Intersection Types in Java: back to the future

Mariangiola Dezani-Ciancaglini[1⋆], Paola Giannini[2⋆⋆], and Betti Venneri[3]

[1] Dipartimento di Informatica, Università di Torino, Italy
[2] Dipartimento di Scienze e Innovazione Tecnologica,
Università del Piemonte Orientale, Italy
[3] Dipartimento di Statistica, Informatica, Applicazioni,
Università di Firenze, Italy

Abstract. In this paper we figure out the future of intersection types in Java developments, based both on the primary meaning of the intersection type constructor and on the present approach in Java. In our vision, the current use of intersection types will be extended in two directions. Firstly, intersections will be allowed to appear as types of fields, types of formal parameters and return values of methods, therefore they will be significantly used as target types for λ-expressions anywhere. Secondly, the notion of functional interface will be extended to any intersection of interfaces, including also several abstract methods with different signatures. Thus a single target type will be able to express multiple, possibly unrelated, properties of one λ-expression.
We formalise our proposal trough a minimal Java core extended with these novel features and we prove the type safety property.

1 Introduction

Intersection types have been invented originally for the λ-calculus to increase the set of terms having meaning types [3]. The power of this type system lies on the fact that the set of untyped λ-terms having types is exactly the set of normalising terms, that is terminating programs. This prevents the adoption of this full system in programming languages, but the intuition behind intersection types can be particularly inspiring for language designers looking for mechanisms to improve flexibility of typechecking.

The development of the successive versions of Java shows a careful and even more significant entry of intersection types in the shape of intersection (denoted by &) of nominal types. In particular, we remark that the version Java 5 introduced generic types and intersection types later appeared as bounds of generic type variables. It is worth noting that already in the last century Büchi and Weck [2] had proposed to extend Java 1 by allowing intersection types (called

⋆ Partially supported by EU H2020-644235 Rephrase project, EU H2020-644298 HyVar project, IC1402 ARVI and Ateneo/CSP project RunVar.
⋆⋆ This original research has the financial support of the Università del Piemonte Orientale.

there compound types) as parameter types and return types of methods, by showing interesting examples of their use for structuring and reusing code. Java still does not allow these uses of intersections, that are confined to be target types of type-casts. However, one could assume that the proposal of [2] is now realised, in some fashion, in Java with generic types, given that a generic type variable, bounded by an intersection type, can appear as parameter type as well as return type of a method. Unfortunately, the above argument does not fit the case of λ-expressions, which are the key novelty of Java 8, since a generic type variable cannot be instantiated by the type of a λ-expression. Java λ-expressions are *poly expressions* in the sense they can have various types according to context requirements. Each context prescribes the *target type* for the λ-expression used in that context, and Java does not compile when this target type is unspecified. The target type can be either a *functional interface* (i.e., an interface with a single abstract method) or an intersection of interfaces that induces a functional interface. Notably, the λ-expression must match the signature of the unique abstract method of its functional interface. When we cast a λ-expression to an intersection type, this intersection becomes its target type, and so the λ-expression exploits its poly expression nature, having all the types of the intersection. So, in addition to implementing the abstract method, it also acquires all the behaviours that are represented by the default methods defined in the various interfaces forming the intersection. However, when the λ-expression is passed as argument to a method or returned by a method, then its target type cannot be an intersection type. Our proposal wants to free intersection types of all those bindings and restrictions, so that they can appear as types of fields, types of formal parameters and value results of methods, thus playing the role of target types for λ-expressions anywhere these expressions can be used.

The second limitation of Java we want to overcome relates to the definition of functional interface, which must contain one and *only one* abstract method. In Java a λ-expression is able to match multiple headers of abstract methods, with different signatures, but its target type in each context expresses just one of such signatures. This can be frustrating for Java programmers in many situations, where they are compelled to use several copies of the same λ-expression, each one matching a single signature. Completely different, the main idea behind intersection type theory is that an intersection type expresses multiple, possibly unrelated, properties of one term in a single type. The prototypical example is represented by the term $\lambda x.x\,x$, denoting the auto-application function, which can be assigned, for instance, the type $(\alpha\&(\alpha \to \beta)) \to \beta$, where α, β are arbitrary types and the arrow denotes the function type constructor. The intersection type $\alpha\&(\alpha \to \beta)$ says that the parameter must behave both as function and as argument of itself. We can retrieve this powerful feature from intersection type theory to Java, by allowing any intersection of interfaces to be a functional interface having multiple abstract methods. For example, let us consider the method

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

where `C` is any class (without fields for simplicity), `Arg` and `Fun` are two Java in-

terfaces with the abstract methods `C mArg (C y)` and `Arg mFun (Arg z)`, respectively. Although the method is greedy with requirements about its argument, many λ-expressions are ready to match the target type `Arg&Fun`, first of all the simple identity `x->x`.

In conclusion, this paper wants to flash forwards to a future development of Java, in which the use of intersection types is extended in the two directions discussed above, in order to study its formal properties. To this end, we formalise the calculus FJP&λ (Featherweight Java with Polymorphic Intersection types and λ-expressions), based on the core language FJ&λ [1], that models the treatment of λ-expressions and intersection types in Java 8. As main result, we prove that FJP&λ preserves type-safety.

2 Syntax

In defining the syntax of FJP&λ we follow the notational convention of [1], recalled here for self-containment. We use $\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}$ to denote classes, I, J to denote interfaces, T, U to denote nominal pre-types, i.e., either classes or interfaces; f, g to denote field names; m to denote method names; t to denote terms; x, y to denote variables, including the special variable this; τ, σ to denote pre-types as defined below. We use $\overrightarrow{\mathsf{I}}$ as a shorthand for the (comma separated) list $\mathsf{I}_1, \ldots, \mathsf{I}_n$ and $\overline{\mathsf{M}}$ as a shorthand for the sequence $\mathsf{M}_1 \ldots \mathsf{M}_n$ and similarly for the other names. Sometimes order in lists and sequences is unimportant. In rules, we write both $\overline{\mathsf{N}}$ as a declaration and $\overrightarrow{\mathsf{N}}$ for some name N: the meaning is that a sequence is declared and the list is obtained from the sequence adding commas. The notation $\overline{\tau}\,\overline{\mathsf{f}};$ abbreviates $\tau_1\mathsf{f}_1; \ldots \tau_n\mathsf{f}_n;$ and $\overrightarrow{\tau}\,\overrightarrow{\mathsf{f}}$ abbreviates $\tau_1\mathsf{f}_1, \ldots, \tau_n\mathsf{f}_n$ (likewise $\overrightarrow{\tau}\,\overrightarrow{\mathsf{x}}$) and $\mathsf{this}.\overline{\mathsf{f}} = \overline{\mathsf{f}};$ abbreviates $\mathsf{this}.\mathsf{f}_1 = \mathsf{f}_1; \ldots \mathsf{this}.\mathsf{f}_n = \mathsf{f}_n;$. Lists and 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. Figure 1 gives declarations: CD ranges over class declarations; ID ranges over interface declarations; K ranges over constructor declarations; H ranges over method header (or abstract method) declarations; M ranges over method (or concrete method) declarations.

$\mathsf{CD} ::= \mathsf{class\ C\ extends\ D\ implements}\ \overrightarrow{\mathsf{I}}\ \{\overline{\tau}\,\overline{\mathsf{f}};\ \mathsf{K}\ \overline{\mathsf{M}}\}$	class declarations
$\mathsf{ID} ::= \mathsf{interface\ I\ extends}\ \overrightarrow{\mathsf{I}}\ \{\overline{\mathsf{H}};\ \overline{\mathsf{M}}\}$	interface declarations
$\mathsf{K} ::= \mathsf{C}(\overrightarrow{\tau}\,\overrightarrow{\mathsf{f}})\{\mathsf{super}(\overrightarrow{\mathsf{f}});\ \mathsf{this}.\overline{\mathsf{f}} = \overline{\mathsf{f}};\}$	constructor declarations
$\mathsf{H} ::= \tau\mathsf{m}(\overrightarrow{\tau}\,\overrightarrow{\mathsf{x}})$	header declarations
$\mathsf{M} ::= \mathsf{H}\,\{\mathsf{return\ t};\}$	method declarations

Fig. 1: Declarations

$$\frac{CT(\mathsf{I}) = \mathsf{interface\ I\ extends}\ \overrightarrow{\mathsf{I}}\ \{\overline{\mathsf{H}}; \overline{\mathsf{M}}\}}{\texttt{A-mh}(\mathsf{I}) = \overrightarrow{\mathsf{H}} \uplus \texttt{A-mh}(\overrightarrow{\mathsf{I}})}$$

$$\frac{CT(\mathsf{I}) = \mathsf{interface\ I\ extends}\ \overrightarrow{\mathsf{I}}\ \{\overline{\mathsf{H}}; \overline{\mathsf{M}}\} \quad \overline{\mathsf{M}} = \overline{\mathsf{H}'\ \{\mathsf{return\ t};\}}}{\texttt{D-mh}(\mathsf{I}) = \overrightarrow{\mathsf{H}'} \uplus \texttt{D-mh}(\overrightarrow{\mathsf{I}})}$$

$$\texttt{A-mh}(\mathsf{I}_1, \ldots, \mathsf{I}_n) = \texttt{A-mh}(\mathsf{I}_1 \& \ldots \& \mathsf{I}_n) = \texttt{A-mh}(\mathsf{C} \& \mathsf{I}_1 \& \ldots \& \mathsf{I}_n) = \biguplus_{1 \le i \le n} \texttt{A-mh}(\mathsf{I}_i)$$

$$\texttt{D-mh}(\mathsf{I}_1, \ldots, \mathsf{I}_n) = \texttt{D-mh}(\mathsf{I}_1 \& \ldots \& \mathsf{I}_n) = \texttt{D-mh}(\mathsf{C} \& \mathsf{I}_1 \& \ldots \& \mathsf{I}_n) = \biguplus_{1 \le i \le n} \texttt{D-mh}(\mathsf{I}_i)$$
$$\text{if } \texttt{D-mh}(\mathsf{I}_j) \cap \texttt{D-mh}(\mathsf{I}_\ell) \neq \epsilon \text{ implies either } \mathsf{I}_j <: \mathsf{I}_\ell \text{ or } \mathsf{I}_\ell <: \mathsf{I}_j$$

