$

To show that

$$\langle x, t_S[\phi], \eta m [\Phi'](\overline{x_i\ \tau_i}^{n})\ \tau, \eta e\rangle \approx \lambda(Y_1, Y_2, Y_3)\,.\,X\ (W, Y_1, Y_2, Y_3) \in [\![\eta m M]\!]_{k'}$$

holds, we assume the left-hand side of the implication in the premise of rule EQUIV-METHOD-DICT-ENTRY for some $k'', \phi', W', u, U, \overline{v}^n, \overline{V}^n$:

$$k'' \leq k' \tag{4}$$

$$\eta\Phi' \mapsto \phi' : \eta' \tag{5}$$

$$\phi' \approx W' \in [\![\eta\Phi']\!]_{k''} \tag{6}$$

$$u \approx U \in [\![t_S[\phi]]\!]_{k''} \tag{7}$$

$$(\forall i \in [n]) \,.\, v_i \approx V_i \in [\![\eta'\eta\tau_i]\!]_{k''} \tag{8}$$

We then need to prove (9) to show the overall goal.

$$\theta\eta'\eta e \approx V\ (U, W', (\overline{V}^n)) \in [\![\eta'\eta\tau]\!]_{k''} \tag{9}$$

$$\theta = \langle x \mapsto u, \overline{x_i \mapsto v_i}^n \rangle$$

$$V = \lambda(\overline{Y}^3)\,.\,X\ (W, Y_1, Y_2, Y_3) \tag{10}$$

Let $\Phi = \overline{\alpha\,\sigma}^p$, $\Phi' = \overline{\beta\,\sigma'}^q$, $\phi = \overline{\sigma''}^p$, $\phi' = \overline{\sigma'''}^q$. Then by (3) and (5)

$$\eta = \langle \overline{\alpha_i \mapsto \sigma''_i}^p \rangle \tag{11}$$

$$\eta' = \langle \overline{\beta_i \mapsto \sigma'''_i}^q \rangle \tag{12}$$

Define

$$\Psi := \overline{\alpha_i\,\sigma_i}^p\,\overline{\beta_i\,\sigma'_i}^q$$

$$\phi'' := \overline{\sigma''}^p\,\overline{\sigma'''}^q$$

$$\eta'' := \langle \overline{\alpha_i \mapsto \sigma''_i}^p\,\overline{\beta_i \mapsto \sigma'''_i}^q \rangle \tag{13}$$

Then

$$\Psi \mapsto \phi'' : \eta'' \tag{14}$$

The $\overline{\beta}^q$ are sufficiently fresh, so $\mathsf{ftv}(\overline{\sigma''}^p) \cap \overline{\beta}^q = \emptyset$. Hence by (11), (12), (13)

$$\eta'\eta\overline{\sigma'_i}^q = \eta''\overline{\sigma'_i}^q \qquad \eta'\eta\overline{\tau_i}^n = \eta''\overline{\tau_i}^n \qquad \eta'\eta\tau = \eta''\tau \qquad \eta'\eta e = \eta'' e \tag{15}$$

We have from (2) and (6)

$$W = (\overline{W}^p) \tag{16}$$

$$W' = (\overline{W'}^q) \tag{17}$$

We now prove

$$\phi'' \approx (\overline{W}^p, \overline{W'}^q) \in [\![\Psi]\!]_{k''} \tag{18}$$

by verifying the implication in the premise of rule EQUIV-BOUNDED-TYPARAMS. We consider two cases for every $\ell \leq k''$.
Case distinction whether i in $[p]$ or in $[q]$.

- *Case* $i \in [p]$: We need to prove $\forall u, U\,.\, u \approx U \in [\![\sigma''_i]\!]_\ell \implies u \approx W_i\ U \in [\![\eta''\sigma_i]\!]_\ell$. From (2) we get with $u \approx U \in [\![\sigma''_i]\!]_\ell$ and $\ell \leq k'' \overset{(4)}{\leq} k'$ that $u \approx W_i\ U \in [\![\eta\sigma_i]\!]_\ell$. By assumption 5.2.1 $\mathsf{ftv}(\sigma_i) \subseteq \overline{\alpha}^p$, so $\eta''\sigma_i = \eta\sigma_i$ by (11) and (13).

- *Case* $i \in [q]$: We need to prove $\forall u, U \,.\, u \approx U \in [\![\sigma_i''']\!]_\ell \implies u \approx W_i'\, U \in [\![\eta''\sigma_i']\!]_\ell$. From (6) we get with $u \approx U \in [\![\sigma_i''']\!]_\ell$ that $u \approx W_i'\, U \in [\![\eta'\eta\sigma_i']\!]_\ell$. Also, $\eta'\eta\sigma_i' = \eta''\sigma_i'$ by (15).

End case distinction. This finishes the proof of (18).

From (7) and (13) we have

$$u \approx U \in [\![\eta'' t_S[\overline{\alpha}^p]]\!]_{k''} \tag{19}$$

From (8) and (15)

$$v_i \approx V_i \in [\![\eta''\tau_i]\!]_{k''} \tag{20}$$

From the assumption $\textbf{func}\,(x \;\; t_S[\Phi])\, mM\, \{\textbf{return}\, e\} \approx_k X$, we can invert rule EQUIV-METHOD-DECL. Noting that $k'' \overset{(4)}{\leq} k' \overset{(1)}{<} k$ and that (14), (18), (19), (20) give us the left-hand side of the implication in the premise of the rule, we get by the right-hand side of the implication

$$\theta\eta'' e \approx X\ ((\overline{W}^p), U, (\overline{W'}^q), (\overline{V}^n)) \in [\![\eta''\tau]\!]_{k''}$$

With (15), (16), (17)

$$\theta\eta'\eta e \approx X\ (W, U, W', (\overline{V}^n)) \in [\![\eta'\eta\tau]\!]_{k''} \tag{21}$$

We have by (10)

$$V\ (U, W', (\overline{V}^n)) = (\lambda(\overline{Y}^3)\,.\,X\ (W, Y_1, Y_2, Y_3))\ (U, W', (\overline{V}^n))$$

so

$$V\ (U, W', (\overline{V}^n)) \longrightarrow^* X\ (W, U, W', (\overline{V}^n))$$

Thus, with (21) and Lemma 5.2.2

$$\theta\eta'\eta e \approx V\ (U, W', (\overline{V}^n)) \in [\![\eta'\eta\tau]\!]_{k''}$$

as required to prove (9). ■

Definition A.3.11 (Domain)**.** We write $\mathsf{dom}(\cdot)$ for the domain of a substitution η, θ, ρ or μ, of a type environment Δ, of a value environment Γ, or some type parameters Φ.

Definition A.3.12 (Free variables)**.** We write $\mathsf{fv}(\cdot)$ for the set of free term variables, and $\mathsf{ftv}(\cdot)$ for the set of free type variables.

Lemma A.3.13 (Subtyping preserves equivalence)**.** *Let $\overline{D} \approx_k \mu$. Assume $\Delta \vdash_{\mathsf{coerce}} \tau <: \sigma \rightsquigarrow V$ and $\eta \approx \rho \in [\![\Delta]\!]_k$ and $e \approx E \in [\![\eta\tau]\!]_k$. Then $e \approx (\rho V)\ E \in [\![\eta\sigma]\!]_k$.*

We prove Lemma A.3.13 together with the following two lemmas.

Lemma A.3.14. *Assume $\overline{D} \approx_k \mu$ and $\eta \approx \rho \in [\![\Delta]\!]_k$. Let $\langle R, V\rangle \in \mathsf{methods}(\Delta, t_S[\phi])$ and define $U = \lambda(\overline{Y}^3)\,.\,X_{m,t_S}\ (\rho V, Y_1, Y_2, Y_3)$. Then we have for all $k' < k$ that $\mathsf{methodLookup}(m, \eta t_S[\phi]) \approx U \in [\![\eta R]\!]_{k'}$.*

Lemma A.3.15 (Substitution preserves equivalence)**.** *Assume* $\overline{D} \approx_k \mu$ *and* $\eta \approx \rho \in [\![\Delta]\!]_k$. *If* $\Delta \vdash_{\mathsf{subst}} \Phi \mapsto \phi : \eta' \rightsquigarrow V$ *then* $\eta\phi \approx \rho V \in [\![\eta\Phi]\!]_k$.

Proof of Lemmas A.3.13, A.3.14, and A.3.15. We show the three lemmas by induction on the combined height of the derivations for $\Delta \vdash_{\mathsf{coerce}} \tau <: \sigma \rightsquigarrow V$ and $\langle R, V\rangle \in \mathsf{methods}(\Delta, t_S[\phi])$ and $\Delta \vdash_{\mathsf{subst}} \Phi \mapsto \phi : \eta' \rightsquigarrow V$.

We start with the proof for *Lemma A.3.13*. We have from the assumptions

$$e \approx E \in [\![\eta\tau]\!]_k \tag{1}$$

Assume $k' < k$ and $e \longrightarrow^{k'} e'$. The second implication in the premise of rule EQUIV-EXP holds obviously, because with $\mathsf{diverge}(e')$ we get from (1) $\mathsf{diverge}(E)$, so also $\mathsf{diverge}((\rho V)\ E)$.

Thus, we only need to prove the first implication. Assume that $e' = v$ for some value v. Then via (1) for some U

$$E \longrightarrow^* U \tag{2}$$

$$v \equiv U \in [\![\eta\tau]\!]_{k-k'} \tag{3}$$

We then need to verify that $(\rho V)\ U \longrightarrow^* U'$ for some U' with $v \equiv U' \in [\![\eta\sigma]\!]_{k-k'}$. In fact, $k' < k$, so with Lemma A.3.8 it suffices to show that $v \approx U' \in [\![\eta\sigma]\!]_{k-k'}$.
Case distinction on the last rule in the derivation of $\Delta \vdash_{\mathsf{coerce}} \tau <: \sigma \rightsquigarrow V$.

