Gradual Guarantee for FJ with lambda-Expressions

Pedro Ângelo
DCC-FCUP & LIACC, Universidade do Porto, Porto, Portugal, up201207861@edu.fc.up.pt

Viviana Bono
Dipartimento di Informatica, University of Torino, Italy, viviana.bono@unito.it

Mariangiola Dezani-Ciancaglini
Dipartimento di Informatica, University of Torino, Italy, dezani@di.unito.it

Mário Florido
DCC-FCUP & LIACC, Universidade do Porto, Porto, Portugal, amflorid@fc.up.pt

ABSTRACT

We present FJ&$\lambda\star$, a new core calculus that extends *Featherweight Java* (FJ) with interfaces, λ-*expressions*, *intersection types* and a form of *dynamic type*. Intersection types can be used anywhere, in particular to specify *target types* of λ-expressions. The dynamic type is exploited to specify parts of the class tables and programs we want to exclude temporarily from static typing. Our main result is the *gradual guarantee*, which says that if a program is well typed in a class table, then replacing type annotations (from the program and from the class table) with the dynamic type always produces a program that is still well typed in the obtained class table. Furthermore, if a typed program evaluates to a value in a class table, then replacing type annotations with dynamic types always produces a program that evaluates to the same value in the obtained class table.

CCS CONCEPTS

• **Theory of computation → Object oriented constructs**; **Type structures**.

KEYWORDS

Featherweight Java, λ-expressions, Gradual Typing, Intersection Types

ACM Reference Format:
Pedro Ângelo, Viviana Bono, Mariangiola Dezani-Ciancaglini, and Mário Florido. 2023. Gradual Guarantee for FJ with lambda-Expressions. In *Proceedings of the 25th ACM International Workshop on Formal Techniques for Java-like Programs (FTfJP '23), July 18, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3605156.3606453

This work was partially supported by: *(i)* the Portuguese Fundação para a Ciência e a Tecnologia, under the PhD grant number SFRH/BD/145183/2019, and by Base Funding - UIDB/00027/2020 of the Artificial Intelligence and Computer Science Laboratory – LIACC - funded by national funds through the FCT/MCTES (PIDDAC); *(ii)* the EuroHPC JU by way of the ADMIRE project (G.A. n. 956748) and by the Spoke 1 "FutureHPC & BigData" of ICSC – Centro Nazionale di Ricerca in High-Performance Computing, Big Data and Quantum Computing, funded by European Union – NextGenerationEU.

FTfJP '23, July 18, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0246-4/23/07…$15.00
https://doi.org/10.1145/3605156.3606453

1 INTRODUCTION

Type verification may be performed either at compile type or at run-time. These two approaches are called *static* and *dynamic* typing, respectively, and each has its own benefits and drawbacks. Integrating these two disciplines has become quite an active area of research in recent years, especially since the introduction of gradual typing [9, 10]. Gradual typing accomplishes this by allowing programmers to choose which portions of the program are statically typed and which are dynamically typed. By annotating code with the dynamic type $\star$, the type system is relaxed, allowing type checking for that portion of code to be delayed until runtime. In this way, the expressiveness of the type system is increased, because possible type errors are limited to dynamically annotated portions of code, then plausibly correct programs type-check. To ensure errors are eventually caught at run-time, runtime type checks are inserted during compilation, into the borders between statically and dynamically typed code, via a phase called *cast insertion*.

Following the dichotomy above, Java [4] and its formalisations [3, 6] employ static typing. In this paper we aim to add the dynamic type $\star$ to the calculus of [3], where *Featherweight Java* (FJ) [6] is enriched with interfaces, λ-expressions and intersection types. According to [4] (page 93), Java uses for λ-expressions the types required by the contexts enclosing them, called *target types*. In [3] target types can be intersections of interfaces including several abstract methods with different signatures. Thus target types are able to express multiple, possibly unrelated, properties of λ-expressions. Similarly to [3] we achieve this by:

• adding casts in the reduction rules for field access and method invocation;
• reducing the cast of a λ-expression to the λ-expression decorated by its target type (decorated λ-expressions are values).

We are thus working in the direction of getting a core Java 8 calculus (dubbed FJ&$\lambda\star$) with the addition of a form of dynamic type and target types as intersections of interfaces with more than one abstract method. However, the full gradual typing approach, that is, compiling programs written in the source calculus into an intermediate language with dynamic casts inserted into the borders between statically and dynamically typed code, seems not possible in our setting (or at least, it appears to be very difficult). Indeed, FJ&$\lambda\star$ (and Java) λ-expressions can have target types containing a different set of default methods for each possible target type and there is no evident way the right target type can be inferred *statically*, that is, without knowing the context a λ-expression will be used in at runtime. In particular, target types are needed for calling default methods on λ-expressions (see Rule [MethDλ] in Figure 3), therefore they must be present *at runtime*.

Nevertheless, a novel approach to partial static typing, still exploiting dynamic type annotations, is possible in our setting, too, where no compilation step is present and no casts to ⋆ are thus inserted before evaluation. Note that such casts can appear during evaluation, together with the other casts. We propose then to use ⋆ in declarations and to type with ⋆ field accesses and method invocations requiring only that the subterms can be typed. In this way, we obtain a calculus enjoying the *gradual guarantee*, relating the behaviour of programs that differ only with respect to their dynamic type annotations. As pointed out in [11], this is the keystone in the formal characterisation of being gradually typed.

The prototypical example showing the full power of λ-calculi with intersection types is the type $(\alpha\&(\alpha \to \beta)) \to \beta$ (where α, β are arbitrary types) for the auto-application function $\lambda x.x\,x$. The arrow denotes the function type constructor and the intersection type $\alpha\&(\alpha \to \beta)$ says that the parameter must behave both as function and as argument of itself. *We can type the auto-application function in FJ&λ⋆*, too, since we can define the method

```
⋆ auto (Arg&Fun x){return x.mFun(x).mArg(new C( ));}
```

where `C` is any class without fields, `Arg` and `Fun` are two interfaces with the abstract methods `⋆ mArg (⋆ y)` and `⋆ mFun (⋆ z)`, respectively. This method `auto` can take as argument any λ-expression of arity one and, as expected, the resulting term can reduce to a value, or diverge or reduce to error.

Outline. In Section 2 we supply the class table and the lookup function definitions. In the following two sections we present the type assignment system and the operational semantics, respectively. Section 5 is devoted to the proof of safety, which we dub *weak* due to the possibility of evaluating well-typed terms to error. Our main result, i.e., the gradual guarantee, is the content of Section 6. We conclude with related works. The full version of the paper (with complete proofs) can be found as auxiliary material.

2 SYNTAX

We use C, D to denote classes, I, J to denote interfaces, T, U to denote nominal types, i.e., either classes or interfaces, and ⋆ to denote the dynamic type. Types are ranged over by τ, σ and *target types* are ranged over by φ (see Definition 2.2).