$$\texttt{mh}(\texttt{Object}) = \epsilon$$

$$\frac{CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D\ implements}\ \overrightarrow{\mathsf{I}}\ \{\overline{\tau}\,\overline{\mathsf{f}}; \mathsf{K}\,\overline{\mathsf{M}}\} \quad \overline{\mathsf{M}} = \overline{\mathsf{H}\ \{\mathsf{return\ t};\}}}{\texttt{mh}(\mathsf{C}) = \overrightarrow{\mathsf{H}} \uplus \texttt{mh}(\mathsf{D}) \uplus \texttt{mh}(\overrightarrow{\mathsf{I}})}$$

$$\texttt{mh}(\overrightarrow{\mathsf{I}}) = \texttt{A-mh}(\overrightarrow{\mathsf{I}}) \uplus \texttt{D-mh}(\overrightarrow{\mathsf{I}}) \qquad \text{if } \texttt{A-mh}(\overrightarrow{\mathsf{I}}) \cap \texttt{D-mh}(\overrightarrow{\mathsf{I}}) = \epsilon$$

$$\texttt{mh}(\mathsf{I}_1 \& \ldots \& \mathsf{I}_n) = \texttt{mh}(\mathsf{I}_1, \ldots, \mathsf{I}_n) \qquad \texttt{mh}(\mathsf{C} \& \mathsf{I}_1 \& \ldots \& \mathsf{I}_n) = \texttt{mh}(\mathsf{C}) \uplus \texttt{mh}(\mathsf{I}_1, \ldots, \mathsf{I}_n)$$

Fig. 2: Functions `A-mh`, `D-mh` and `mh`

A main novelty of FJP&λ with respect to Java is that the types of fields, parameters and return terms are intersections instead of nominal types. The syntax of FJP&λ is obtained from the syntax of FJ&λ with default methods of [1] by replacing everywhere nominal pre-types by arbitrary pre-types. As usual a class declaration specifies the name of the class, its superclass and the implemented interfaces. A class has fields $\overline{\mathsf{f}}$, a single constructor K and methods $\overline{\mathsf{M}}$. The instance variables $\overline{\mathsf{f}}$ are added to the ones of its superclasses and should have names disjoint from these. An interface declaration lists the extended interfaces, the method headers, and the default methods (omitting the keyword default). The arguments of the constructors correspond to the immutable values of the class fields. The inherited fields are initialised by the call to super, while the new fields are initialised by assignments. Headers relate method names with result and parameter pre-types. Methods are headers with bodies, i.e., return expressions. We omit implements and extends when the lists of interfaces are empty.

`Object` is a special class without fields and methods, does not require a declaration and it is the top of the class hierarchy.

A *class table* CT is a mapping from nominal types to their declarations. A *program* is a pair $(CT, \mathtt{t})$. In the following we assume a fixed class table.

Pre-types (ranged over by τ, σ) are either nominal types or intersections of:

– interfaces or

– a class (in leftmost position) and any number of interfaces.

Using ι to denote either an interface or an intersection of interfaces we define:

$$\tau ::= \mathsf{C} \mid \iota \mid \mathsf{C}\&\iota \quad \text{where} \quad \iota ::= \mathsf{I} \mid \iota\&\mathsf{I}$$

The notation $\mathsf{C}[\&\iota]$ means either the class C or the pre-type $\mathsf{C}\&\iota$.

Types are pre-types whose method declarations are consistent. To define consistency we use the partial functions `A-mh`, `D-mh` and `mh` that map pre-types to lists of method headers, considered as sets, see Figure 2. As in [1], `A-mh` and `D-mh` collect abstract and default methods in interfaces, whereas the function `mh` collects all method headers of interfaces and classes. We use ϵ for the empty list. With $\biguplus$ we denote the set-theoretic union of lists of method headers, that is defined only if the same method name does not occur with different pre-types.

For example `C mArg(C x)` $\biguplus$ `Arg mFun(Arg x)` is defined, while

`C mArg(C x)` $\biguplus$ `Arg mArg(Arg x)`

is not defined.

As we can see from the penultimate line of Figure 2, function `mh` is defined for a list of interfaces only if the same method name is not declared both as abstract and default method, see page 292 of [6].

Definition 1 (Types). *A pre-type τ is a type if* $\mathtt{mh}(\tau)$ *is defined.*

For example, if we consider

`interface Arg {C mArg(C x);}` and `interface Fun {Arg mFun(Arg x);}`

then `Arg&Fun` is a type; whereas if we define

`interface ArgD {D mArg(D x);}`

where the class `D` differs from the class `C`, then the pre-type `Arg&ArgD` is not a type, since `mh(Arg)` $\biguplus$ `mh(ArgD)` is not defined.

Notice that the present definition of type coincides with that in [1], and therefore all types here are Java types.

In the following we will always restrict $\mathsf{T}, \mathsf{U}, \tau, \sigma, \iota$ to range over types. The typing rules for classes and interfaces (see Figure 11) assure that all nominal pre-types in a well-formed class table are types.

Terms are defined in Figure 3. Terms are a subset of Java terms, with the addition that *λ-expressions* may or may not be *decorated by intersections* of

$\mathsf{t} ::=$		terms	$\mathsf{v} ::=$		values
	v	value		w	proper value
	x	variable		$\overrightarrow{\mathsf{p}} \to \mathsf{t}$	pure λ-expression
	$\mathsf{t.f}$	field access	$\mathsf{w} ::=$		proper values
	$\mathsf{t.m}(\overrightarrow{\mathsf{t}})$	method invocation		$\mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}})$	object
	$\mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{t}})$	object		$(\overrightarrow{\mathsf{p}} \to \mathsf{t})^{\iota}$	decorated λ-expression
	$(\tau)\,\mathsf{t}$	cast	$\mathsf{p} ::=$		parameters
				x	untyped
				$\tau\ \mathsf{x}$	typed

Fig. 3: Terms

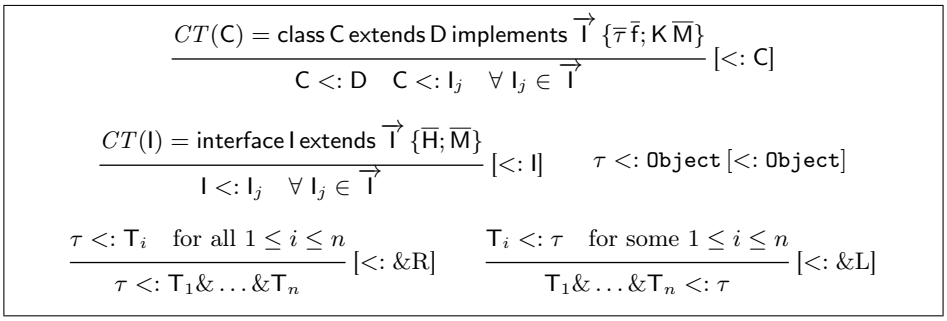

Fig. 4: Subtyping

interfaces. The intersection of interfaces decorating a λ-expression represents its *target type*. *Values*, ranged over by v, u, are either proper values or pure λ-expressions. *Proper values*, ranged over by w, are either objects or decorated λ-expressions. Pure λ-expressions are written by the user, whereas decorated λ-expressions are generated by reduction. A parameter p of a λ-expression can be either untyped or typed, but typing rules forbid to mix untyped and typed parameters in the same λ-expression. A difference with [1] is the freedom of decorating λ-expression with interfaces and intersections of interfaces without requiring exactly one abstract method as in Java (see [6], page 321).

We use t_λ to range over pure λ-expressions.

The *subtype relation* $<:$ is the reflexive and transitive closure of the relation induced by the rules of Figure 4. It takes into account both the hierarchy between nominal types induced by the class table and the set theoretic properties of intersection. Rule $[<:\&\mathrm{R}]$ formalises the statement in the last two lines of page 677 in [6].

Our definition of intersection types is consistent with the requirements of the Java Language Specification [6] (pages 70-71). In particular, on one side from Definition 1, τ is a type if "$\mathtt{mh}(\tau)$ defined", and therefore we can define a nominal class that is a subtype of τ. On the other, the existence of a nominal class which is a subtype of τ assures $\mathtt{mh}(\tau)$ defined since rule $[\mathsf{C}\ \mathit{OK}]$ in Figure 11 requires $\mathtt{mh}(\mathsf{C})$ defined and $\mathtt{mh}(\tau) \subseteq \mathtt{mh}(\mathsf{C})$, see the proof of Lemma 1(2).

Notice that $\iota <: \mathtt{Object}\&\iota <: \iota$ for all ι, but these types cannot be considered equivalent, since ι can be the target type of a λ-expression whereas $\mathtt{Object}\&\iota$ cannot.

3 Operational Semantics

For the evaluation and typing rules we need the auxiliary definitions of Figures 5 and 6. The fields of a class C, dubbed $\mathtt{fields}(\mathsf{C})$, are specified by a list of pairs associating names and types of the fields that are defined in C or in one of its superclasses. To give the signature of methods, i.e. the parameters and result types, we specify method name and type. Moreover, it is useful to distinguish between

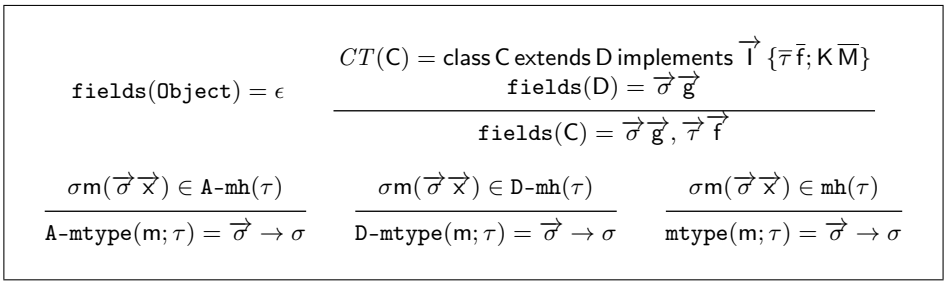

Fig. 5: Lookup Fields and Method Types

abstract and default method in interfaces. Therefore, we have three lookup functions: $\texttt{A-mtype}(\mathsf{m};\tau)$, $\texttt{D-mtype}(\mathsf{m};\tau)$ and $\texttt{mtype}(\mathsf{m};\tau)$. Their definition uses the functions given in Figure 2.