- *Case* COERCE-TYVAR:

$$\frac{X \text{ fresh} \qquad (a : \tau_I) \in \Delta \qquad \Delta \vdash_{\mathsf{coerce}} \tau_I <: \sigma \rightsquigarrow W}{\Delta \vdash_{\mathsf{coerce}} \underset{\tau}{\overset{}{\underset{\shortparallel}{\alpha}}} <: \sigma \rightsquigarrow \underbrace{\lambda X\,.\,W\ (X_\alpha\ X)}_{=V}}$$

  Our goal to show is

$$e \approx (\rho X_\alpha)\ U \in [\![\eta\tau_I]\!]_k \tag{4}$$

  With (4) and the IH for Lemma A.3.13 we then get

$$e \approx (\rho W)\ ((\rho X_\alpha)\ U) \in [\![\eta\sigma]\!]_k$$

  Then, with $e \longrightarrow^{k'} v$, we get $(\rho V)\ U \longrightarrow (\rho W)\ ((\rho X_\alpha)\ U) \longrightarrow^* U'$ for some U' with $v \equiv U' \in [\![\eta\sigma]\!]_{k-k'}$.
  We now prove (4). From the assumption $\eta \approx \rho \in [\![\Delta]\!]_k$ we have

$$\Delta = \overline{\alpha_i : \tau_i}^n$$

$$\eta = \langle \overline{\alpha_i \mapsto \sigma_i}^n \rangle$$

$$\rho = \langle \overline{X_{\alpha_i} \mapsto V_i}^n \rangle$$

$$\overline{\sigma}^n \approx \overline{V}^n \in [\![\overline{\alpha_i\ \tau_i}^n]\!]_k \tag{5}$$

  such that $\alpha = \alpha_j$ and $\tau_I = \tau_j$ for some $j \in [n]$. Inverting rule EQUIV-BOUNDED-TYPARAMS on (5) yields

$$\forall k'' \le k, w, W'\,.\, w \approx W' \in [\![\sigma_j]\!]_{k''} \implies w \approx V_j\ W' \in [\![\eta\tau_j]\!]_{k''} \tag{6}$$

From (3) by $\eta\tau = \sigma_j$ then $v \equiv U \in [\![\sigma_j]\!]_{k-k'}$. Thus with (6) and Lemma A.3.9

$$v \approx V_j\ U \in [\![\eta\tau_j]\!]_{k-k'}$$

With Lemma 5.2.3 and $e \longrightarrow^{k'} v$ then

$$e \approx V_j\ U \in [\![\eta\tau_j]\!]_k$$

But $\tau_j = \tau_I$ and $V_j = \rho X_\alpha$, so this proves (4).

- *Case* COERCE-STRUCT-IFACE:

$$\frac{\begin{array}{lr} X, Y_1, Y_2, Y_3 \text{ fresh} & \\ \textbf{type } t_I[\Phi] \textbf{ interface } \{\overline{mM}^n\} \in \overline{D} & (7) \\ \Phi \mapsto \phi : \eta' & (8) \\ \langle \eta'(m_i M_i), V_i \rangle \in \mathsf{methods}(\Delta, t_S[\psi]) & (9) \\ V_i' = \lambda(Y_1, Y_2, Y_3).X_{m_i, t_S}\ (V_i, Y_1, Y_2, Y_3) \quad (\forall i \in [n]) & (10) \end{array}}{\Delta \vdash_{\mathsf{coerce}} \underset{\tau}{\underset{\shortparallel}{t_S[\psi]}} <: \underset{\sigma}{\underset{\shortparallel}{t_I[\phi]}} \rightsquigarrow \underbrace{\lambda X.(X, (\overline{V_i'}^n))}_{=V}}$$

Hence $(\rho V)\ U \longrightarrow (U, \rho(\overline{V'}^n))$ and $U' := (U, \rho(\overline{V'}^n))$ is a value. We now want to show $v \approx U' \in [\![\eta\sigma]\!]_{k-k'}$ via rule EQUIV-IFACE. Define the σ_S in the premise of EQUIV-IFACE as $\eta\tau = t_S[\eta\psi]$.
The first premise of EQUIV-IFACE

$$\forall k_1 < k - k'.\ v \equiv U \in [\![\eta\tau]\!]_{k_1} \tag{11}$$

follows from (3) and Lemma A.3.3. From (7) and (8) we get with $\sigma = t_I[\phi]$ the second premise as

$$\mathsf{methods}(\eta\sigma) = \eta\eta'\overline{mM}^n \tag{12}$$

We next prove the third premise of EQUIV-IFACE. Pick some $j \in [n]$ and $k_2 < k - k'$. With the assumptions $\overline{D} \approx_k \mu$ and $\eta \approx \rho \in [\![\Delta]\!]_k$ and with (9), (10), and the IH for Lemma A.3.14 we get

$$\mathsf{methodLookup}(m_j, \eta t_S[\psi]) \approx \rho V_j' \in [\![\eta\eta' m_j M_j]\!]_{k_2} \tag{13}$$

(11), (12), (13), and the definition of U' are the pieces required to derive $v \approx U' \in [\![\eta\sigma]\!]_{k-k'}$ via rule EQUIV-IFACE.

- *Case* COERCE-IFACE-IFACE:

$$\frac{\begin{array}{lr} Y, \overline{X}^n \text{ fresh} \qquad \pi : [q] \longrightarrow [n] \text{ total} & \\ \textbf{type } t_I[\Phi_1] \textbf{ interface } \{\overline{R}^n\} \in \overline{D} & (14) \\ \textbf{type } u_I[\Phi_2] \textbf{ interface } \{\overline{R'}^q\} \in \overline{D} & (15) \\ \Phi_1 \mapsto \phi_1 : \eta_1 \qquad \Phi_2 \mapsto \phi_2 : \eta_2 & \\ \eta_2 R_i' = \eta_1 R_{\pi(i)} \quad (\forall i \in [q]) & (16) \end{array}}{\Delta \vdash_{\mathsf{coerce}} \underset{\tau}{\underset{\shortparallel}{t_I[\phi_1]}} <: \underset{\sigma}{\underset{\shortparallel}{u_I[\phi_2]}} \rightsquigarrow \underbrace{\lambda(Y, (\overline{X}^n)).(Y, (X_{\pi(1)}, \ldots, X_{\pi(q)}))}_{=V\quad(17)}}$$

As $\eta\tau = \eta t_I[\phi_1]$ is an interface type, we get from (3) by inverting rule EQUIV-IFACE for some $W, \sigma_S, \overline{W}^n$ that

$$\forall k_1 < k - k' \,.\, v \approx W \in [\![\sigma_S]\!]_{k_1} \tag{18}$$

$$\mathsf{methods}(\eta\tau) = \eta\eta_1\overline{R}^n = \overline{mM}^n \tag{19}$$

$$\forall i \in [n], k_2 < k - k' \,.\, \mathsf{methodLookup}(m_i, \sigma_s) \approx W_i \in [\![m_i M_i]\!]_{k_2} \tag{20}$$

$$U = (W, (\overline{W}^n)) \tag{21}$$

Our goal is to show $(\rho V)\ U \longrightarrow^* U'$ for some U' with $v \approx U' \in [\![\eta\sigma]\!]_{k-k'}$. Via (17) and (21)

$$(\rho V)\ U = V\ U \longrightarrow^* (W, (W_{\pi(1)}, \ldots, W_{\pi(q)})) =: U' \tag{22}$$

From (14), (15), (16), and (19) we have

$$\begin{aligned}\overline{m'M'}^q = \eta\eta_2\overline{R'}^q &= \eta\eta_1 R_{\pi(1)}, \ldots, \eta\eta_1 R_{\pi(q)} \\ &= m_{\pi(1)}M_{\pi(1)}, \ldots, m_{\pi(q)}M_{\pi(q)}\end{aligned} \tag{23}$$

$$\mathsf{methods}(\eta\sigma) = \{\overline{m'M'}^q\} \tag{24}$$

Pick $j \in [q]$. Then via (23)

$$\mathsf{methodLookup}(m'_j, \sigma_S) = \mathsf{methodLookup}(m_{\pi(j)}, \sigma_s)$$

Hence with (20) and (23)

$$\forall j \in [q], k_2 < k - k' \,.\, \mathsf{methodLookup}(m'_j, \sigma_S) \approx W_{\pi(j)} \in [\![m'_j M'_j]\!]_{k_2} \tag{25}$$

With (18), (25), (24) and the definition of U' in (22) we then get by applying rule EQUIV-IFACE $v \approx U' \in [\![\eta\sigma]\!]_{k-k'}$ and with (22) also $(\rho V)\ U \longrightarrow^* U'$.

End case distinction on the last rule in the derivation of $\Delta \vdash_{\mathsf{coerce}} \tau <: \sigma \rightsquigarrow V$.

This finishes the proof of Lemma A.3.13.

We next prove Lemma A.3.14. By inverting rule METHODS-STRUCT for the assumption $\langle R, V\rangle \in \mathsf{methods}(\Delta, t_S[\phi])$ we get

$$\textbf{func}\ (x\ t_S[\Phi])\ mM\ \{\textbf{return}\ e\} \in \overline{D} \tag{26}$$

$$\Delta \vdash_{\mathsf{subst}} \Phi \mapsto \phi : \eta' \rightsquigarrow V \tag{27}$$

$$R = \eta' mM \tag{28}$$

Inverting (27) yields

$$\begin{gathered}\Phi = \overline{\alpha\ \tau}^n \\ \eta' = \langle \overline{\alpha_i \mapsto \sigma_i}^n \rangle \\ \phi = \overline{\sigma}^n \\ \Delta \vdash_{\mathsf{coerce}} \sigma_i <: \eta'\tau_i \rightsquigarrow V_i \quad (\forall i \in [n]) \\ V = (\overline{V}^n)\end{gathered}$$

Define $\eta'' = \langle \overline{\alpha_i \mapsto \eta\sigma_i}^n \rangle$. Then by rule METHOD-LOOKUP and (26)

$$\mathsf{methodLookup}(m, \eta t_S[\phi]) = \langle x, \eta t_S[\phi], \eta'' mM, \eta'' e\rangle \tag{29}$$

By assumption 5.2.1, the $\overline{\alpha}^n$ can be assumed to be fresh, $\mathsf{ftv}(\Phi) \subseteq \overline{\alpha}^n$, and $\eta\Phi = \Phi$. Applying the IH for Lemma A.3.15 on (27) yields $\eta\phi \approx \rho V \in [\![\eta\Phi]\!]_k$.

$$\eta\phi \approx \rho V \in [\![\Phi]\!]_k \tag{30}$$

From the assumption $\overline{D} \approx_k \mu$ we get with (26)

$$\textbf{func}\ (x\ t_S[\Phi])\ mM\ \{\textbf{return}\ e\} \approx_k X_{m,t_S}$$

Then for any $k' < k$ by Lemma A.3.10, where (30) and Lemma A.3.5 give the left-hand side of the implication

$$\begin{gathered}\Phi \mapsto \eta\phi : \eta'' \\ \langle x, \eta t_S[\phi], \eta'' mM, \eta'' e\rangle \approx \\ \underbrace{\lambda(\overline{Y}^3)\,.\,X_{m,t_S}\ (\rho V, Y_1, Y_2, Y_3)}_{=U} \in [\![\eta'' mM]\!]_{k'}\end{gathered} \tag{31}$$

We have $\eta R \overset{(28)}{=} \eta\eta' mM = \eta'' mM$, where the last equality holds because $\mathsf{ftv}(mM) \subseteq \overline{\alpha}^n$ and $\overline{\alpha}^n$ fresh by assumption 5.2.1. Hence (29) and (31) give the desired claim.

