Deconfined Intersection Types in Java

Dedicated to Maurizio Gabbrielli on the Occasion of His 60th Birthday

Mariangiola Dezani-Ciancaglini
Computer Science Department, University of Torino, Italy
dezani@di.unito.it

Paola Giannini
Department of Advanced Science and Technology, University of Eastern Piedmont, Alessandria, Italy
paola.giannini@uniupo.it

Betti Venneri
Department of Statistics, Computer Science, Applications, University of Firenze, Italy
betti.venneri@unifi.it

Abstract

We show how Java intersection types can be freed from their confinement in type casts, in such a way that the proposed Java extension is safe and fully compatible with the current language. To this aim, we exploit two calculi which formalise the simple Java core and the extended language, respectively. Namely, the second calculus extends the first one by allowing an intersection type to be used anywhere in place of a nominal type. We define a translation algorithm, compiling programs of the extended language into programs of the former calculus. The key point is the interaction between λ-expressions and intersection types, that adds safe expressiveness while being the crucial matter in the translation. We prove that the translation preserves typing and semantics. Thus, typed programs in the proposed extension are translated to typed Java programs. Moreover, semantics of translated programs coincides with the one of the source programs.

2012 ACM Subject Classification Theory of computation → Type structures; Theory of computation → Semantics and reasoning

Keywords and phrases Intersection Types, Featherweight Java, Lambda Expressions

Digital Object Identifier 10.4230/OASIcs.Gabbrielli.2020.3

Acknowledgements We are very pleased to dedicate this work to Maurizio, whose notable contributions to the theory of programming languages are well known and appreciated by the scientific community.
We would like to thank the anonymous referees for their helpful comments.

1 Introduction

Intersection types have been proposed at the beginning of the Eighties [6, 7, 21, 8, 1] for typing all strongly normalising λ-terms and for building models of λ-calculus. Intersection types provide a form of polymorphism that is an attractive suggestion for programming languages. Indeed, they are able to express a huge (potentially infinite) amount of types about a component of a program, simply by listing the ones that matter in each context. On the other hand, intersection type inference turns out to be quite powerful, since it allows the typing of exactly all terminating programs; this power results in many difficult issues for their implementation. John Reynolds designed the first programming language including intersection types, the Algol-like academic language Forsyte [23, 24], in late Nineties. Since then, only recently intersection types have won the attention of language designers and have been successfully included in real programming languages in some restricted forms, e.g., Scala [9, 19, 22]. Concerning Java, in the last years intersection types have gained small spaces, step by step, in the successive releases of the language [14] (Section 4.9).

Java intersection types take the form $\mathsf{T}\&\mathsf{I}_1\&\ldots\&\mathsf{I}_n$, where T is a class or an interface and $\mathsf{I}_1,\ldots,\mathsf{I}_n$ are interfaces. Thus intersection types break the nominal type system of Java: they allow the user to combine nominal types on demand, and so to express the formal specification of pieces of code as combinations of those interfaces which are exactly needed in each context. This is handy for keeping interfaces very thin (according to the Interface Segregation Principle [18]) and so avoiding interface pollution. However, despite these advantages, intersection types are among the features of Java which have not been adopted by the bulk of the programming community. The main motivation is, in our opinion, the fact that in essence they have never gained full dignity of language types, that is in all sense equivalent to nominal types and therefore as usable as classes and interfaces. We can sum up the boundaries into which Java actually confines their use, as follows:

- as bounds of type variables in generics definitions;
- as target types in explicit type casts;
- as anonymous types that are used only by the type inference system, in particular for typing conditional expressions.

What we immediately notice is that Java lacks support for using intersection types for fields, parameter and return types of methods. This becomes a crucial restriction when considering, in particular, λ-expressions. In Java each λ-expression is always associated with its target type, that is inferred from the enclosing context by the typing system. The target type must be a *functional type*: either a functional interface or an intersection of interfaces, containing exactly one abstract method and any number of *default method* definitions. Thus, the λ-expression provides the implementation for the abstract method, while default methods add new behaviour to the λ-expression. The primary motivation for introducing default methods in Java 8 was interface evolution, i.e., the ability of extending interfaces with new functionalities without breaking down the existing subclasses. But an interesting development arises from the combination of default methods with intersection types and λ-expressions. Assume we have defined several interfaces, containing default methods that can be used and reused in distinct pieces of code. Then each context can choose on the fly the functionalities that are needed and compose them with the abstract method, by casting the λ-expression to the suited intersection type. For instance, in the term $(\mathsf{I}\&\mathsf{I}_1\&\ldots\&\mathsf{I}_n)\,(\mathsf{x}\rightarrow\mathsf{t})$, the intersection type becomes the target type of $\mathsf{x}\rightarrow\mathsf{t}$. In this case, in addition to implementing the abstract method of I, the λ-expression $\mathsf{x}\rightarrow\mathsf{t}$ can be immediately used as the receiver of any default method defined in the other interfaces. This really increases flexibility in using the function $\mathsf{x}\rightarrow\mathsf{t}$. The use of cast is needed in Java, where signatures only contain nominal types. Instead, in the proposed extension a λ-expression can get an intersection type as target type even occurring as parameter or return value of a method (see the last example of Section 2).

Concerning generics [4], it is convenient to dispel the false belief that the use of intersection types as bound of generic type variables in some way can make up for missing freely used intersection types. The generic type variable is a placeholder for a type which is unknown to the compiler, until the caller chooses an actual type to replace the type variable. Differently, using intersection types in parameter and return types of a method declaration, we express a precise constraint, that is the type of the actual parameter must implement a given list of types (formal specifications) and so for the return type. For example, the declaration $<\mathsf{X}\,\mathsf{extends}\,\mathsf{I}_1\&\mathsf{I}_2>\mathsf{X}\,\mathsf{mGen}(\mathsf{X}\,\mathsf{x})$ is totally different from $\mathsf{I}_1\&\mathsf{I}_2\;\mathsf{mInt}(\mathsf{I}_1\&\mathsf{I}_2\mathsf{x})$. Let C and D be two unrelated classes implementing both I_1 and I_2. By instantiating X with C (or D) in mGen, we can apply mGen to an object of type C and the result is of type C (or D, respectively). Differently, method mInt can be applied to an object of type C, or to an object of type D; the result can be an object of any type implementing both I_1 and I_2. For instance, the result will be an object of class C or D, when the body of the method is a conditional expression returning an object C and an object D in the two branches, respectively. Therefore, since the expressive power of intersection types is totally orthogonal to the use of generic types, we leave out generics from the minimal core languages that are exploited in this paper.

In Section 2, we present many examples that show the high degree of boilerplate coding required in Java because of the above restrictions on the usage of intersection types. Our proposal is to desegregate Java intersection types from those restrictions, so that the programmer can use them as field types in class declarations and as parameter and return types in method signatures, as it does with nominal types. This extension of Java is proven to be safe and fully compatible with the current language, that is it does not require any modification of the Java Virtual Machine (JVM), thus keeping the essential property of backward compatibility. The main contribution of this paper is the compilation of the proposed extension into Java core and the proof that compiled programs preserve types and semantics of the source program. To this aim, we exploit two calculi, which formalise the simple Java core and the extended language, respectively. These calculi are minimal core languages in that they omit all the features that are not significant for our purpose.

The first calculus, TJ&, is presented in Section 3. It models how Java 8 deals with λ-expressions and intersection types, that are confined within the above restrictions. TJ& is a lightweight version of FJ&λ, defined in [2], since some features, such as conditional expressions, are avoided to direct attention to the essential points for our issue.

The calculus SJ&+, presented in Section 4, formalises the proposed Java extension. The calculus defined in [10] is a conservative extension of SJ&+, so we inherit from [10] the main properties of type preservation and progress.

In Section 5, we present the compilation algorithm for translating typed SJ&+ programs into TJ& programs. As the first step, we erase all the intersection types appearing in field declarations and in method signatures. Each erased intersection is replaced with its most relevant component, that is either the class or the functional interface (if any). Then the lost type information is recovered by inserting several downcasts into the source code. The intrinsic goal of the added type casts is preserving typing and semantics. As expected, the crucial issue is the translation of λ-expressions with their target types.

Properties of the translation are discussed in Section 6. We prove that translated programs are typed too. Then we show that inserted casts are guaranteed to not fail at run time. Thus source and target programs either produce “indistinguishable” values or both reduce forever. By “indistinguishable” we roughly mean that the difference between the values are type casts which never fail at run-time.

We conclude in Section 7 discussing related and future works.

2 Motivating Examples

This section is split into three parts. The first two parts show the advantages of the proposed extension also in presence of generic types and of the `var` construct. The third part exemplifies the expressivity of deconfined intersection types for typing λ-expressions. The actual Java code is on light-grey background, while the proposal Java code is on light-green background.

Generic Types

We want to implement a game in which players with different moving capabilities explore a world picking up objects as they go along. Object oriented modelling lends itself well to directly translating “real-world” *entities* and their *capabilities* into code, by describing capabilities via *interfaces* and entities via *classes* implementing their capabilities.

In our game some players can fly, some can swim and others can do both. Java interfaces are used to model the two moving capabilities, so we can have players implementing both interfaces as the following code shows.

```
interface Flyable { void fly(); }

interface Swimmable { void swim(); }

public class NaviatorDrone implements Flyable, Swimmable {
 public void fly() { ... }
 public void swim() { ... };
}

public class Pelican implements Flyable, Swimmable {
 public void fly() { ... }
 public void swim() { ... };
}
```

Moreover, we can define *actions*, implemented by *methods*, that may require players with either one of the two capabilities or both. We concentrate on modelling the latter.

We want to write a method, `goAcrossRavine`, that requires to fly over a ravine. If there is an object in the stream at the bottom, the player has to fly down, dive and swim into the water and then fly up, after having picked the object up. So we need `Flyable` and `Swimmable` players.

```
public class Game {
 public static void goAcrossRavine(XXX player, boolean underwaterObj){
  System.out.println("Reached␣the␣ravine");
  if (underwaterObj) {
   player.fly();
   player.swim();
   System.out.println("Picked␣Object");
   player.swim();
   player.fly();
  } else player.fly();
  System.out.println("Crossed␣the␣ravine");
 }
 // Other methods of the game using the capabilities of players
 }
```

We write `XXX` as type of the `player` parameter, since it must implement `Flyable` and `Swimmable`, but in Java we cannot specify `Flyable & Swimmable`. One natural solution is to define a new interface

```
interface FlyableSwimmable extends Flyable, Swimmable {}
```

and use it for `XXX`. Now, if we want to apply the method to our `NaviatorDrone` and `Pelican` we have to change their class definitions and make them implement `FlyableSwimmable`.

```
public class NaviatorDrone implements FlyableSwimmable { ... }

public class Pelican implements FlyableSwimmable { ... }

public class Game {
 public static void
        goAcrossRavine(FlyableSwimmable player, boolean underwaterObj){
    ...
 }
 // Other methods of the game using the capabilities of players
}
```

This change raises some problems, in particular

1. we may not have access to the implementation of `NaviatorDrone` and `Pelican` (that could come from different sources), and
2. we have to anticipate and define all combinations of the capabilities we will need in the evolution of the class `Game`.

The second point is particularly delicate, as it may lead to *interface proliferation*, one of the motivations behind the introduction of intersection types.

To avoid these problems we could use a generic variable with the intersection as a bound. So, instead of defining a new interface, the `player` parameter has a generic variable as type, whose bound is the intersection of `Flyable` and `Swimmable`.

```
public class Game {
 public static void <X extends Swimmable & Flyable>
        goAcrossRavine (X player, boolean underwaterObj){
       ...
 }
 // Other methods of the game using the capabilities of players
 public static void main() {
  // ...
  goAcrossRavine(new Pelican(), true);
  goAcrossRavine(new NaviatorDrone(), false);
 }
}
```

In Java *intersection types cannot be used in the declaration of variables*, i.e., declarations such as

```
Swimmable & Flyable player = new Pelican();
```

are not permitted, even though, as the last two lines of the previous code show, `new Pelican()` or `new NaviatorDrone()` can be arguments of the method `goAcrossRavine`.

We could try to use generic variables with the intersection `Swimmable & Flyable` as bound also for the variable definitions. However we need *type cast* that may cause type errors at run time.

```
public static <X extends Swimmable & Flyable> void main(){
 // ...
 X player = (X) new Pelican();
 goAcrossRavine(player, true);
}
```

Moreover, the interface of the method `main` would espose the type of a local variable!

The `var` Constructor

In Java 10 [16] the `var` construct was introduced having the prominent feature that the type of the declared variable is inferred from the expression assigned to it. The static type of the `var` variable is the type inferred for the expression on the right side. The expression cannot be a λ-expression. The inferred type can be an intersection type, for instance when the expression is a conditional expression. This can be useful to avoid boilerplate code in the body of a method: if the type of the variable is an intersection type you can call on it all methods of this intersection. However, the above benefits are restricted to local variables. In fact `var` variables cannot be used for fields, method parameters and return types, that require explicitly declared types (never inferred types).

For example, with this construct we can declare `player` variables that can be used where we require objects of intersection type, as the following code shows:

```java
public static void main() {
 // ...
 var player = new Pelican();
 goAcrossRavine(player, true);
}
```

Allowing *intersections as return types of methods*, we can write the following method, which builds a player for our game:

```java
enum Version{HIGHTECH, CLASSICAL}
```

```java
public static Swimmable & Flyable makePlayer(Version v){
 return (v==Version.HIGHTECH)? new NaviatorDrone(): new Pelican());
}
```

In Java, we write a corresponding method using both generics and the `var` construct

```java
public static<X extends Swimmable & Flyable> X makePlayer(Version v){
  var res=((v==Version.HIGHTECH)? new NaviatorDrone(): new Pelican());
  return (X)res;
}
```

We observe that a cast is needed for a correct compilation and the code is less readable.

λ-expressions

Intersection types and type inference are crucial for typing λ-*expressions*, another feature added to Java 8 [14] (Section 15.27).