The body of a method m for a type τ is specified by $\texttt{mbody}(\mathsf{m};\tau)$ of Figure 6. If τ is a class C, we first look for a definition of m in C, then in its superclass, and, if not found, we look for a default method with name m in the interfaces $\overrightarrow{\mathsf{I}}$ implemented by C. The fact that the function $\texttt{D-mh}(\tau)$ of Figure 2 (used by $\texttt{D-mtype}(\mathsf{m};\ \overrightarrow{\mathsf{I}})$ of Figure 5) is defined ensures that, if there is more than one definition of m, then there is a most specific one which is the one returned. This is enforced by the rules for $\texttt{mbody}(\mathsf{m};\ \overrightarrow{\mathsf{I}})$ and $\texttt{mbody}(\mathsf{m};\mathsf{I}_1\&\ldots\&\mathsf{I}_n)$.

In typing the source code, Java uses for λ-expressions the types prescribed by the contexts enclosing them. These types are called *target types*. This means that λ-expressions are *poly expressions*, i.e. they can have different types in different contexts, see page 93 of [6]. More precisely:

(1) the target type of a λ-expression that occurs as a actual parameter of a constructor call is the type of the field in the class declaration;
(2) the target type of a λ-expression that occurs as a actual parameter of a method call is the type of the parameter in the method declaration;
(3) the target type of a λ-expression that occurs as a return term of a method is the result type in the method declaration;
(4) the target type of a λ-expression that occurs as the body of another λ-expression is the result type of the target type of the external λ-expression;
(5) the target type of a λ-expression that occurs as argument of a cast is the cast type.

According to [6] (page 602): "It is a compile-time error if a lambda expression occurs in a program in someplace other than an assignment context, an invocation context (like (1), (2), (3) and (4) above), or a casting context (like (5) above)."

FJP&λ extends Java allowing any intersection of interfaces as target type, while Java requires exactly one abstract method.

Following [1] the reduction rules assure that the pure λ-expressions are decorated by their target types in the evaluated terms. The mapping $(\mathsf{t})^{?\tau}$ defined as follows:

$$\frac{CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D\ implements}\ \overrightarrow{\mathsf{I}}\ \{\overline{\tau}\,\overline{\mathsf{f}};\mathsf{K}\,\overline{\mathsf{M}}\} \quad \sigma\mathsf{m}(\overrightarrow{\sigma}\,\overrightarrow{\mathsf{x}})\{\mathsf{return\ t};\} \in \overrightarrow{\mathsf{M}}}{\mathtt{mbody}(\mathsf{m};\mathsf{C}) = (\overrightarrow{\mathsf{x}},\mathsf{t})}$$

$$\frac{CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D\ implements}\ \overrightarrow{\mathsf{I}}\ \{\overline{\tau}\,\overline{\mathsf{f}};\mathsf{K}\,\overline{\mathsf{M}}\} \quad \mathsf{m}\text{ is not defined in }\overrightarrow{\mathsf{M}} \quad \mathtt{mbody}(\mathsf{m};\mathsf{D})\text{ is defined}}{\mathtt{mbody}(\mathsf{m};\mathsf{C}) = \mathtt{mbody}(\mathsf{m};\mathsf{D})}$$

$$\frac{CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D\ implements}\ \overrightarrow{\mathsf{I}}\ \{\overline{\tau}\,\overline{\mathsf{f}};\mathsf{K}\,\overline{\mathsf{M}}\} \quad \mathsf{m}\text{ is not defined in }\overrightarrow{\mathsf{M}} \quad \mathtt{mbody}(\mathsf{m};\mathsf{D})\text{ is not defined}}{\mathtt{mbody}(\mathsf{m};\mathsf{C}) = \mathtt{mbody}(\mathsf{m};\ \overrightarrow{\mathsf{I}})}$$

$$\frac{CT(\mathsf{I}) = \mathsf{interface\ I\ extends}\ \overrightarrow{\mathsf{I}}\ \{\overline{\mathsf{H}};\overline{\mathsf{M}}\} \quad \tau\mathsf{m}(\overrightarrow{\tau}\,\overrightarrow{\mathsf{x}})\{\mathsf{return\ t};\} \in \overrightarrow{\mathsf{M}}}{\mathtt{mbody}(\mathsf{m};\mathsf{I}) = (\overrightarrow{\mathsf{x}},\mathsf{t})}$$

$$\frac{CT(\mathsf{I}) = \mathsf{interface\ I\ extends}\ \overrightarrow{\mathsf{I}}\ \{\overline{\mathsf{H}};\overline{\mathsf{M}}\} \quad \mathsf{m}\text{ is not defined in }\overrightarrow{\mathsf{M}}}{\mathtt{mbody}(\mathsf{m};\mathsf{I}) = \mathtt{mbody}(\mathsf{m};\ \overrightarrow{\mathsf{I}})}$$

$$\mathtt{mbody}(\mathsf{m};\mathsf{I}_1,\ldots,\mathsf{I}_n) = \mathtt{mbody}(\mathsf{m};\mathsf{I}_1\&\ldots\&\mathsf{I}_n) = \mathtt{mbody}(\mathsf{m};\mathsf{I}_j)$$
$$\text{if } \mathtt{mbody}(\mathsf{m};\mathsf{I}_\ell) \text{ defined implies } \mathsf{I}_j <: \mathsf{I}_\ell$$

$$\mathtt{mbody}(\mathsf{m};\mathsf{C}\&\mathsf{I}_1\&\ldots\&\mathsf{I}_n) = \begin{cases} \mathtt{mbody}(\mathsf{m};\mathsf{C}) & \text{if defined} \\ \mathtt{mbody}(\mathsf{m};\mathsf{I}_1,\ldots,\mathsf{I}_n) & \text{otherwise} \end{cases}$$

Fig. 6: Method Body Lookup

$$(\mathsf{t})^{?\tau} = \begin{cases} (\mathsf{t})^{\tau} & \text{if t is a pure } \lambda\text{-expression,} \\ \mathsf{t} & \text{otherwise} \end{cases}$$

decorates with τ pure λ-expressions, whereas leaves all the other terms unchanged. This mapping is used in propagating the type expected for λ-expressions in constructor and method calls and in casts. The typing rules assure that if t is a pure λ-expression, then τ is an interface or an intersection of interfaces, i.e. reducing well-typed terms we only get decorated terms of the shape $(\mathsf{t}_\lambda)^\iota$.

As usual $[\mathsf{x} \mapsto \mathsf{t}]$ denotes the substitution of x by t and it generalises to an arbitrary number of variables/terms as expected.
The notation $\overrightarrow{\mathsf{x}} \mapsto (\overrightarrow{\mathsf{v}})^{?\overrightarrow{\tau}}$ is short for $\mathsf{x}_1 \mapsto (\mathsf{v}_1)^{?\tau_1}, \ldots, \mathsf{x}_n \mapsto (\mathsf{v}_n)^{?\tau_n}$.

The reduction rules are given in Figures 7 and 8. The rules for method calls decorate actual parameters and bodies that are λ-expressions with the expected types. This is also the case for the rule of field access and the rule for cast of λ-expressions. It is easy to verify that all pure λ-expressions being actual parameters or resulting terms in the l.h.s. are decorated by their target types in the r.h.s. We only comment the rules regarding method calls when

$$\frac{\tau\ \mathsf{f}_j \in \mathtt{fields}(\mathsf{C})}{\mathsf{new}\ \mathsf{C}(\overrightarrow{\mathsf{v}}).\mathsf{f}_j \longrightarrow (\mathsf{v}_j)^{?\tau}}\ [\text{E-ProjNew}] \qquad \frac{\mathsf{C} <: \tau}{(\tau)\,\mathsf{new}\ \mathsf{C}(\overrightarrow{\mathsf{v}}) \longrightarrow \mathsf{new}\ \mathsf{C}(\overrightarrow{\mathsf{v}})}\ [\text{E-CastNew}]$$

$$\frac{\mathtt{mbody}(\mathsf{m};\mathsf{C}) = (\overrightarrow{\mathsf{x}},\mathsf{t}) \quad \mathtt{mtype}(\mathsf{m};\mathsf{C}) = \overrightarrow{\tau} \to \tau}{\mathsf{new}\ \mathsf{C}(\overrightarrow{\mathsf{v}}).\mathsf{m}(\overrightarrow{\mathsf{u}}) \longrightarrow [\overrightarrow{\mathsf{x}} \mapsto (\overrightarrow{\mathsf{u}})^{?\overrightarrow{\tau}}, \mathsf{this} \mapsto \mathsf{new}\ \mathsf{C}(\overrightarrow{\mathsf{v}})](\mathsf{t})^{?\tau}}\ [\text{E-InvkNew}]$$

$$\frac{\mathtt{A\text{-}mtype}(\mathsf{m};\iota) = \overrightarrow{\tau} \to \tau}{(\overrightarrow{\mathsf{y}} \to \mathsf{t})^{\iota}.\mathsf{m}(\overrightarrow{\mathsf{v}}) \longrightarrow [\overrightarrow{\mathsf{y}} \mapsto (\overrightarrow{\mathsf{v}})^{?\overrightarrow{\tau}}](\mathsf{t})^{?\tau}}\ [\text{E-Invk}\lambda\text{U-A}]$$