Finally, we prove Lemma A.3.15. By inverting rule TYPE-INST-CHECKED for the assumption $\Delta \vdash_{\mathsf{subst}} \Phi \mapsto \phi : \eta' \rightsquigarrow V$ we get

$$\begin{gathered}\Phi = \overline{\alpha\ \tau}^n \\ \phi = \overline{\sigma}^n \\ \eta' = \langle\overline{\alpha_i \mapsto \sigma_i}^n\rangle \\ \Delta \vdash_{\mathsf{coerce}} \sigma_i <: \eta'\tau_i \rightsquigarrow V_i \quad (\forall i \in [n]) \\ V = (\overline{V}^n)\end{gathered} \tag{32}$$

Define $\eta'' = \langle\overline{\alpha_i \mapsto \eta\sigma_i}^n\rangle$. To prove $\eta\phi \approx \rho V \in [\![\eta\Phi]\!]_k$ we need to show the implication $\forall j \in [n], k' \le k\,.\,u \approx U \in [\![\eta\sigma_j]\!]_{k'} \implies u \approx (\rho V)\ U \in [\![\eta''\eta\tau_j]\!]_{k'}$ from the premise of rule EQUIV-BOUNDED-TYPARAMS. Assume $j \in [n]$, $k' \le k$, and $u \approx U \in [\![\eta\sigma_j]\!]_{k'}$. Applying the IH for Lemma A.3.13 on (32) yields together with Lemma A.3.5 and Lemma A.3.7 that

$$u \approx (\rho V_j)\ U \in [\![\eta\eta'\tau_j]\!]_{k'} \tag{33}$$

As the $\overline{\alpha}$ are bound in Φ, we may assume that $\mathsf{dom}(\eta) \cap \overline{\alpha} = \emptyset = \mathsf{ftv}(\eta) \cap \overline{\alpha}$. We now argue that

$$\eta\eta'\tau_j = \eta''\eta\tau_j \tag{34}$$

by induction on the structure of τ_j. The interesting case is were τ_j is a type variable (otherwise the claim follows by the IH). If $\tau_j \in \overline{\alpha}$ then

$$\eta\eta'\tau_j \overset{\text{def. of } \eta''}{=} \eta''\tau_j \overset{\mathsf{dom}(\eta)\cap\overline{\alpha}=\emptyset}{=} \eta''\eta\tau_j$$

If $\tau_j \in \mathsf{dom}(\eta)$ then

$$\eta\eta'\tau_j \overset{\mathsf{dom}(\eta)\cap\overline{\alpha}=\emptyset}{=} \eta\tau_j \overset{\mathsf{ftv}(\eta)\cap\overline{\alpha}=\emptyset}{=} \eta''\eta\tau_j$$

If τ_j is some other type variable, (34) holds obviously. With (33) and (34) we get $u \approx (\rho V_j)\ U \in [\![\eta''\eta\tau_j]\!]_{k'}$ as required. ∎

Lemma A.3.16 (Free variables of coercion values). *If* $\Delta \vdash_{\mathsf{coerce}} \tau <: \sigma \rightsquigarrow V$ *then* $\mathsf{fv}(V) \subseteq \{X_\alpha \mid \alpha \in \mathsf{dom}(\Delta)\} \cup \mathcal{X}$ *where* $\mathcal{X} = \{X_{m,t_S} \mid m \text{ method name}, t_S \text{ struct name}\}$.

Proof. By straightforward induction on the derivation of $\Delta \vdash_{\mathsf{coerce}} \tau <: \sigma \rightsquigarrow V$. ■

A.3.2.1 Proof of Lemma 5.2.4. By induction on the derivation of $\langle\Delta, \Gamma\rangle \vdash_{\mathsf{exp}} e : \tau \rightsquigarrow E$. *Case distinction* on the last rule in the derivation.

- *Case* VAR:

$$\frac{(x : \tau) \in \Gamma}{\langle\Delta, \Gamma\rangle \vdash_{\mathsf{exp}} x : \tau \rightsquigarrow X}$$

  with $\theta\eta e = \theta x$ and $\rho E = \rho X$. From the assumption $\theta \approx \rho \in [\![\eta\Gamma]\!]_k$ we get $\theta x \approx \rho X \in [\![\eta\tau]\!]_k$ as required.

- *Case* STRUCT:

$$\frac{\begin{array}{cr} \Delta \vdash_{\mathsf{ok}} t_S[\phi] & \\ \textbf{type } t_S[\Phi] \textbf{ struct } \{\overline{f\ \tau}^n\} \in \overline{D} & (1) \\ \Phi \mapsto \phi : \eta' & (2) \\ \langle\Delta, \Gamma\rangle \vdash_{\mathsf{exp}} e_i : \eta'\tau_i \rightsquigarrow E_i \quad (\forall i \in [n]) & (3) \end{array}}{\langle\Delta, \Gamma\rangle \vdash_{\mathsf{exp}} \underbrace{t_S[\phi]\{\overline{e}^n\}}_{=e} : \underbrace{t_S[\phi]}_{=\tau} \rightsquigarrow \underbrace{(\overline{E}^n)}_{=E}} \quad (4)$$

  Applying the IH to (3) yields

$$\theta\eta e_i \approx \rho E_i \in [\![\eta\eta'\tau_i]\!]_k \quad (\forall i \in [n]) \tag{5}$$

  We now consider the two implications in the premise of rule EQUIV-EXP

  (a) Assume $k' < k$ and $\theta\eta e \longrightarrow^{k'} v$ for some value v. The goal is to show that there exists some value V with $\rho E \longrightarrow^* V$ and $v \equiv V \in [\![\eta\tau]\!]_{k-k'}$.
  With $\theta\eta e \longrightarrow^{k'} v$ there must exist values $\overline{v}^n$ such that

$$\theta\eta e_i \longrightarrow^{k_i} v_i \qquad (\forall i \in [n])$$

$$k_i \le k' \qquad (\forall i \in [n]) \tag{6}$$

$$v = t_S[\eta\phi]\{\overline{v}^n\} \tag{7}$$

  Via (5) and $k_i \le k' < k$ then for all $i \in [n]$

$$\rho E_i \longrightarrow^* V_i \text{ for some } V_i \quad (\forall i \in [n]) \tag{8}$$

$$v_i \equiv V_i \in [\![\eta\eta'\tau_i]\!]_{k-k_i} \quad (\forall i \in [n]) \tag{9}$$

  We have $k - k' \le k - k_i$ for all $i \in [n]$ by (6). Thus with (9) and Lemma A.3.3

$$v_i \equiv V_i \in [\![\eta\eta'\tau_i]\!]_{k-k'} \qquad (\forall i \in [n]) \tag{10}$$

  We also have with (8) and the definition of E in (4)

$$\rho E \longrightarrow^* (\overline{V}^n) \tag{11}$$

Assume $\hat{\Phi} = \overline{\alpha}^p$ and $\phi = \overline{\sigma}^p$. Then by (2) $\eta' = \langle \overline{\alpha_i \mapsto \sigma_i}^p \rangle$ and for $\eta'' = \langle \overline{\alpha_i \mapsto \eta\sigma_i}^p \rangle$ we have

$$\Phi \mapsto \eta\phi : \eta'' \tag{12}$$

By assumption 5.2.1 we have $\mathsf{ftv}(\overline{\tau}^n) \subseteq \{\overline{\alpha}\}$, so

$$\eta\eta'\tau_i = \eta''\tau_i \qquad (\forall i \in [n]) \tag{13}$$

With (1), (10), (7), (12), (13), and rule EQUIV-STRUCT then

$$v \equiv (\overline{V}^n) \in [\![t_S[\eta\phi]]\!]_{k-k'}$$

Together with (11), this finishes subcase (a) for $V = (\overline{V}^n)$.

(b) Assume $k' < k$ and $\theta\eta e \longrightarrow^{k'} e'$ and $\mathsf{diverge}(e')$. Then $\mathsf{diverge}(e_j)$ for some $j \in [n]$, so with (5) and (6) also $\mathsf{diverge}(\rho E_j)$. Thus, by definition of E in (4), $\mathsf{diverge}(\rho E)$ as required.