Intersections of interfaces may be *target types* of λ-expressions. The *intersection of interfaces must be functional*, i.e., to have exactly one abstract method, the one implemented by the λ-expression. The limitations on the use of intersection types imposed by Java reduce the usability of λ-expressions, as the following example shows.

Assume we want to write a method `finalPrice` that computes the amount to charge for a purchase. This method can vary according to the algorithm for defining the discount and it must choose a policy for charging a delivery cost. We assume that different strategies for the delivery cost are encapsulated in the default methods of several interfaces. The reason motivating the use of interfaces with default methods, instead of classes (as in the Strategy Design Pattern [13]), is to allow these interfaces to appear in the type of a λ-expression, when combined in an intersection type with a functional interface. Thus the behaviour of `finalPrice` can be parametric with respect to one single λ-expression, to which the method delegates the definition of the delivery cost as well as the implementation of the discount algorithm.

For example, we assume the following simple declarations.

```java
interface Discount { double discount(int price); }

interface DeliveryPrice {
 default double deliveryPrice(int price) {
  return (price>30)? 0: 5;
 }
}
```

Then the method for the final price would be very compact and clean in our proposed extension of Java, by using intersection types in parameter types. It could be defined as follows:

```
public static double
       finalPrice(Discount & DeliveryPrice funPrice, int price){
 return funPrice.discount(price)+funPrice.deliveryPrice(price);
}
```

For example, the following call of the method applies a 1% discount if the price is more than 100 euros and charges 5 euros for the delivery only if the price is less than or equal to 30 euros (for simplicity, the actual price is denoted by `n`):

```
double computedPrice=finalPrice(x->x-((x>100)? x*0.01: 0),n);
```

Differently, in Java, given the restrictions on the use of intersection types, we have to move the parameter into a local variable inside the method body, in order to obtain the behaviour above. Namely:

```
public static double finalPrice(int price) {
 var funPrice=(Discount & DeliveryPrice)(x->x-((x>100)? x*0.01: 0));
 return funPrice.discount(price)+funPrice.deliveryPrice(price);
}
```

Notice that this code compiles only if the λ-expression `x->x-((x>100)? x*0.01: 0)` is type cast. Most importantly, `finalPrice` is not parametric on the discount policy, i.e., we have to modify the method body for changing the discount algorithm.

Therefore, we can try to use Java generics, where the intersection type can be the bound of the type variable:

```
public static <X extends Discount & DeliveryPrice> double
       finalPrice(X funPrice, int price) {
 return funPrice.discount(price)+funPrice.deliveryPrice(price);
}
```

In this case, the call of `finalPrice` compiles if the passed λ-expression

`x->x-((x>100)? x*0.01: 0)`

is cast to the intersection type, i.e.,

```
double computedPrice=
      finalPrice((Discount & DeliveryPrice)(x->x-((x>100)? x*0.01: 0)),n)
          ;
```

compiles, while the code

```
double computedPrice=finalPrice(x->x-((x>100)? x*0.01: 0),n);
```

gives the error

```
Example.java:20:error:incompatible types: cannot infer type-variable(s) X
double computedPrice = finalPrice(x->x-((x>100)? x*0.01: 0),n);
                                 ^
    X extends Discount,DeliveryPrice declared in
                                        method<X>finalPrice(X,int)
  where INT#1 is an intersection type:
    INT#1 extends Object,Discount,DeliveryPrice
```

The discussion of this example shows the utility of default methods in interfaces, since they can be invoked not only on objects but also on λ-expressions.

3 Java with Confined Intersection Types (TJ&)

In this section we present our *target calculus* TJ& formalising the use of intersection types and λ-expressions in Java 8. A small extension of this calculus has been introduced in [2].

We use $\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}$ to denote classes, I, J to denote interfaces, T, U to denote nominal types, i.e., either classes or interfaces; 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 $\overrightarrow{\mathsf{I}}$ as a shorthand for the list $\mathsf{I}_1, \dots, \mathsf{I}_n$, $\overline{\mathsf{M}}$ as a shorthand for the sequence $\mathsf{M}_1 \dots \mathsf{M}_n$, 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 $\overline{\mathsf{N}}$ 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 $\overline{\mathsf{T}}\,\overline{\mathsf{f}};$ abbreviates $\mathsf{T}_1\mathsf{f}_1; \dots \mathsf{T}_n\mathsf{f}_n;$ and $\overrightarrow{\mathsf{T}}\,\overrightarrow{\mathsf{f}}$ abbreviates $\mathsf{T}_1\mathsf{f}_1, \dots, \mathsf{T}_n\mathsf{f}_n$ (likewise $\overrightarrow{\mathsf{T}}\,\overrightarrow{\mathsf{x}}$) and $\mathsf{this}.\overline{\mathsf{f}} = \overline{\mathsf{f}};$ abbreviates $\mathsf{this}.\mathsf{f}_1 = \mathsf{f}_1; \dots \mathsf{this}.\mathsf{f}_n = \mathsf{f}_n;$. This convention on $^{-}$ and $^{\rightarrow}$ is also used in the reduction and typing 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.

Types (ranged over by τ, σ) are generated by the grammar:

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

assuming that classes and interfaces in the intersection type have different method names. The notation $\mathsf{C}[\&\iota]$ means either the class C or the type $\mathsf{C}\&\iota$.

The syntax of terms, classes and interfaces of TJ& is defined by:

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

Terms are values, variables, field accesses, method calls, object creations and casts. Values include λ-expressions. 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. φ denotes a *functional* type, that is an interface or an intersection of interfaces with *exactly* one abstract method. Decorated λ-expressions are produced at run-time only. We use t_λ to range over pure λ-expressions.

CD^{T} ranges over class declarations; ID^{T} ranges over interface declarations; K^{T} ranges over constructor declarations; H^{T} ranges over method header (abstract method) declarations; M^{T} 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. Except for these simplifications, every TJ& program is an executable Java program.

In writing examples, we omit implements and extends when the list of interfaces is empty.

A *class table* CT^{T} 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.

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

- `A-mtypeT`(φ) gives the parameter and return types of the unique abstract method in φ;
- `A-nameT`(φ) gives the name of the unique abstract method in φ;
- `fieldsT`(C) gives the sequence of fields declarations in class C;
- `mtypeT`$(\mathsf{m};\tau)$ gives the parameter and return types of method m in τ;
- `mbodyT`$(\mathsf{m};\tau)$ gives the formal parameters and the body of method m in τ.

$$\frac{\texttt{fieldsT}(\mathsf{C}) = \overrightarrow{\mathsf{T}}\ \overrightarrow{\mathsf{f}}}{\mathsf{new\ C}(\overrightarrow{\mathsf{v}}).\mathsf{f}_j \longrightarrow_{\mathcal{T}} (\mathsf{v}_j)^{?\mathsf{T}_j}}\ [\text{T-ProjNew}] \qquad \frac{\mathsf{C} <: \tau}{(\tau)\ \mathsf{new\ C}(\overrightarrow{\mathsf{v}}) \longrightarrow_{\mathcal{T}} \mathsf{new\ C}(\overrightarrow{\mathsf{v}})}\ [\text{T-CastNew}]$$

$$\frac{\texttt{mbodyT}(\mathsf{m};\mathsf{C}) = (\overrightarrow{\mathsf{x}},\mathsf{t}) \quad \texttt{mtypeT}(\mathsf{m};\mathsf{C}) = \overrightarrow{\mathsf{T}} \to \mathsf{T}}{\mathsf{new\ C}(\overrightarrow{\mathsf{v}}).\mathsf{m}(\overrightarrow{\mathsf{u}}) \longrightarrow_{\mathcal{T}} [\overrightarrow{\mathsf{x}} \mapsto (\overrightarrow{\mathsf{u}})^{?\overrightarrow{\mathsf{T}}}, \mathsf{this} \mapsto \mathsf{new\ C}(\overrightarrow{\mathsf{v}})](\mathsf{t})^{?\mathsf{T}}}\ [\text{T-InvkNew}]$$

$$\frac{\texttt{A-nameT}(\varphi) = \mathsf{m} \quad \texttt{A-mtypeT}(\varphi) = \overrightarrow{\mathsf{T}} \to \mathsf{T}}{(\overrightarrow{\mathsf{y}} \to \mathsf{t})^{\varphi}.\mathsf{m}(\overrightarrow{\mathsf{v}}) \longrightarrow_{\mathcal{T}} [\overrightarrow{\mathsf{y}} \mapsto (\overrightarrow{\mathsf{v}})^{?\overrightarrow{\mathsf{T}}}](\mathsf{t})^{?\mathsf{T}}}\ [\text{T-Invk}\lambda\text{-A}]$$

$$\frac{\texttt{mbodyT}(\mathsf{m};\varphi) = (\overrightarrow{\mathsf{x}},\mathsf{t}) \quad \texttt{mtypeT}(\mathsf{m};\varphi) = \overrightarrow{\mathsf{T}} \to \mathsf{T}}{(\mathsf{t}_\lambda)^{\varphi}.\mathsf{m}(\overrightarrow{\mathsf{v}}) \longrightarrow_{\mathcal{T}} [\overrightarrow{\mathsf{x}} \mapsto (\overrightarrow{\mathsf{v}})^{?\overrightarrow{\mathsf{T}}}, \mathsf{this} \mapsto (\mathsf{t}_\lambda)^{\varphi}](\mathsf{t})^{?\mathsf{T}}}\ [\text{T-Invk}\lambda\text{-D}]$$

$$(\varphi)\,\mathsf{t}_\lambda \longrightarrow_{\mathcal{T}} (\mathsf{t}_\lambda)^{\varphi} [\text{T-C}\lambda] \qquad \frac{\varphi <: \varphi'}{(\varphi')\,(\mathsf{t}_\lambda)^{\varphi} \longrightarrow_{\mathcal{T}} (\mathsf{t}_\lambda)^{\varphi}}\ [\text{T-CC}\lambda] \qquad \frac{\mathsf{t} \longrightarrow_{\mathcal{T}} \mathsf{t}'}{\mathcal{E}[\mathsf{t}] \longrightarrow_{\mathcal{T}} \mathcal{E}[\mathsf{t}']}\ [\text{T-Ctx}]$$

Figure 1 Reduction Rules of TJ&.

We assume that there are no cycles in the subclass relation between nominal types induced by the class table. The *subtype relation* <: takes into account both the subclass relation induced by the class table, and the set theoretic properties of intersection, which give the following relations:

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

In what follows, to lighten the notation of reduction and typing rules, we assume a fixed class table $CT^{\mathcal{T}}$.

The reduction rules are given in Figure 1, where evaluation contexts $\mathcal{E}$ are defined by:

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

Following [2], reduction rules guarantee that pure λ-expressions are decorated by their target types in the evaluated terms. This is realised by means of the mapping $(\mathsf{t})^{?\tau}$ defined as follows:

$$(\mathsf{t})^{?\tau} = \begin{cases} (\mathsf{t})^{\tau} & \text{if } \mathsf{t} = \mathsf{t}_\lambda, \\ \mathsf{t} & \text{otherwise} \end{cases}$$

Namely, this mapping decorates pure λ-expressions with τ, whereas leaves all the other terms unchanged. It is used in propagating the types expected for λ-expressions in object constructors, method calls and type casts. The typing rules assure that if t is a pure λ-expression, then its target type τ is a functional type. So we only get decorated terms of the shape $(\mathsf{t}_\lambda)^{\varphi}$. The reduction of a method call on a λ-expression distinguishes the case of abstract methods from that of default methods. As usual, $\longrightarrow^*_{\mathcal{T}}$ is the reflexive and transitive closure of $\longrightarrow_{\mathcal{T}}$.

The typing rules for terms are given in Figure 2. These rules are standard, but for [T-λ], [T-DC], [T-UC] and the judgment $\vdash^*$. Rule [T-λ] checks that a λ-expression is typed as required by the only abstract method in φ, and the subject of the conclusion is the decorated

$$\dfrac{\mathsf{x} : \mathsf{T} \in \Delta}{\Delta \vdash_{\mathcal{T}} \mathsf{x} : \mathsf{T}}\,[\text{T-VAR}] \qquad \dfrac{\Delta \vdash_{\mathcal{T}} \mathsf{t} : \mathsf{C}[\&\iota] \quad \mathsf{T}\,\mathsf{f} \in \mathtt{fieldsT}(\mathsf{C})}{\Delta \vdash_{\mathcal{T}} \mathsf{t.f} : \mathsf{T}}\,[\text{T-FIELD}]$$

$$\dfrac{\Delta \vdash_{\mathcal{T}} \mathsf{t} : \tau \quad \mathtt{mtypeT}(\mathsf{m};\tau) = \overrightarrow{\mathsf{T}} \to \mathsf{T} \quad \Delta \vdash^{*}_{\mathcal{T}} \bar{\mathsf{t}} : \bar{\mathsf{T}}}{\Delta \vdash_{\mathcal{T}} \mathsf{t.m}(\overrightarrow{\mathsf{t}}) : \mathsf{T}}\,[\text{T-INVK}]$$

$$\dfrac{\mathtt{fieldsT}(\mathsf{C}) = \overrightarrow{\mathsf{T}}\,\overrightarrow{\mathsf{f}} \quad \Delta \vdash^{*}_{\mathcal{T}} \bar{\mathsf{t}} : \bar{\mathsf{T}}}{\Delta \vdash_{\mathcal{T}} \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{t}}) : \mathsf{C}}\,[\text{T-NEW}]$$

$$\dfrac{\mathtt{A\text{-}mtypeT}(\varphi) = \overrightarrow{\mathsf{T}} \to \mathsf{T} \quad \Delta, \overrightarrow{\mathsf{y}} : \overrightarrow{\mathsf{T}} \vdash^{*}_{\mathcal{T}} \mathsf{t} : \mathsf{T}}{\Delta \vdash_{\mathcal{T}} (\overrightarrow{\mathsf{y}} \to \mathsf{t})^{\varphi} : \varphi}\,[\text{T-}\lambda]$$

$$\dfrac{\Delta \vdash_{\mathcal{T}} (\mathsf{t}_\lambda)^{\varphi} : \varphi}{\Delta \vdash^{*}_{\mathcal{T}} \mathsf{t}_\lambda : \varphi}\,[\text{T-*-}\lambda] \qquad \dfrac{\Delta \vdash_{\mathcal{T}} \mathsf{t} : \sigma \quad \mathsf{t} \neq \mathsf{t}_\lambda \quad \sigma <: \tau}{\Delta \vdash^{*}_{\mathcal{T}} \mathsf{t} : \tau}\,[\text{T-*-N}]$$

$$\dfrac{\Delta \vdash^{*}_{\mathcal{T}} \mathsf{t} : \tau}{\Delta \vdash_{\mathcal{T}} (\tau)\,\mathsf{t} : \tau}\,[\text{T-UC}] \qquad \dfrac{\Delta \vdash_{\mathcal{T}} \mathsf{t} : \mathsf{T}}{\Delta \vdash_{\mathcal{T}} (\mathsf{T}[\&\iota])\,\mathsf{t} : \mathsf{T}[\&\iota]}\,[\text{T-DC}]$$

Figure 2 Term Typing Rules of TJ&.