$$\frac{\mathtt{A\text{-}mtype}(\mathsf{m};\iota) = \overrightarrow{\tau} \to \tau}{(\overrightarrow{\tau}\,\overrightarrow{\mathsf{y}} \to \mathsf{t})^{\iota}.\mathsf{m}(\overrightarrow{\mathsf{v}}) \longrightarrow [\overrightarrow{\mathsf{y}} \mapsto (\overrightarrow{\mathsf{v}})^{?\overrightarrow{\tau}}](\mathsf{t})^{?\tau}}\ [\text{E-Invk}\lambda\text{T-A}]$$

$$\frac{\mathtt{mbody}(\mathsf{m};\iota) = (\overrightarrow{\mathsf{x}},\mathsf{t}) \quad \mathtt{D\text{-}mtype}(\mathsf{m};\iota) = \overrightarrow{\tau} \to \tau}{(\mathsf{t}_\lambda)^{\iota}.\mathsf{m}(\overrightarrow{\mathsf{v}}) \longrightarrow [\overrightarrow{\mathsf{x}} \mapsto (\overrightarrow{\mathsf{v}})^{?\overrightarrow{\tau}}, \mathsf{this} \mapsto (\mathsf{t}_\lambda)^{\iota}](\mathsf{t})^{?\tau}}\ [\text{E-Invk}\lambda\text{-D}]$$

$$(\iota)\,\mathsf{t}_\lambda \longrightarrow (\mathsf{t}_\lambda)^{\iota}\ [\text{E-Cast}\lambda] \qquad \frac{\iota <: \iota'}{(\iota')\,(\mathsf{t}_\lambda)^{\iota} \longrightarrow (\mathsf{t}_\lambda)^{\iota'}}\ [\text{E-Cast}\lambda\text{Target}]$$

Fig. 7: Computational Rules

the receivers are λ-expressions, since the others are obvious. A decorated λ-expression implements all the abstract methods declared in the interfaces of its target type, so in rules [E-InvkλU-A] and [E-InvkλT-A] 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. In this way we follow Java 8 specification [6] (page 480), but for the decoration of the λ-expression.

The reduction rules in Figure 8 specify the (standard) execution strategy.

$$\frac{\mathsf{t} \longrightarrow \mathsf{t}'}{\mathsf{t.f} \longrightarrow \mathsf{t}'.\mathsf{f}}\ [\text{E-Field}] \qquad \frac{\mathsf{t} \longrightarrow \mathsf{t}'}{(\tau)\,\mathsf{t} \longrightarrow (\tau)\,\mathsf{t}'}\ [\text{E-Cast}]$$

$$\frac{\mathsf{t} \longrightarrow \mathsf{t}'}{\mathsf{t.m}(\overrightarrow{\mathsf{t}}) \longrightarrow \mathsf{t}'.\mathsf{m}(\overrightarrow{\mathsf{t}})}\ [\text{E-Invk-Recv}]$$

$$\frac{\mathsf{t} \longrightarrow \mathsf{t}'}{\mathsf{w.m}(\overrightarrow{\mathsf{v}},\mathsf{t},\overrightarrow{\mathsf{t}}) \longrightarrow \mathsf{w.m}(\overrightarrow{\mathsf{v}},\mathsf{t}',\overrightarrow{\mathsf{t}})}\ [\text{E-Invk-Arg}]$$

$$\frac{\mathsf{t} \longrightarrow \mathsf{t}'}{\mathsf{new}\ \mathsf{C}(\overrightarrow{\mathsf{v}},\mathsf{t},\overrightarrow{\mathsf{t}}) \longrightarrow \mathsf{new}\ \mathsf{C}(\overrightarrow{\mathsf{v}},\mathsf{t}',\overrightarrow{\mathsf{t}})}\ [\text{E-New-Arg}]$$

Fig. 8: Congruence Rules

For example, assuming that the method `auto` of the Introduction is defined in class `AutoApp` we get

$$\begin{aligned}\texttt{new AutoApp().auto(x->x)} &\longrightarrow \texttt{(x->x)}^{\texttt{Arg\&Fun}}\texttt{.mFun((y->y)}^{\texttt{Arg\&Fun}}\texttt{).mArg(new C())}\\ &\longrightarrow \texttt{(y->y)}^{\texttt{Arg\&Fun}}\texttt{.mArg(new C())}\\ &\longrightarrow \texttt{new C()}\end{aligned}$$

where in duplicating the parameter of `auto` we used two renamings of the identity $\texttt{(x->x)}^{\texttt{Arg\&Fun}}$ and $\texttt{(y->y)}^{\texttt{Arg\&Fun}}$ to clarify that they have different roles.

4 Typing rules

FJP&λ generalises FJ&λ allowing to use intersections everywhere and avoiding the restriction that target types must have a single abstract method. We start discussing the rules for terms shown in Figure 9. The typing judgment is $\Gamma \vdash \mathtt{t} : \tau$, where an environment Γ is a finite mapping from variables to types.

Field access is well typed, rule [T-FIELD], if the type is an intersection with at least one class (our interfaces cannot have fields) and the class contains the required field. In rules [T-INVK] and [T-NEW] the actual parameters are typed with the type judgement $\vdash^*$ which behaves differently depending on the fact that the term is a pure λ-expression or any other term. The judgment $\vdash^*$ is defined as follows

$$\frac{\Gamma \vdash \mathtt{t} : \sigma \quad \sigma <: \tau}{\Gamma \vdash^* \mathtt{t} : \tau} \qquad \frac{\Gamma \vdash (\mathtt{t}_\lambda)^\iota : \iota}{\Gamma \vdash^* \mathtt{t}_\lambda : \iota}$$

and taking advantage of the notation $(\,)^?$, can be synthesised by:

$$\frac{\Gamma \vdash (\mathtt{t})^{?\tau} : \sigma \quad \sigma <: \tau}{\Gamma \vdash^* \mathtt{t} : \tau}\,[\vdash \vdash^*]$$

As usual $\Gamma \vdash^* \overrightarrow{\mathtt{t}} : \overrightarrow{\tau}$ is short for $\Gamma \vdash^* \mathtt{t}_1 : \tau_1, \ldots, \Gamma \vdash^* \mathtt{t}_n : \tau_n$.
According to the judgment $\vdash^*$, actual parameters that are not pure λ-expressions can have any type which is a subtype of the type of the matching parameter, whereas pure λ-expression can only have the type required by the context, which is the type of the matching parameter. The premise of the rule $\vdash^*$ for pure λ-expressions requires that we derive for λ-expressions decorated with their target type exactly their target type. The rules for typing decorated λ-expressions are [T-λU], if the parameters are untyped, and [T-λT], if they are typed. Note that, in the type system $\vdash$ there is no typing rule for pure λ-expressions, since we expect each λ-expression to be decorated with its target type. Rule [T-λU] 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 $\vdash^*$ to use the correct typing judgement for pure λ-expressions and other terms. In addition to the requirements of [T-λU], when the types of the parameters are specified, rule [T-λT] prescribes that they coincide with the types of the parameters of all the abstract methods declared in the interfaces occurring in the target type of the λ-expression. We observe that rules [T-λU] and [T-λT] can give type to any λ-expression when $\texttt{A-mh}(\iota)$ is the empty list. This is consistent with our formal setting, where we use any λ-expression just for calling default methods in the absence of abstract methods.

$$\frac{\mathsf{x} : \tau \in \Gamma}{\Gamma \vdash \mathsf{x} : \tau}\ [\text{T-VAR}] \qquad \frac{\Gamma \vdash \mathsf{t} : \mathsf{C}[\&\iota] \quad \tau\,\mathsf{f} \in \mathtt{fields}(\mathsf{C})}{\Gamma \vdash \mathsf{t.f} : \tau}\ [\text{T-FIELD}]$$

$$\frac{\Gamma \vdash \mathsf{t} : \tau \quad \mathtt{mtype}(\mathsf{m}; \tau) = \overrightarrow{\sigma} \to \sigma \quad \Gamma \vdash^* \overrightarrow{\mathsf{t}} : \overrightarrow{\sigma}}{\Gamma \vdash \mathsf{t.m}(\overrightarrow{\mathsf{t}}) : \sigma}\ [\text{T-INVK}]$$

$$\frac{\mathtt{fields}(\mathsf{C}) = \overrightarrow{\tau}\,\overrightarrow{\mathsf{f}} \quad \Gamma \vdash^* \overrightarrow{\mathsf{t}} : \overrightarrow{\tau}}{\Gamma \vdash \mathsf{new\ C}(\overrightarrow{\mathsf{t}}) : \mathsf{C}}\ [\text{T-NEW}]$$

$$\frac{\tau\mathsf{m}(\overrightarrow{\tau}\,\overrightarrow{\mathsf{x}}) \in \mathtt{A\text{-}mh}(\iota) \text{ implies } \Gamma, \overrightarrow{\mathsf{y}} : \overrightarrow{\tau} \vdash^* \mathsf{t} : \tau}{\Gamma \vdash (\overrightarrow{\mathsf{y}} \to \mathsf{t})^{\iota} : \iota}\ [\text{T-}\lambda\text{U}]$$

$$\frac{\Gamma \vdash (\overrightarrow{\mathsf{y}} \to \mathsf{t})^{\iota} : \iota \quad \tau\mathsf{m}(\overrightarrow{\sigma}\,\overrightarrow{\mathsf{x}}) \in \mathtt{A\text{-}mh}(\iota) \text{ implies } \overrightarrow{\sigma} = \overrightarrow{\tau}}{\Gamma \vdash (\overrightarrow{\tau}\,\overrightarrow{\mathsf{y}} \to \mathsf{t})^{\iota} : \iota}\ [\text{T-}\lambda\text{T}]$$

Fig. 9: Syntax Directed Typing Rules

$$\frac{\Gamma \vdash^* \mathsf{t} : \tau}{\Gamma \vdash (\tau)\,\mathsf{t} : \tau}\ [\text{T-UCAST}]$$

$$\frac{\Gamma \vdash \mathsf{t} : \tau \quad \tau \sim \mathsf{C}[\&\iota] \quad \sigma \sim \mathsf{D}[\&\iota'] \quad \tau \not<: \sigma \quad \text{either } \mathsf{C} <: \mathsf{D} \text{ or } \mathsf{D} <: \mathsf{C}}{\Gamma \vdash (\sigma)\,t : \sigma}\ [\text{T-UDCAST}]$$

Fig. 10: Cast Typing Rules

The rules for type casts in Figure 10 are as in [1]. We use $\tau' \sim \sigma'$ as short for $\tau' <: \sigma'$ and $\sigma' <: \tau'$. The condition $\tau \not<: \sigma$ forbids to apply rule [T-UDCAST] when rule [T-UCAST] can be used instead.