- *Case* ACCESS:

$$\frac{\begin{array}{c}\langle \Delta, \Gamma \rangle \vdash_{\mathsf{exp}} e' : t_S[\phi] \rightsquigarrow E' \qquad (14)\\ \textbf{type } t_S[\Phi] \textbf{ struct } \{\overline{f\ \tau}^n\} \in \overline{D} \\ \Phi \mapsto \phi : \eta'\end{array}}{\langle \Delta, \Gamma \rangle \vdash_{\mathsf{exp}} \underbrace{e'.f_j}_{=e} : \underbrace{\eta'\tau_j}_{=\tau} \rightsquigarrow \underbrace{\textbf{case } E' \textbf{ of } (\overline{X}^n) \to X_i}_{=E}} \quad (15)$$

Applying the IH to (14) yields

$$\theta\eta e' \approx \rho E' \in [\![t_S[\eta\phi]]\!]_k \tag{16}$$

We now consider the two implications in the premise of rule EQUIV-EXP

(a) Assume $k' < k$ and $\theta\eta e \longrightarrow^{k'} v$ for some value v. The goal is to show that there exists some value V with $\rho E \longrightarrow^* V$ and $v \equiv V \in [\![\eta\tau]\!]_{k-k'}$.
With $\theta\eta e \longrightarrow^{k'} v$ then $\theta\eta e' \longrightarrow^{k''} v'$ for some v' and $k'' < k'$. With (16) then for some V'

$$\rho E' \longrightarrow^* V' \tag{17}$$
$$v' \equiv V' \in [\![t_S[\eta\phi]]\!]_{k-k''} \tag{18}$$

Inverting rule EQUIV-STRUCT on (18) yields

$$v' = t_S[\eta\phi]\{\overline{v}^n\} \tag{19}$$
$$V' = (\overline{V}^n) \quad \text{for some } \overline{V}^n$$
$$v_i \equiv V_i \in [\![\eta''\tau_i]\!]_{k-k''} \quad (\forall i \in [n]) \tag{20}$$

where $\eta'' = \langle \overline{\alpha_i \mapsto \eta\sigma_i}^p \rangle$, assuming $\hat{\Phi} = \overline{\alpha}^p$ and $\phi = \overline{\sigma}^p$. By assumption 5.2.1 we have $\mathsf{ftv}(\tau_j) \subseteq \{\overline{\alpha}\}$. Thus, $\eta''\tau_j = \eta\eta'\tau_j \overset{(15)}{=} \eta\tau$. Also, $k'' \leq k'$, so $k - k' \leq k - k''$. Hence with (20) and Lemma A.3.3

$$v_j \equiv V_j \in [\![\eta\tau]\!]_{k-k'} \tag{21}$$

With (17) and the definition of E in (15) we get

$$\rho E \longrightarrow^{*} V_j \tag{22}$$

With $\theta\eta e \longrightarrow^{k'} v$ and $\theta\eta e' \longrightarrow^{k''} v'$ and the form of v' in (19), we get $\theta\eta e \longrightarrow^{k'} v_j$ and $v = v_j$ by rule FG-FIELD. Define $V = V_j$ and we are done with subcase (a) by (21) and (22).

(b) Assume $k' < k$ and $\theta\eta e \longrightarrow^{k'} e''$ and $\mathsf{diverge}(e'')$. Then we must have that $\theta\eta e' \longrightarrow^{k''} e'''$ for some $k'' < k'$ and some e'''. Thus, $\mathsf{diverge}(e''')$ by the definition of e in (15) and the evaluation rules for FGG^{-}. With (16) then $\mathsf{diverge}(\rho E')$. By definition of E in (15) then $\mathsf{diverge}(\rho E)$ as required.

- *Case* CALL-STRUCT:

$$\frac{\begin{array}{lr}\langle\Delta,\Gamma\rangle \vdash_{\mathsf{exp}} g : t_S[\phi] \rightsquigarrow G & (23)\\ \langle m[\Psi](\overline{x\ \sigma}^n)\sigma, W\rangle \in \mathsf{methods}(\Delta, t_S[\phi]) & (24)\\ \Delta \vdash_{\mathsf{subst}} \Psi \mapsto \psi : \eta_1 \rightsquigarrow W' & (25)\\ \langle\Delta,\Gamma\rangle \vdash_{\mathsf{exp}} e_i : \eta_1\sigma_i \rightsquigarrow E_i \quad (\forall i \in [n]) & (26)\end{array}}{\langle\Delta,\Gamma\rangle \vdash_{\mathsf{exp}} \underbrace{g.m[\psi](\overline{e}^n)}_{=e} : \underbrace{\eta_1\sigma}_{=\tau} \rightsquigarrow \underbrace{X_{m,t_S}\ \mathtt{(}W, G, W', \mathtt{(}\overline{E}^n\mathtt{))}}_{=E}} \quad (27)$$

From the IH applied to (23) and (26)

$$\theta\eta g \approx \rho G \in [\![\eta t_S[\phi]]\!]_k \tag{28}$$
$$\theta\eta e_i \approx \rho E_i \in [\![\eta\eta_1\sigma_i]\!]_k \quad (\forall i \in [n]) \tag{29}$$

Assume $\theta\eta e \longrightarrow^{k'} e'$ for some $k' < k$. We first consider the following situation for some values $u, \overline{v}^n$:

$$\theta\eta g \longrightarrow^{k''} u \tag{30}$$
$$\theta\eta e_i \longrightarrow^{k_i} v_i \tag{31}$$
$$\theta\eta e \longrightarrow^{k''+\Sigma k_i} u.m[\eta\psi](\overline{v}^n) \longrightarrow^{k'-k''-\Sigma k_i} e' \tag{32}$$

with $k'' + \Sigma k_i \leq k'$. (30), (31), and (32) are intermediate assumptions, which become true when we later prove the two implications of rule EQUIV-EXP.
We have from (28), (30), (29), and (31)

$$\rho G \longrightarrow^{*} U \text{ for some } U \text{ with } u \equiv U \in [\![\eta t_S[\phi]]\!]_{k-k''} \tag{33}$$
$$(\forall i \in [n])\ \rho E_i \longrightarrow^{*} V_i \text{ for some } V_i \text{ with } v_i \equiv V_i \in [\![\eta\eta_1\sigma_i]\!]_{k-k_i} \tag{34}$$

From (24) we get by inverting rule METHODS-STRUCT:

$$\textsf{func}\ (x\ t_S[\Phi])\ m[\Psi'](\overline{x\ \sigma'}^n)\ \sigma'\ \{\textsf{return}\ g'\} \in \overline{D} \tag{35}$$
$$\Delta \vdash_{\mathsf{subst}} \Phi \mapsto \phi : \eta_2 \rightsquigarrow W \tag{36}$$
$$m[\Psi](\overline{x\ \sigma}^n)\ \sigma = \eta_2(m[\Psi'](\overline{x\ \sigma'}^n)\ \sigma') \tag{37}$$

From the assumption $\overline{D} \approx_k \mu$ and (35)

$$\textsf{func}\ (x\ t_S[\Phi])\ m[\Psi'](\overline{x\ \sigma'}^n)\ \sigma'\ \{\textsf{return}\ g'\} \approx_k X_{m,t_S} \tag{38}$$

Define

$$k''' := min(k - k'', k - \Sigma k_i) - 1 \tag{39}$$

We have $k' - k'' - \Sigma k_i < k''' + 1$ by the following reasoning:

$$\begin{aligned} k''' + 1 &\overset{(39)}{=} min(k - k'', k - \Sigma k_i) \\ &= k - max(k'', \Sigma k_i) \\ &\geq k - k'' - \Sigma k_i \\ &\overset{k'<k}{>} k' - k'' - \Sigma k_i \end{aligned} \tag{40}$$

With (38) and $k''' < k$, we now want to use the implication from the premise of rule EQUIV-METHOD-DECL. We instantiate the universally quantified variables of the implication as follows: $k' = k'''$, $\phi = \eta(\phi, \psi)$, $\overline{W}^p = \rho W$, $\overline{W'}^q = \rho W'$, $v = u$, $V = U$, $\overline{v}^n = \overline{v}^n$, $\overline{V}^n = \overline{V}^n$. Next, we prove the left-hand side of the implication. But first assume (see (36), (25), (37))

$$\Phi = \overline{\alpha\,\tau}^p \qquad \phi = \overline{\tau'}^p \qquad W = \overline{W}^p \tag{41}$$

$$\Psi' = \overline{\beta\,\tau''}^q \qquad \psi = \overline{\tau'''}^q \qquad W' = \overline{W'}^q \tag{42}$$

and define

$$\eta_3 = \langle \overline{\alpha_i \mapsto \eta\tau'_i}^p, \overline{\beta_i \mapsto \eta\tau'''_i}^q \rangle \tag{43}$$

- We start by showing the first two conjuncts of the implication's left-hand side.

$$\Phi, \Psi' \mapsto \eta(\phi, \psi) : \eta_3 \wedge \eta(\phi, \psi) \approx \rho(\overline{W}, \overline{W'}) \in [\![\Phi, \Psi']\!]_{k'''} \tag{44}$$

  The left part of the conjunction follows from (43). We then show $\eta(\phi, \psi) \approx \rho(\overline{W}, \overline{W'}) \in [\![\Phi, \Psi']\!]_k$ by proving the two implications required to fulfill the premise of rule EQUIV-BOUNDED-TYPARAMS. The right part of the conjunction in (44) then follows via Lemma A.3.5.

  * First implication: $u_j \approx U_j \in [\![\eta\tau'_j]\!]_k \implies u_j \approx (\rho W_j)\, U_j \in [\![\eta_3\tau_j]\!]_k$ for all $j \in [p]$ and all u_j, U_j.
    From (36) and Lemma A.3.15 we have $\eta\phi \approx \rho W \in [\![\eta\Phi]\!]_k$ Hence, with $u_j \approx U_j \in [\![\eta\tau'_j]\!]_k$ and the implication in the premise of rule EQUIV-BOUNDED-TYPARAMS, we have $u_j \approx (\rho W_j)\, U_j \in [\![\langle\overline{\alpha_i \mapsto \eta\tau'_i}\rangle\eta\tau_j]\!]_k$. From assumption 5.2.1, (35), and (41), we know that $\mathsf{ftv}(\tau_j) \subseteq \{\overline{\alpha}\}$ and $\overline{\alpha}$ fresh, so $\langle\overline{\alpha_i \mapsto \eta\tau'_i}\rangle\eta\tau_j = \eta_3\tau_j$. Thus $u_j \approx (\rho W_j)\, U_j \in [\![\eta_3\tau_j]\!]_k$ as required.
  * Second implication: $u_j \approx U_j \in [\![\eta\tau'''_j]\!]_k \implies u_j \approx (\rho W'_j)\, U_j \in [\![\eta_3\tau''_j]\!]_k$ for all $i \in [q]$ and all u_j, U_j.
    From (25) and Lemma A.3.15 we have $\eta\psi \approx \rho W' \in [\![\eta\Psi]\!]_k$. Hence, with $u_j \approx U_j \in [\![\eta\tau'''_j]\!]_k$, the implication in the premise of rule EQUIV-BOUNDED-TYPARAMS, and (37) then $u_j \approx (\rho W'_j)\, U_j \in [\![\langle\overline{\beta_i \mapsto \eta\tau'''_i}\rangle\eta\eta_2\tau''_j]\!]_k$. We have with (36) and (41) that $\eta_2 = \langle\overline{\alpha_i \mapsto \tau'_i}\rangle$. Because of assumption 5.2.1, (35), and (42), we know that $\mathsf{ftv}(\tau''_j) \subseteq \{\overline{\alpha}, \overline{\beta}\}$ and $\overline{\alpha}, \overline{\beta}$ fresh. Hence, $\langle\overline{\beta_i \mapsto \eta\tau'''_i}\rangle\eta\eta_2\tau''_j =$

$\langle\overline{\beta_i \mapsto \eta\tau_i'''}\rangle\langle\overline{\alpha_i \mapsto \eta\tau_i'}\rangle\tau_j'' \overset{(43)}{=} \eta_3\tau_j''$. Thus $u_j \approx (\rho W_j')\, U_j \in [\![\eta_3\tau_j'']\!]_k$ as required.