λ-expression. Rule [T-DC] is a restricted form of the standard typing rule for downcast. Indeed, it represents exactly the form of the downcasts that we will introduce in a term of SJ&+ when translating it into the target language TJ&. Then an immediate reason for including this rule is that translated terms must be typed. Conversely, by omitting a general downcast rule, we can focus only on the downcasts that are the crucial modification of a term during translation. Concerning the judgment $\vdash^*$, it has a different meaning for decorated λ-expressions and other terms. Rule [T-*-λ] states that we derive a type for a pure λ-expression only by checking that the decorated λ-expression is typed. Instead, for a term which is not a λ-expression, rule [T-*-N] makes subsumption explicit, going from $\vdash$ to $\vdash^*$. The utility of the judgment $\vdash^*$ consists in simplifying the formulation of rules [T-INVK] and [T-NEW]. Rule [T-UC] is shorter than usual, by taking advantage from the judgment $\vdash^*$.

$$\dfrac{\mathtt{mtypeT}(\mathsf{m};\mathsf{T}) = \overrightarrow{\mathsf{T}} \to \mathsf{T}' \quad \overrightarrow{\mathsf{x}} : \overrightarrow{\mathsf{T}}, \mathsf{this} : \mathsf{T} \vdash^{*}_{\mathcal{T}} \mathsf{t} : \mathsf{T}'}{\mathsf{T}'\mathsf{m}(\overrightarrow{\mathsf{T}}\,\overrightarrow{\mathsf{x}})\{\mathsf{return}\ \mathsf{t};\}\ \text{T-}\mathit{OK}\ \text{in}\ \mathsf{T}}\,[\mathsf{M}\ \text{T-}\mathit{OK}\ \text{in}\ \mathsf{T}]$$

$$\dfrac{\begin{array}{c}\mathsf{K}^{\mathcal{T}} = \mathsf{C}(\overrightarrow{\mathsf{U}}\,\overrightarrow{\mathsf{g}}, \overrightarrow{\mathsf{T}}\,\overrightarrow{\mathsf{f}})\{\mathsf{super}(\overrightarrow{\mathsf{g}}); \mathsf{this}.\bar{\mathsf{f}} = \bar{\mathsf{f}};\} \quad \mathtt{fieldsT}(\mathsf{D}) = \overrightarrow{\mathsf{U}}\,\overrightarrow{\mathsf{g}} \quad \overline{\mathsf{M}^{\mathcal{T}}}\ \text{T-}\mathit{OK}\ \text{in}\ \mathsf{C} \\ \mathtt{mtypeT}(\mathsf{m};\mathsf{C})\ \text{defined implies}\ \mathtt{mbodyT}(\mathsf{m};\mathsf{C})\ \text{defined}\end{array}}{\mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{D}\ \mathsf{implements}\ \overrightarrow{\mathsf{I}}\ \{\overline{\mathsf{T}}\,\bar{\mathsf{f}}; \mathsf{K}^{\mathcal{T}}\ \overline{\mathsf{M}^{\mathcal{T}}}\}\ \mathit{OK}}\,[\mathsf{C}\ \text{T-}\mathit{OK}]$$

$$\dfrac{\overline{\mathsf{M}^{\mathcal{T}}}\ \text{T-}\mathit{OK}\ \text{in}\ \mathsf{I}}{\mathsf{interface}\ \mathsf{I}\ \mathsf{extends}\ \overrightarrow{\mathsf{I}}\ \{\overline{\mathsf{H}^{\mathcal{T}}}; \overline{\mathsf{M}^{\mathcal{T}}}\}\ \mathit{OK}}\,[\mathsf{I}\ \text{T-}\mathit{OK}]$$

Figure 3 Method, Class and Interface Declaration Typing Rules of TJ&.

Moreover, a feature of [T-UC] is the possibility of obtaining a judgment $\vdash$ from a judgment $\vdash^*$. Notice that, if a closed term is typed in $\vdash$, then the type derivation is unique. This is clearly false for $\vdash^*$.

If the body of a typed method contains a λ-expression, then the λ-expression may contain the formal parameters of the enclosing method, which are effectively final variables, as prescribed in [14] (page 607). Moreover, no other final variable from the enclosing environment can occur in this λ-expression, since we are in a purely functional model without assignments.

Typing statements for methods, classes and interfaces are checked by the rules in Figure 3: they say that a method is well formed in a class, a class declaration is well formed and an interface declaration is well formed, respectively. In rule [M T-$\mathtt{OK}$ in T] we omit the standard condition on soundness for overriding, see [20] (Figure 19-2). A class table is well formed if all class and interface declarations are well formed.

A *program* is a pair $(CT^{\mathcal{T}}, \mathsf{t})$ of a class table $CT^{\mathcal{T}}$ and a closed term t. We say that the program is *typed* if $CT^{\mathcal{T}}$ is well formed and t is typed by using $CT^{\mathcal{T}}$.

Finally, this calculus enjoys Subject Reduction, thanks to the above restriction of the downcast rule. Moreover, a program without downcasts has Progress.

▶ **Theorem 1** (Subject Reduction and Progress).

1. *If* $\Delta \vdash_{\mathcal{T}} \mathsf{t} : \tau$ *and* $\mathsf{t} \longrightarrow_{\mathcal{T}} \mathsf{t}'$*, then* $\Delta \vdash_{\mathcal{T}} \mathsf{t} : \sigma$ *for some* $\sigma <: \tau$.
2. *If* $(CT^{\mathcal{T}}, \mathsf{t})$ *is typed without using rule* [T-DC]*, then* t *either is a proper value or reduces.*

Both properties are proved in [2].

4 Java with Deconfined Intersection Types (SJ&+)

In this Sections we extend TJ& with a new crucial feature: intersection types are first class types, that is they can be used everywhere a type is expected. Thus intersection types are allowed to appear as types of fields and as return and parameter types of methods, rather than being confined within a type cast as in TJ&. This extension is formalised by the *source calculus* SJ&+. Terms in SJ&+ are defined as terms in TJ&. Instead, the extended use of intersection types requires the following new definitions:

$$\frac{\mathtt{fieldsS}(\mathsf{C}) = \vec{\tau}\ \vec{\mathsf{f}}}{\mathsf{new\ C}(\vec{\mathsf{v}}).\mathsf{f}_j \longrightarrow_{S} (\mathsf{v}_j)^{?\tau_j}}\ \text{[S-ProjNew]}$$

$$\frac{\mathtt{mbodyS}(\mathsf{m};\mathsf{C}) = (\vec{\mathsf{x}}, \mathsf{t}) \quad \mathtt{mtypeS}(\mathsf{m};\mathsf{C}) = \vec{\tau} \to \tau}{\mathsf{new\ C}(\vec{\mathsf{v}}).\mathsf{m}(\vec{\mathsf{u}}) \longrightarrow_{S} [\vec{\mathsf{x}} \mapsto (\vec{\mathsf{u}})^{?\vec{\tau}}, \mathsf{this} \mapsto \mathsf{new\ C}(\vec{\mathsf{v}})](\mathsf{t})^{?\tau}}\ \text{[S-InvkNew]}$$

$$\frac{\mathtt{A\text{-}nameS}(\varphi) = \mathsf{m} \quad \mathtt{A\text{-}mtypeS}(\varphi) = \vec{\tau} \to \tau}{(\vec{\mathsf{y}} \to \mathsf{t})^{\varphi}.\mathsf{m}(\vec{\mathsf{v}}) \longrightarrow_{S} [\vec{\mathsf{y}} \mapsto (\vec{\mathsf{v}})^{?\vec{\tau}}](\mathsf{t})^{?\tau}}\ \text{[S-Invk}\lambda\text{-A]}$$

$$\frac{\mathtt{mbodyS}(\mathsf{m};\varphi) = (\vec{\mathsf{x}}, \mathsf{t}) \quad \mathtt{mtypeS}(\mathsf{m};\varphi) = \vec{\tau} \to \tau}{(\mathsf{t}_\lambda)^{\varphi}.\mathsf{m}(\vec{\mathsf{v}}) \longrightarrow_{S} [\vec{\mathsf{x}} \mapsto (\vec{\mathsf{v}})^{?\vec{\tau}}, \mathsf{this} \mapsto (\mathsf{t}_\lambda)^{\varphi}](\mathsf{t})^{?\tau}}\ \text{[S-Invk}\lambda\text{-D]}$$

Figure 4 Reduction Rules of SJ&+: rules [S-CastNew], [S-Cλ], [S-CCλ] and [S-Ctx] are omitted.

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

$$\frac{\Gamma \vdash_S \mathsf{t} : \tau \quad \mathtt{mtypeS}(\mathsf{m}; \tau) = \vec{\sigma} \to \sigma \quad \Gamma \vdash^*_S \bar{\mathsf{t}} : \bar{\sigma}}{\Gamma \vdash_S \mathsf{t.m}(\vec{\mathsf{t}}) : \sigma}\,[\text{S-INVK}]$$

$$\frac{\mathtt{fieldsS}(\mathsf{C}) = \vec{\tau}\,\vec{\mathsf{f}} \quad \Gamma \vdash^*_S \bar{\mathsf{t}} : \bar{\tau}}{\Gamma \vdash_S \mathsf{new}\,\mathsf{C}(\vec{\mathsf{t}}) : \mathsf{C}}\,[\text{S-NEW}]$$

$$\frac{\mathtt{A\text{-}mtypeS}(\varphi) = \vec{\tau} \to \tau \text{ implies } \Gamma, \vec{\mathsf{y}} : \vec{\tau} \vdash^*_S \mathsf{t} : \tau}{\Gamma \vdash_S (\vec{\mathsf{y}} \to \mathsf{t})^{\varphi} : \varphi}\,[\text{S-}\lambda]$$

Figure 5 Term Typing Rules of SJ&+: rules [S-*-λ], [S-*-N] and [S-UC] are omitted.

$$\frac{\mathtt{mtypeS}(\mathsf{m}; \mathsf{T}) = \vec{\tau} \to \tau \quad \vec{\mathsf{x}} : \vec{\tau}, \mathsf{this} : \mathsf{T} \vdash^*_S \mathsf{t} : \tau}{\tau\mathsf{m}(\vec{\tau}\,\vec{\mathsf{x}})\{\mathsf{return\ t};\}\ \text{S-}\mathit{OK}\ \text{in}\ \mathsf{T}}\,[\mathsf{M}\ \text{S-}\mathit{OK}\ \text{in}\ \mathsf{T}]$$

$$\frac{\begin{array}{c}\mathsf{K}^S = \mathsf{C}(\vec{\sigma}\,\vec{\mathsf{g}}, \vec{\tau}\,\vec{\mathsf{f}})\{\mathsf{super}(\vec{\mathsf{g}}); \mathsf{this}.\bar{\mathsf{f}} = \bar{\mathsf{f}};\} \quad \mathtt{fieldsS}(\mathsf{D}) = \vec{\sigma}\,\vec{\mathsf{g}} \quad \overline{\mathsf{M}^S}\ \text{S-}\mathit{OK}\ \text{in}\ \mathsf{C} \\ \mathtt{mtypeS}(\mathsf{m}; \mathsf{C}) \text{ defined implies } \mathtt{mbodyS}(\mathsf{m}; \mathsf{C}) \text{ defined}\end{array}}{\mathsf{class\ C\ extends\ D\ implements}\ \vec{\mathsf{I}}\ \{\bar{\tau}\,\bar{\mathsf{f}}; \mathsf{K}^S\ \overline{\mathsf{M}^S}\}\ \mathit{OK}}\,[\mathsf{C}\ \text{S-}\mathit{OK}]$$

Figure 6 Method and Class Declaration Typing Rules of SJ&+: rule [I S-*OK*] is omitted.

$$\begin{array}{lcl}
\mathsf{CD}^S & ::= & \mathsf{class\ C\ extends\ D\ implements}\ \vec{\mathsf{I}}\ \{\bar{\tau}\,\bar{\mathsf{f}}; \mathsf{K}^S\ \overline{\mathsf{M}^S}\} \\
\mathsf{ID}^S & ::= & \mathsf{interface\ I\ extends}\ \vec{\mathsf{I}}\ \{\overline{\mathsf{H}^S}; \overline{\mathsf{M}^S}\} \\
\mathsf{K}^S & ::= & \mathsf{C}(\vec{\tau}\,\vec{\mathsf{f}})\{\mathsf{super}(\vec{\mathsf{f}}); \mathsf{this}.\bar{\mathsf{f}} = \bar{\mathsf{f}};\} \\
\mathsf{H}^S & ::= & \tau\mathsf{m}(\vec{\tau}\,\vec{\mathsf{x}}) \\
\mathsf{M}^S & ::= & \mathsf{H}^S\,\{\mathsf{return\ t};\}
\end{array}$$

Lookup functions for a given class table in SJ&+ are defined as in TJ&. To distinguish the two calculi, the suffix or prefix `S` replaces `T` in the arrow denoting reduction, in the derivation symbol, and in the labels of the rules. Figures 4, 5 and 6 show the rules which are different from the corresponding rules in Figures 1, 2 and 3, respectively. The other rules are unchanged, namely:

- the reduction rules [S-CastNew], [S-Cλ], [S-CCλ] and [S-Ctx] must be added to Figure 4;
- the typing rules [S-*-λ], [S-*-N] and [S-UC] must be added to Figure 5;
- the rule [I S-*OK*] must be added to Figure 6.