We use f, g to denote field names; m, n to denote method names; t to denote terms; x, y to denote variables, including the special variable this. We use $\overrightarrow{t}$ as a shorthand for the list $t_1, \ldots, t_n$, $\overline{M}$ as a shorthand for the sequence $M_1 \ldots M_n$, and similarly for the other names. The order in lists and sequences is sometimes unimportant, and this is clear from the context. In rules, we write both $\overline{N}$ as a declaration and $\overrightarrow{N}$ for some name N: the meaning is that a sequence is declared and the list is obtained from the sequence adding commas. The notation $\overline{\tau}\,\overline{f}$; abbreviates $\tau_1 f_1; \ldots \tau_n f_n$; and $\overrightarrow{\tau}\,\overrightarrow{f}$ abbreviates $\tau_1 f_1, \ldots, \tau_n f_n$ (likewise $\overrightarrow{\tau}\,\overrightarrow{x}$) and this.$\overline{f} = \overline{f}$; abbreviates this.$f_1 = f_1; \ldots$ this.$f_n = f_n;$. This convention on $\overline{\ }$ and $\overrightarrow{\ }$ is also used in the typing and reduction rules. Sequences of interfaces, fields, parameters and methods are assumed to contain no duplicate names. The keyword super, used only in constructor's body, refers to the superclass constructor.

The syntax of terms, classes and interfaces of FJ&λ⋆ is given in the following definition.

Definition 2.1 (Terms, Classes, Interfaces).

$$
\begin{array}{lcl}
t & ::= & v \mid x \mid t.f \mid t.m(\overrightarrow{t}) \mid \mathsf{new}\,C(\overrightarrow{t}) \mid (\tau)\,t \\
v & ::= & w \mid \overrightarrow{x} \to t \\
w & ::= & \mathsf{new}\,C(\overrightarrow{v}) \mid (\overrightarrow{x} \to t)^{\varphi} \\
CD & ::= & \mathsf{class}\,C\,\mathsf{extends}\,D\,\mathsf{implements}\,\overrightarrow{I}\,\{\overline{\tau}\,\overline{f}; K\,\overline{M}\} \\
ID & ::= & \mathsf{interface}\,I\,\mathsf{extends}\,\overrightarrow{I}\,\{\overline{H}; \overline{M}\} \\
K & ::= & C(\overrightarrow{\tau}\,\overrightarrow{f})\{\mathsf{super}(\overrightarrow{f}); \mathsf{this}.\overline{f} = \overline{f};\} \\
H & ::= & \tau m(\overrightarrow{\tau}\,\overrightarrow{x}) \\
M & ::= & H\,\{\mathsf{return}\,t;\}
\end{array}
$$

Terms are values, variables, field accesses, method calls, object creations and casts. Values include λ-expressions, ranged over by t_λg. We distinguish between values (ranged over by v, u) and *proper values* (ranged over by w). A pure λ-expression is a value, while a λ-expression decorated by its *target type* φ is a proper value. Decorated λ-expressions are produced at run-time only.

CD ranges over class declarations; ID ranges over interface declarations; K ranges over constructor declarations; H ranges over method header (abstract method) declarations; M ranges over method declarations. Thus, an interface declaration can contain not only abstract methods, but also concrete methods with a default implementation. For simplicity, we omit the keyword *default* and the parentheses around parameters of λ-expressions.

A *class table CT* is a mapping from nominal types to their declarations. `Object` is a special class without fields and methods and it is not included in the class table.

A special role is played by the *target types*, that can be assigned to λ-expressions by the contexts of usage. These types for us are intersections of interfaces, which can declare a method name at most once. In Java, instead, target types are intersections of interfaces which must declare exactly one abstract method. *Types* can also be intersections of one class with interfaces and with the dynamic type ⋆, which can declare a method name at most once. The type syntax is given in the following definition, proviso that intersections satisfy the unicity of method names.

Definition 2.2 (Types). (1) *Target types* are: $\varphi ::= I \mid \varphi\&I$
(2) *Types* are: $\tau ::= C \mid \iota \mid C\&\iota$ where $\iota ::= \star \mid \varphi \mid \iota\&I$

The notation $C[\&\iota]$ means either the class C or the type $C\&\iota$.

We assume that there are no cycles in the subclass relation between nominal types induced by the class table.

Lookup functions for a given class table *CT* are as follows, where we use inheritance and overriding as expected:

- $\mathtt{Am}(CT; \varphi)$ gives the set of abstract methods defined in φ;
- $\mathtt{fields}(CT; C)$ gives the sequence of fields declarations in class C;
- $\mathtt{mtype}(CT; m; \tau)$ gives the parameter and return types of method m in τ;
- $\mathtt{mbody}(CT; m; \tau)$ gives the formal parameters and the body of method m in τ.

3 TYPE ASSIGNMENT SYSTEM

We compare types by means of the subtyping, precision and convertibility relations.

$$\frac{\mathsf{x} : \tau \in \Delta}{\Delta \vdash_{CT} \mathsf{x} : \tau}\ [\text{VAR}] \qquad \frac{\Delta \vdash_{CT} \mathsf{t} : \mathsf{C}[\&\iota] \quad \tau\,\mathsf{f} \in \mathtt{fields}(CT;\mathsf{C})}{\Delta \vdash_{CT} \mathsf{t.f} : \tau}\ [\text{FIELD}] \qquad \frac{\Delta \vdash_{CT} \mathsf{t} : \tau}{\Delta \vdash_{CT} \mathsf{t.f} : \star}\ [\star\text{FIELD}]$$

$$\frac{\Delta \vdash_{CT} \mathsf{t} : \tau \quad \mathtt{mtype}(CT;\mathsf{m};\tau) = \overrightarrow{\sigma} \to \sigma \quad \Delta \vdash^{\flat}_{CT} \overline{\mathsf{t}} : \overline{\sigma}}{\Delta \vdash_{CT} \mathsf{t.m}(\overrightarrow{\mathsf{t}}) : \sigma}\ [\text{INVK}] \qquad \frac{\Delta \vdash_{CT} \mathsf{t} : \tau \quad \Delta \vdash^{\flat}_{CT} \overline{\mathsf{t}} : \overline{\tau}}{\Delta \vdash_{CT} \mathsf{t.m}(\overrightarrow{\mathsf{t}}) : \star}\ [\star\text{INVK}]$$

$$\frac{\mathtt{fields}(CT;\mathsf{C}) = \overrightarrow{\tau}\,\overrightarrow{\mathsf{f}} \quad \Delta \vdash^{\flat}_{CT} \overline{\mathsf{t}} : \overline{\tau}}{\Delta \vdash_{CT} \mathsf{new\ C}(\overrightarrow{\mathsf{t}}) : \mathsf{C}}\ [\text{NEW}] \qquad \frac{\mathsf{m} \in \mathtt{Am}(CT;\varphi) \text{ and } \mathtt{mtype}(CT;\mathsf{m};\varphi) = \overrightarrow{\tau} \to \tau \text{ imply } \Delta, \overrightarrow{\mathsf{y}} : \overrightarrow{\tau} \vdash^{\flat}_{CT} \mathsf{t} : \tau}{\Delta \vdash_{CT} (\overrightarrow{\mathsf{y}} \to \mathsf{t})^{\varphi} : \varphi}\ [\lambda]$$

$$\frac{\Delta \vdash_{CT} (\mathsf{t}_{\lambda})^{\varphi} : \varphi}{\Delta \vdash^{\flat}_{CT} \mathsf{t}_{\lambda} : \varphi}\ [\flat\lambda] \qquad \frac{\Delta \vdash_{CT} \mathsf{t} : \sigma \quad \mathsf{t} \neq \mathsf{t}_{\lambda} \quad \sigma \sqsubseteq \tau}{\Delta \vdash^{\flat}_{CT} \mathsf{t} : \tau}\ [\flat\neg\lambda] \qquad \frac{\Delta \vdash_{CT} \mathsf{t} : \sigma \quad \sigma \Rightarrow \tau}{\Delta \vdash_{CT} (\tau)\,\mathsf{t} : \tau}\ [\text{UC}]$$

$$\frac{\Delta \vdash_{CT} \mathsf{t} : \sigma \quad \tau \Rightarrow \sigma}{\Delta \vdash_{CT} (\tau)\,\mathsf{t} : \tau}\ [\text{DC}] \qquad \frac{\Delta \vdash_{CT} \mathsf{t} : \sigma \quad \sigma \not\Rightarrow \tau \quad \tau \not\Rightarrow \sigma}{\Delta \vdash_{CT} (\tau)\,\mathsf{t} : \tau}\ [\text{SC}] \qquad \frac{\Delta \vdash_{CT} (\mathsf{t}_{\lambda})^{\varphi} : \varphi}{\Delta \vdash_{CT} (\varphi)\,\mathsf{t}_{\lambda} : \varphi}\ [\lambda\text{C}]$$

