Deconfined Intersection Types in Java

joint work with Paola Giannini and Betti Venneri

IFIP 2.2 meeting 2021, 21 September

https://drops.dagstuhl.de/opus/volltexte/2020/13225/

Intersection types in Java

- Intersection types τ are defined by

Intersection types in Java

- Intersection types τ are defined by

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

  C is a class name and I is an interface name.

Intersection types in Java

- Intersection types τ are defined by

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

  C is a class name and I is an interface name.
- Intersection types are not first-class citizens of Java type system, they can occur only as

Intersection types in Java

- Intersection types τ are defined by

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

  C is a class name and I is an interface name.
- Intersection types are not first-class citizens of Java type system, they can occur only as
  - bounds of type variables in generic types:

```
<X extends C & I1 & I2> void m(X x) { ... }
```

Intersection types in Java

- Intersection types τ are defined by

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

  C is a class name and I is an interface name.
- Intersection types are not first-class citizens of Java type system, they can occur only as
  - bounds of type variables in generic types:

    ```
    <X extends C & I1 & I2> void m(X x) { ... }
    ```

  - as types of type-casts and as target types of λ-expressions

    (I1 & I2) λ-expr

    is the λ-expr with the target type `(I1 & I2)`

Intersection types in Java

- Intersection types τ are defined by

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

  C is a class name and I is an interface name.
- Intersection types are not first-class citizens of Java type system, they can occur only as
  - bounds of type variables in generic types:

    ```
    <X extends C & I1 & I2> void m(X x) { ... }
    ```

  - as types of type-casts and as target types of λ-expressions

    `(I1 & I2)` λ`-expr`

    is the λ-expr with the target type `(I1 & I2)`
- The target types must be functional, i.e., have

Intersection types in Java

- Intersection types τ are defined by

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

  C is a class name and I is an interface name.
- Intersection types are not first-class citizens of Java type system, they can occur only as
  - bounds of type variables in generic types:

    ```
    <X extends C & I1 & I2> void m(X x) { ... }
    ```

  - as types of type-casts and as target types of λ-expressions

    (I1 & I2) λ-expr

    is the λ-expr with the target type `(I1 & I2)`
- The target types must be functional, i.e., have
  - exactly one abstract method (implemented by λ-expressions)

Intersection types in Java

- Intersection types τ are defined by

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

  C is a class name and I is an interface name.
- Intersection types are not first-class citizens of Java type system, they can occur only as
  - bounds of type variables in generic types:

    ```
    <X extends C & I1 & I2> void m(X x) { ... }
    ```

  - as types of type-casts and as target types of λ-expressions

    (I1 & I2) λ-expr

    is the λ-expr with the target type `(I1 & I2)`
- The target types must be functional, i.e., have
  - exactly one abstract method (implemented by λ-expressions)
  - any number of default methods (with λ-expressions as receivers)

Our proposal: intersections as first-class types

An intersection type can occur everywhere, i.e. as

Our proposal: intersections as first-class types

An intersection type can occur everywhere, i.e. as

- a return type

Our proposal: intersections as first-class types

An intersection type can occur everywhere, i.e. as

- a return type
- a parameter type

Our proposal: intersections as first-class types

An intersection type can occur everywhere, i.e. as

- a return type
- a parameter type
- the explicit type of a variable

Our proposal: intersections as first-class types

An intersection type can occur everywhere, i.e. as

- a return type
- a parameter type
- the explicit type of a variable
- a field type

Our proposal: intersections as first-class types

An intersection type can occur everywhere, i.e. as

- a return type
- a parameter type
- the explicit type of a variable
- a field type

for example we can write `I1&I2 m (I3&I4 x)`

Our proposal: intersections as first-class types

An intersection type can occur everywhere, i.e. as

- a return type
- a parameter type
- the explicit type of a variable
- a field type

for example we can write `I1&I2 m (I3&I4 x)`

Advantages:

- We avoid the use of obscure generic signatures

Our proposal: intersections as first-class types

An intersection type can occur everywhere, i.e. as

- a return type
- a parameter type
- the explicit type of a variable
- a field type

for example we can write `I1&I2 m (I3&I4 x)`

Advantages:

- We avoid the use of obscure generic signatures
- We avoid unsafe type casts

Our proposal: intersections as first-class types

An intersection type can occur everywhere, i.e. as

- a return type
- a parameter type
- the explicit type of a variable
- a field type

for example we can write `I1&I2 m (I3&I4 x)`

Advantages:

- We avoid the use of obscure generic signatures
- We avoid unsafe type casts
- We exploit the polytype nature of λ-expressions

Game example

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

interface Swimmable { void swim(); }
```

Game example

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

interface Swimmable { void swim(); }
```

Players that can do both implement both interfaces.

Game example

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

interface Swimmable { void swim(); }
```

Players that can do both implement both interfaces.

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

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

Game example

```
public class Game {
 public static void goAcrossRavine(XXX player,boolean unWatObj){
   System.out.println("Reached the ravine");
   if (unWatObj) {
     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
 }
```

Game example

```
public class Game {
 public static void goAcrossRavine(XXX player,boolean unWatObj){
   System.out.println("Reached the ravine");
   if (unWatObj) {
     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
 }
```

For XXX we need a player implementing `Flyable` and `Swimmable`, which is allowed in our extension

Game example

Can generic types help?

Game example

Can generic types help?

NO, we can declare `player` of type `X` with

```
<X extends Flyable&Swimmable>
```

Game example

Can generic types help?

NO, we can declare `player` of type `X` with

```
<X extends  Flyable&Swimmable>
```

but we cannot declare

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

Game example

Can generic types help?

NO, we can declare `player` of type `X` with