As usual, $\longrightarrow^*_S$ is the reflexive and transitive closure of $\longrightarrow_S$.

We observe that SJ&+ does not have a downcast rule. The reason of this choice is the same which justifies the restricted form of downcast in the typing rule of TJ&. When translating a term of SJ&+ without downcasts, we have the property that the downcasts in the translated term are those and only those introduced by the translation. In this case, the proofs of properties concerning the translation become more concise and compact.

As in TJ&, a *program* of SJ&+ is a pair (CT^S, t) of a class table CT^S and a closed term t. We say that the program is *typed* if CT^S is well formed and t is typed by using CT^S.

Finally, both Subject Reduction and Progress hold. Since SJ&+ does not have any downcast typing rule, Progress requires no conditions.

▶ **Theorem 2** (Subject Reduction and Progress)**.**

1. *If* $\Gamma \vdash_S \mathsf{t} : \tau$ *and* $\mathsf{t} \longrightarrow_S \mathsf{t}'$*, then* $\Gamma \vdash_S \mathsf{t} : \sigma$ *for some* $\sigma <: \tau$*.*
2. *If* (CT^S, t) *is typed, then* t *either is a proper value or reduces.*

Both properties are proved in [10] for a conservative extension of SJ&+.

5 Translation

In this section we want to translate a typed program (CT^S, t) in the source calculus SJ&+ into a program (CT^T, t') in the target calculus TJ&.

To lighten definitions, we adopt the following convention for writing components of intersections in a given order: if ι is a functional type such that $\iota = \mathsf{I}\&\iota'$, then I is a functional interface.

We define the erasure on intersection types as the *erasure* mapping:

$$|\mathsf{T}[\&\iota]| = \mathsf{T}$$

This mapping replaces an intersection type with its first component, a class or an interface. Observe that the above convention on the order in intersection types ensures that the mapping of a functional type is a functional type too.

Now we have to define the translation of a well formed class table CT^S into a corresponding CT^T. We give a preliminary informal discussion about the main underlying ideas of the proposed translation technique.

The initial step must be the erasure of type intersections which are (i) parameter and return types in signatures of methods (ii) types of fields in class declarations. Then the crucial matter becomes the translation of method bodies, which have to recover the type information lost by erasure, bearing in mind two different issues. First, we must ensure that typing is preserved under translation. Second, the behaviour of the translated term must mimic the behaviour of the original term.

To resolve the first issue, we enclose any method call in a type cast to the original return type, since this return type has been erased in the translated signature. Analogously, we insert type casts on field selections and variable occurrences. To address the second issue, we take into account that λ-expressions need to be decorated by their target types to define their behaviour. However, a pure λ-expression can only get its target type from the enclosing context, namely:

- the target type of a λ-expression that occurs as an actual parameter of a constructor call is the type of the field in the class declaration;
- the target type of a λ-expression that occurs as an actual parameter of a method call is the type of the parameter in the method declaration;
- the target type of a λ-expression that occurs as a return term of a method is the result type in the method declaration;
- the target type of a λ-expression that occurs as the argument of a type cast is the type of the cast.

$$\left\llparenthesis \frac{\mathsf{x} : \tau \in \Gamma}{\Gamma \vdash_S \mathsf{x} : \tau}\,[\text{S-VAR}] \right\rrparenthesis = (\tau)\,\mathsf{x}$$

$$\left\llparenthesis \frac{\mathcal{D} :: \Gamma \vdash_S \mathsf{t} : \mathsf{C}[\&\iota] \quad \tau\,\mathsf{f} \in \mathtt{fieldsS}(\mathsf{C})}{\Gamma \vdash_S \mathsf{t.f} : \tau}\,[\text{S-FIELD}] \right\rrparenthesis = (\tau)\,(\llparenthesis \mathcal{D} \rrparenthesis.\mathsf{f})$$

$$\left\llparenthesis \frac{\mathcal{D} :: \Gamma \vdash_S \mathsf{t} : \tau \quad \mathtt{mtypeS}(\mathsf{m};\tau) = \overrightarrow{\sigma} \to \sigma \quad \overline{\mathcal{D}} :: \Gamma \vdash^*_S \overline{\mathsf{t}} : \overline{\sigma}}{\Gamma \vdash_S \mathsf{t.m}(\overrightarrow{\mathsf{t}}) : \sigma}\,[\text{S-INVK}] \right\rrparenthesis = (\sigma)\,(\llparenthesis \mathcal{D} \rrparenthesis.\mathsf{m}(\overrightarrow{\llparenthesis \mathcal{D} \rrparenthesis}))$$

$$\left\llparenthesis \frac{\mathtt{fieldsS}(\mathsf{C}) = \overrightarrow{\tau}\,\overrightarrow{\mathsf{f}} \quad \overline{\mathcal{D}} :: \Gamma \vdash^*_S \overline{\mathsf{t}} : \overline{\tau}}{\Gamma \vdash_S \mathsf{new\,C}(\overrightarrow{\mathsf{t}}) : \mathsf{C}}\,[\text{S-NEW}] \right\rrparenthesis = \mathsf{new\,C}(\overrightarrow{\llparenthesis \mathcal{D} \rrparenthesis})$$

$$\left\llparenthesis \frac{\mathtt{A\text{-}mtypeS}(\varphi) = \overrightarrow{\tau} \to \tau \quad \mathcal{D} :: \Gamma, \overrightarrow{\mathsf{y}} : \overrightarrow{\tau} \vdash^*_S \mathsf{t} : \tau}{\Gamma \vdash_S (\overrightarrow{\mathsf{y}} \to \mathsf{t})^\varphi : \varphi}\,[\text{S-}\lambda] \right\rrparenthesis = (\overrightarrow{\mathsf{y}} \to \llparenthesis \mathcal{D} \rrparenthesis)^\varphi$$

$$\left\llparenthesis \frac{\mathcal{D} :: \Gamma \vdash_S (\mathsf{t}_\lambda)^\varphi : \varphi}{\Gamma \vdash^*_S \mathsf{t}_\lambda : \varphi}\,[\text{S-*-}\lambda] \right\rrparenthesis = (\varphi)\mathsf{t}'_\lambda \quad \text{if } \llparenthesis \mathcal{D} \rrparenthesis = (\mathsf{t}'_\lambda)^\varphi$$

$$\left\llparenthesis \frac{\mathcal{D} :: \Gamma \vdash_S \mathsf{t} : \sigma \quad \mathsf{t} \neq \mathsf{t}_\lambda \quad \sigma <: \tau}{\Gamma \vdash^*_S \mathsf{t} : \tau}\,[\text{S-*-N}] \right\rrparenthesis = \llparenthesis \mathcal{D} \rrparenthesis$$

$$\left\llparenthesis \frac{\mathcal{D} :: \Gamma \vdash^*_S \mathsf{t} : \tau}{\Gamma \vdash_S (\tau)\,\mathsf{t} : \tau}\,[\text{S-UC}] \right\rrparenthesis = (\tau)\,\llparenthesis \mathcal{D} \rrparenthesis$$

Figure 7 Term Translation.

In the first three cases above, the erasure of method signatures and field types requires to recover the original target types. Thus the translation algorithm can add type casts to pure λ-expressions to preserve their target types. However, a crucial question arises: where do we find the target type of each occurrence of a λ-expression in the original term? The target type is included not in the syntactic structure of the term but in its typing. This motivates our formulation of translation from (typed) terms to terms, that is defined as a mapping from the type derivation of a term in SJ&+ into a term in TJ&.

Going into formalities, the translation of the program is based on three mappings.

The first mapping is the erasure mapping already defined.

The second mapping, dubbed $\llparenthesis\ \rrparenthesis$, applied to a type derivation for a term in SJ&+ gives a term of TJ&. The translation is defined by induction on derivations considering the last rule applied, as shown in Figure 7. The type derivation is used to cast λ-expressions to their target types in the source term. We convene that $\mathcal{D}$ ranges over derivations, i.e. $\mathcal{D} :: \Gamma \vdash_S \mathsf{t} : \tau$ means a derivation with conclusion $\Gamma \vdash_S \mathsf{t} : \tau$, and similarly for $\vdash^*_S$. We denote by $\llparenthesis \mathcal{D} \rrparenthesis$ the result of the translation of $\mathcal{D} :: \Gamma \vdash_S \mathsf{t} : \tau$ or $\mathcal{D} :: \Gamma \vdash^*_S \mathsf{t} : \tau$. With an abuse of language, when t is closed and typed we define $\llparenthesis \mathsf{t} \rrparenthesis = \llparenthesis \mathcal{D} \rrparenthesis$, where $\mathcal{D}$ is the unique derivation in $\vdash_S$ such that t is the subject of the conclusion and the typing context is empty. Notice that the condition for the translation of rule [S-*-λ] is always satisfied, since the judgment $\Gamma \vdash_S (\mathsf{t}_\lambda)^\varphi : \varphi$ can only be the conclusion of rule [S-λ]. We point out that decorated λ-expressions are produced only by translating decorated λ-expressions in the source term. Therefore a program of SJ&+ is compiled to a program in TJ& which is Java code.

The third mapping, dubbed $[\![\ ]\!]$, has as arguments and results declarations in class tables of the two calculi, respectively. We start by translating method headers and constructors, which only requires mapping intersections to nominal types.

▶ **Definition 3** (Translation of Headers and Constructors). *We define*

$$[\![\tau\,\mathsf{m}(\overrightarrow{\tau}\,\overrightarrow{\mathsf{x}})]\!] = |\tau|\,\mathsf{m}(\overrightarrow{|\tau|}\,\overrightarrow{\mathsf{x}})$$
$$[\![\mathsf{C}(\overrightarrow{\sigma}\,\overrightarrow{\mathsf{g}},\overrightarrow{\tau}\,\overrightarrow{\mathsf{f}})\{\mathsf{super}(\overrightarrow{\mathsf{g}});\mathsf{this}.\overline{\mathsf{f}}=\overline{\mathsf{f}};\}]\!] = \mathsf{C}(\overrightarrow{|\sigma|}\,\overrightarrow{\mathsf{g}},\overrightarrow{|\tau|}\,\overrightarrow{\mathsf{f}})\{\mathsf{super}(\overrightarrow{\mathsf{g}});\mathsf{this}.\overline{\mathsf{f}}=\overline{\mathsf{f}};\}$$

The translation of methods requires the translation of bodies, which being terms need type derivations depending on the class or the interface in which they are defined. For this reason the translation of methods is parametrised on nominal types. We use $[\![\ ]\!]^{\mathsf{T}}$ to denote the dependency on type T.

▶ **Definition 4** (Translation of Methods). *The translation of a method in a class or interface* T *is defined by:*

$$[\![\tau\,\mathsf{m}(\overrightarrow{\tau}\,\overrightarrow{\mathsf{x}})\{\mathsf{return}\ \ \mathsf{t};\}]\!]^{\mathsf{T}} = |\tau|\,\mathsf{m}(\overrightarrow{|\tau|}\,\overrightarrow{\mathsf{x}})\{\mathsf{return}\ \ (\![\mathcal{D}]\!);\}$$

where $\mathcal{D} :: \mathsf{x} : \overrightarrow{\tau}, \mathsf{this} : \mathsf{T} \vdash^*_S \mathsf{t} : \tau$.

We are now able to define the application of $[\![\ ]\!]$ to class and interface declarations.

▶ **Definition 5** (Translation of Class/Interface Declarations). *We define*

$$[\![\mathsf{class\,C\,extends\,D\,implements}\ \overrightarrow{\mathsf{I}}\ \{\overline{\tau}\,\overline{\mathsf{f}};\mathsf{K}^S\,\overline{\mathsf{M}^S}\}]\!] = \mathsf{class\,C\,extends\,D\,implements}\ \overrightarrow{\mathsf{I}}\ \{\overline{|\tau|}\,\overline{\mathsf{f}};[\![\mathsf{K}^S]\!]\,\overline{[\![\mathsf{M}^S]\!]^{\mathsf{C}}}\}$$
$$[\![\mathsf{interface\,I\,extends}\ \overrightarrow{\mathsf{I}}\ \{\overline{\mathsf{H}^S};\overline{\mathsf{M}^S}\}]\!] = \mathsf{interface\,I\,extends}\ \overrightarrow{\mathsf{I}}\ \{\overline{[\![\mathsf{H}^S]\!]};\overline{[\![\mathsf{M}^S]\!]^{\mathsf{I}}}\}$$

▶ **Definition 6** (Translation of Typed Programs). *Let* (CT^S, t) *be a typed program. Its translation is the program* $(CT^T, (\![\mathsf{t}]\!])$ *such that:*

$$\text{if } CT^S(\mathsf{C}) = \mathsf{CD}^S, \text{ then } CT^T(\mathsf{C}) = [\![\mathsf{CD}^S]\!]$$
$$\text{if } CT^S(\mathsf{I}) = \mathsf{ID}^S, \text{ then } CT^T(\mathsf{I}) = [\![\mathsf{ID}^S]\!]$$

Notice that $(\![\mathsf{t}]\!])$ is well defined since t is closed and typed.