This finishes the proof of (44).

– We next show the third conjunct of the implication's left-hand side.

$$u \approx U \in [\![t_S[\eta_3\overline{\alpha}]]\!]_{k'''} \tag{45}$$

We have $t_S[\eta_3\overline{\alpha}] = t_S[\eta\phi]$ by (43) and (41). Hence, with (33), Lemma A.3.9, and Lemma A.3.3, it suffices to show that $k''' \leq k - k''$. But this follows from construction of k''' in (39).

– Finally, we show the fourth conjunct:

$$v_i \approx V_i \in [\![\eta_3\sigma_i']\!]_{k'''} (\forall i \in [n]) \tag{46}$$

By (25), (37), (42) we have $\eta_1 = \langle\overline{\beta_i \mapsto \tau_i'''}^q\rangle$. By (36) and (41) we have $\eta_2 = \langle\overline{\alpha_i \mapsto \tau_i'}^p\rangle$. Thus,

$$\eta\eta_1\sigma_i \overset{(37)}{=} \eta\eta_1\eta_2\sigma_i' \overset{(43)}{=} \eta_3\sigma_i'$$

For the last equation, note that $\mathsf{ftv}(\sigma_i') \subseteq \{\overline{\alpha}, \overline{\beta}\}$ by assumption 5.2.1 and (35), (41), (42). Hence, with (34), Lemma A.3.9, and Lemma A.3.3, it suffices to show that $k''' \leq k - k_i$. But this follows from construction of k''' in (39).

Now (44), (45), and (46) are the left-hand side of the implication of rule EQUIV-METHOD-DECL, which we get from (38). The right-hand side of the implication then yields

$$\underbrace{\langle x \mapsto u, \overline{x_i \mapsto v_i}^n\rangle}_{=:\theta'}\eta_3 g' \approx X_{m,t_S}\ (\rho W, U, \rho W', (\overline{V})) \in [\![\eta_3\sigma']\!]_{k'''} \tag{47}$$

From (33) we have $u = t_S[\eta\phi]$ by inverting rule EQUIV-STRUCT. Hence by (35), (43), and rule FG-CALL

$$u.m[\eta\psi](\overline{v}) \longrightarrow \theta'\eta_3 g' \tag{48}$$

Also we have

$$\eta\tau \overset{(27)}{=} \eta\eta_1\sigma \overset{(37)}{=} \eta\eta_1\eta_2\sigma' = \eta_3\sigma'$$

where the last equation follows from (25), (37), (42), (36), (41) and $\mathsf{ftv}(\sigma') \subseteq \{\overline{\alpha}, \overline{\beta}\}$ with assumption 5.2.1. Thus, with (47), (48), and Lemma 5.2.3

$$u.m[\eta\psi](\overline{v}) \approx X_{m,t_S}\ (\rho W, U, \rho W', (\overline{V})) \in [\![\eta\tau]\!]_{k'''+1} \tag{49}$$

By definition of E in (27) and with (33) and (34) we have

$$\rho E \longrightarrow^* \mu(X_{m,t_S})\ (\rho W, U, \rho W', (\overline{V})) \tag{50}$$

Also, we have by rules TL-CONTEXT and TL-METHOD

$$X_{m,t_S}\ (\rho W, U, \rho W', (\overline{V})) \longrightarrow \mu(X_{m,t_S})\ (\rho W, U, \rho W', (\overline{V})) \tag{51}$$

So far, we proved everything under the assumptions (30), (31), (32). We next consider the two implications of rule EQUIV-EXP.

(a) Assume $e' = v$ for some value v. Our goal is to prove that there exists some value V such that $\rho E \longrightarrow^* V$ and $v \equiv V \in [\![\eta\tau]\!]_{k-k'}$. Noting that (30), (31), (32) hold, we have together with (48)

$$\theta\eta e \longrightarrow^{k''+\Sigma k_i} u.m[\eta\psi](\overline{v}) \longrightarrow^{k'-k''-\Sigma k_i} v \tag{52}$$

with $k'' + \Sigma k_i < k'$. We have $k' - k'' - \Sigma k_i < k''' + 1$ by (40). Hence with (49) and (52) we know that there exists some value V with

$$X_{m,ts}\ (\rho W, U, \rho W', (\overline{V})) \longrightarrow^* V \tag{53}$$

$$v \equiv V \in [\![\eta\tau]\!]_{k'''+1-k'+k''+\Sigma k_i} \tag{54}$$

We have $k - k' \leq k''' + 1 - k' + k'' + \Sigma k_i$ by the following reasoning:

$$\begin{aligned} k''' + 1 - k' + k'' + \Sigma k_i &\overset{(39)}{=} min(k - k'', k - \Sigma k_i) - k' + k'' + \Sigma k_i \\ &= k - max(k'', \Sigma k_i) - k' + k'' + \Sigma k_i \\ &\geq k - k'' - \Sigma k_i - k' + k'' + \Sigma k_i \\ &= k - k' \end{aligned}$$

With (54) and Lemma A.3.3 then $v \equiv V \in [\![\eta\tau]\!]_{k-k'}$. And from (50), (51), (53), and Lemma A.1.2 we have that $\rho E \longrightarrow^* V$.

(b) Assume $\mathsf{diverge}(e')$. We then have to show $\mathsf{diverge}(\rho E)$.
Case distinction whether receiver, argument or method call diverges.
 - *Case* receiver diverges: Then $\theta\eta g \longrightarrow^{k'} g''$ and $\mathsf{diverge}(g'')$. With (28) and $k' < k$ then $\mathsf{diverge}(\rho G)$, so by the definition of E in (27) we get $\mathsf{diverge}(\rho E)$.
 - *Case* j-th argument diverges: Then $\theta\eta g \longrightarrow^{k''} u$ and $\theta\eta e_i \longrightarrow^{k_i} v_i$ for all $i < j$ and $\theta\eta e_j \longrightarrow^{k_j} e''$ and $\mathsf{diverge}(e'')$. With (29) and $k_j \leq k' < k$ we get $\mathsf{diverge}(\rho E_j)$. By definition of E in (27) then $\mathsf{diverge}(\rho E)$.
 - *Case* method call diverges: Then we are in the situation that (30), (31), and (32) hold. We then have

$$u.m[\eta\psi](\overline{v}^n) \longrightarrow^{k'-k''-\Sigma k_i} e'$$

 Hence, with (40), (49), and the second implication in the premise of rule EQUIV-EXP, we have that $\mathsf{diverge}(X_{m,ts}\ (\rho W, U, \rho W', (\overline{V})))$. With (50) and (51) and Lemma A.1.2 then also $\mathsf{diverge}(\rho E)$ as required.

End case distinction.

This finishes the proof for rule CALL-STRUCT.

- *Case* CALL-IFACE:

$$\frac{\begin{array}{ll} \langle\Delta, \Gamma\rangle \vdash_{\mathsf{exp}} g : \tau_I \rightsquigarrow G & (55) \\ \mathsf{methods}(\tau_I) = \overline{R}^q & (56) \\ R_j = m[\Psi](\overline{x\ \sigma}^n)\sigma \quad (\text{for some } j \in [q]) & (57) \\ \Delta \vdash_{\mathsf{subst}} \Psi \mapsto \psi : \eta_1 \rightsquigarrow V & (58) \\ \langle\Delta, \Gamma\rangle \vdash_{\mathsf{exp}} e_i : \eta_1\sigma_i \rightsquigarrow E_i \quad (\forall\, i \in [n]) & (59) \\ Y, \overline{X}^q \text{ fresh} & \end{array}}{\langle\Delta, \Gamma\rangle \vdash_{\mathsf{exp}} \underbrace{g.m[\psi](\overline{e}^n)}_{=e} : \underbrace{\eta_1\sigma}_{=\tau} \rightsquigarrow E \qquad (60)}$$

with

$$E = \textbf{case}\ G\ \textbf{of}\ (Y, (\overline{X}^q)) \to X_j\ (Y, V, (\overline{E}^n)) \tag{61}$$

From the IH applied to (55), (59)

$$\theta\eta g \approx \rho G \in [\![\eta\tau_I]\!]_k \tag{62}$$

$$\theta\eta e_i \approx \rho E_i \in [\![\eta\eta_1\sigma_i]\!]_k \quad (\forall i \in [n]) \tag{63}$$

Assume $\theta\eta e \longrightarrow^{k'} e'$ for some $k' < k$. We first consider the following situation for some values $u, \overline{v}^n$:

$$\theta\eta g \longrightarrow^{k''} u \tag{64}$$

$$\theta\eta e_i \longrightarrow^{k_i} v_i \tag{65}$$

$$\theta\eta e \longrightarrow^{k''+\Sigma k_i} u.m[\eta\psi](\overline{v}^n) \longrightarrow^{k'-k''-\Sigma k_i} e' \tag{66}$$

with $k'' + \Sigma k_i \le k'$. (64), (65), and (66) are intermediate assumptions, which become true when we later prove the two implications of rule EQUIV-EXP.
We have from (62), (63), (64), and (65)

$$\rho G \longrightarrow^* U \text{ for some } U \text{ with } u \equiv U \in [\![\eta\tau_I]\!]_{k-k''} \tag{67}$$

$$(\forall i \in [n])\ \rho E_i \longrightarrow^* V_i \text{ for some } V_i \text{ with } v_i \equiv V_i \in [\![\eta\eta_1\sigma_i]\!]_{k-k_i} \tag{68}$$