Figure 1: Term typing rules for class table CT.

$$\frac{\mathtt{mtype}(CT;\mathsf{m};\tau) = \overrightarrow{\tau} \to \tau \quad \overrightarrow{\mathsf{x}} : \overrightarrow{\tau}, \mathsf{this} : \mathsf{T} \vdash^{\flat}_{CT} \mathsf{t} : \tau}{\tau\mathsf{m}(\overrightarrow{\tau}\,\overrightarrow{\mathsf{x}})\{\mathsf{return\ t;}\}\ CT\text{-OK in T}}\ [\mathsf{M}\ CT\text{-OK in T}] \qquad \frac{\overline{\mathsf{M}\ CT\text{-OK in I}}}{\mathsf{interface\ I\ extends}\ \overrightarrow{\mathsf{I}}\ \{\overline{\mathsf{H}};\overline{\mathsf{M}}\}\ \mathsf{OK}}\ [\mathsf{I}\ CT\text{-OK}]$$

$$\frac{\begin{array}{c}\mathsf{K} = \mathsf{C}(\overrightarrow{\tau}\,\overrightarrow{\mathsf{g}}, \overrightarrow{\sigma}\,\overrightarrow{\mathsf{f}})\{\mathsf{super}(\overrightarrow{\mathsf{g}}); \mathsf{this.}\overline{\mathsf{f}} = \overline{\mathsf{f}};\} \quad \mathtt{fields}(CT;\mathsf{D}) = \overrightarrow{\tau}\,\overrightarrow{\mathsf{g}} \quad \overline{\mathsf{M}\ CT\text{-OK in C}} \\ \mathtt{mtype}(CT;\mathsf{m};\mathsf{C}) \text{ defined implies } \mathtt{mbody}(CT;\mathsf{m};\mathsf{C}) \text{ defined}\end{array}}{\mathsf{class\ C\ extends\ D\ implements}\ \overrightarrow{\mathsf{I}}\ \{\overline{\tau}\,\overline{\mathsf{f}};\mathsf{K}\,\overline{\mathsf{M}}\}\ \mathsf{OK}}\ [\mathsf{C}\ CT\text{-OK}]$$

Figure 2: Method, class and interface declaration typing rules for class table CT.

The *subtype relation* $<:$ takes into account both the subclass relation induced by the class table, and the set theoretic properties of intersection:

$$\frac{\tau <: \mathsf{T}_i \quad \text{for all } 1 \le i \le n}{\tau <: \mathsf{T}_1 \& \ldots \& \mathsf{T}_n}\ [<:\&\text{R}] \qquad \frac{\mathsf{T}_i <: \tau \quad \text{for some } 1 \le i \le n}{\mathsf{T}_1 \& \ldots \& \mathsf{T}_n <: \tau}\ [<:\&\text{L}]$$

The dynamic type is only subtype of itself: $\star <: \star$.

In the *precision relation* $\star$ is the top type. This relation plays a fundamental role in formalising the gradual guarantee, see [11]. It is denoted by $\sqsubseteq$, and it is the transitive closure of

$$\tau \sqsubseteq \star \qquad \frac{\tau <: \sigma}{\tau \sqsubseteq \sigma} \qquad \frac{\tau \sqsubseteq \tau' \ \sigma \sqsubseteq \sigma'}{\tau\&\sigma \sqsubseteq \tau'\&\sigma'}$$

The *convertibility relation* relates $\star$ to any type. It is denoted by $\Rightarrow$, and it is defined by

$$\star \Rightarrow \tau \qquad \tau \Rightarrow \star \qquad \frac{\tau <: \sigma}{\tau \Rightarrow \sigma} \qquad \frac{\tau \Rightarrow \tau' \ \sigma \Rightarrow \sigma'}{\tau\&\sigma \Rightarrow \tau'\&\sigma'}$$

Note that convertibility is not transitive.

We write $\tau \not\sqsubseteq \sigma$ if $\tau \sqsubseteq \sigma$ does not hold and similarly for $\not\Rightarrow$.

These relations depend on the subclass relation induced by a given class table. Such a class table is contextually identifiable when we use the relations in typing and reduction rules.

Figure 1 presents the typing rules for terms. As in [3], we can use intersections everywhere and we avoid the restriction that target types must have a single abstract method. Rules [VAR] and [FIELD] are standard.

The auxiliary judgments $\vdash^{\flat}_{CT}$ derive a target type for a pure λ-expression (Rule [$\flat\lambda$]) and a less precise type for a term which is not a pure λ-expression (Rule [$\flat\neg\lambda$]). These judgments are used in Rules [INVK], [$\star$INVK] and [NEW] to deal with actual parameters, since pure λ-expression can only have the exact target type required by the context, while all others terms can have any type which is more precise than the type of the matching formal parameter.

To type a decorated λ-expression Rule [λ] requires that the body of the λ-expression be well typed for all the headers of the abstract methods declared in the interfaces occurring in its target type. The body of the λ-expression is typed by means of $\vdash^{\flat}_{CT}$ to use the correct typing judgement for pure λ-expressions and other terms. Note that, in the type system $\vdash_{CT}$ there is no typing rule for pure λ-expressions, since we expect each λ-expression to be decorated with its target type before typing its body.

Rules [$\star$FIELD] and [$\star$INVK] type field selection and method call with the dynamic type $\star$ requiring only that all subterms must be typed. Note that we can use these rules also when the corresponding standard rules are applicable. This implies that the same term can have more than one type, which is typical of intersection types.

As in [3], a pure λ-expression can be casted to a target type if the corresponding decorated λ-expression can be typed by exactly that target type (Rule [λC]). The other cast rules are the rules of [6] where subtyping is replaced by convertibility.

We write $\not\vdash^{\flat}_{CT} v : \tau$ if $\vdash^{\flat}_{CT} v : \tau$ does not hold, and similarly for $\not\vdash^{\flat}_{CT} \overline{v} : \overline{\tau}$, where $\overline{v}$ and $\overline{\tau}$ can have different lengths.

The typing rules for method, class and interface declaration given in Figure 2 are standard, see [3].

4 OPERATIONAL SEMANTICS

Figure 3 depicts the reduction rules producing terms. As usual, $[x \mapsto t]$ denotes the substitution of x by t and it generalises to an arbitrary number of variables/terms as expected.