▶ **Example 7.** Consider the program (CT^S, t) where $\mathsf{t} = \mathsf{new\,C(\ ).m(x' \to x')}$ and CT^S is the class table on the left-side of Figure 8. Its translation is (CT^T, t'), where

$$\mathsf{t}' = (\mathsf{I}_1\&\mathsf{I}_2)\,(\mathsf{new\,C(\ ).m}((\mathsf{I}_1\&\mathsf{I}_2)\,(\mathsf{x'} \to (\mathsf{I}_1\&\mathsf{I}_2)\,\mathsf{x'})))$$

and CT^T is the class table on the right-side of Figure 8.

```
interface I1 { I1&I2 m(I1&I2 x); }
interface I2 { I1&I2 n(I1&I2 y){return y; } }
class C extends Object implements I1, I2 {
   C( ) { super( ); }
   I1&I2 m(I1&I2 x){
      return x.n(z → z);
   }
}
```

SJ&+ Class Table

```
interface I1 { I1 m(I1 x); }
interface I2 { I1 n(I1 y){return (I1&I2)y; } }
class C extends Object implements I1, I2 {
   C( ) { super( ); }
   I1 m(I1 x){
      return (I1&I2)((I1&I2)x.n((I1&I2)(z → (I1&I2)z)));
   }
}
```

TJ& Class Table

Figure 8 Class Tables of Example 7.

It is interesting to compare the reduction of t:

$$\begin{aligned} \mathsf{t} &\longrightarrow_S (\mathsf{x'} \to \mathsf{x'})^{\mathsf{I}_1\&\mathsf{I}_2}.\mathsf{n}(\mathsf{z} \to \mathsf{z}) \\ &\longrightarrow_S (\mathsf{z} \to \mathsf{z})^{\mathsf{I}_1\&\mathsf{I}_2} \end{aligned}$$

with the reduction of t′:

$$\begin{aligned} \mathsf{t}' &\longrightarrow_T (\mathsf{I}_1\&\mathsf{I}_2)\,(\mathsf{new\,C(\,).m}((\mathsf{x'} \to (\mathsf{I}_1\&\mathsf{I}_2)\,\mathsf{x'})^{\mathsf{I}_1\&\mathsf{I}_2})) \\ &\longrightarrow_T (\mathsf{I}_1\&\mathsf{I}_2)\,((\mathsf{I}_1\&\mathsf{I}_2)((\mathsf{I}_1\&\mathsf{I}_2)((\mathsf{x'} \to (\mathsf{I}_1\&\mathsf{I}_2)\,\mathsf{x'})^{\mathsf{I}_1\&\mathsf{I}_2}).\mathsf{n}((\mathsf{I}_1\&\mathsf{I}_2)(\mathsf{z} \to (\mathsf{I}_1\&\mathsf{I}_2)\mathsf{z})))) \\ &\longrightarrow_T (\mathsf{I}_1\&\mathsf{I}_2)\,((\mathsf{I}_1\&\mathsf{I}_2)((\mathsf{I}_1\&\mathsf{I}_2)((\mathsf{x'} \to (\mathsf{I}_1\&\mathsf{I}_2)\,\mathsf{x'})^{\mathsf{I}_1\&\mathsf{I}_2}).\mathsf{n}((\mathsf{z} \to (\mathsf{I}_1\&\mathsf{I}_2)\mathsf{z})^{\mathsf{I}_1\&\mathsf{I}_2}))) \\ &\longrightarrow_T (\mathsf{I}_1\&\mathsf{I}_2)\,((\mathsf{I}_1\&\mathsf{I}_2)((\mathsf{I}_1\&\mathsf{I}_2)((\mathsf{I}_1\&\mathsf{I}_2)\,(\mathsf{z} \to (\mathsf{I}_1\&\mathsf{I}_2)\mathsf{z})^{\mathsf{I}_1\&\mathsf{I}_2}))) \\ &\longrightarrow_T^* (\mathsf{z} \to (\mathsf{I}_1\&\mathsf{I}_2)\mathsf{z})^{\mathsf{I}_1\&\mathsf{I}_2} \end{aligned}$$

where in the last steps we only apply rules [T-CCλ] and [T-Ctx]. The difference between the obtained values $(\mathsf{z} \to \mathsf{z})^{\mathsf{I}_1\&\mathsf{I}_2}$ and $(\mathsf{z} \to (\mathsf{I}_1\&\mathsf{I}_2)\mathsf{z})^{\mathsf{I}_1\&\mathsf{I}_2}$ is the cast of z to $\mathsf{I}_1\&\mathsf{I}_2$. This is due to the difference between the bodies of method m in the class tables CT^S and CT^T. Notice that $(\mathsf{z} \to (\mathsf{I}_1\&\mathsf{I}_2)\mathsf{z})^{\mathsf{I}_1\&\mathsf{I}_2}$ is the translation of $(\mathsf{z} \to \mathsf{z})^{\mathsf{I}_1\&\mathsf{I}_2}$.

6 Translation Correctness

In this section we show that the translation preserves the static and dynamic semantics of programs. To this aim we prove that typed SJ&+ programs are translated into typed TJ& programs (Theorem 11) and that the original and the translated programs either produce values related by the translation or both have infinite computations (Theorem 16 and Corollary 17).

The proof of these results relies on two main features of the translation. The first is that any subterm is cast to the same type τ that was used for typing the subterm in the source code. The second insight is about pure λ-expressions, that become arguments of type casting in translated terms, and so can become decorated λ-expressions in their reducts.

In the following we consider a fixed typed program (CT^S, t) and its translation $(CT^T, (\![\mathsf{t}]\!))$ as defined in Section 5.

6.1 Typing is preserved under translation

The typability of programs obtained by translation is shown by first proving that the translated terms are typed (Lemma 9) and then that class and interface declarations are well formed (Lemma 10).

We start by stating the relations between the class table CT^S and its translation CT^T. These relations can be easily checked looking at the translations of method headers and constructors (Definition 3).

▶ **Lemma 8.**

1. *The nominal type* T *belongs to* CT^S *if and only if* T *belongs to* CT^T*.*
2. *The subtyping* $\mathsf{T} <: \mathsf{T}'$ *holds in* CT^S *if and only if* $\mathsf{T} <: \mathsf{T}'$ *holds in* CT^T*.*
3. $\mathtt{fieldsS(C)} = \overrightarrow{\tau}\,\overrightarrow{\mathsf{f}}$ *in* CT^S *if and only if* $\mathtt{fieldsT(C)} = \overrightarrow{|\tau|}\,\overrightarrow{\mathsf{f}}$ *in* CT^T*.*
4. $\mathtt{mtypeS}(\mathsf{m};\tau) = \overrightarrow{\sigma} \to \sigma$ *in* CT^S *if and only if* $\mathtt{mtypeT}(\mathsf{m};\tau) = \overrightarrow{|\sigma|} \to |\sigma|$ *in* CT^T*.*
5. $\mathtt{A\text{-}mtypeS}(\varphi) = \overrightarrow{\tau} \to \tau$ *in* CT^S *if and only if* $\mathtt{A\text{-}mtypeT}(\varphi) = \overrightarrow{|\tau|} \to |\tau|$ *in* CT^T*.*

Building on previous lemma we can show that the translation of a term t typed by τ in $\vdash_S$ gives a term t′ typed by τ in $\vdash_T$ and similarly for $\vdash_S^*$ and $\vdash_T^*$. Figure 9 shows the mapping $\{\![\;]\!\}$ between type derivations. We also need to translate typing contexts Γ. We

$$\left\{\!\!\left[\dfrac{\mathsf{x}:\tau\in\Gamma}{\Gamma\vdash_S \mathsf{x}:\tau}\,[\text{S-VAR}]\right]\!\!\right\} = \dfrac{\dfrac{\mathsf{x}:|\tau|\in|\Gamma|}{|\Gamma|\vdash_T \mathsf{x}:|\tau|}\,[\text{T-VAR}]}{|\Gamma|\vdash_T (\tau)\,\mathsf{x}:\tau}\,[\text{T-DC}]$$

$$\left\{\!\!\left[\dfrac{\mathcal{D}::\Gamma\vdash_S \mathsf{t}:\mathsf{C}[\&\iota]\quad \tau\,\mathsf{f}\in\mathtt{fieldsS}(\mathsf{C})}{\Gamma\vdash_S \mathsf{t.f}:\tau}\,[\text{S-FIELD}]\right]\!\!\right\} =$$
$$\dfrac{\dfrac{\{\![\mathcal{D}]\!\}::|\Gamma|\vdash_T \mathsf{t}':\mathsf{C}[\&\iota]\quad |\tau|\,\mathsf{f}\in\mathtt{fieldsT}(\mathsf{C})}{|\Gamma|\vdash_T \mathsf{t}'.\mathsf{f}:|\tau|}\,[\text{T-FIELD}]}{|\Gamma|\vdash_T (\tau)\,(\mathsf{t}'.\mathsf{f}):\tau}\,[\text{T-DC}]$$

$$\left\{\!\!\left[\dfrac{\mathcal{D}::\Gamma\vdash_S \mathsf{t}:\tau\quad \mathtt{mtypeS}(\mathsf{m};\tau)=\overrightarrow{\sigma}\to\sigma\quad \overline{\mathcal{D}}::\Gamma\vdash_S^* \bar{\mathsf{t}}:\overline{\sigma}}{\Gamma\vdash_S \mathsf{t.m}(\overrightarrow{\mathsf{t}}):\sigma}\,[\text{S-INVK}]\right]\!\!\right\} =$$
$$\dfrac{\dfrac{\{\![\mathcal{D}]\!\}::|\Gamma|\vdash_T \mathsf{t}':\tau\quad \mathtt{mtypeT}(\mathsf{m};\tau)=\overrightarrow{|\sigma|}\to|\sigma|\qquad \dfrac{\overline{\{\![\mathcal{D}]\!\}}:|\Gamma|\vdash_T^* \overline{\mathsf{t}'}:\overline{\sigma}\quad \overline{\sigma}<:|\overline{\sigma}|}{|\Gamma|\vdash_T^* \overline{\mathsf{t}'}:|\overline{\sigma}|}\,[\text{T-}{*}{*}\text{-N}]}{|\Gamma|\vdash_T \mathsf{t}'.\mathsf{m}(\overrightarrow{\mathsf{t}'}):|\sigma|}\,[\text{T-INVK}]}{|\Gamma|\vdash_T (\sigma)\,(\mathsf{t}'.\mathsf{m}(\overrightarrow{\mathsf{t}'})):\sigma}\,[\text{T-DC}]$$

$$\left\{\!\!\left[\dfrac{\mathtt{fieldsS}(\mathsf{C})=\overrightarrow{\tau}\,\overrightarrow{\mathsf{f}}\quad \overline{\mathcal{D}}::\Gamma\vdash_S^* \bar{\mathsf{t}}:\overline{\tau}}{\Gamma\vdash_S \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{t}}):\mathsf{C}}\,[\text{S-NEW}]\right]\!\!\right\} =$$
$$\dfrac{\mathtt{fieldsT}(\mathsf{C})=\overrightarrow{|\tau|}\,\overrightarrow{\mathsf{f}}\qquad \dfrac{\overline{\{\![\mathcal{D}]\!\}}:|\Gamma|\vdash_T^* \overline{\mathsf{t}'}:\overline{\tau}\quad \overline{\tau}<:|\overline{\tau}|}{|\Gamma|\vdash_T^* \overline{\mathsf{t}'}:|\overline{\tau}|}\,[\text{T-}{*}{*}\text{-N}]}{|\Gamma|\vdash_T \mathsf{new}\,\mathsf{C}(\overrightarrow{\mathsf{t}'}):\mathsf{C}}\,[\text{T-NEW}]$$

$$\left\{\!\!\left[\dfrac{\mathtt{A\text{-}mtypeS}(\varphi)=\overrightarrow{\tau}\to\tau\quad \mathcal{D}::\Gamma,\overrightarrow{\mathsf{y}}:\overrightarrow{\tau}\vdash_S^* \mathsf{t}:\tau}{\Gamma\vdash_S (\overrightarrow{\mathsf{y}}\to\mathsf{t})^{\varphi}:\varphi}\,[\text{S-}\lambda]\right]\!\!\right\} =$$
$$\dfrac{\mathtt{A\text{-}mtypeT}(\varphi)=\overrightarrow{|\tau|}\to|\tau|\qquad \dfrac{\{\![\mathcal{D}]\!\}::|\Gamma|,\overrightarrow{\mathsf{y}}:\overrightarrow{|\tau|}\vdash_T^* \mathsf{t}':\tau\quad \tau<:|\tau|}{|\Gamma|,\overrightarrow{\mathsf{y}}:\overrightarrow{|\tau|}\vdash_T^* \mathsf{t}':|\tau|}\,[\text{T-}{*}{*}\text{-N}]}{|\Gamma|\vdash_T (\overrightarrow{\mathsf{y}}\to\mathsf{t}')^{\varphi}:\varphi}\,[\text{T-}\lambda]$$

$$\left\{\!\!\left[\dfrac{\mathcal{D}::\Gamma\vdash_S (\mathsf{t}_\lambda)^{\varphi}:\varphi}{\Gamma\vdash_S^* \mathsf{t}_\lambda:\varphi}\,[\text{S-*-}\lambda]\right]\!\!\right\} = \dfrac{\dfrac{\dfrac{\{\![\mathcal{D}]\!\}::|\Gamma|\vdash_T (\mathsf{t}'_\lambda)^{\varphi}:\varphi}{|\Gamma|\vdash_T^* \mathsf{t}'_\lambda:\varphi}\,[\text{T-*-}\lambda]}{|\Gamma|\vdash_T (\varphi)\mathsf{t}'_\lambda:\varphi\quad \varphi<:\varphi}\,[\text{T-UC}]}{|\Gamma|\vdash_T^* (\varphi)\mathsf{t}'_\lambda:\varphi}\,[\text{T-}{*}\text{-N}]$$

$$\left\{\!\!\left[\dfrac{\mathcal{D}::\Gamma\vdash_S \mathsf{t}:\sigma\quad \mathsf{t}\neq\mathsf{t}_\lambda\quad \sigma<:\tau}{\Gamma\vdash_S^* \mathsf{t}:\tau}\,[\text{S-*-N}]\right]\!\!\right\} = \dfrac{\{\![\mathcal{D}]\!\}::|\Gamma|\vdash_T \mathsf{t}':\sigma\quad \mathsf{t}'\neq\mathsf{t}_\lambda\quad \sigma<:\tau}{|\Gamma|\vdash_T^* \mathsf{t}':\tau}\,[\text{T-*-N}]$$

$$\left\{\!\!\left[\dfrac{\mathcal{D}::\Gamma\vdash_S^* \mathsf{t}:\tau}{\Gamma\vdash_S (\tau)\,\mathsf{t}:\tau}\,[\text{S-UC}]\right]\!\!\right\} = \dfrac{\{\![\mathcal{D}]\!\}::|\Gamma|\vdash_T^* \mathsf{t}':\tau}{|\Gamma|\vdash_T (\tau)\,\mathsf{t}':\tau}\,[\text{T-UC}]$$

Figure 9 Mapping from derivations in SJ&+ to derivations in TJ&.

extend the mapping $|\ |$ to typing contexts in the expected way: $|\Gamma| = \{\mathsf{x} : |\tau| \mid x : \tau \in \Gamma\}$. Each derivation $\mathcal{D} :: \Gamma \vdash_S \mathsf{t} : \tau$ is translated to a derivation $\{[\mathcal{D}]\}$ of $|\Gamma| \vdash_T \mathsf{t}' : \tau$. This modification of the context implies the need to add a downcast of the variable from $|\tau|$ to τ, when translating the axiom. Analogously, we need to add a downcast when the last applied rule is [S-FIELD] or [S-INVK]. This is due to the relations between lookup functions in CT^S and CT^T, see Lemma 8(3) and (4).