From (67) and (56) we get by inverting rule EQUIV-IFACE

$$\exists \sigma_S = t_S[\phi] \tag{69}$$

$$U = (U', (\overline{U}^q)) \tag{70}$$

$$\forall \ell_1 < k - k''\ .\ u \equiv U' \in [\![\sigma_S]\!]_{\ell_1} \tag{71}$$

$$\forall \ell_2 < k - k''\ .\ \mathsf{methodLookup}(m_j, \sigma_S) \approx U_j \in [\![\eta R_j]\!]_{\ell_2} \tag{72}$$

Hence we have by (69), (72), (57), and rule METHOD-LOOKUP

$$\textbf{func}\ (x\ t_S[\Phi])\ \underbrace{m[\Psi'](\overline{x\ \sigma'}^n)\ \sigma'}_{=:R'}\ \{\textbf{return}\ e''\} \in \overline{D} \tag{73}$$

$$\Phi \mapsto \phi : \eta_2 \tag{74}$$

$$\mathsf{methodLookup}(m_j, \sigma_S) = \langle x, t_S[\phi], \eta_2 R', \eta_2 e''\rangle \tag{75}$$

$$\eta_2 R' = \eta R_j = \eta(m[\Psi](\overline{x_i\ \sigma_i}^n)\ \sigma) \tag{76}$$

Then by (72) and (75)

$$\langle x, t_S[\phi], \eta_2 R', \eta_2 e''\rangle \approx U_j \in [\![\eta R_j]\!]_{k-k''-1} \tag{77}$$

Define $k''' := \min(k - k'' - 1, k - \Sigma k_i - 1)$. Then

$$k''' \le k - k'' - 1 \tag{78}$$

$$k''' < k \tag{79}$$

$$k''' < k - k'' \tag{80}$$

$$k''' < k - k_i \quad (\forall i \in [n]) \tag{81}$$

$$k' - k'' - \Sigma k_i < k''' + 1 \tag{82}$$

The first four of these claims are straightforward to verify. The last can be shown with the following reasoning:

$$\begin{aligned} k''' + 1 &= k - \max(k'' + 1, \Sigma k_i + 1) + 1 \\ &> k - (k'' + 1 + \Sigma k_i + 1) + 1 \\ &= k - 1 - k'' - \Sigma k_i \\ &\overset{k'<k}{\geq} k' - k'' - \Sigma k_i \end{aligned}$$

From (77) we get the implication in the premise of rule EQUIV-METHOD-DICT-ENTRY. We now show that the left-hand side of the implication holds. The universally quantified variables of the rule's premise are instantiated as follows: $k' = k'''$, $\phi = \eta\psi$, $W = \rho V$, $v = u$, $V = U'$, $\overline{v}^n = \overline{v}^n$, $\overline{V}^n = \overline{V}^n$. The variables in the conclusion are instantiated as follows: $x = x$, $\tau_S = t_S[\phi]$, $m[\Phi](\overline{x_i\ \tau_i}^n)\ \tau = \eta_2 R'$, $e = \eta_2 e''$. The requirement $k''' \leq k - k'' - 1$ follows from (78).
We have from (58) and (76) the first conjunct:

$$\eta\Psi \mapsto \eta\psi : \underbrace{\langle\overline{\alpha \mapsto \eta\tau}\rangle}_{=\eta_4} \ \text{(assuming } \eta_1 = \langle\overline{\alpha \mapsto \tau}\rangle, \hat{\Psi} = \overline{\alpha}, \psi = \overline{\tau}\text{)} \tag{83}$$

From (58) we get the second conjunct by Lemma A.3.15, (79), by the assumptions $\overline{D} \approx_k \mu$ and $\eta \approx \rho \in [\![\Delta]\!]_k$, and by Lemma A.3.5:

$$\eta\psi \approx \rho V \in [\![\eta\Psi]\!]_{k'''}$$

With (80), (71), (69), and Lemma A.3.9 we get the third conjunct:

$$u \approx U' \in [\![t_S[\phi]]\!]_{k'''}$$

With (81), (68), Lemma A.3.9, and Lemma A.3.3 we have

$$v_i \approx V_i \in [\![\eta\eta_1\sigma_i]\!]_{k'''} \quad (\forall i \in [n]) \tag{84}$$

We next prove

$$\eta\eta_1\sigma_i = \eta_4\eta\sigma_i \quad (\forall i \in [n]) \tag{85}$$

$$\eta\eta_1\sigma = \eta_4\eta\sigma \tag{86}$$

by induction on σ_i or σ. The interesting case is where σ_i or σ is a type variable $\alpha \in \mathsf{dom}(\eta_1) \cup \mathsf{dom}(\eta)$. As the $\overline{\alpha} = \mathsf{dom}(\eta_1) = \mathsf{dom}(\eta_4)$ are bound in Ψ (see (83)), we may assume that $\overline{\alpha} \cap \mathsf{dom}(\eta) = \emptyset = \overline{\alpha} \cap \mathsf{ftv}(\eta)$. If $\alpha \in \mathsf{dom}(\eta_1)$ then

$$\eta\eta_1\alpha \overset{(83)}{=} \eta_4\alpha \overset{\mathsf{dom}(\eta_1)\cap\mathsf{dom}(\eta)=\emptyset}{=} \eta_4\eta\alpha$$

If $\alpha \in \mathsf{dom}(\eta)$ then

$$\eta\eta_1\alpha \overset{\mathsf{dom}(\eta)\cap\mathsf{dom}(\eta_1)=\emptyset}{=} \eta\alpha \overset{\mathsf{dom}(\eta_1)\cap\mathsf{ftv}(\eta)=\emptyset}{=} \eta_4\eta\alpha$$

We now get with (76) and (85) that $\eta\eta_1\sigma_i = \eta_4\eta_2\sigma_i'$. Hence with (84) the fourth conjunct:

$$(\forall i \in [n]) \quad v_i \approx V_i \in [\![\eta_4\eta_2\sigma_i']\!]_{k'''}$$

Now the right-hand side of the implication of rule EQUIV-METHOD-DICT-ENTRY yields with (77)

$$\langle x \mapsto u, \overline{x_i \mapsto v_i}^n \rangle \eta_4 \eta_2 e'' \approx U_j\ (U', \rho V, (\overline{V}^n)) \in [\![\eta_4 \eta_2 \sigma']\!]_{k'''} \tag{87}$$

Define η_3 such that

$$\Phi, \Psi' \mapsto \phi, \eta\psi : \eta_3 \tag{88}$$

Then with (74) and (83)

$$\eta_4 \eta_2 e'' = \eta_3 e'' \tag{89}$$

by induction on e''. The interesting case is the one for a type variable α. By assumption 5.2.1 and (73), we know that $\alpha \in \hat{\Phi} \cup \hat{\Psi}'$. Further we may assume that the type variables $\hat{\Psi}'$ are fresh, and we have $\mathsf{dom}(\eta_2) = \hat{\Phi}$ by (74) and $\mathsf{dom}(\eta_4) = \hat{\Psi}'$ by (83). Thus, if $\alpha \in \hat{\Phi}$ then $\eta_4 \eta_2 \alpha = \eta_2 \alpha$ because $\hat{\Psi}'$ fresh, and $\eta_3 \alpha = \eta_2 \alpha$ by (74) and (88). If $\alpha \in \hat{\Psi}'$ then $\eta_4 \eta_2 \alpha = \eta_4 \alpha$ because $\hat{\Psi}'$ fresh, and $\eta_3 \alpha = \eta_4 \alpha$ by (88) and (83).
With (86) and (76) $\eta_4 \eta_2 \sigma' = \eta \eta_1 \sigma$. Hence we have with (89), (87)

$$\langle x \mapsto u, \overline{x_i \mapsto v_i}^n \rangle \eta_3 e'' \approx U_j\ (U', \rho V, (\overline{V}^n)) \in [\![\eta \eta_1 \sigma]\!]_{k'''} \tag{90}$$

From (69) and (71) we get by inverting rule EQUIV-STRUCT that $u = t_S[\phi]\{\ldots\}$. Hence by rule FG-CALL with (73) and (88)

$$u.m[\eta\psi](\overline{v}^n) \longrightarrow \langle x \mapsto u, \overline{x_i \mapsto v_i}^n \rangle \eta_3 e''$$

Then with (90) and Lemma 5.2.3

$$u.m[\eta\psi](\overline{v}^n) \approx U_j\ (U', \rho V, (\overline{V}^n)) \in [\![\eta \eta_1 \sigma]\!]_{k'''+1} \tag{91}$$

We also have

$$\begin{aligned} \rho E &\overset{(67),(61)}{\longrightarrow^*} \textsf{\textbf{case}}\ U\ \textsf{\textbf{of}}\ (Y, (\overline{X}^q)) \to X_j\ (Y, V, (\overline{E}^n)) \\ &\overset{(70)}{\longrightarrow} U_j\ (U', \rho V, \rho(\overline{E}^n)) \\ &\overset{(68)}{\longrightarrow^*} U_j\ (U', \rho V, (\overline{V}^n)) \end{aligned} \tag{92}$$

So far, we proved everything under the assumptions (64), (65), (66). We next consider the two implications of rule EQUIV-EXP.