Rules [FieldNew] and [MethNew] are standard, but for the addition of the casts which are needed in order to associate λ-expression with their target types. A decorated λ-expression implements all the abstract methods declared in the interfaces of its target type, therefore in Rule [Meth$A\lambda$] the call of one of such methods reduces to the body of the λ-expression in which the formal parameters are substituted by the actual ones. In case the method called is one of the default methods with body t, the λ-expression acts as the object on which the method is called. Then the call reduces to t in which the (decorated) λ-expression replaces this and the actual parameters replace the formal ones (Rule [Meth$D\lambda$]).

Rule [CastNew] uses convertibility in agreement with the typing Rules [UC], [DC] and [SC]. The cast of a pure λ-expression to a target type reduces to the λ-expression decorated by the type (Rule [Castλ]). The cast of a decorated λ-expression behaves similarly to the cast of an object (Rule [CastλD]).

Finally Rule [Ctx] reduces inside evaluation contexts $\mathcal{E}$, which are defined by:

$$\mathcal{E} ::= [\,] \mid \mathcal{E}.f \mid \mathcal{E}.m(\overrightarrow{t}) \mid w.m(\overrightarrow{v}\mathcal{E}\overrightarrow{t}) \mid \mathsf{new}\,C(\overrightarrow{v}\mathcal{E}\overrightarrow{t}) \mid (\tau)\,\mathcal{E}$$

We use $\longrightarrow^{*}_{CT}$ to denote the reflexive and transitive closure of $\longrightarrow_{CT}$.

To formalise the gradual guarantee it is handy to have reduction rules producing Error when the reduction rules of Figure 3 cannot be applied. We use $\twoheadrightarrow_{CT}$ to denote reduction to Error and we give these rules in Figure 4, where $|\overrightarrow{t}|$ denotes the length of the list $\overrightarrow{t}$.

We write $t \longrightarrow_{CT}$ as short for $t \longrightarrow_{CT} t'$ for some t' and similarly for $\twoheadrightarrow_{CT}$.

5 PROPERTIES

This section is devoted to the proof of the safety property (Theorem 5.4). This ensures *only* that executions of typed terms never encounters untrapped errors [10], because of the presence of the dynamic type.

As usual, soundness is proved in two steps, commonly known as progress and preservation [8, Section 8.3], see Theorems 5.2 and 5.3. Also in these theorems we need to consider the cases in which terms reduce to Error. Therefore, we dub these properties *weak safety*, *weak progress* and *weak preservation*.

The key lemma to show weak progress relates the two reduction relations. It states that if a closed term cannot be reduced using $\longrightarrow_{CT}$, then it is a typable value, or a λ-expression (pure or decorated), or it reduces using $\twoheadrightarrow_{CT}$.

Lemma 5.1. *If* t *is a closed normal form for* $\longrightarrow_{CT}$ *different from a* λ*-expression with or without decorations, then either* t *is a typable value or* $t \twoheadrightarrow_{CT}$ Error.

Proof. By structural induction on t. We consider only the case $t = \mathsf{new}\,C(\overrightarrow{t})$. By induction, each term in $\overrightarrow{t}$ either reduces to Error or it is a typable value. If a term in $\overrightarrow{t}$ reduces to Error, then $t \twoheadrightarrow_{CT}$ Error by Rule [E-Ctx]. If all terms in $\overrightarrow{t}$ are typable values, then either t can be typed by Rule [NEW] or $t \twoheadrightarrow_{CT}$ Error by Rule [E-NewA]. □

The property of weak progress follows from the previous lemma.

Theorem 5.2 (Weak Progress). *If* $\vdash_{CT} t : \tau$*, then* $t \longrightarrow_{CT}$ *or* $t \twoheadrightarrow_{CT}$ *or* t *is a typable value.*

Proof. The condition $\vdash_{CT} t : \tau$ excludes t to be a λ-expression without decoration. If t is a decorated λ-expression, then t is a typable value since $\vdash_{CT} t : \tau$. Otherwise the statement follows from Lemma 5.1. □

By reducing a typed term, either we get a term with a more precise type, or we get a untypable term that reduces to Error.

Theorem 5.3 (Weak Preservation). *If* $\vdash_{CT} t : \tau$ *and* $t \longrightarrow_{CT} t'$*, then either* $\vdash_{CT} t' : \tau'$ *for some* τ' *such that* $\tau' \sqsubseteq \tau$ *or* $t' \twoheadrightarrow_{CT}$ Error.

Proof. The proof is by induction on $\longrightarrow_{CT}$ and by cases on the reduction rules. We present here some interesting cases only. Rule [Ctx]. In this case $t = \mathcal{E}[t_0]$, $t_0 \longrightarrow_{CT} t'_0$ and $t' = \mathcal{E}[t'_0]$. By definition of evaluation contexts t_0 and t'_0 are closed terms and t_0 can be typed. By induction either t'_0 can be typed or $t'_0 \twoheadrightarrow_{CT}$ Error. In the second case $t' \twoheadrightarrow_{CT}$ Error. We show the first case by structural induction on $\mathcal{E}$.

$\mathcal{E} = \mathsf{new}\,C(\overrightarrow{v}\mathcal{E}'\overrightarrow{t})$. Then t is typed using Rule [NEW], which implies $\tau = C$ and $\mathtt{fields}(CT; C) = \overrightarrow{\tau}\sigma\overrightarrow{\sigma}\overrightarrow{f}$ and $\vdash^{\flat}_{CT} \overline{v} : \overline{\tau}$ and $\vdash^{\flat}_{CT} \mathcal{E}'[t_0] : \sigma$ and $\vdash^{\flat}_{CT} \overline{t} : \overline{\sigma}$. Since a λ-expression cannot be reduced, $\vdash^{\flat}_{CT} \mathcal{E}'[t_0] : \sigma$ is derived by Rule [$\flat\neg\lambda$], which implies $\vdash_{CT} \mathcal{E}'[t_0] : \rho$ with $\rho \sqsubseteq \sigma$. By structural induction $\vdash_{CT} \mathcal{E}'[t'_0] : \rho'$ and $\rho' \sqsubseteq \rho$. By transitivity of $\sqsubseteq$ we get $\rho' \sqsubseteq \sigma$. We then derive $\vdash^{\flat}_{CT} \mathcal{E}'[t'_0] : \sigma$ by Rule [$\flat\neg\lambda$] and so $\vdash_{CT} t' : C$ by Rule [NEW].

$\mathcal{E} = (\tau)\,\mathcal{E}'$. Then t is typed using Rule [UC], [DC], or [SC], since a λ-expression cannot be reduced. This implies $\vdash_{CT} \mathcal{E}'[t_0] : \sigma$. By structural induction $\vdash_{CT} \mathcal{E}'[t'_0] : \sigma'$ and $\sigma' \sqsubseteq \sigma$. We can then derive $\vdash_{CT} t' : \tau$ using one of the Rules [UC], [DC], and [SC] according to the relations between τ and σ'. □

Weak safety follows easily from weak progress (Theorem 5.3) and weak preservation (Theorem 5.2).

Theorem 5.4 (Weak Safety). *If* $\vdash_{CT} t : \tau$*, then one of the following holds:*

- $t \longrightarrow_{CT} t'$ *and either* $\vdash_{CT} t' : \tau'$ *for some* τ' *such that* $\tau' \sqsubseteq \tau$ *or* $t' \twoheadrightarrow_{CT}$ Error
- $t \twoheadrightarrow_{CT}$ Error
- t *is a typable value.*

If we avoid the dynamic type we obtain the usual type safety as shown in the following theorem.