In the translations of rules [S-INVK], [S-NEW] and [S-λ] it is handy to consider the following rule:

$$\frac{\Delta \vdash^*_T \mathsf{t} : \sigma \quad \mathsf{t} \neq \mathsf{t}_\lambda \quad \sigma <: \tau}{\Delta \vdash^*_T \mathsf{t} : \tau}\ \text{[T-**-N]}$$

This rule is admissible in the type systems of Figure 2 since, if $\mathsf{t} \neq \mathsf{t}_\lambda$, then the only way to derive $\Delta \vdash^*_T \mathsf{t} : \sigma$ is

$$\frac{\Delta \vdash_T \mathsf{t} : \sigma' \quad \mathsf{t} \neq \mathsf{t}_\lambda \quad \sigma' <: \sigma}{\Delta \vdash^*_T \mathsf{t} : \sigma}\ \text{[T-*-N]}$$

Therefore, being $<:$ transitive

$$\frac{\Delta \vdash_T \mathsf{t} : \sigma' \quad \mathsf{t} \neq \mathsf{t}_\lambda \quad \sigma' <: \tau}{\Delta \vdash^*_T \mathsf{t} : \tau}\ \text{[T-*-N]}$$

This rule does the subsumption needed for the relations between the lookup functions in CT^S and CT^T, see Lemma 8(3), (4) and (5).

Looking at rule [S-λ] we see that the translation of a type derivation for a decorated λ-expression is still a derivation for a decorated λ-expression. This is crucial for the translation of rule [S-*-λ], since $\mathcal{D} :: \Gamma \vdash_S (\mathsf{t}_\lambda)^\varphi : \varphi$ can only be the conclusion of rule [S-λ]. Then $(\![\mathcal{D}]\!) = (\mathsf{t}'_\lambda)^\varphi$ and we can use rule [T-*-λ] before doing the upcast.

The translation of rules [S-*-N] and [S-UC] is the identity. It works since thanks to Lemma 8(2) $\sigma <: \tau$ holds in CT^S if and only if $\sigma <: \tau$ holds in CT^T.

As expected, the subject in the conclusion of $\{[\mathcal{D}]\}$ is $(\![\mathcal{D}]\!)$: it can be easily checked by induction on derivations comparing the definitions of $\{[\ ]\}$ (Figure 9) and $(\![\ ]\!)$ (Figure 7). This implies that no t' in Figure 9 can be a pure λ-expression and so justifies the applications of rule [T-**-N].

To sum up we proved:

▶ **Lemma 9.**
1. *If* $\mathcal{D} :: \Gamma \vdash_S \mathsf{t} : \tau$*, then* $|\Gamma| \vdash_T (\![\mathcal{D}]\!) : \tau$.
2. *If* $\mathcal{D} :: \Gamma \vdash^*_S \mathsf{t} : \tau$*, then* $|\Gamma| \vdash^*_T (\![\mathcal{D}]\!) : \tau$.

We end this subsection by showing that the well-formedness of classes and interfaces in CT^S implies the well-formedness of their translations, i.e. the well-formedness of the obtained class table CT^T.

▶ **Lemma 10.**
1. *If the method declaration* M^S *is S-*`OK` *in the class or interface* T *for the class table* CT^S*, then its translation* $[\![\mathsf{M}^S]\!]^{\mathsf{T}}$ *is T-*`OK` *in* T *for the class table* CT^T.
2. *If the class declaration* CD^S *is S-*`OK` *for the class table* CT^S*, then its translation* $[\![\mathsf{CD}^S]\!]$ *is T-*`OK` *for the class table* CT^T.
3. *If the interface declaration* ID^S *is S-*`OK` *for the class table* CT^S*, then its translation* $[\![\mathsf{ID}^S]\!]$ *is T-*`OK` *for the class table* CT^T.

Proof. (1). Let $\mathsf{M}^S = \tau\mathsf{m}(\overrightarrow{\tau}\,\overrightarrow{\mathsf{x}})\{\mathsf{return\ t;}\}$, then, by rule [M S-*OK* in T] of Figure 6, $\mathtt{mtypeS}(\mathsf{m};\mathsf{T}) = \overrightarrow{\tau} \to \tau$ and $\mathcal{D} :: \overrightarrow{\mathsf{x}} : \overrightarrow{\tau}, \mathsf{this} : \mathsf{T} \vdash^*_S \mathsf{t} : \tau$ for some $\mathcal{D}$. Lemma 8(4) implies $\mathtt{mtypeT}(\mathsf{m};\mathsf{T}) = \overrightarrow{|\tau|} \to |\tau|$. We have $\overrightarrow{\mathsf{x}} : \overrightarrow{|\tau|}, \mathsf{this} : \mathsf{T} \vdash^*_T (\![\mathcal{D}]\!) : \tau$ by Lemma 9(2). By Definition 4 we get $[\![\mathsf{M}^S]\!]^\mathsf{T} = |\tau|\mathsf{m}(\overrightarrow{|\tau|}\,\overrightarrow{\mathsf{x}})\{\mathsf{return}\ (\![\mathcal{D}]\!);\}$. We can apply rule [M T-*OK* in T] of Figure 3.

(2). Let $\mathsf{CD}^S = \mathsf{class\,C\,extends\,D\,implements}\ \overrightarrow{\mathsf{I}}\ \{\overline{\tau}\,\overline{\mathsf{f}}; \mathsf{K}^S\,\overline{\mathsf{M}^S}\}$, then, by rule [C S-*OK*] of Figure 6, $\mathsf{K}^S = \mathsf{C}(\overrightarrow{\sigma}\,\overrightarrow{\mathsf{g}}, \overrightarrow{\tau}\,\overrightarrow{\mathsf{f}})\{\mathsf{super}(\overrightarrow{\mathsf{g}}); \mathsf{this}.\overline{\mathsf{f}} = \overline{\mathsf{f}};\}$ and $\mathtt{fieldsS}(\mathsf{D}) = \overrightarrow{\sigma}\,\overrightarrow{\mathsf{g}}$ and $\overline{\mathsf{M}^S}$ S-*OK* in C. By Definition 3 $[\![\mathsf{K}^S]\!] = \mathsf{C}(\overrightarrow{|\sigma|}\,\overrightarrow{\mathsf{g}}, \overrightarrow{|\tau|}\,\overrightarrow{\mathsf{f}})\{\mathsf{super}(\overrightarrow{\mathsf{g}}); \mathsf{this}.\overline{\mathsf{f}} = \overline{\mathsf{f}};\}$. Lemma 8(4) implies $\mathtt{fieldsS}(\mathsf{D}) = \overrightarrow{|\sigma|}\,\overrightarrow{\mathsf{g}}$. Point (1) gives $\overline{[\![\mathsf{M}^S]\!]^\mathsf{C}}$ T-*OK* in C. By Definition 5 $[\![\mathsf{CD}^S]\!] = \mathsf{class\,C\,extends\,D\,implements}\ \overrightarrow{\mathsf{I}}\ \{\overline{|\tau|}\,\overline{\mathsf{f}}; [\![\mathsf{K}^S]\!]\,\overline{[\![\mathsf{M}^S]\!]^\mathsf{C}}\}$. Therefore we can apply rule [C T-*OK*] of Figure 3.

The proof of Point (3) is similar and simpler than the proof of Point (2). ◀

Therefore, the final result is an immediate consequence of Lemmas 10 and 9(1).

▶ **Theorem 11.** *The translation of a typed program in* SJ&+ *is a typed program in* TJ&.

6.2 Semantics is preserved under translation

Given a typed program (CT^S, t) and its translation $(CT^T, (\![\mathsf{t}]\!))$ we want to state the relations between the reduction of t with $\longrightarrow_S$ and the reduction of $(\![\mathsf{t}]\!)$ with $\longrightarrow_T$.

Looking at Figure 7 it is clear that $(\![\mathsf{t}]\!)$ is obtained from t by adding casts. We prove that translated terms can be typed in $\vdash_S$. This implies that translated terms cannot be stuck and that either t and $(\![\mathsf{t}]\!)$ both reduce to values or both have infinite computations.

▶ **Lemma 12.**
1. *If* $\mathcal{D} :: \Gamma \vdash_S \mathsf{t} : \tau$, *then* $\Gamma \vdash_S (\![\mathcal{D}]\!) : \tau$.
2. *If* $\mathcal{D} :: \Gamma \vdash^*_S \mathsf{t} : \tau$, *then* $\Gamma \vdash^*_S (\![\mathcal{D}]\!) : \tau$.

Proof. The proof of (1) and (2) is by simultaneous induction on the construction of $(\![\mathcal{D}]\!)$ done in Figure 7. We only consider some interesting cases.

Rule [S-INVK].

$$\left(\!\!\left[\frac{\mathcal{D} :: \Gamma \vdash_S \mathsf{t} : \tau \quad \mathtt{mtypeS}(\mathsf{m};\tau) = \overrightarrow{\sigma} \to \sigma \quad \overline{\mathcal{D}} :: \Gamma \vdash^*_S \overline{\mathsf{t}} : \overline{\sigma}}{\Gamma \vdash_S \mathsf{t}.\mathsf{m}(\overrightarrow{\mathsf{t}}) : \sigma}\,[\text{S-INVK}]\right]\!\!\right) = (\sigma)\,((\![\mathcal{D}]\!).\mathsf{m}(\overrightarrow{(\![\mathcal{D}]\!)}))$$

By IH $\Gamma \vdash_S (\![\mathcal{D}]\!) : \tau$ and $\Gamma \vdash^*_S \overline{(\![\mathcal{D}]\!)} : \overline{\sigma}$. By applying [S-INVK] we derive $\Gamma \vdash_S (\![\mathcal{D}]\!).\mathsf{m}(\overrightarrow{(\![\mathcal{D}]\!)}) : \sigma$. Rule [S-*-N] gives $\Gamma \vdash^*_S (\![\mathcal{D}]\!).\mathsf{m}(\overrightarrow{(\![\mathcal{D}]\!)}) : \sigma$. We conclude $\Gamma \vdash_S (\sigma)\,((\![\mathcal{D}]\!).\mathsf{m}(\overrightarrow{(\![\mathcal{D}]\!)})) : \sigma$ using [S-UC].

Rule [S-*-λ].

$$\left(\!\!\left[\frac{\mathcal{D} :: \Gamma \vdash_S (\mathsf{t}_\lambda)^\varphi : \varphi}{\Gamma \vdash^*_S \mathsf{t}_\lambda : \varphi}\,[\text{S-*-}\lambda]\right]\!\!\right) = (\varphi)\mathsf{t}'_\lambda \qquad \text{if } (\![\mathcal{D}]\!) = (\mathsf{t}'_\lambda)^\varphi$$

By IH and the condition $(\![\mathcal{D}]\!) = (\mathsf{t}'_\lambda)^\varphi$ we get $\Gamma \vdash_S (\mathsf{t}'_\lambda)^\varphi : \varphi$. By applying [S-*-$\lambda$] we derive $\Gamma \vdash^*_S \mathsf{t}'_\lambda : \varphi$. Rule [S-UC] gives $\Gamma \vdash_S (\varphi)\,\mathsf{t}'_\lambda : \varphi$. We conclude $\Gamma \vdash^*_S (\varphi)\,\mathsf{t}'_\lambda : \varphi$ using [S-*-N]. ◀

In the remaining of this section we want to establish the relations between the reduction of a term in SJ&+, according to a given class table, and the reduction of its translation in TJ&, using the translated class table. This issue deserves a preliminary discussion, because of difficulties arising from the presence of λ-expressions. The value obtained computing a typed program (if any) is always a proper value, i.e., either an object or a λ-expression with its target type.

The first remark is that, in general, *the translation of a proper value is not a proper value.* For example, let us assume a class D that has a field of type I, where I is a functional interface with abstract method of type $\mathsf{C} \to \mathsf{C}$. Then we have $([\![\mathsf{new\,D(x \to x)}]\!]) = \mathsf{new\,D((I)\,(x \to (C)\,x))}$, where $\mathsf{new\,D((I)\,(x \to (C)\,x))}$ reduces to $\mathsf{new\,D((x \to (C)\,x)^I)}$. The pure λ-expression $\mathsf{x \to x}$ has type I in SJ&+, as provided by the class table, namely by the field type. As expected, $\mathsf{x \to x}$ is explicitly decorated by this target type in $\mathsf{new\,D((x \to (C)\,x)^I)}$ and its body is translated (as well as method bodies are translated in the class table).