A type derivation for the term reduced at the end of Section 3 is:

$$\frac{\vdash \texttt{new AutoApp()} : \texttt{AutoApp} \quad \mathtt{mtype}(\texttt{AutoApp}; \texttt{auto}) = \texttt{Arg\&Fun} \to \texttt{C} \quad \Delta}{\vdash \texttt{new AutoApp().auto(x->x)} : \texttt{C}}$$

where Δ is the derivation:

$$\frac{\texttt{A-mh(Arg\&Fun)}=\{\texttt{C mArg(C x), Arg mFun(Arg x)}\} \quad \texttt{y:C} \vdash \texttt{y} : \texttt{C} \quad \texttt{y:Arg} \vdash \texttt{y} : \texttt{Arg}}{\frac{\vdash \texttt{(y->y)}^{\texttt{Arg\&Fun}} : \texttt{Arg\&Fun}}{\vdash^* \texttt{(y->y)} : \texttt{Arg\&Fun}}}$$

Finally, we define the rules for checking that method, class and interface declarations are well formed. Note the use of the judgement $\vdash^*$ for typing the bodies of the methods. For methods the key difference with respect to the corresponding rule of FJ&λ is the presence of intersection types in place of nominal types.

$$\frac{\overrightarrow{\mathsf{x}} : \overrightarrow{\tau}, \mathsf{this} : \mathsf{T} \vdash^* \mathsf{t} : \tau \quad \tau\mathsf{m}(\overrightarrow{\tau}\,\overrightarrow{\mathsf{x}}) \in \mathtt{mh}(\mathsf{T})}{\tau\mathsf{m}(\overrightarrow{\tau}\,\overrightarrow{\mathsf{x}})\{\mathsf{return}\ \mathsf{t};\}\ \mathit{OK}\ \text{in}\ \mathsf{T}}\ [\mathsf{M}\ \mathit{OK}\ \text{in}\ \mathsf{T}]$$

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

$$\frac{\overline{\mathsf{M}}\ \mathit{OK}\ \text{in}\ \mathsf{I} \quad \mathtt{mh}(\mathsf{I})\ \text{defined}}{\mathsf{interface}\ \mathsf{I}\ \mathsf{extends}\ \overrightarrow{\mathsf{I}}\ \{\overline{\mathsf{H}}; \overline{\mathsf{M}}\}\ \mathit{OK}}\ [\mathsf{I}\ \mathit{OK}]$$

Fig. 11: Method, Class and Interface Declaration Typing Rules

To sum up, the program (CT, t) is well typed if the class table CT is well formed and for some τ we have that $\vdash \mathsf{t} : \tau$, using the declarations and the subtyping of CT.

5 Subject Reduction and Progress

The subject reduction proof of FJP&λ extends that of FJ&λ taking into account the replacement of nominal types by intersection types and the generalisation of target types.

As usual our type system enjoys *weakening*, i.e., $\Gamma \vdash \mathsf{t} : \tau$ implies $\Gamma, \mathsf{x} : \sigma \vdash \mathsf{t} : \tau$ and $\Gamma \vdash^* \mathsf{t} : \tau$ implies $\Gamma, \mathsf{x} : \sigma \vdash^* \mathsf{t} : \tau$.

Lemma 1. *(1) If* $\mathsf{C}[\&\iota] <: \mathsf{D}[\&\iota']$, *then* $\mathtt{fields}(\mathsf{D}) \subseteq \mathtt{fields}(\mathsf{C})$.
(2) If $\mathtt{mtype}(\mathsf{m};\tau) = \overrightarrow{\rho} \to \rho$, *then* $\mathtt{mtype}(\mathsf{m};\sigma) = \overrightarrow{\rho} \to \rho$ *for all* $\sigma <: \tau$.

Proof. (1) Assume that ι and ι' are present and D is not `Object`, the proof in the other cases is simpler. From $\mathsf{C}\&\iota <: \mathsf{D}\&\iota'$ and rule [<: &R] of Figure 4 we get $\mathsf{C}\&\iota <: \mathsf{D}$. Therefore, since $\iota <: \mathsf{D}$ cannot hold, from rule [<: &L] of Figure 4 we have that $\mathsf{C} <: \mathsf{D}$.
(2) By induction on the derivation of $\sigma <: \tau$ it is easy to prove $\mathtt{mh}(\tau) \subseteq \mathtt{mh}(\sigma)$.

Lemma 2 (Substitution).

(1) If $\Gamma, \mathsf{x} : \sigma \vdash^* \mathsf{t} : \tau$ *and* $\Gamma \vdash^* \mathsf{v} : \sigma$, *then* $\Gamma \vdash^* [\mathsf{x} \mapsto (\mathsf{v})^{?\sigma}]\mathsf{t} : \tau$.
(2) If $\Gamma, \mathsf{x} : \sigma \vdash \mathsf{t} : \tau$ *and* $\Gamma \vdash^* \mathsf{v} : \sigma$, *then* $\Gamma \vdash [\mathsf{x} \mapsto (\mathsf{v})^{?\sigma}]\mathsf{t} : \rho$ *for some* $\rho <: \tau$.

Proof. (1) and (2) are proved by simultaneous induction on type derivations.

(1). If $\Gamma, \mathsf{x} : \sigma \vdash^* \mathsf{t} : \tau$, then the last rule applied is [$\vdash \vdash^*$]. We consider first the case of t being a pure λ-expression, and then t being any of the other terms. **Case** $\mathsf{t} = \overrightarrow{\mathsf{y}} \to \mathsf{t}'$**.** The premise of rule [$\vdash \vdash^*$] must be $\Gamma, \mathsf{x} : \sigma \vdash (\overrightarrow{\mathsf{y}} \to \mathsf{t}')^{\tau} : \tau$. By part (2) of the induction hypothesis we have that $\Gamma \vdash ([\mathsf{x} \mapsto (\mathsf{v})^{?\sigma}](\overrightarrow{\mathsf{y}} \to \mathsf{t}'))^{\tau} : \rho$ for some $\rho <: \tau$. Since the last rule applied in the derivation is [T-λU], we get

$\rho = \tau$. Using rule $[\vdash \vdash^*]$ we conclude $\Gamma \vdash^* [\mathsf{x} \mapsto (\mathsf{v})^{?\sigma}](\overrightarrow{\mathsf{y}} \to \mathsf{t}') : \tau$. The proof for the case $\mathsf{t} = \overrightarrow{\tau}\,\overrightarrow{\mathsf{y}} \to \mathsf{t}'$ is similar.

Case t not a pure λ-expression. The premise of rule $[\vdash \vdash^*]$ is $\Gamma, \mathsf{x} : \sigma \vdash \mathsf{t} : \rho$ for some $\rho <: \tau$. By part (2) of the induction hypothesis $\Gamma \vdash [\mathsf{x} \mapsto (\mathsf{v})^{?\sigma}]\mathsf{t} : \rho'$ for some $\rho' <: \rho$. The transitivity of $<:$ gives $\rho' <: \tau$. Applying rule $[\vdash \vdash^*]$ we conclude $\Gamma \vdash^* [\mathsf{x} \mapsto (\mathsf{v})^{?\sigma}]\mathsf{t} : \tau$.

(2). By cases on the last rule used in the derivation of $\Gamma, \mathsf{x} : \sigma \vdash \mathsf{t} : \tau$.

Case [T-VAR]. $\Gamma, \mathsf{x} : \sigma \vdash \mathsf{x} : \tau$ implies $\sigma = \tau$. The judgment $\Gamma \vdash^* \mathsf{v} : \tau$ must be obtained by applying rule $[\vdash \vdash^*]$ with premise $\Gamma \vdash (\mathsf{v})^{?\tau} : \rho$ for some $\rho <: \tau$, as required.

Case [T-FIELD]. In this case $\mathsf{t} = \mathsf{t}'.\mathsf{f}$ and

$$\frac{\Gamma, \mathsf{x} : \sigma \vdash \mathsf{t}' : \mathsf{C}\&\iota \quad \tau\,\mathsf{f} \in \mathtt{fields}(\mathsf{C})}{\Gamma, \mathsf{x} : \sigma \vdash \mathsf{t}'.\mathsf{f} : \tau}$$

(the case in which $\&\iota$ is missing is easier). The induction hypothesis implies $\Gamma \vdash [\mathsf{x} \mapsto (\mathsf{v})^{?\sigma}]\mathsf{t}' : \rho$ for some $\rho <: \mathsf{C}\&\iota$. The subtyping rules of Figure 4 give $\rho = \mathsf{D}[\&\iota']$ for some D and ι'. By Lemma 1(1) we have that $\mathtt{fields}(\mathsf{C}) \subseteq \mathtt{fields}(\mathsf{D})$ and then $\tau\,\mathsf{f} \in \mathtt{fields}(\mathsf{D})$. Therefore applying rule [T-FIELD] we conclude $\Gamma \vdash [\mathsf{x} \mapsto (\mathsf{v})^{?\sigma}]\mathsf{t}'.\mathsf{f} : \tau$.