Theorem 5.5 (Safety). *If CT,* t *and* τ *do not contain the dynamic type, then* $\vdash_{CT} t : \tau$ *without using Rules* [DC], [SC], [$\star$FIELD], [$\star$INVK] *and* $t \longrightarrow^{*}_{CT} t'$ *where* t' *is a normal form imply that* t' *is a proper value and* $\vdash_{CT} t' : \tau'$*, for some* τ' *such that* $\tau' <: \tau$.

$$\frac{\texttt{fields}(CT;\mathsf{C}) = \overrightarrow{\tau}\ \overrightarrow{\mathsf{f}}}{\mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}}).\mathsf{f}_j \longrightarrow_{CT} (\tau_j)\mathsf{v}_j}\ [\text{FieldNew}] \qquad \frac{\texttt{mbody}(CT;\mathsf{m};\mathsf{C}) = (\overrightarrow{\mathsf{x}},\mathsf{t}) \quad \texttt{mtype}(CT;\mathsf{m};\mathsf{C}) = \overrightarrow{\tau} \to \tau}{\mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}}).\mathsf{m}(\overrightarrow{\mathsf{u}}) \longrightarrow_{CT} [\overrightarrow{\mathsf{x}} \mapsto (\overrightarrow{\tau})\overrightarrow{\mathsf{u}}, \mathsf{this} \mapsto \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}})](\tau)\mathsf{t}}\ [\text{MethNew}]$$

$$\frac{\mathsf{m} \in \texttt{Am}(CT;\varphi) \quad \texttt{mtype}(CT;\mathsf{m};\varphi) = \overrightarrow{\tau} \to \tau}{(\overrightarrow{\mathsf{y}} \to \mathsf{t})^{\varphi}.\mathsf{m}(\overrightarrow{\mathsf{v}}) \longrightarrow_{CT} [\overrightarrow{\mathsf{y}} \mapsto (\overrightarrow{\tau})\overrightarrow{\mathsf{v}}](\tau)\mathsf{t}}\ [\text{MethA}\lambda] \qquad \frac{\texttt{mbody}(CT;\mathsf{m};\varphi) = (\overrightarrow{\mathsf{x}},\mathsf{t}) \quad \texttt{mtype}(\mathsf{m};\varphi) = \overrightarrow{\tau} \to \tau}{(\mathsf{t}_\lambda)^{\varphi}.\mathsf{m}(\overrightarrow{\mathsf{v}}) \longrightarrow_{CT} [\overrightarrow{\mathsf{x}} \mapsto (\overrightarrow{\tau})\overrightarrow{\mathsf{v}}, \mathsf{this} \mapsto (\mathsf{t}_\lambda)^{\varphi}](\tau)\mathsf{t}}\ [\text{MethD}\lambda]$$

$$\frac{\mathsf{C} \Rrightarrow \tau}{(\tau)\,\mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}}) \longrightarrow_{CT} \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}})}\ [\text{CastNew}] \qquad \frac{}{(\varphi)\,\mathsf{t}_\lambda \longrightarrow_{CT} (\mathsf{t}_\lambda)^{\varphi}}\ [\text{Cast}\lambda]$$

$$\frac{\varphi \Rrightarrow \tau}{(\tau)\,(\mathsf{t}_\lambda)^{\varphi} \longrightarrow_{CT} (\mathsf{t}_\lambda)^{\varphi}}\ [\text{Cast}\lambda\text{D}] \qquad \frac{\mathsf{t} \longrightarrow_{CT} \mathsf{t}'}{\mathcal{E}[\mathsf{t}] \longrightarrow_{CT} \mathcal{E}[\mathsf{t}']}\ [\text{Ctx}]$$

Figure 3: Reduction rules of $\longrightarrow_{CT}$.

$$\frac{\texttt{fields}(CT;\mathsf{C}) = \overrightarrow{\tau}\ \overrightarrow{\mathsf{f}} \quad \mathsf{g} \notin \overrightarrow{\mathsf{f}}}{\mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}}).\mathsf{g} \hookrightarrow_{CT} \text{Error}}\ [\text{E-FNew}] \qquad \frac{}{\mathsf{t}_\lambda.\mathsf{f} \hookrightarrow_{CT} \text{Error}}\ [\text{E-F}\lambda] \qquad \frac{}{(\mathsf{t}_\lambda)^{\varphi}.\mathsf{f} \hookrightarrow_{CT} \text{Error}}\ [\text{E-F}\lambda\text{D}]$$

$$\frac{\texttt{mbody}(CT;\mathsf{m};\mathsf{C})\ \text{undefined}}{\mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}}).\mathsf{m}(\overrightarrow{\mathsf{u}}) \hookrightarrow_{CT} \text{Error}}\ [\text{E-MNew1}] \qquad \frac{\texttt{mbody}(CT;\mathsf{m};\mathsf{C}) = (\overrightarrow{\mathsf{x}},\mathsf{t}) \quad |\overrightarrow{\mathsf{x}}| \neq |\overrightarrow{\mathsf{u}}|}{\mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}}).\mathsf{m}(\overrightarrow{\mathsf{u}}) \hookrightarrow_{CT} \text{Error}}\ [\text{E-MNew2}]$$

$$\frac{}{\mathsf{t}_\lambda.\mathsf{m}(\overrightarrow{\mathsf{v}}) \hookrightarrow_{CT} \text{Error}}\ [\text{E-M}\lambda] \qquad \frac{\mathsf{m} \notin \texttt{Am}(CT;\varphi) \quad \texttt{mbody}(CT;\mathsf{m};\varphi)\ \text{undefined}}{(\mathsf{t}_\lambda)^{\varphi}.\mathsf{m}(\overrightarrow{\mathsf{v}}) \hookrightarrow_{CT} \text{Error}}\ [\text{E-M}\lambda\text{D1}]$$

$$\frac{\mathsf{m} \in \texttt{Am}(CT;\varphi) \quad |\overrightarrow{\mathsf{x}}| \neq |\overrightarrow{\mathsf{v}}|}{(\overrightarrow{\mathsf{x}} \to \mathsf{t})^{\varphi}.\mathsf{m}(\overrightarrow{\mathsf{v}}) \hookrightarrow_{CT} \text{Error}}\ [\text{E-M}\lambda\text{D2}] \qquad \frac{\texttt{mbody}(CT;\mathsf{m};\varphi) = (\overrightarrow{\mathsf{x}},\mathsf{t}) \quad |\overrightarrow{\mathsf{x}}| \neq |\overrightarrow{\mathsf{v}}|}{(\mathsf{t}_\lambda)^{\varphi}.\mathsf{m}(\overrightarrow{\mathsf{v}}) \hookrightarrow_{CT} \text{Error}}\ [\text{E-M}\lambda\text{D3}]$$

$$\frac{\texttt{fields}(CT;\mathsf{C}) = \overrightarrow{\tau}\ \overrightarrow{\mathsf{f}} \quad \not\vdash^{b}_{CT} \overline{\mathsf{v}} : \overline{\tau}}{\mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}}) \hookrightarrow_{CT} \text{Error}}\ [\text{E-NewA}] \qquad \frac{\mathsf{C} \not\Rrightarrow \tau}{(\tau)\,\mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}}) \hookrightarrow_{CT} \text{Error}}\ [\text{E-CastNew}]$$

$$\frac{\not\vdash_{CT} (\mathsf{t}_\lambda)^{\tau} : \tau}{(\tau)\,\mathsf{t}_\lambda \hookrightarrow_{CT} \text{Error}}\ [\text{E-Cast}\lambda] \qquad \frac{\varphi \not\Rrightarrow \tau}{(\tau)\,(\overrightarrow{\mathsf{x}} \to \mathsf{t})^{\varphi} \hookrightarrow_{CT} \text{Error}}\ [\text{E-Cast}\lambda\text{D}] \qquad \frac{\mathsf{t} \hookrightarrow_{CT} \text{Error}}{\mathcal{E}[\mathsf{t}] \hookrightarrow_{CT} \text{Error}}\ [\text{E-Ctx}]$$

Figure 4: Reduction rules of $\hookrightarrow_{CT}$.