(a) Assume $e' = v$ for some value v. Then (64), (65), and (66) hold. We now need to show that there exists some W with $\rho E \longrightarrow^* W$ and $v \equiv W \in [\![\eta\tau]\!]_{k-k'}$. We have $k' - k'' - \Sigma k_i < k''' + 1$ by (82) Also, we have with (66) that

$$u.m[\eta\psi](\overline{v}^n) \longrightarrow^{k'-k''-\Sigma k_i} v$$

Hence, (91) gives us the existence of some W such that

$$U_j\ (U', \rho V, (\overline{V}^n)) \longrightarrow^* W \tag{93}$$
$$v \approx W \in [\![\eta \eta_1 \sigma]\!]_{k'''+1-(k'-k''-\Sigma k_i)}$$

We get $k - k' \leq k''' + 1 - (k' - k'' - \Sigma k_i)$ by

$$\begin{aligned}
&k''' + 1 - (k' - k'' - \Sigma k_i) \\
&= k - max(k'' + 1, \Sigma k_i + 1) + 1 - k' + k'' + \Sigma k_i \\
&= k - max(k'', \Sigma k_i) - k' + k'' + \Sigma k_i \\
&\geq k - (k'' + \Sigma k_i) - k' + (k'' + \Sigma k_i) \\
&= k - k'
\end{aligned}$$

Hence by Lemma A.3.3

$$v \approx W \in [\![\eta\eta_1\sigma]\!]_{k-k'}$$

By (60) $\eta_1\sigma = \tau$ so $v \approx W \in [\![\eta\tau]\!]_{k-k'}$ and with (92) and (93) $\rho E \longrightarrow^* W$.

(b) Assume $\mathsf{diverge}(e')$. We then have to show $\mathsf{diverge}(\rho E)$.
Case distinction whether receiver, argument or method call diverges.
- *Case* receiver diverges: Then $\theta\eta g \longrightarrow^{k'} g'$ and $\mathsf{diverge}(g')$. With (62) and $k' < k$ then $\mathsf{diverge}(\rho G)$, so by the definition of E in (61) we get $\mathsf{diverge}(\rho E)$.
- *Case* j-th argument diverges: Then $\theta\eta g \longrightarrow^{k''} u$. By (62) and rule EQUIV-IFACE we know that $U = (U', \overline{U}^q)$ for some $U', \overline{U}^q$. Hence

$$\rho E \longrightarrow^* U_j\ (U', \rho V, \rho(\overline{E}^n)) \tag{94}$$

Because the j-th argument diverges, we also have $\theta\eta e_i \longrightarrow^{k_i} v_i$ for all $i < j$ and $\theta\eta e_j \longrightarrow^{k_j} e''$ and $\mathsf{diverge}(e'')$. With (63) we get $\mathsf{diverge}(\rho E_j)$, so with (94) also $\mathsf{diverge}(\rho E)$.
- *Case* method call diverges: Then we are in the situation that (64), (65), and (66) hold. Thus, we get with (66), (91), (82)

$$\begin{gathered}
u.m[\eta\psi](\overline{v}^n) \longrightarrow^{k'-k''-\Sigma k_i} e' \\
u.m[\eta\psi](\overline{v}^n) \approx U_j\ (U', \rho V, (\overline{V}^n)) \in [\![\eta\eta_1\sigma]\!]_{k'''+1} \\
k' - k'' - \Sigma k_i < k''' + 1
\end{gathered}$$

Hence $\mathsf{diverge}(U_j\ (U', \rho V, (\overline{V}^n)))$ by the implication in the premise of rule EQUIV-EXP. So by (92) also $\mathsf{diverge}(\rho E)$ as required.

End case distinction.

- *Case* SUB:

$$\frac{\langle\Delta, \Gamma\rangle \vdash_{\mathsf{exp}} e : \sigma \rightsquigarrow E' \qquad \Delta \vdash_{\mathsf{coerce}} \sigma <: \tau \rightsquigarrow V}{\langle\Delta, \Gamma\rangle \vdash_{\mathsf{exp}} e : \tau \rightsquigarrow \underbrace{V\ E'}_{=E}}\ \text{SUB}$$

From the IH then $\theta\eta e \approx \rho E' \in [\![\eta\sigma]\!]_k$. From Lemma A.3.13 we get $\theta\eta e \approx (\rho V)\ \rho E' \in [\![\eta\tau]\!]_k$ with $\rho E = (\rho V)\ (\rho E')$ as required.

End case distinction on the last rule in the derivation of $\langle\Delta, \Gamma\rangle \vdash_{\mathsf{exp}} e : \tau \rightsquigarrow E$. ■

A.3.2.2 Proof of Lemma 5.2.5. We proceed by induction on k. For $k = 0$, we first note that $e \approx E \in [\![\tau]\!]_0$ holds for any e, E, τ because the two implications in the premise of rule EQUIV-EXP hold trivially. Thus, we get $D \approx_0 X_{m,t_S}$ for all $D =$

func $(x\ \ t_S[\Phi])\, mM\, \{$**return** $e\} \in \overline{D}$ by rule EQUIV-METHOD-DECL. Hence $\overline{D} \approx_0 \mu$ by rule EQUIV-DECLS.

Now assume $\overline{D} \approx_k \mu$ (IH) for some k and prove $\overline{D} \approx_{k+1} \mu$. By rule EQUIV-DECLS, we need to show $D \approx_{k+1} X_{m,t_S}$ for all

$$D = \textbf{func}\ (x\ t_S[\Phi])\ m[\Phi'](\overline{x\ \tau}^n)\ \tau\ \{\textbf{return}\ e\} \in \overline{D} \tag{95}$$

Thus, we assume the left-hand side of the implication in the premise of rule EQUIV-METHOD-DECL and then show the right-hand side of the implication. More specifically, let

$$\Phi = \overline{\alpha_i\ \sigma_i}^p \qquad \Phi' = \overline{\beta_i\ \sigma'_i}^q \qquad \Psi = \Phi, \Phi' = \overline{\alpha_i\ \sigma_i}^p\ \overline{\beta_i\ \sigma'_i}^q$$

and assume for arbitrary $k' < k+1$, $\phi = \overline{\sigma''}^p$, $\phi' = \overline{\sigma'''}^q$, $\overline{W}^p$, $\overline{W'}^q$, u, U, $\overline{v}^n$, $\overline{V}^n$ the left-hand side of the implication:

$$\Psi \mapsto \phi, \phi' : \eta \text{ with } \eta = \langle \overline{\alpha_i \mapsto \sigma''_i}^p\ \overline{\beta_i \mapsto \sigma'''_i}^q \rangle \tag{96}$$

$$\phi, \phi' \approx (\overline{W}^p, \overline{W'}^q) \in [\![\Psi]\!]_{k'} \tag{97}$$

$$u \approx U \in [\![t_S[\eta\overline{\alpha}^p]]\!]_{k'} \tag{98}$$

$$v_i \approx V_i \in [\![\eta\tau_i]\!]_{k'} \quad (\forall i \in [n]) \tag{99}$$

From this we need to prove the following goal:

$$\underbrace{\langle x \mapsto u, \overline{x_i \mapsto v_i}^n \rangle}_{=:\theta} \eta e \approx X_{m,t_S}\ ((\overline{W}^p), U, (\overline{W'}^q), (\overline{V}^n)) \in [\![\eta\tau]\!]_{k'} \tag{100}$$

Define

$$\rho = \langle \overline{X_{\alpha_i} \mapsto W_i}^p, \overline{X_{\beta_i} \mapsto W'_i}^q, X \mapsto U, \overline{X_i \mapsto V_i}^n \rangle \tag{101}$$

$$\Delta = \{\overline{\alpha_i : \sigma_i}^p,\ \overline{\beta_i : \sigma'_i}^q\}$$

$$\Gamma = \{x : t_S[\overline{\alpha}^p], \overline{x_i : \tau_i}^n\}$$

Then with (96), (97), and rule EQUIV-TY-SUBST

$$\eta \approx \rho \in [\![\Delta]\!]_{k'} \tag{102}$$

And with (98), (99), the definition of θ in (100), and rule EQUIV-VAL-SUBST

$$\theta \approx \rho \in [\![\eta\Gamma]\!]_{k'} \tag{103}$$

From the assumption $\vdash_{\mathsf{meth}} D \rightsquigarrow X_{m,t_S} = V$ we get by inverting rule METHOD

$$\langle \Delta, \Gamma \rangle \vdash_{\mathsf{exp}} e : \tau \rightsquigarrow E \tag{104}$$

$$V = \lambda((\overline{X_{\alpha_i}}^p), X, (\overline{X_{\beta_i}}^q), (\overline{X}^n)).E \tag{105}$$

With $k' < k+1$ we have $k' \leq k$. With the IH and Lemma A.3.7 then

$$\overline{D} \approx_{k'} \mu \tag{106}$$

(106), (102), (103) and (104) are the requirements of Lemma 5.2.4. The lemma then yields

$$\theta\eta e \approx \rho E \in [\![\eta\tau]\!]_{k'} \tag{107}$$

We also have

$$X_{m,t_S}\ ((\overline{W}^p), U, (\overline{W'}^q), (\overline{V}^n)) \longrightarrow V\ ((\overline{W}^p), U, (\overline{W'}^q), (\overline{V}^n)) \longrightarrow^* \rho E$$

where the first reduction follows from assumption $\mu(X_{m,t_S}) = V$ and rule TL-METHOD, the remaining steps by (105) and (101). With (107) and Lemma 5.2.2 we then get (100) as required. ∎

A.3.2.3 Proof of Theorem 5.2.6. We first prove that the assumptions of the theorem imply $e \approx E \in [\![\tau]\!]_k$ for any k. $\overline{D}$ and μ are the declarations and the substitution whose existence we assumed globally. Obviously, they meet the requirements of Assumption 5.2.1.

Assume $k \in \mathbb{N}$. From Lemma 5.2.5 we get $\overline{D} \approx_k \mu$. By the assumption $\vdash_{\mathsf{prog}} \overline{D}$ **func** $\mathsf{main}()\{\_ = e\} \rightsquigarrow$ **let** $\overline{X_i = V_i}$ **in** E, by inverting rule PROG, and by the assumption that e has type τ, we find $\langle\emptyset,\emptyset\rangle \vdash_{\mathsf{exp}} e : \tau \rightsquigarrow E$. Lemma 5.2.4 then yields $e \approx E \in [\![\tau]\!]_k$ as required.

From $e \approx E \in [\![\tau]\!]_k$ for any k and the two implications in the premise of rule EQUIV-EXP, we then get the two claims needed to show. ∎

A.3.3 Equivalence Between Different Translations

Lemma A.3.17. *If $v \equiv (\overline{V}) \in [\![\tau]\!]_k$ then none of the V_i is a lambda.*

Proof.
Case distinction on the last rule in the derivation of $v \equiv (\overline{V}) \in [\![\tau]\!]_k$.