Case [T-INVK]. In this case $\mathsf{t} = \mathsf{t}'.\mathsf{m}(\overrightarrow{\mathsf{t}})$ and

$$\frac{\Gamma, \mathsf{x} : \sigma \vdash \mathsf{t}' : \rho \quad \mathtt{mtype}(\mathsf{m}; \rho) = \overrightarrow{\tau} \to \tau \quad \Gamma, \mathsf{x} : \sigma \vdash^* \overrightarrow{\mathsf{t}} : \overrightarrow{\tau}}{\Gamma, \mathsf{x} : \sigma \vdash \mathsf{t}'.\mathsf{m}(\overrightarrow{\mathsf{t}}) : \tau}$$

From $\Gamma, \mathsf{x} : \sigma \vdash^* \overrightarrow{\mathsf{t}} : \overrightarrow{\tau}$ we get $\Gamma \vdash^* [\mathsf{x} \mapsto (\mathsf{v})^{?\sigma}]\overrightarrow{\mathsf{t}} : \overrightarrow{\tau}$ by part (1) of the induction hypothesis. By induction hypothesis on $\Gamma, \mathsf{x} : \sigma \vdash \mathsf{t}' : \rho$ we have that $\Gamma \vdash [\mathsf{x} \mapsto (\mathsf{v})^{?\sigma}]\mathsf{t}' : \rho'$ for some $\rho' <: \rho$. Lemma 1(2) gives $\mathtt{mtype}(\mathsf{m}; \rho') = \overrightarrow{\tau} \to \tau$. Applying rule [T-INVK] we conclude $\Gamma \vdash [\mathsf{x} \mapsto (\mathsf{v})^{?\sigma}](\mathsf{t}'.\mathsf{m}(\overrightarrow{\mathsf{t}})) : \tau$.

Case [T-NEW]. By part (1) of the induction hypothesis on the judgments for the parameters.

Case [T-λU]. In this case $\mathsf{t} = (\overrightarrow{\mathsf{y}} \to \mathsf{t}')^{\tau}$ and

$$\frac{\rho\,(\overrightarrow{\mathsf{m}}\,\overrightarrow{\rho})\mathsf{x} \in \mathtt{A\text{-}mh}(\tau) \text{ implies } \Gamma, \mathsf{x} : \sigma, \overrightarrow{\mathsf{y}} : \overrightarrow{\rho} \vdash^* \mathsf{t}' : \rho}{\Gamma, \mathsf{x} : \sigma \vdash (\overrightarrow{\mathsf{y}} \to \mathsf{t}')^{\tau} : \tau}$$

By part (1) of the induction hypothesis we have that $\rho\,(\overrightarrow{\mathsf{m}}\,\overrightarrow{\rho})\mathsf{x} \in \mathtt{A\text{-}mh}(\tau)$ implies $\Gamma, \overrightarrow{\mathsf{y}} : \overrightarrow{\rho} \vdash^* [\mathsf{x} \mapsto (\mathsf{v})^{?\sigma}]\mathsf{t}' : \rho$. Applying rule [T-$\lambda$U] we conclude

$$\Gamma \vdash ([\mathsf{x} \mapsto (\mathsf{v})^{?\sigma}](\overrightarrow{\mathsf{y}} \to \mathsf{t}'))^{\tau} : \tau$$

Case [T-λT]. In this case $\mathsf{t} = (\overrightarrow{\tau}\,\overrightarrow{\mathsf{y}} \to \mathsf{t}')^{\tau}$ and

$$\frac{\Gamma, \mathsf{x} : \sigma \vdash (\overrightarrow{\mathsf{y}} \to \mathsf{t}')^{\tau} : \tau \quad \rho\,\mathsf{m}(\overrightarrow{\rho}\,\overrightarrow{\mathsf{x}}) \in \mathtt{A\text{-}mh}(\iota) \text{ implies } \overrightarrow{\rho} = \overrightarrow{\tau}}{\Gamma, \mathsf{x} : \sigma \vdash (\overrightarrow{\tau}\,\overrightarrow{\mathsf{y}} \to \mathsf{t}')^{\tau} : \tau}$$

By induction hypothesis we have that $\Gamma \vdash ([\mathsf{x} \mapsto (\mathsf{v})^{?\sigma}](\overrightarrow{\mathsf{y}} \to \mathsf{t}'))^{\tau} : \tau$. Applying rule [T-$\lambda$T] we conclude $\Gamma \vdash ([\mathsf{x} \mapsto (\mathsf{v})^{?\sigma}](\overrightarrow{\tau}\,\overrightarrow{\mathsf{y}} \to \mathsf{t}'))^{\tau} : \tau$.

Lemma 3. *If* $\mathtt{mtype}(\mathsf{m}; \tau) = \overrightarrow{\sigma} \to \sigma$ *and* $\mathtt{mbody}(\mathsf{m}; \tau) = (\overrightarrow{\mathsf{x}}, \mathsf{t})$*, then* $\overrightarrow{\mathsf{x}} : \overrightarrow{\sigma}, \mathsf{this} : \mathsf{T} \vdash^* \mathsf{t} : \sigma$ *for some* T *such that* $\tau <: \mathsf{T}$.

Proof. Let $\tau = \mathsf{C}\&\iota$, the other cases being simpler. By definition of `mbody` method m must be declared in:

– class C or

– some interface in ι or
– some class or interface from which either C or an interface in ι inherits.

In all cases rule [M *OK* in C] or [M *OK* in I] of Figure 11 gives the desired typing judgement.

Lemma 4. *If* $\Gamma \vdash^* \mathsf{t} : \tau$*, then* $\Gamma \vdash (\mathsf{t})^{?\tau} : \sigma$ *for some* $\sigma <: \tau$*.*

Proof. The judgment $\Gamma \vdash^* \mathsf{t} : \tau$ must be obtained by applying rule [$\vdash \vdash^*$] with premise $\Gamma \vdash (\mathsf{t})^{?\tau} : \sigma$ for some $\sigma <: \tau$, as required.

Theorem 1 (Subject Reduction). *If* $\Gamma \vdash \mathsf{t} : \tau$ *without using rule* [T-UDCAST] *and* $\mathsf{t} \longrightarrow \mathsf{t}'$*, then* $\Gamma \vdash \mathsf{t}' : \sigma$ *for some* $\sigma <: \tau$*.*

Proof. By induction on a derivation of $\mathsf{t} \longrightarrow \mathsf{t}'$, with a case analysis on the final rule. We only consider interesting cases.

$$\textbf{Case}\ \frac{\tau\,\mathsf{f}_j \in \mathtt{fields}(\mathsf{C})}{\mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}}).\mathsf{f}_j \longrightarrow (\mathsf{v}_j)^{?\tau}}\ [\text{E-ProjNew}]$$

The l.h.s. is typed as follows:
$$\frac{\mathtt{fields}(\mathsf{C}) = \overrightarrow{\tau}\ \overrightarrow{\mathsf{f}} \quad \Gamma \vdash^* \overrightarrow{\mathsf{v}} : \overrightarrow{\tau}}{\dfrac{\Gamma \vdash \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}}) : \mathsf{C} \qquad \tau\,\mathsf{f}_j \in \mathtt{fields}(\mathsf{C})}{\Gamma \vdash \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}}).\mathsf{f}_j : \tau}}$$

From Lemma 4 and $\Gamma \vdash^* \overrightarrow{\mathsf{v}} : \overrightarrow{\tau}$ we derive that
$$\Gamma \vdash (\mathsf{v}_j)^{?\tau} : \sigma \text{ for some } \sigma <: \tau.$$

$$\textbf{Case}\ \frac{\mathtt{mbody}(\mathsf{m};\mathsf{C}) = (\overrightarrow{\mathsf{x}}, \mathsf{t}'') \quad \mathtt{mtype}(\mathsf{m};\mathsf{C}) = \overrightarrow{\tau} \to \tau}{\mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}}).\mathsf{m}(\overrightarrow{\mathsf{u}}) \longrightarrow [\overrightarrow{\mathsf{x}} \mapsto (\overrightarrow{\mathsf{u}})^{?\overrightarrow{\tau}}, \mathsf{this} \mapsto \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}})](\mathsf{t}'')^{?\tau}}\ [\text{E-InvkNew}]$$

The l.h.s. is typed as follows:
$$\frac{\Gamma \vdash \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}}) : \mathsf{C} \quad \mathtt{mtype}(\mathsf{m};\mathsf{C}) = \overrightarrow{\tau} \to \tau \quad \Gamma \vdash^* \overrightarrow{\mathsf{u}} : \overrightarrow{\tau}}{\Gamma \vdash \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}}).\mathsf{m}(\overrightarrow{\mathsf{u}}) : \tau}$$

By Lemma 3 $\mathtt{mbody}(\mathsf{m};\mathsf{C}) = (\overrightarrow{\mathsf{x}}, \mathsf{t}'')$ implies $\overrightarrow{\mathsf{x}} : \overrightarrow{\tau}, \mathsf{this} : \mathsf{T} \vdash^* \mathsf{t}'' : \tau$ with $\mathsf{C} <: \mathsf{T}$ for some T. Let $\Gamma' = \overrightarrow{\mathsf{x}} : \overrightarrow{\tau}, \mathsf{this} : \mathsf{T}$. By Lemma 4 $\Gamma' \vdash (\mathsf{t}'')^{?\tau} : \rho$ for some $\rho <: \tau$ and by weakening $\Gamma, \Gamma' \vdash (\mathsf{t}'')^{?\tau} : \rho$. From $\Gamma \vdash \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}}) : \mathsf{C}$ and $\mathsf{C} <: \mathsf{T}$ we get $\Gamma \vdash^* \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}}) : \mathsf{T}$. From $\Gamma \vdash^* \overrightarrow{\mathsf{u}} : \overrightarrow{\tau}$ and $\Gamma \vdash^* \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}}) : \mathsf{T}$ and $\Gamma, \Gamma' \vdash (\mathsf{t}'')^{?\tau} : \rho$ and Lemma 2(2) we get
$$\Gamma \vdash [\overrightarrow{\mathsf{x}} \mapsto (\overrightarrow{\mathsf{u}})^{?\overrightarrow{\tau}}, \mathsf{this} \mapsto \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}})](\mathsf{t}'')^{?\tau} : \sigma \text{ for some } \sigma <: \rho.$$
Finally by transitivity of $<:$ we have $\sigma <: \tau$.