Proof. The type system without Rules [⋆FIELD] and [⋆INVK] is the type system of [3], where the theorem is proved. □

6 GRADUAL GUARANTEE

In order to formalise the gradual guarantee (Theorem 6.5) we need to define the *precision* relation between terms, class and interface declarations, object constructors, method headers, method declarations and class tables.

Definition 6.1 (Term Precision). Term precision is the reflexive and transitive relation defined by:

(1) if $\mathsf{t} \sqsubseteq \mathsf{t}'$, then $\mathsf{t.f} \sqsubseteq \mathsf{t}'.\mathsf{f}$;
(2) if $\mathsf{t} \sqsubseteq \mathsf{t}'$ and $\overrightarrow{\mathsf{t}} \sqsubseteq \overrightarrow{\mathsf{t}'}$, then $\mathsf{t.m}(\overrightarrow{\mathsf{t}}) \sqsubseteq \mathsf{t}'.\mathsf{m}(\overrightarrow{\mathsf{t}'})$;
(3) if $\overrightarrow{\mathsf{t}} \sqsubseteq \overrightarrow{\mathsf{t}'}$, then $\mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{t}}) \sqsubseteq \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{t}'})$;
(4) if $\mathsf{t} \sqsubseteq \mathsf{t}'$ and $\tau \sqsubseteq \tau'$, then $(\tau)\,\mathsf{t} \sqsubseteq (\tau')\,\mathsf{t}'$;
(5) if $\mathsf{t} \sqsubseteq \mathsf{t}'$, then $\overrightarrow{\mathsf{x}} \to \mathsf{t} \sqsubseteq \overrightarrow{\mathsf{x}} \to \mathsf{t}'$;
(6) if $\mathsf{t} \sqsubseteq \mathsf{t}'$ and $\varphi \sqsubseteq \varphi'$, then $(\overrightarrow{\mathsf{x}} \to \mathsf{t})^{\varphi} \sqsubseteq (\overrightarrow{\mathsf{x}} \to \mathsf{t}')^{\varphi'}$.

Definition 6.2.
(1) if $\overline{\tau} \sqsubseteq \overline{\tau'}$ and $\mathsf{K} \sqsubseteq \mathsf{K}'$ and $\overline{\mathsf{M}} \sqsubseteq \overline{\mathsf{M}'}$, then class C extends D implements $\overrightarrow{\mathsf{I}}$ $\{\overline{\tau}\,\overline{\mathsf{f}}; \mathsf{K}\,\overline{\mathsf{M}}\} \sqsubseteq$ class C extends D implements $\overrightarrow{\mathsf{I}}$ $\{\overline{\tau'}\,\overline{\mathsf{f}}; \mathsf{K}'\,\overline{\mathsf{M}'}\}$;
(2) if $\overline{\mathsf{H}} \sqsubseteq \overline{\mathsf{H}'}$ and $\overline{\mathsf{M}} \sqsubseteq \overline{\mathsf{M}'}$, then interface I extends $\overrightarrow{\mathsf{I}}$ $\{\overline{\mathsf{H}}; \overline{\mathsf{M}}\} \sqsubseteq$ interface I extends $\overrightarrow{\mathsf{I}}$ $\{\overline{\mathsf{H}'}; \overline{\mathsf{M}'}\}$;
(3) if $\overrightarrow{\tau} \sqsubseteq \overrightarrow{\tau'}$, then $\mathsf{C}(\overrightarrow{\tau}\ \overrightarrow{\mathsf{f}})\{\mathsf{super}(\overrightarrow{\mathsf{f}}); \mathsf{this.}\overline{\mathsf{f}} = \overline{\mathsf{f}};\} \sqsubseteq$ $\mathsf{C}(\overrightarrow{\tau'}\ \overrightarrow{\mathsf{f}})\{\mathsf{super}(\overrightarrow{\mathsf{f}}); \mathsf{this.}\overline{\mathsf{f}} = \overline{\mathsf{f}};\}$;
(4) if $\tau \sqsubseteq \tau'$ and $\overrightarrow{\tau} \sqsubseteq \overrightarrow{\tau'}$, then $\tau\mathsf{m}(\overrightarrow{\tau}\ \overrightarrow{\mathsf{x}}) \sqsubseteq \tau'\mathsf{m}(\overrightarrow{\tau'}\ \overrightarrow{\mathsf{x}})$;
(5) if $\mathsf{H} \sqsubseteq \mathsf{H}'$ and $\mathsf{t} \sqsubseteq \mathsf{t}'$, then $\mathsf{H}\,\{\mathsf{return\ t;}\} \sqsubseteq \mathsf{H}'\,\{\mathsf{return\ t}';\}$.

Definition 6.3 (Class Table Precision). A class table CT is more or equally precise than a class table CT', notation $CT \sqsubseteq CT'$, if all the class and interface declarations of CT are in the relation $\sqsubseteq$ with all the class and interface declarations of CT'.

Note that if $CT \sqsubseteq CT'$, then CT and CT' define the same subtype relation and hence the same precision and convertibility relations between types.

The following lemma connects precision relations with the look up functions. The proof easily follows from the definitions.

Lemma 6.4. *Let $CT \sqsubseteq CT'$.*

(1) *If* $\mathsf{C}[\&\iota] \sqsubseteq \mathsf{D}[\&\iota']$ *and* $\sigma\mathsf{f} \in \mathtt{fields}(CT;\mathsf{C})$, *then* $\sigma'\mathsf{f} \in \mathtt{fields}(CT';\mathsf{D})$ *with* $\sigma \sqsubseteq \sigma'$.
(2) *If* $\tau \sqsubseteq \tau'$ *and* $\mathtt{mtype}(CT;\mathsf{m};\tau) = \overrightarrow{\sigma} \to \sigma$, *then* $\mathtt{mtype}(CT';\mathsf{m};\tau') = \overrightarrow{\sigma'} \to \sigma'$ *with* $\overrightarrow{\sigma} \sqsubseteq \overrightarrow{\sigma'}$ *and* $\sigma \sqsubseteq \sigma'$.
(3) *If* $\mathtt{fields}(CT;\mathsf{C}) = \overrightarrow{\tau}$, *then* $\mathtt{fields}(CT';\mathsf{C}) = \overrightarrow{\tau'}$ *with* $\overrightarrow{\tau} \sqsubseteq \overrightarrow{\tau'}$.
(4) *If* $\mathtt{mbody}(CT;\mathsf{m};\mathsf{C}) = (\overrightarrow{\mathsf{x}},\mathsf{t})$, *then* $\mathtt{mbody}(CT';\mathsf{m};\mathsf{C}) = (\overrightarrow{\mathsf{x}},\mathsf{t}')$ *with* $\mathsf{t} \sqsubseteq \mathsf{t}'$.
(5) *If* $\mathtt{mtype}(CT;\mathsf{m};\mathsf{C}) = \overrightarrow{\tau} \to \tau$, *then* $\mathtt{mtype}(CT';\mathsf{m};\mathsf{C}) = \overrightarrow{\tau'} \to \tau'$ *with* $\overrightarrow{\tau} \sqsubseteq \overrightarrow{\tau'}$ *and* $\tau \sqsubseteq \tau'$.