- *Case* rule EQUIV-STRUCT: Then we know that $v = \tau_S\{\overline{v}\}$ and $\tau = \tau_S$ and for all i exists some σ_i with $v_i \equiv V_i \in [\![\sigma_i]\!]_k$. But then obviously V_i cannot be a lambda.
- *Case* rule EQUIV-IFACE: Obvious.

End case distinction. ∎

Lemma A.3.18. *If $v \equiv (U, (\overline{W})) \in [\![\tau_I]\!]_k$ with $k > 0$, then all W_i are lambdas.*

Proof. The derivation of $v \equiv (U, (\overline{W})) \in [\![\tau_I]\!]_k$ ends with rule EQUIV-IFACE. The premise of the rule gives us for each W_i

$$\mathsf{methodLookup}(\mathsf{methodName}(R_i), \sigma_S) \approx W_i \in [\![R_i]\!]_{k_2} \tag{1}$$

for some method signature R_i, struct type σ_S and all $k_2 < k$. As $k > 0$ we know that (1) holds for at least one k_2. Further, the derivation of (1) ends with rule EQUIV-METHOD-DICT-ENTRY and this rule requires that W_i is a lambda. ∎

Lemma A.3.19. *If $v \equiv V \in [\![t_I[\overline{\tau}]]\!]_k$ for all $k \in \mathbb{N}$, then $V = (U, (\overline{W}^n))$ where n is the number of methods defined by t_I, $v = \tau_S\{\overline{v}\}$, and $v \equiv U \in [\![\tau_S]\!]_k$ for all $k \in \mathbb{N}$.*

Proof. The derivation of $v \equiv V \in [\![t_I[\overline{\tau}]]\!]_k$ ends with rule EQUIV-IFACE for any $k \in \mathbb{N}$. Also for all $k \in \mathbb{N}$, the conclusion of this rule requires $V = (U, (\overline{W}^n))$, the premise of this rule states that interface t_I has n methods and further gives us

$$\exists \sigma_S . \forall k_1 < k . v \equiv U \in [\![\sigma_S]\!]_{k_1} \tag{1}$$

Obviously, $v \equiv U \in [\![\sigma_S]\!]_{k_1}$ ends with rule EQUIV-STRUCT. Because value v must have the form $v = \tau_S\{\overline{v}\}$, we then know that the existentially quantified σ_S is the same as τ_S. Because (1) holds for any $k \in \mathbb{N}$, we then get $v \equiv U \in [\![\tau_S]\!]_k$ for all $k \in \mathbb{N}$ as required. ■

Lemma A.3.20. *If* $v \equiv V \in [\![\tau]\!]_k$ *and* $v \equiv V' \in [\![\tau]\!]_k$ *for any* $k \in \mathbb{N}$*, then* $\mathsf{erase}(V) = \mathsf{erase}(V')$.

Proof. Define a measure function

$$\mathcal{M}(v,\tau) = \begin{cases} (|v|, 0) & \text{if } \tau \text{ is a struct type} \\ (|v|, 1) & \text{if } \tau \text{ is an interface type} \\ (|v|, 2) & \text{if } \tau \text{ is a type variable} \end{cases}$$

and proceed by induction on $\mathcal{M}(v,\tau)$. We first note that the derivations of $v \equiv V \in [\![\tau]\!]_k$ all end with the same rule, independent from $k \in \mathbb{N}$.
Case distinction on the last rule in the derivations of $v \equiv V \in [\![\tau]\!]_k$.

- *Case* rule EQUIV-STRUCT: Then τ is a struct type, so the derivations of $v \equiv V' \in [\![\tau]\!]_k$ also all end with EQUIV-STRUCT. Thus we have

$$v = t_S[\overline{\tau}]\{\overline{v}^n\}$$
$$V = (\overline{V}^n)$$
$$V' = (\overline{V'}^n)$$
$$\textsf{type } t_S[\overline{\alpha\ \tau_I}]\ \textsf{struct } \{\overline{f\ \sigma}^n\} \in \overline{D}$$
$$\eta = \langle \overline{\alpha \mapsto \tau} \rangle$$
$$(\forall i \in [n])\ v_i \equiv V_i \in [\![\eta\sigma_i]\!]_k \tag{1}$$
$$(\forall i \in [n])\ v_i \equiv V'_i \in [\![\eta\sigma_i]\!]_k \tag{2}$$

  As this holds for any $k \in \mathbb{N}$ and we have $\mathcal{M}(v_i, \eta\sigma_i) < \mathcal{M}(v,\tau)$, we may apply the IH to (1) and (2) and get $\mathsf{erase}(V_i) = \mathsf{erase}(V'_i)$ for all $i \in [n]$. Then $\mathsf{erase}(V) = \mathsf{erase}(V')$ follows by definition of erase.

- *Case* rule EQUIV-IFACE: Then τ is interface type. With Lemma A.3.19 then for some σ_S and n

$$V = (U, (\overline{W}^n))$$
$$V' = (U', (\overline{W'}^n))$$
$$(\forall k \in \mathbb{N}).v \equiv U \in [\![\sigma_S]\!]_k \tag{3}$$
$$(\forall k \in \mathbb{N}).v \equiv U' \in [\![\sigma_S]\!]_k \tag{4}$$

  Noting that $\mathcal{M}(v,\sigma_S) < \mathcal{M}(v,\tau)$, we apply the IH to (3) and (4) and get $\mathsf{erase}(U) = \mathsf{erase}(U')$. With Lemma A.3.18 applied to assumptions $(\forall k \in \mathbb{N}).v \equiv V \in [\![\tau]\!]_k$ and $(\forall k \in \mathbb{N}).v \equiv V' \in [\![\tau]\!]_k$, we know that all W_i, W'_i are lambdas. Thus by definition of erase

$$\mathsf{erase}(V) = (\mathsf{erase}(U), (\overline{\mathbf{K}_\lambda}^n)) = \mathsf{erase}(V')$$

End case distinction. ■

Lemma A.3.21. *If* $v \equiv V \in [\![\tau]\!]_k$ *and* $v \equiv V' \in [\![\tau']\!]_k$ *for any* $k \in \mathbb{N}$*, then* $\mathsf{erase}(\tau, V) = \mathsf{erase}(\tau', V')$.

Proof. We label the assumptions:

$$(\forall k \in \mathbb{N}).v \equiv V \in [\![\tau]\!]_k \tag{1}$$

$$(\forall k \in \mathbb{N}).v \equiv V' \in [\![\tau']\!]_k \tag{2}$$

We then perform a case distinction on the form of τ and τ'. Note that neither of them can be a type variable, otherwise (1) and (2) would not be derivable.
Case distinction on the forms of τ and τ'.

- *Case* τ and τ' are both struct types: Then all derivations of (1) and (2) end with rule EQUIV-STRUCT. Hence $\tau = \tau'$. Thus $\mathsf{erase}(V) = \mathsf{erase}(V')$ by Lemma A.3.20. But by definition of erase, we also have $\mathsf{erase}(\tau, V) = \mathsf{erase}(V)$ and $\mathsf{erase}(\tau', V') = \mathsf{erase}(V)$.
- *Case* τ is a struct type and τ' is an interface type: Then all derivations of (1) end with rule EQUIV-STRUCT, so we know that $v = \tau\{\overline{v}\}$. Then we get with Lemma A.3.19 and (2)

  $$V' = (U, W)$$
  $$(\forall k \in \mathbb{N}).v \equiv U \in [\![\tau]\!]_k$$

  With (1) and Lemma A.3.20 and the definition of erase then

  $$\mathsf{erase}(\tau, \mathsf{erase}(V)) = \mathsf{erase}(V) = \mathsf{erase}(U) = \mathsf{erase}(\tau', V')$$

- *Case* τ is an interface type and τ' is a struct type: Analogously to the preceding case.
- *Case* τ and τ' are both interface types: Then with Lemma A.3.19 and (1) and (2)

  $$v = \sigma_S\{\overline{v}\}$$
  $$V = (U, W)$$
  $$V' = (U', W')$$
  $$(\forall k \in \mathbb{N}).v \equiv U \in [\![\sigma_S]\!]_k$$
  $$(\forall k \in \mathbb{N}).v \equiv U' \in [\![\sigma_S]\!]_k$$

  Now Lemma A.3.20 and the definition of erase

  $$\mathsf{erase}(\tau, V) = \mathsf{erase}(U) = \mathsf{erase}(U') = \mathsf{erase}(\tau', V')$$

  as required.

End case distinction. ■

A.3.3.1 Proof of Theorem 5.2.7.

From $\vdash_{\mathsf{prog}} P \rightsquigarrow \mathbf{let}\ \overline{X_i = V_i}\ \mathbf{in}\ E$ and $\vdash_{\mathsf{prog}} P \rightsquigarrow \mathbf{let}\ \overline{X_i' = V_i'}\ \mathbf{in}\ E'$ and e having type τ and τ', respectively, we get

$$\langle \emptyset, \emptyset \rangle \vdash_{\mathsf{exp}} e : \tau \rightsquigarrow E$$
$$\langle \emptyset, \emptyset \rangle \vdash_{\mathsf{exp}} e : \tau' \rightsquigarrow E'$$

With Corollary 5.1.2, we get that either e reduces to some value v or diverges.

We now start with the first claim. Assume $E \longrightarrow_{\mu}^{*} V$ for some V. Then e must reduce to some value v because of Theorem 5.2.6. Again with Theorem 5.2.6 and with Lemma A.1.2:

$$v \equiv V \in [\![\tau]\!]_k \quad (\forall k \in \mathbb{N}) \tag{3}$$

$$E \longrightarrow_{\mu'}^{*} V' \quad \text{for some } V'$$

$$v \equiv V' \in [\![\tau']\!]_k \quad (\forall k \in \mathbb{N}) \tag{4}$$

Applying Lemma A.3.21 yields $\mathsf{erase}(\tau, V) = \mathsf{erase}(\tau', V')$ as required.

For the second claim, we assume that E diverges. With Theorem 5.2.6, we know that e must diverge as well. Again with Theorem 5.2.6 we get that E' also diverges. ■