The second remark is that $\mathsf{t} \longrightarrow_S \mathsf{t'}$ *does not imply* $([\mathsf{t}]) \longrightarrow_T^* \mathsf{t''}$ *with* $([\mathsf{t'}]) \longrightarrow_T^* \mathsf{t''}$, for some $\mathsf{t''}$. For example, let us assume a further class E without fields, containing a method m which has return type J, parameter y of type D and body $\mathsf{z \to y}$. Let t be the term $\mathsf{t = new\,E().m(new\,D(x \to x))}$. Then t reduces as follows:

$$\mathsf{new\,E().m(new\,D(x \to x))} \longrightarrow_S \mathsf{(z \to new\,D(x \to x))^J}$$

We consider the translation $([\mathsf{t}]) = \mathsf{(J)\,(new\,E().m(new\,D((I)\,(x \to (C)\,x))))}$ and its reduction:

$$\begin{array}{lll} ([\mathsf{t}]) & \longrightarrow_T & \mathsf{(J)\,(new\,E().m(new\,D((x \to (C)\,x)^I)))} \\ & \longrightarrow_T & \mathsf{(J)\,((J)\,(z \to (D)\,(new\,D((x \to (C)\,x)^I))))} \\ & \longrightarrow_T^* & \mathsf{(z \to (D)\,(new\,D((x \to (C)\,x)^I)))^J} \end{array}$$

It is clear that $([\mathsf{(z \to new\,D(x \to x))^J}]) = \mathsf{(z \to new\,D((I)\,(x \to (C)\,x)))^J}$ is a value, since the type cast is included in the body of the λ-expression and so no reduction rule applies. Therefore it does not reduce to $\mathsf{(z \to (D)\,(new\,D((x \to (C)\,x)^I)))^J}$.

Finally, we assume the method n in the class E, which has return type J, no parameters and body $\mathsf{z \to new\,D(x \to x)}$. Then we get

$$\mathsf{new\,E().n()} \longrightarrow_S \mathsf{(z \to new\,D(x \to x))^J}$$

and its translation so reduces

$$\begin{array}{lll} \mathsf{(J)\,(new\,E().n())} & \longrightarrow_T & \mathsf{(J)\,((J)\,(z \to new\,D((I)\,(x \to (C)\,x))))} \\ & \longrightarrow_T^* & \mathsf{(z \to new\,D((I)\,(x \to (C)\,x))^J} \end{array}$$

Comparing methods m and n, we observe that the body of m is a function returning the parameter, while the body of n is a function returning the expression $\mathsf{new\,D(x \to x)}$. Therefore, the evaluations of the two calls, $\mathsf{m(new\,D(x \to x))}$ and $\mathsf{n(\,)}$, return the same value, while their translations reduce to different values. This happens because, when calling m, the translation of the parameter $\mathsf{new\,D(x \to x)}$ is reduced before replacing it to the formal parameter in the λ-expression. Moreover the formal parameter is cast to D.

Notice that, in the above examples, there are no intersection types: this directs our attention to the fact that the casts, that are introduced by translation, are reduced only when they are in evaluation contexts.

This discussion suggests that *the relation between* $([\mathsf{t'}])$ *and* $\mathsf{t''}$ *when* $\mathsf{t} \longrightarrow_S^* \mathsf{t'}$ *and* $([\mathsf{t}]) \longrightarrow_T^* \mathsf{t''}$ *cannot be expressed as a function.* Intuitively, this relation is an equivalence that can be realised by:

- ignoring type casts on closed terms different from λ-expressions;
- identifying type casts of λ-expressions with the decorated λ-expressions obtained by reducing them.

▶ **Definition 13.** *The* cast-equivalence *relation* $\approx$ *on closed and typed terms of* SJ&+ *is the smallest congruence which satisfies:*

$$\frac{\vdash_S^* \mathsf{t} : \tau \quad \mathsf{t} \neq \mathsf{t}'_\lambda}{(\tau)\,\mathsf{t} \approx \mathsf{t}} \qquad\qquad (\varphi)\,\mathsf{t}_\lambda \approx (\mathsf{t}_\lambda)^\varphi$$

Notice that cast-equivalence allows to eliminate all casts introduced by the translation of closed and typed source terms. Lemma 12 assures that all these casts are upcasts when typing in SJ&+. It is easy to verify that cast-equivalence preserves typing in SJ&+.

In general $(\!|\mathsf{t}|\!) \approx \mathsf{t}$ does not hold. Let class D be as in previous example, then the translation of $\mathsf{new\,D(x \to x)}$ is $\mathsf{new\,D((I)\,(x \to (C)\,x))}$.

We would like to prove:

$$\text{if } \mathsf{t} \longrightarrow_S \mathsf{t}', \text{ then } (\!|\mathsf{t}|\!) \longrightarrow_T^* \mathsf{t}'' \text{ and } (\!|\mathsf{t}'|\!) \approx \mathsf{t}'' \text{ for some } \mathsf{t}''$$

Lemma 15 shows a more general formulation of this relation, which better fits the proof of Theorem 16.

We first show that terms cast-equivalent to translations of values reduce to cast-equivalent proper values using $\longrightarrow_T$.

▶ **Proposition 14.** *Let* v *be a closed and typed value of* SJ&+ *and* $(\!|\mathsf{v}|\!) \approx \mathsf{t}$*. Then* $\mathsf{t} \longrightarrow_T^* \mathsf{w}$ *and* $(\!|\mathsf{v}|\!) \approx \mathsf{w}$ *for some* w*.*

Proof. The first observation is that in $(\!|\mathsf{v}|\!)$ all λ-expressions are enclosed by a type cast, so reducing $(\!|\mathsf{v}|\!)$ we cannot obtain a pure λ-expression. In general, $(\!|\mathsf{v}|\!)$ is not a value since it contains added casts. If t is a proper value we are done. Otherwise, from t we can get a proper value w by reducing the casts inside evaluation contexts, since by Lemma 12 all casts added by the translation do not fail at run time. By definition of $\approx$ we also get $\mathsf{t} \approx \mathsf{w}$. From the transitivity of $\approx$ we conclude $(\!|\mathsf{v}|\!) \approx \mathsf{w}$. ◀

▶ **Lemma 15.** *Let* t *be a closed and typed term of* SJ&+*. If* $\mathsf{t} \longrightarrow_S \mathsf{t}_1$ *and* $(\!|\mathsf{t}|\!) \approx \mathsf{t}_2$*, then* $\mathsf{t}_2 \longrightarrow_T^* \mathsf{t}'$ *and* $(\!|\mathsf{t}_1|\!) \approx \mathsf{t}'$ *for some* t'*.*

Proof. The proof is by cases and by induction on the reduction rules of Figure 4. We only consider some interesting cases.

Rule [S-InvkNew].

$$\frac{\mathtt{mbodyS}(\mathsf{m};\mathsf{C}) = (\overrightarrow{\mathsf{x}}, \mathsf{t_m}) \quad \mathtt{mtypeS}(\mathsf{m};\mathsf{C}) = \overrightarrow{\tau} \to \tau}{\mathsf{new\,C}(\overrightarrow{\mathsf{v}}).\mathsf{m}(\overrightarrow{\mathsf{u}}) \longrightarrow_S [\overrightarrow{\mathsf{x}} \mapsto (\overrightarrow{\mathsf{u}})^{?\overrightarrow{\tau}}, \mathsf{this} \mapsto \mathsf{new\,C}(\overrightarrow{\mathsf{v}})](\mathsf{t_m})^{?\tau}}\ \text{[S-InvkNew]}$$

We get $(\!|\mathsf{new\,C}(\overrightarrow{\mathsf{v}}).\mathsf{m}(\overrightarrow{\mathsf{u}})|\!) = (\tau)\,(\mathsf{new\,C}(\overrightarrow{(\!|\mathsf{v}|\!)}).\mathsf{m}(\overrightarrow{(\!|\mathsf{u}|\!)}))$. From $(\tau)\,(\mathsf{new\,C}(\overrightarrow{(\!|\mathsf{v}|\!)}).\mathsf{m}(\overrightarrow{(\!|\mathsf{u}|\!)})) \approx \mathsf{t}_2$ we get $\mathsf{t}_2 = (\overline{\sigma})\,(\mathsf{new\,C}(\overrightarrow{\mathsf{r}}).\mathsf{m}(\overrightarrow{\mathsf{s}}))$, where $\overrightarrow{(\!|\mathsf{v}|\!)} \approx \overrightarrow{\mathsf{r}}$ and $\overrightarrow{(\!|\mathsf{u}|\!)} \approx \overrightarrow{\mathsf{s}}$. This implies $\mathsf{t}_2 \longrightarrow_T^* (\overline{\sigma})\,(\mathsf{new\,C}(\overrightarrow{\mathsf{w}}).\mathsf{m}(\overrightarrow{\mathsf{w}'}))$, where $\overrightarrow{\mathsf{r}} \longrightarrow_T^* \overrightarrow{\mathsf{w}}$ and $\overrightarrow{\mathsf{s}} \longrightarrow_T^* \overrightarrow{\mathsf{w}'}$ by Proposition 14. By rule [T-InvkNew] $(\overline{\sigma})\,(\mathsf{new\,C}(\overrightarrow{\mathsf{w}}).\mathsf{m}(\overrightarrow{\mathsf{w}'})) \longrightarrow_T (\overline{\sigma})\,([\overrightarrow{\mathsf{x}} \mapsto \overrightarrow{\mathsf{w}'}, \mathsf{this} \mapsto \mathsf{new\,C}(\overrightarrow{\mathsf{w}})](\!|\mathsf{t_m}|\!))$, since by construction $(\!|\mathsf{t_m}|\!)$ is the body of method m for class C in the class table CT^T and by definition a translation is never a pure λ-expression. We have

$$\mathsf{t}_1 = [\overrightarrow{\mathsf{x}} \mapsto (\overrightarrow{\mathsf{u}})^{?\overrightarrow{\tau}}, \mathsf{this} \mapsto \mathsf{new\,C}(\overrightarrow{\mathsf{v}})](\mathsf{t_m})^{?\tau}$$

which implies $(\!|\mathsf{t}_1|\!) = [\overrightarrow{\mathsf{x}} \mapsto \overrightarrow{(\!|\mathsf{u}|\!)}, \mathsf{this} \mapsto \mathsf{new\,C}(\overrightarrow{(\!|\mathsf{v}|\!)})](\!|\mathsf{t_m}|\!)$ since the translation of a decorated λ-expression is a λ-expression decorated with the same type, see rules [S-λ] and [S-*-λ]. From Proposition 14 we have $\overrightarrow{(\!|\mathsf{v}|\!)} \approx \overrightarrow{\mathsf{w}}$ and $\overrightarrow{(\!|\mathsf{u}|\!)} \approx \overrightarrow{\mathsf{w}'}$. If $\overline{\sigma} = \overline{\sigma}'\sigma$, since $\mathsf{t}_2 = (\overline{\sigma}')\,(\sigma)\,(\mathsf{new\,C}(\overrightarrow{\mathsf{r}}).\mathsf{m}(\overrightarrow{\mathsf{s}}))$ is typable, we have $\vdash_S^* \mathsf{new\,C}(\overrightarrow{\mathsf{r}}).\mathsf{m}(\overrightarrow{\mathsf{s}}) : \sigma$, which implies $\vdash_S^* [\overrightarrow{\mathsf{x}} \mapsto \overrightarrow{\mathsf{w}'}, \mathsf{this} \mapsto \mathsf{new\,C}(\overrightarrow{\mathsf{w}})](\!|\mathsf{t_m}|\!) : \sigma$ by Subject Reduction (Theorem 2(1)). We conclude

$$(\!|\mathsf{t}_1|\!) = [\overrightarrow{\mathsf{x}} \mapsto \overrightarrow{(\!|\mathsf{u}|\!)}, \mathsf{this} \mapsto \mathsf{new\,C}(\overrightarrow{(\!|\mathsf{v}|\!)})](\!|\mathsf{t_m}|\!) \approx (\overline{\sigma})\,([\overrightarrow{\mathsf{x}} \mapsto \overrightarrow{\mathsf{w}'}, \mathsf{this} \mapsto \mathsf{new\,C}(\overrightarrow{\mathsf{w}})](\!|\mathsf{t_m}|\!))$$

If $\overline{\sigma}$ is empty, then

$$\llparenthesis \mathsf{t}_1 \rrparenthesis = [\overrightarrow{\mathsf{x}} \mapsto \overrightarrow{\llparenthesis \mathsf{u} \rrparenthesis}, \mathsf{this} \mapsto \mathsf{new\,C}(\overrightarrow{\llparenthesis \mathsf{v} \rrparenthesis})]\llparenthesis \mathsf{t_m} \rrparenthesis \approx ([\overrightarrow{\mathsf{x}} \mapsto \overrightarrow{\mathsf{w}}', \mathsf{this} \mapsto \mathsf{new\,C}(\overrightarrow{\mathsf{w}})]\llparenthesis \mathsf{t_m} \rrparenthesis)$$

Rule [S-Invkλ-A].

$$\frac{\texttt{A-nameS}(\varphi) = \mathsf{m} \quad \texttt{A-mtypeS}(\varphi) = \overrightarrow{\tau} \to \tau}{(\overrightarrow{\mathsf{y}} \to \mathsf{t_m})^{\varphi}.\mathsf{m}(\overrightarrow{\mathsf{v}}) \longrightarrow_S [\overrightarrow{\mathsf{y}} \mapsto (\overrightarrow{\mathsf{v}})^{?\overrightarrow{\tau}}](\mathsf{t_m})^{?\tau}} \text{[S-Invk}\lambda\text{-A]}$$