The gradual guarantee ensures that a less precise program with a less precise class table behaves the same as a more precise program with a more precise class table, except that the less precise program with a less precise class table might have less trapped errors.

Theorem 6.5 (Gradual Guarantee). *Let $CT \sqsubseteq CT'$ and $\mathsf{t} \sqsubseteq \mathsf{t}'$ and $\vdash_{CT} \mathsf{t} : \tau$, then*

(1) $\vdash_{CT'} \mathsf{t}' : \tau'$ *and* $\tau \sqsubseteq \tau'$;
(2) *if* $\mathsf{t} \longrightarrow_{CT} \mathsf{t}_0$, *then* $\mathsf{t}' \longrightarrow_{CT'} \mathsf{t}'_0$ *and* $\mathsf{t}_0 \sqsubseteq \mathsf{t}'_0$;
(3) *if* $\mathsf{t}' \longrightarrow_{CT'} \mathsf{t}'_0$, *then either* $\mathsf{t} \longrightarrow_{CT} \mathsf{t}_0$ *and* $\mathsf{t}_0 \sqsubseteq \mathsf{t}'_0$ *or* $\mathsf{t} \nrightarrow_{CT}$ Error.

Proof. We only consider the most interesting cases.

(1). The proof is by structural induction on t and t′ and by cases on the definition of $\sqsubseteq$.

If $\mathsf{t} = \mathsf{t}_0.\mathsf{f}$, then $\mathsf{t}' = \mathsf{t}'_0.\mathsf{f}$ and $\mathsf{t}_0 \sqsubseteq \mathsf{t}'_0$. If $\vdash_{CT} \mathsf{t} : \tau$ is the conclusion of Rule [FIELD], then $\vdash_{CT} \mathsf{t}_0 : \mathsf{C}[\&\iota]$ and $\tau\mathsf{f} \in \mathtt{fields}(CT;\mathsf{C})$. By structural induction $\vdash_{CT'} \mathsf{t}'_0 : \sigma$ and $\mathsf{C}[\&\iota] \sqsubseteq \sigma$. If $\sigma = \mathsf{D}[\&\iota']$ and $\tau'\mathsf{f} \in \mathtt{fields}(CT';\mathsf{D})$, then $\tau \sqsubseteq \tau'$ by Lemma 6.4(1). In this case, we derive $\vdash_{CT'} \mathsf{t}' : \tau'$ using Rule [FIELD]. Otherwise we derive $\vdash_{CT'} \mathsf{t}' : \star$ using Rule [⋆FIELD]. If $\vdash_{CT} \mathsf{t} : \tau$ is the conclusion of Rule [⋆FIELD], then $\tau = \star$ and we derive $\vdash_{CT'} \mathsf{t}' : \star$ with Rule [⋆FIELD].

If $\mathsf{t} = \mathsf{t}_0.\mathsf{m}(\overrightarrow{\mathsf{t}})$, then $\mathsf{t}' = \mathsf{t}'_0.\mathsf{m}(\overrightarrow{\mathsf{t}'})$ with $\mathsf{t}_0 \sqsubseteq \mathsf{t}'_0$ and $\overrightarrow{\mathsf{t}} \sqsubseteq \overrightarrow{\mathsf{t}'}$. If $\vdash_{CT} \mathsf{t} : \tau$ is the conclusion of Rule [INVK], then $\vdash_{CT} \mathsf{t}_0 : \sigma$ and $\mathtt{mtype}(CT;\mathsf{m};\sigma) = \overrightarrow{\tau} \to \tau$ and $\vdash^{\flat}_{CT} \overline{\mathsf{t}} : \overline{\tau}$. By structural induction we get $\vdash_{CT'} \mathsf{t}'_0 : \sigma'$ with $\sigma \sqsubseteq \sigma'$. If $\mathtt{mtype}(CT';\mathsf{m};\sigma') = \overrightarrow{\tau'} \to \tau'$, then $\overrightarrow{\tau} \sqsubseteq \overrightarrow{\tau'}$ and $\tau \sqsubseteq \tau'$ by Lemma 6.4(2). The judgments $\vdash^{\flat}_{CT} \overline{\mathsf{t}} : \overline{\tau}$ mean $\vdash_{CT} \mathsf{t}_i : \rho_i$ with $\rho_i \sqsubseteq \tau_i$ if t_i is not a λ-expression and $\vdash_{CT} (\mathsf{t}_i)^{\rho_i} : \rho_i$ if t_i is a λ-expression with $\rho_i = \tau_i$ a target type. By structural induction $\vdash_{CT'} \overline{\mathsf{t}} : \overline{\rho'}$ with $\overline{\rho} \sqsubseteq \overline{\rho'}$. If these judgments imply $\vdash^{\flat}_{CT} \overline{\mathsf{t}} : \overline{\tau'}$, then we derive $\vdash_{CT'} \mathsf{t}' : \tau'$ using Rule [INVK]. Otherwise we derive $\vdash_{CT'} \mathsf{t}' : \star$ using Rule [⋆INVK]. If $\vdash_{CT} \mathsf{t} : \tau$ is the conclusion of Rule [⋆INVK], then $\tau = \star$ and we derive $\vdash_{CT'} \mathsf{t}' : \star$ using Rule [⋆INVK].

(2). The proof is by cases on the reduction rules.

Rule [FieldNew]. In this case $\mathsf{t} = \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}}).\mathsf{f}_j$ and $\mathtt{fields}(CT;\mathsf{C}) = \overrightarrow{\tau}$ and $\mathsf{t}' = \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}'}).\mathsf{f}_j$ with $\overrightarrow{\mathsf{v}} \sqsubseteq \overrightarrow{\mathsf{v}'}$. By Lemma 6.4(3) $\mathtt{fields}(CT';\mathsf{C}) = \overrightarrow{\tau'}$ with $\overrightarrow{\tau} \sqsubseteq \overrightarrow{\tau'}$. We get $\mathsf{t}' \longrightarrow_{CT'} (\tau'_j)\mathsf{v}'_j$ by Rule [FieldNew].