$$\textbf{Case}\ \frac{\mathtt{A\text{-}mtype}(\mathsf{m};\iota) = \overrightarrow{\tau} \to \tau}{(\overrightarrow{\mathsf{y}} \to \mathsf{t}'')^{\iota}.\mathsf{m}(\overrightarrow{\mathsf{v}}) \longrightarrow [\overrightarrow{\mathsf{y}} \mapsto (\overrightarrow{\mathsf{v}})^{?\overrightarrow{\tau}}](\mathsf{t}'')^{?\tau}}\ [\text{E-Invk}\lambda\text{U-A}]$$

The l.h.s. is typed as follows:

$$\frac{\pi\mathsf{n}(\overrightarrow{\pi}\,\overrightarrow{\mathsf{x}}) \in \mathtt{A\text{-}mh}(\iota) \text{ implies } \Gamma, \overrightarrow{\mathsf{y}} : \overrightarrow{\pi} \vdash^* \mathsf{t}'' : \pi}{\dfrac{\Gamma \vdash (\overrightarrow{\mathsf{y}} \to \mathsf{t}'')^{\iota} : \iota \qquad \mathtt{mtype}(\mathsf{m};\iota) = \overrightarrow{\tau} \to \tau \quad \Gamma \vdash^* \overrightarrow{\mathsf{v}} : \overrightarrow{\tau}}{\Gamma \vdash (\overrightarrow{\mathsf{y}} \to \mathsf{t}'')^{\iota}.\mathsf{m}(\overrightarrow{\mathsf{v}}) : \tau}}$$

The premise of rule [E-InvkλU-A] implies $\Gamma, \overrightarrow{\mathsf{y}} : \overrightarrow{\tau} \vdash^* \mathsf{t}'' : \tau$. By Lemma 4 $\Gamma, \overrightarrow{\mathsf{y}} : \overrightarrow{\tau} \vdash (\mathsf{t}'')^{?\tau} : \rho$ for some $\rho <: \tau$. By Lemma 2(2) we derive

$$\Gamma \vdash [\overrightarrow{\mathsf{y}} \mapsto (\overrightarrow{\mathsf{v}})^{?\overrightarrow{\tau}}](\mathsf{t}'')^{?\tau} : \sigma \text{ for some } \sigma <: \rho$$

Finally by transitivity of $<:$ we have $\sigma <: \tau$.

$$\textbf{Case}\ \frac{\mathtt{mbody}(\mathsf{m};\iota) = (\overrightarrow{\mathsf{x}}, \mathsf{t}'') \quad \mathtt{D\text{-}mtype}(\mathsf{m};\iota) = \overrightarrow{\tau} \to \tau}{(\mathsf{t}_\lambda)^\iota.\mathsf{m}(\overrightarrow{\mathsf{v}}) \longrightarrow [\overrightarrow{\mathsf{x}} \mapsto (\overrightarrow{\mathsf{v}})^{?\overrightarrow{\tau}}, \mathsf{this} \mapsto (\mathsf{t}_\lambda)^\iota](\mathsf{t}'')^{?\tau}}\ [\text{E-Invk}\lambda\text{-D}]$$

The l.h.s. is typed as follows:

$$\frac{\Gamma \vdash (\mathsf{t}_\lambda)^\iota : \iota \quad \mathtt{mtype}(\mathsf{m};\iota) = \overrightarrow{\tau} \to \tau \quad \Gamma \vdash^* \overrightarrow{\mathsf{v}} : \overrightarrow{\tau}}{\Gamma \vdash (\mathsf{t}_\lambda)^\iota.\mathsf{m}(\overrightarrow{\mathsf{v}}) : \tau}$$

By Lemma 3 $\mathtt{mbody}(\mathsf{m};\iota) = (\overrightarrow{\mathsf{x}}, \mathsf{t}'')$ implies $\overrightarrow{\mathsf{x}} : \overrightarrow{\tau}, \mathsf{this} : \mathsf{T} \vdash^* \mathsf{t}'' : \tau$ with $\iota <: \mathsf{T}$ for some T. Let $\Gamma' = \overrightarrow{\mathsf{x}} : \overrightarrow{\tau}, \mathsf{this} : \mathsf{T}$. By Lemma 4 $\Gamma' \vdash (\mathsf{t}'')^{?\tau} : \rho$ for some $\rho <: \tau$ and by weakening $\Gamma, \Gamma' \vdash^* (\mathsf{t}'')^{?\tau} : \rho$. From $\Gamma \vdash (\mathsf{t}_\lambda)^\iota : \iota$ and $\iota <: \mathsf{T}$ by rule [$\vdash \vdash^*$] we derive $\Gamma \vdash^* (\mathsf{t}_\lambda)^\iota : \mathsf{T}$. Therefore, by Lemma 2(2) we derive

$$\Gamma \vdash [\overrightarrow{\mathsf{x}} \mapsto (\overrightarrow{\mathsf{v}})^{?\overrightarrow{\tau}}, \mathsf{this} \mapsto (\mathsf{t}_\lambda)^\iota](\mathsf{t}'')^{?\tau} : \sigma \text{ where } \sigma <: \rho$$

The transitivity of $<:$ implies $\sigma <: \tau$.

$$\textbf{Case}\ \frac{\mathsf{t} \longrightarrow \mathsf{t}'}{\mathsf{w}.\mathsf{m}(\overrightarrow{\mathsf{v}}, \mathsf{t}, \overrightarrow{\mathsf{t}}) \longrightarrow \mathsf{w}.\mathsf{m}(\overrightarrow{\mathsf{v}}, \mathsf{t}', \overrightarrow{\mathsf{t}})}\ [\text{E-Invk-Arg}]$$

The l.h.s. is typed as follows:

$$\frac{\Gamma \vdash \mathsf{w} : \rho \quad \mathtt{mtype}(\mathsf{m};\rho) = \overrightarrow{\tau} \to \tau \quad \Gamma \vdash^* \overrightarrow{\mathsf{v}} : \overrightarrow{\tau_\mathsf{v}} \quad \Gamma \vdash^* \mathsf{t} : \sigma \quad \Gamma \vdash^* \overrightarrow{\mathsf{t}} : \overrightarrow{\tau_\mathsf{t}}}{\Gamma \vdash \mathsf{w}.\mathsf{m}(\overrightarrow{\mathsf{v}}, \mathsf{t}, \overrightarrow{\mathsf{t}}) : \tau}$$

where $\overrightarrow{\tau} = \overrightarrow{\tau_\mathsf{v}}, \sigma, \overrightarrow{\tau_\mathsf{t}}$. By Lemma 4 $\Gamma \vdash^* \mathsf{t} : \sigma$ implies $\Gamma \vdash (\mathsf{t})^{?\sigma} : \sigma'$ for some $\sigma' <: \sigma$. Since $\mathsf{t} \longrightarrow \mathsf{t}'$ implies that t cannot be a λ-expression we get $(\mathsf{t})^{?\sigma} = \mathsf{t}$. By induction hypothesis $\Gamma \vdash \mathsf{t}' : \rho'$ for some $\rho' <: \sigma'$. Being $\rho' <: \sigma$ applying rule [$\vdash \vdash^*$] we derive $\Gamma \vdash^* \mathsf{t}' : \sigma$. Therefore using the typing rule [T-INVK] we conclude

$$\Gamma \vdash \mathsf{w}.\mathsf{m}(\overrightarrow{\mathsf{v}}, \mathsf{t}, \overrightarrow{\mathsf{t}}) : \tau$$

Rule [T-UDCAST] breaks subject reduction already for FJ, as shown in [8] (Section 19.4). Following [8] we can recover subject reduction by erasing the condition "either $\mathsf{C} <: \mathsf{D}$ or $\mathsf{D} <: \mathsf{C}$" in rule [T-UDCAST]. In this way the rule becomes:

$$\frac{\Gamma \vdash \mathsf{t} : \tau \quad \tau \not<: \sigma}{\Gamma \vdash (\sigma)\,\mathsf{t} : \sigma}\ [\text{T-STUPIDCAST}]$$

The closed terms that are typed without using rule [T-UDCAST] enjoy the standard progress property. This can be easily proven by just looking at the shapes of well-typed irreducible terms.

Theorem 2 (Progress). *If* $\vdash^* \mathsf{t} : \tau$ *without using rule* [T-UDCAST] *and* t *cannot reduce, then* t *is a proper value.*

Using rule [T-UDCAST] we can type casts of proper values which cannot be reduced, for example, $(\mathsf{C})\,(\mathtt{new\ Object}())$ with C different from `Object`. An

example involving a λ-expression is $(\mathsf{C})\,(\epsilon \to \mathtt{new}\,\mathtt{Object}(\,))^{\mathsf{I}}$, where I is the interface with the only signature $\mathtt{Object}\,\mathsf{m}(\,)$. This term can be obtained by reducing $(\mathsf{C})\,(\mathsf{I})\,(\epsilon \to \mathtt{new}\,\mathtt{Object}(\,))$.

To characterise the stuck terms, i.e., the irreducible terms which can be obtained by reducing typed terms and are not values, we resort to the notion of evaluation context, as done in [8] (Theorem 19.5.4). *Evaluation contexts* $\mathcal{E}$ are defined as expected:

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

Stuck terms are evaluation contexts with holes filled by casts of typed proper values which cannot reduce, i.e., terms of the shapes $(\tau)\,\mathtt{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}})$ with $\mathsf{C} \not<: \tau$ and $(\tau)\,(\mathsf{t}_\lambda)^\iota$ with $\iota \not<: \tau$. Notice that $(\mathsf{A}[\&\iota])\,\mathtt{new}\,\mathsf{C}(\overrightarrow{\mathsf{v}})$ cannot be typed when A, C are unrelated classes. Instead rule [T-UDCAST] allows us to type all terms of the shape $(\tau)\,(\mathsf{t}_\lambda)^\iota$, when $(\mathsf{t}_\lambda)^\iota$ has a type.