We get $\llparenthesis (\overrightarrow{\mathsf{y}} \to \mathsf{t_m})^{\varphi}.\mathsf{m}(\overrightarrow{\mathsf{v}}) \rrparenthesis = (\tau)\,((\overrightarrow{\mathsf{y}} \to \llparenthesis \mathsf{t_m} \rrparenthesis)^{\varphi}.\mathsf{m}(\overrightarrow{\llparenthesis \mathsf{v} \rrparenthesis}))$.
From $(\tau)\,((\overrightarrow{\mathsf{y}} \to \llparenthesis \mathsf{t_m} \rrparenthesis)^{\varphi}.\mathsf{m}(\overrightarrow{\llparenthesis \mathsf{v} \rrparenthesis})) \approx \mathsf{t}_2$ we get $\mathsf{t}_2 = (\overline{\sigma})\,((\overrightarrow{\mathsf{y}} \to \mathsf{r})^{\varphi}.\mathsf{m}(\overrightarrow{\mathsf{r}}))$, where $\llparenthesis \mathsf{t_m} \rrparenthesis \approx \mathsf{r}$ and $\overrightarrow{\llparenthesis \mathsf{v} \rrparenthesis} \approx \overrightarrow{\mathsf{r}}$. This implies $\mathsf{t}_2 \longrightarrow_T (\overline{\sigma})\,((\overrightarrow{\mathsf{y}} \to \mathsf{r})^{\varphi}.\mathsf{m}(\overrightarrow{\mathsf{w}}))$, where $\overrightarrow{\mathsf{r}} \longrightarrow_T^* \overrightarrow{\mathsf{w}}$ by Proposition 14. By rule [T-Invkλ-A]

$$(\overline{\sigma})\,((\overrightarrow{\mathsf{y}} \to \mathsf{r})^{\varphi}.\mathsf{m}(\overrightarrow{\mathsf{w}})) \longrightarrow_T^* (\overline{\sigma})\,([\overrightarrow{\mathsf{y}} \mapsto \overrightarrow{\mathsf{w}}]\mathsf{r})$$

We have $\mathsf{t}_1 = [\overrightarrow{\mathsf{y}} \mapsto (\overrightarrow{\mathsf{v}})^{?\overrightarrow{\tau}}](\mathsf{t_m})^{?\tau}$, which implies $\llparenthesis \mathsf{t}_1 \rrparenthesis = [\overrightarrow{\mathsf{y}} \mapsto \overrightarrow{\llparenthesis \mathsf{v} \rrparenthesis}]\llparenthesis \mathsf{t_m} \rrparenthesis$. From Proposition 14 we have $\overrightarrow{\llparenthesis \mathsf{v} \rrparenthesis} \approx \overrightarrow{\mathsf{w}}$. If $\overline{\sigma} = \overline{\sigma'}\sigma$, since $\mathsf{t}_2 = (\overline{\sigma'})\,(\sigma)\,((\overrightarrow{\mathsf{y}} \to \mathsf{r})^{\varphi}.\mathsf{m}(\overrightarrow{\mathsf{r}}))$ is typable, we have $\vdash_S^* (\overrightarrow{\mathsf{y}} \to \mathsf{r})^{\varphi}.\mathsf{m}(\overrightarrow{\mathsf{r}}) : \sigma$, which implies $\vdash_S^* [\overrightarrow{\mathsf{y}} \mapsto \overrightarrow{\mathsf{w}}]\mathsf{r} : \sigma$ by Subject Reduction (Theorem 2(1)). We conclude $\llparenthesis \mathsf{t}_1 \rrparenthesis = [\overrightarrow{\mathsf{y}} \mapsto \overrightarrow{\llparenthesis \mathsf{v} \rrparenthesis}]\llparenthesis \mathsf{t_m} \rrparenthesis \approx (\overline{\sigma})\,([\overrightarrow{\mathsf{y}} \mapsto \overrightarrow{\mathsf{w}}]\mathsf{r})$. If $\overline{\sigma}$ is empty, then $\llparenthesis \mathsf{t}_1 \rrparenthesis = [\overrightarrow{\mathsf{y}} \mapsto \overrightarrow{\llparenthesis \mathsf{v} \rrparenthesis}]\llparenthesis \mathsf{t_m} \rrparenthesis \approx ([\overrightarrow{\mathsf{y}} \mapsto \overrightarrow{\mathsf{w}}]\mathsf{r})$. ◀

We now show our main result.

▶ **Theorem 16.** *Let* t_1 *be a closed and typed term of* SJ&+*. If*

$$\mathsf{t}_1 \longrightarrow_S \mathsf{t}_2 \longrightarrow_S \ldots \mathsf{t}_i \longrightarrow_S \ldots$$

is a finite or infinite reduction sequence in SJ&+*, then*

$$\llparenthesis \mathsf{t}_1 \rrparenthesis \longrightarrow_T^* \mathsf{t}'_2 \longrightarrow_T^* \ldots \mathsf{t}'_i \longrightarrow_T^* \ldots$$

where $\llparenthesis \mathsf{t}_i \rrparenthesis \approx \mathsf{t}'_i$*,* $i > 1$*, is a finite or infinite reduction sequence in* TJ&*.*

Proof. If $i > 1$, then from $\mathsf{t}_i \longrightarrow_S \mathsf{t}_{i+1}$ and $\llparenthesis \mathsf{t}_i \rrparenthesis \approx \mathsf{t}'_i$ we get $\mathsf{t}'_i \longrightarrow_T^* \mathsf{t}'_{i+1}$ and $\llparenthesis \mathsf{t}_{i+1} \rrparenthesis \approx \mathsf{t}'_{i+1}$ by Lemma 15. For $i = 1$ we can take $\mathsf{t}'_1 = \llparenthesis \mathsf{t}_1 \rrparenthesis$ since $\approx$ is reflexive. ◀

We end this section by providing the relation between values obtained by reducing a closed and typed source term and its translation. The proof is an easy consequence of previous theorem.

▶ **Corollary 17.** *Let* t *be a closed and typed term of* SJ&+*. If* $\mathsf{t} \longrightarrow_S^* \mathsf{w}$*, then* $\llparenthesis \mathsf{t} \rrparenthesis \longrightarrow_T^* \mathsf{w}'$ *with* $\llparenthesis \mathsf{w} \rrparenthesis \approx \mathsf{w}'$*.*

This corollary assures that the behaviour of a translated program in TJ& with value w' exactly reflects the behaviour of the original program in SJ&+ with value w. A simple example is $\mathsf{w} = \mathsf{new\,C}(\mathsf{x} \to \mathsf{x})$, where the field of C has type $\mathsf{I\&J}$ and the abstract method of I maps objects of class D to objects of class D. Then $\llparenthesis \mathsf{w} \rrparenthesis = \mathsf{new\,C}((\mathsf{I\&J})\,(\mathsf{x} \to (\mathsf{D})\,\mathsf{x}))$ and $\mathsf{w}' = \mathsf{new\,C}((\mathsf{x} \to (\mathsf{D})\,\mathsf{x})^{\mathsf{I\&J}})$. The translation only adds in w redundant type information (with respect to the class table CT^S) in the form of upcasts on subterms. Then, loosely speaking, we can say that w and $\llparenthesis \mathsf{w} \rrparenthesis$ are contextually equivalent in SJ&+, since their occurrences in a complete program can be interchanged without affecting the result of executing the program in a significant way. Differently, when considering $\llparenthesis \mathsf{w} \rrparenthesis$ in TJ&, all those type casts which are added by translation are necessary for typing and preserve types. Corollary 17 says that w' is cast-equivalent to $\llparenthesis \mathsf{w} \rrparenthesis$, which is contextually equivalent to w in SJ&+.

7 Related Work and Conclusion

We have presented a compilation of programs of an extension of Java, where intersection types are completely deconfined from the present boundary within type casts, to the unextended language. Thus the proposed Java extension is guaranteed to be fully backward compatible with the current Java, namely its code can be compiled into JVM.

Formally, we exploited two calculi, TJ& and SJ&+. Since the keystone paper [15], Featherweight Java (FJ) has become the standard calculus for formalising extensions and variants of Java, some of them are listed in the references of [2]. Not surprisingly the two calculi discussed in the present paper are extensions of FJ. Furthermore, they follow the style of FJ, in omitting all the features that do not interact with our issue in a significant way.

TJ& is a core model of Java 8, focusing on the two main novelties: λ-expressions and intersection types in type casts. SJ&+ comes from FJP&λ, presented in [10], the first formal account proposing to extend Java with λ-expressions by the capability of using intersection types as parameter types and return types of methods, as well as field types.

The main contribution of the present paper is a translation of typed programs of SJ&+ into typed programs of TJ&. The translation basically consists in erasing intersection types in method signatures and field types and, consistently, adding type downcasts in terms where needed. Checking these downcasts at run time will always succeed. Namely, we proved that our translation preserves typing and semantics of the source programs.

Intersection types as parameter types and return types of methods were first proposed in [5], which contains interesting examples of structuring and reusing code. Our main inspiration in stating the properties of the translation has been [15], where the safety of GJ (Featherweight Java with generic classes [4]) is shown by compiling GJ into FJ. Also in [15] the translation only adds downcasts, called “synthetic casts”, which are proved not to fail at run time.

In [10] we also proposed to overcome the limitation of exactly one abstract method in functional interfaces. This naturally agrees with the standard meaning of intersection types: intersection types express multiple, possibly unrelated, properties of terms. It magnifies also the polyadic nature of λ-expressions, which can match multiple headers of abstract methods, with different signatures.

For future works, we plan to compile functional interfaces with many abstract methods into functional interfaces with exactly one abstract method. The inspiration will be the formulation of intersection types à la Church, i.e. with types decorating terms [17, 3, 11, 12].

Moreover, the relation between traits in Scala [19] (Chapter 12) and Java interfaces with default methods is clearly worth to be deeply investigated. As a further development of the present paper, we would like to study how to implement a form of traits in Java, by exploiting intersection types. The ability of expressing combinations of traits, as intersections types, also in requirements, that is in parameter and return types of methods, seems to be the key tool for using traits as a valuable design concept (see the discussion in [25]).

References

1 Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A Filter Lambda Model and the Completeness of Type Assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983. `doi:10.2307/2273659`.

2 Lorenzo Bettini, Viviana Bono, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri. Java & Lambda: a Featherweight Story. *Logical Methods in Computer Science*, 14(3):1–24, 2018. `doi:10.23638/LMCS-14(3:17)2018`.

3 Viviana Bono, Betti Venneri, and Lorenzo Bettini. A Typed Lambda Calculus with Intersection Types. *Theoretical Computer Science*, 398(1-3):95–113, 2008. `doi:10.1016/j.tcs.2008.01.046`.

4 Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In Bjørn N. Freeman-Benson and Craig Chambers, editors, *OOPSLA*, pages 183–200. ACM Press, 1998. `doi:10.1145/286936.286957`.

5 Martin Büchi and Wolfgang Weck. Compound Types for Java. In Bjørn N. Freeman-Benson and Craig Chambers, editors, *OOPSLA*, pages 362–373. ACM Press, 1998. `doi:10.1145/286936.286975`.

6 Mario Coppo and Mariangiola Dezani-Ciancaglini. An Extension of the Basic Functionality Theory for the λ-calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, 1980. `doi:10.1305/ndjfl/1093883253`.

7 Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Principal Type Schemes and λ-calculus Semantics. In *To H.B.Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism*, pages 535–560. Academic Press, 1980. ISBN-13: 9780123490506.

8 Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional Characters of Solvable Terms. *Mathematical Logical Quartely*, 27(2-6):45–58, 1981. `doi:10.1002/malq.19810270205`.

9 Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A Core Calculus for Scala Type Checking. In Rastislav Kralovic and Pawel Urzyczyn, editors, *MFCS*, volume 4162 of *LNCS*, pages 1–23, Berlin, 2006. Springer. `doi:10.1007/11821069_1`.

10 Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri. Intersection Types in Java: Back to the Future. In Tiziana Margaria, Susanne Graf, and Kim G. Larsen, editors, *Models, Mindsets, Meta: The What, the How, and the Why Not?*, volume 11200 of *LNCS*, pages 68–86. Springer, 2018. `doi:10.1007/978-3-030-22348-9_6`.

11 Andrej Dudenhefner and Jakob Rehof. Intersection Type Calculi of Bounded Dimension. In Giuseppe Castagna and Andrew D. Gordon, editors, *POPL*, pages 653–665. ACM Press, 2017. `doi:10.15520/jbme.v6i7.2243`.

12 Andrej Dudenhefner and Jakob Rehof. Typability in Bounded Dimension. In Joel Ouaknine, editor, *LICS*, pages 1–12. IEEE Computer Society, 2017. `doi:10.1109/LICS.2017.8005127`.

13 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN-13: 0201633612.

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

15 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001. `doi:10.1145/503502.503505`.

16 Java 10, 2018. URL: `https://docs.oracle.com/javase/10/language/toc.htm#JSLAN-GUID-7D5FDD65-ACE4-4B3C-80F4-CC01CBD211A4`.

17 Luigi Liquori and Simona Ronchi Della Rocca. Intersection-types à la Church. *Information and Computation*, 205(9):1371–1386, 2007. `doi:10.1016/j.ic.2007.03.005`.

18 Robert C. Martin. *Agile Software Development: Principles, Patterns and Practices*. Pearson Education, 2002. ISBN: 0-13-597444-5.

19 Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: Updated for Scala 2.12*. Artima Incorporation, 3rd edition, 2016. ISBN-13: 9780981531687.

20 Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. ISBN: 0262162091.

21 Garrel Pottinger. A Type Assignment for the Strongly Normalizable λ-terms. In *To H.B.Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism*, pages 561–578. Academic Press, 1980. ISBN-13: 9780123490506.

22 Marianna Rapoport, Ifaz Kabir, Paul He, and Ondrej Lhoták. A Simple Soundness Proof for Dependent Object Types. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):46:1–46:27, 2017. doi:10.1145/3133870.

23 John C. Reynolds. Conjunctive Types and Algol-like Languages. In David Gries, editor, *LICS*, page 119. IEEE Computer Society, 1987.

24 John C. Reynolds. Design of the Programming Language Forsythe. In *Algol-like Languages, Progress in Theoretical Computer Science*, pages 173–233. Birkhäuser, 1997. ISBN-978-1-4612-8661-5.

25 Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable Units of Behaviour. In Luca Cardelli, editor, *ECOOP*, volume 2743 of *LNCS*, pages 248–274. Springer, 2003. doi:10.1007/978-3-540-45070-2_12.