Rule [MethNew]. In this case $\mathsf{t} = \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}}).\mathsf{m}(\overrightarrow{\mathsf{u}})$ and $\mathtt{mbody}(CT;\mathsf{m};\mathsf{C}) = (\overrightarrow{\mathsf{x}},\mathsf{t}_0)$ and $\mathtt{mtype}(CT;\mathsf{m};\mathsf{C}) = \overrightarrow{\tau} \to \tau$ and $\mathsf{t}' = \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}'}).\mathsf{m}(\overrightarrow{\mathsf{u}'})$ with $\overrightarrow{\mathsf{v}} \sqsubseteq \overrightarrow{\mathsf{v}'}$ and $\overrightarrow{\mathsf{u}} \sqsubseteq \overrightarrow{\mathsf{u}'}$. By Lemma 6.4(4) $\mathtt{mbody}(CT';\mathsf{m};\mathsf{C}) = (\overrightarrow{\mathsf{x}},\mathsf{t}'_0)$ with $\mathsf{t}_0 \sqsubseteq \mathsf{t}'_0$. By Lemma 6.4(5) $\mathtt{mtype}(CT';\mathsf{m};\mathsf{C}) = \overrightarrow{\tau'} \to \tau'$ with $\overrightarrow{\tau} \sqsubseteq \overrightarrow{\tau'}$ and $\tau \sqsubseteq \tau'$. We get $\mathsf{t}' \longrightarrow_{CT'} [\overrightarrow{\mathsf{x}} \mapsto (\overrightarrow{\tau'})\overrightarrow{\mathsf{u}'}, \mathsf{this} \mapsto \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}'})](\tau')\mathsf{t}'_0$ by Rule [MethNew].

(3). The proof is by induction on reduction and by cases on the reduction rules.

Rule [CastNew]. In this case $\mathsf{t}' = (\tau')\,\mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}'})$ and $\mathsf{C} \Rightarrow \tau'$ and $\mathsf{t} = (\tau)\,\mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}})$ with $\tau \sqsubseteq \tau'$ and $\overrightarrow{\mathsf{v}} \sqsubseteq \overrightarrow{\mathsf{v}'}$. If $\mathsf{C} \Rightarrow \tau$, then $\mathsf{t} \longrightarrow_{CT} \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}})$ using Rule [CastNew]. Otherwise $\mathsf{t} \nrightarrow_{CT}$ Error.

Rule [Ctx]. In this case $\mathsf{t}' = \mathcal{E}'[\mathsf{s}']$ and $\mathsf{s}' \longrightarrow_{CT'} \mathsf{s}'_0$ and $\mathsf{t}'_0 = \mathcal{E}'[\mathsf{s}'_0]$ and $\mathsf{t} = \mathcal{E}[\mathsf{s}]$ with $\mathcal{E}[\mathsf{s}] \sqsubseteq \mathcal{E}'[\mathsf{s}']$. By induction, either $\mathsf{s} \longrightarrow_{CT} \mathsf{s}_0$ with $\mathsf{s}_0 \sqsubseteq \mathsf{s}'_0$, or $\mathsf{s} \nrightarrow_{CT}$ Error. In the first case $\mathsf{t} \longrightarrow_{CT} \mathcal{E}[\mathsf{s}_0]$ by Rule [Ctx]. Note that $\mathcal{E}[\mathsf{s}] \sqsubseteq \mathcal{E}'[\mathsf{s}']$ and $\mathsf{s}_0 \sqsubseteq \mathsf{s}'_0$ imply $\mathcal{E}[\mathsf{s}_0] \sqsubseteq \mathcal{E}'[\mathsf{s}'_0]$. In the second case $\mathsf{t} \nrightarrow_{CT}$ Error by Rule [E-Ctx]. □

7 RELATED WORKS AND CONCLUSION

Java Core Calculi. The calculus FJ [6] is an elegant description of main Java features suitable for extension and variations. We build on [3] where FJ is enriched with interfaces, λ-expressions, first-class intersection types and flexible target types (i.e., they can contain more than one abstract method).

Gradual Typing for Objects. The object calculus of [1] is extended with a gradual typing system in [9]. The main contribution of this work is to show that gradual typing and subtyping are orthogonal and can be combined in a principled fashion. The language of [7] combines dynamic types and generics, extending [6]. There, dynamic types can be used as type arguments of a generic class, permitting a smooth interfacing between dynamically and statically typed code. The work [12] presents the design of a gradual typing system that accommodates class composition across components with different type disciplines. In [2] there is a formal framework for defining and comparing various gradually-typed object oriented languages.

In this paper, we add the dynamic type to the calculus of [3] in a light way, since we were not able to introduce easily an intermediate language with explicit cast to ⋆ for run-time checks. Therefore FJ&λ⋆ is not a gradually typed calculus in the traditional way. However, FJ&λ⋆ enjoys weak safety and satisfies the gradual guarantee. Notably, if we apply the Java restrictions in the use of intersection types and in the definition of target types, we get a core Java 8 calculus with the addition of a form of dynamic type.

As future work, we plan to study which conditions must be satisfied by dynamically annotated parts of code surrounding λ-expressions in order to have enough information (albeit partial) to be able to reduce the λ-expressions themselves. This would be a first step towards a standard gradual typing approach, with compilation in an intermediate calculus with dynamic casts. In another direction we will enrich FJ&λ⋆ with generics, as in [7]. Moreover, it would be interesting to analyse FJ&λ⋆ from the point of view of cast accumulation [5], in particular because FJ&λ⋆ seems not to suffer of this issue (in fact, a cast applied to a λ-expression simply goes away if the annotated target type is convertible to it).

REFERENCES

[1] Martin Abadi and Luca Cardelli. 1996. *A Theory of Objects*. Springer, Berlin. ISBN: 0387947752.

[2] Benjamin Chung, Paley Li, Francesco Zappa Nardelli, and Jan Vitek. 2018. KafKa: Gradual Typing for Objects. In *ECOOP (LIPIcs, Vol. 109)*, Todd Millstein (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, 12:1–12:24. https://doi.org/10.4230/LIPIcs.ECOOP.2018.12

[3] Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri. 2018. Intersection Types in Java: Back to the Future. In *Models, Mindsets, Meta: The What, the How, and the Why Not? (LNCS, Vol. 11200)*, Tiziana Margaria, Susanne Graf, and Kim G. Larsen (Eds.). Springer, Berlin, 68–86. https://doi.org/10.1007/978-3-030-22348-9_6

[4] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. 2015. *The Java Language Specification, Java SE 8 Edition*. Oracle, Austin, TX. ISBN-13: 9780133900699.

[5] David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *Higher Order Symbol. Comput.* 23, 2 (2010), 167–189. https://doi.org/10.1007/s10990-011-9066-z

[6] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (2001), 396–450. https://doi.org/10.1145/503502.503505

[7] Lintaro Ina and Atsushi Igarashi. 2011. Gradual Typing for Generics. In *OOPSLA*. ACM Press, New York, NY, 609–624. https://doi.org/10.1145/2048066.2048114

[8] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press, Cambridge, MA. ISBN: 0262162091.

[9] Jeremy Siek and Walid Taha. 2007. Gradual Typing for Objects. In *ECOOP*, Erik Ernst (Ed.). Springer, Berlin, 2–27. https://doi.org/10.1007/978-3-540-73589-2_2

[10] Jeremy G Siek and Walid Taha. 2006. Gradual typing for functional languages. *Scheme and Functional Programming Workshop*, Robby Findler (Ed.), http://scheme2006.cs.uchicago.edu/scheme2006.pdf, pages 81–92.

[11] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *SNAPL (LIPIcs, Vol. 32)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, 274–293. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

[12] Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Gradual Typing for First-Class Classes. In *OOPSLA*. ACM Press, New York, NY, 793–810. https://doi.org/10.1145/2384616.2384674

Received 2023-05-26; accepted 2023-06-23