6 Type Inference

Our type system naturally uses the technique of *bidirectional checking* [9, 4]. In fact the judgments $\vdash$ operate in synthesis mode, propagating typing upward from subexpressions, while the judgments $\vdash^*$ operate in checking mode, propagating typing downward from enclosing expressions.

$\mathtt{tInf}(\Gamma;\mathsf{x})$	$= \tau$ if $\mathsf{x} : \tau \in \Gamma$
$\mathtt{tInf}(\Gamma;\mathsf{t.f})$	$= \tau$ if $\mathtt{tInf}(\Gamma;\mathsf{t}) = \mathsf{C}[\&\iota]$ and $\tau\,\mathsf{f} \in \mathtt{fields}(\mathsf{C})$
$\mathtt{tInf}(\Gamma;\mathtt{new}\,\mathsf{C}(\overrightarrow{\mathsf{t}}))$	$= \mathsf{C}$ if $\mathtt{fields}(\mathsf{C}) = \overrightarrow{\tau}\,\overrightarrow{\mathsf{f}}$ and $\mathtt{tCk}(\Gamma;\overrightarrow{\mathsf{t}};\overrightarrow{\tau})$
$\mathtt{tInf}(\Gamma;\mathsf{t.m}(\overrightarrow{\mathsf{t}}))$	$= \tau$ if $\mathtt{tInf}(\Gamma;\mathsf{t}) = \sigma$ and $\mathtt{mtype}(\mathsf{m};\sigma) = \overrightarrow{\tau} \to \tau$ and $\mathtt{tCk}(\Gamma;\overrightarrow{\mathsf{t}};\overrightarrow{\tau})$
$\mathtt{tInf}(\Gamma;(\tau)\,\mathsf{t})$	$= \tau$ if one of the following conditions holds • $\mathtt{tCk}(\Gamma;\mathsf{t};\tau)$ • $\tau = \mathsf{C}[\&\iota]$ and $\mathtt{tInf}(\Gamma;\mathsf{t}) = \mathsf{D}[\&\iota']$ and either $\mathsf{C} <: \mathsf{D}$ or $\mathsf{D} <: \mathsf{C}$
$\mathtt{tInf}(\Gamma;(\overrightarrow{\mathsf{y}} \to \mathsf{t})^\iota)$	$= \iota$ if $\mathtt{tCk}(\Gamma, \overrightarrow{\mathsf{y}} : \overrightarrow{\tau};\mathsf{t};\tau)$ for all $\tau\mathsf{m}(\overrightarrow{\tau}\,\overrightarrow{\mathsf{x}}) \in \mathtt{A\text{-}mh}(\iota)$
$\mathtt{tInf}(\Gamma;(\overrightarrow{\tau}\,\overrightarrow{\mathsf{y}} \to \mathsf{t})^\iota)$	$= \iota$ if $\mathtt{tCk}(\Gamma, \overrightarrow{\mathsf{y}} : \overrightarrow{\sigma};\mathsf{t};\tau)$ and $\overrightarrow{\sigma} = \overrightarrow{\tau}$ for all $\tau\mathsf{m}(\overrightarrow{\sigma}\,\overrightarrow{\mathsf{x}}) \in \mathtt{A\text{-}mh}(\iota)$

Fig. 12: Type Inference Function

We assume a given class table to compute the lookup functions and the subtype relation. The partial function $\mathtt{tInf}(\Gamma;\mathsf{t})$ gives (if any) the type τ such that $\Gamma \vdash \mathsf{t} : \tau$. It is defined by mutual recursion with the predicate $\mathtt{tCk}(\Gamma;\mathsf{t};\tau)$ which is true if $\Gamma \vdash^* \mathsf{t} : \tau$. So, according to rule [$\vdash \vdash^*$]:

$$\mathtt{tCk}(\Gamma;\mathsf{t};\tau) \quad \text{if} \quad \mathtt{tInf}(\Gamma;(\mathsf{t})^{?\tau}) = \sigma \text{ and } \sigma <: \tau$$

This asserts that, if we can infer the type of an expression, then we can check that it has this type, and, in case it is not a λ-expression, also all its supertypes. We write $\mathtt{tCk}(\Gamma;\overrightarrow{\mathsf{t}};\overrightarrow{\tau})$ as short for $\mathtt{tCk}(\Gamma;\mathsf{t}_1;\tau_1), \ldots, \mathtt{tCk}(\Gamma;\mathsf{t}_n;\tau_n)$. Figure 12 gives

$\mathsf{OK}(\tau \mathsf{m}(\overrightarrow{\tau}\,\overrightarrow{\mathsf{x}})\{\mathsf{return}\ \mathsf{t};\}, \mathsf{T})$	if $\tau \mathsf{m}(\overrightarrow{\tau}\,\overrightarrow{\mathsf{x}}) \in \mathtt{mh}(\mathsf{T})$ and $\mathtt{tCk}(\Gamma, \overrightarrow{\mathsf{x}} : \overrightarrow{\tau}, \mathsf{this} : \mathsf{T}; \mathsf{t}; \tau)$
$\mathsf{OK}(\mathsf{class\ C\ extends\ D\ implements}\ \overrightarrow{\mathsf{I}}\ \{\overline{\tau}\,\overline{\mathsf{f}}; \mathsf{K}\,\overline{\mathsf{M}}\})$	if $\mathsf{K} = \mathsf{C}(\overrightarrow{\sigma}\,\overrightarrow{\mathsf{g}}, \overrightarrow{\tau}\,\overrightarrow{\mathsf{f}})\{\mathsf{super}(\overrightarrow{\mathsf{g}}); \mathsf{this}.\overline{\mathsf{f}} = \overline{\mathsf{f}};\}$ and $\mathtt{fields}(\mathsf{D}) = \overrightarrow{\sigma}\,\overrightarrow{\mathsf{g}}$ and $\mathsf{OK}(\overrightarrow{\mathsf{M}}, \mathsf{C})$ and $\mathtt{mh}(\mathsf{C})$ def. and for any m $\mathtt{mtype}(\mathsf{m}; \mathsf{C})$ def. impl. $\mathtt{mbody}(\mathsf{m}; \mathsf{C})$ def.
$\mathsf{OK}(\mathsf{interface\ I\ extends}\ \overrightarrow{\mathsf{I}}\ \{\overline{\mathsf{H}}; \overline{\mathsf{M}}\})$	if $\mathsf{OK}(\overrightarrow{\mathsf{M}}, \mathsf{I})$ and $\mathtt{mh}(\mathsf{I})$ def.
$\mathsf{OK}(\overline{\mathsf{C}}\,\overline{\mathsf{I}})$	if $\mathsf{OK}(\overrightarrow{\mathsf{C}})$ and $\mathsf{OK}(\overrightarrow{\mathsf{I}})$

Fig. 13: Well-formedness Function

`tInf`. The definition is an algorithmic reading of the rules of Figures 9 and 10. The definition of `tInf` uses the predicate `tCk` (on subexpressions) to check the types of actual parameters, type casts, and bodies of decorated λ-expressions against the types expected from the contexts. As we can see, `tInf` is undefined for pure λ-expressions.

Building on Figure 11, Figure 13 defines a predicate OK which tests well-formedness of class tables, i.e. of classes, interfaces and methods.
We use the following abbreviations: def. for defined, impl. for implies, $\mathsf{OK}(\overrightarrow{\mathsf{M}}, \mathsf{T})$ for $\mathsf{OK}(\mathsf{M}_1, \mathsf{T}), \ldots, \mathsf{OK}(\mathsf{M}_n, \mathsf{T})$, and $\mathsf{OK}(\overrightarrow{\mathsf{M}})$ for $\mathsf{OK}(\mathsf{M}_1), \ldots, \mathsf{OK}(\mathsf{M}_n)$.

7 Conclusion and Related Works

The core language presented here is essentially based on FJ&λ [1], which in turn extends [7]. Our objective was to investigate how to extend the present use of intersection types in Java through a formal account. As a main result, we proved that the cross fertilisation between intersection types and λ-expressions can be further enhanced, getting a more interesting usability of Java λ-expressions while preserving the language type safety. Notably, nominal intersection types are used everywhere, and the functional interface of a λ-expression can provide more than one abstract method (hence, an intersection type for the function). In this way, a λ-expression can be used with different types in different contexts, similarly to what happens in a functional language.

We refer to [1] for a wide survey of the works that are related to this topic, concerning both intersection type theory and the modelling of object-oriented features by intersection types. We refer to Oracle documentation [6] for Java with λ-expressions and intersections.

This paper concentrates mostly on the formal foundation of our proposal. Concerning feasibility of its implementation, in ongoing work we are devising a translation from our calculus into FJ&λ which exactly models the present Java approach. Moreover, we want to investigate how programming methodologies can benefit from these novel features, that seem to be very promising for avoiding the application of *design patterns* [5] and getting a reduced amount of code.

Acknowledgements

We would like to thank the anonymous referees for their helpful comments.

References

1. L. Bettini, V. Bono, M. Dezani-Ciancaglini, P. Giannini, and B. Venneri. Java & Lambda: a Featherweight Story. *Logical Methods in Computer Science*, 2018. To appear.
2. M. Büchi and W. Weck. Compound Types for Java. In B. N. Freeman-Benson and C. Chambers, editors, *OOPSLA*, pages 362–373. ACM, 1998.
3. M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional Characters of Solvable Terms. *Mathematical Logic Quartely*, 27(2-6):45–58, 1981.
4. R. Davies and F. Pfenning. Intersection Types and Computational Effects. In M. Odersky and P. Wadler, editors, *ICFP*, pages 198–208. ACM, 2000.
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995.
6. J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 8 Edition*. Oracle, 2015.
7. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.
8. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
9. B. C. Pierce and D. N. Turner. Local Type Inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, 2000.