```
<X extends  Flyable&Swimmable>
```

but we cannot declare

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

we can only write

```
X player = (X) new Pelican();
```

this introduces a type cast and exposes the type of a local variable!

Discount example

```
interface Discount { double discount(int price); }
interface DelPrice {
 default double delPrice(int price) {
  return (price>30)? 0: 5;
 }
}
```

Discount example

```
interface Discount { double discount(int price); }
interface DelPrice {
 default double delPrice(int price) {
  return (price>30)? 0: 5;
 }
}
```

in our extension

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

Discount example

```
interface Discount { double discount(int price); }
interface DelPrice {
 default double delPrice(int price) {
  return (price>30)? 0: 5;
 }
}
```

in our extension

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

instead in Java (using **var**)

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

`funPrice` is fixed and type casted!

Translation aerial view

Source calculus: Java with deconfined intersection types

Translation aerial view

Source calculus: Java with deconfined intersection types

Target calculus: standard Java

Translation aerial view

Source calculus: Java with deconfined intersection types

Target calculus: standard Java

Source calculus versus Target calculus

Translation aerial view

Source calculus: Java with deconfined intersection types

Target calculus: standard Java

Source calculus versus Target calculus

- same sets of terms

Translation aerial view

Source calculus: Java with deconfined intersection types

Target calculus: standard Java

Source calculus versus Target calculus

- same sets of terms
- different types in declarations

Translation aerial view

Source calculus: Java with deconfined intersection types

Target calculus: standard Java

Source calculus versus Target calculus

- same sets of terms
- different types in declarations

Compilation strategy

Translation aerial view

Source calculus: Java with deconfined intersection types

Target calculus: standard Java

Source calculus versus Target calculus

- same sets of terms
- different types in declarations

Compilation strategy

- replace intersections with their most relevant components

Translation aerial view

Source calculus: Java with deconfined intersection types

Target calculus: standard Java

Source calculus versus Target calculus

- same sets of terms
- different types in declarations

Compilation strategy

- replace intersections with their most relevant components
- recover the lost type information inserting downcasts (essential to preserve the target types of λ-expressions)

Three mappings

Erasure of types: from intersection types to nominal types

Three mappings

Erasure of types: from intersection types to nominal types

convention: the first interface in intersections of interfaces is functional, if any

Three mappings

Erasure of types: from intersection types to nominal types

convention: the first interface in intersections of interfaces is functional, if any

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

Three mappings

Erasure of types: from intersection types to nominal types

convention: the first interface in intersections of interfaces is functional, if any

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

the erasure maps functional intersections into functional interfaces (essential to preserve target types of λ-expressions)

Three mappings

Erasure of types: from intersection types to nominal types

Compilation of terms: from type derivations for terms of the source calculus to terms of the target calculus

Three mappings

Erasure of types: from intersection types to nominal types

Compilation of terms: from type derivations for terms of the source calculus to terms of the target calculus

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

$$(\sigma)\,((\![\mathcal{D}]\!).\mathsf{m}(\overrightarrow{(\![\mathcal{D}]\!)}))$$

Three mappings

Erasure of types: from intersection types to nominal types

Compilation of terms: from type derivations for terms of the source calculus to terms of the target calculus

Compilation of declarations: from declarations in the source calculus to declarations in the target calculus

Three mappings

Erasure of types: from intersection types to nominal types

Compilation of terms: from type derivations for terms of the source calculus to terms of the target calculus

Compilation of declarations: from declarations in the source calculus to declarations in the target calculus

Headers: $[\![\tau \mathsf{m}(\overrightarrow{\tau}\,\overrightarrow{\mathsf{x}})]\!] = |\tau|\mathsf{m}(\overrightarrow{|\tau|}\,\overrightarrow{\mathsf{x}})$

Three mappings

Erasure of types: from intersection types to nominal types

Compilation of terms: from type derivations for terms of the source calculus to terms of the target calculus

Compilation of declarations: from declarations in the source calculus to declarations in the target calculus

Constructors:

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

Three mappings

Erasure of types: from intersection types to nominal types

Compilation of terms: from type derivations for terms of the source calculus to terms of the target calculus

Compilation of declarations: from declarations in the source calculus to declarations in the target calculus

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

Three mappings

Erasure of types: from intersection types to nominal types

Compilation of terms: from type derivations for terms of the source calculus to terms of the target calculus

Compilation of declarations: from declarations in the source calculus to declarations in the target calculus

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

Three mappings

Erasure of types: from intersection types to nominal types

Compilation of terms: from type derivations for terms of the source calculus to terms of the target calculus

Compilation of declarations: from declarations in the source calculus to declarations in the target calculus

$$\text{Interfaces: } \llbracket \mathsf{interface\,I\,extends}\ \overrightarrow{\mathsf{I}}\ \{\overline{\mathsf{H}^S}; \overline{\mathsf{M}^S}\} \rrbracket = \\ \mathsf{interface\,I\,extends}\ \overrightarrow{\mathsf{I}}\ \{\overline{\llbracket \mathsf{H}^S \rrbracket}; \overline{\llbracket \mathsf{M}^S \rrbracket^{\mathsf{I}}}\}$$

Three mappings

Erasure of types: from intersection types to nominal types

Compilation of terms: from type derivations for terms of the source calculus to terms of the target calculus

Compilation of declarations: from declarations in the source calculus to declarations in the target calculus

Implementation done by Stefano Borsatto available at
`https://github.com/cplrossi/extendj-dit`

Properties of the translation

Type preservation

The translation of a typed program in the source calculus is a program with the same type in the target calculus

Properties of the translation

Type preservation

The translation of a typed program in the source calculus is a program with the same type in the target calculus

Semantic preservation

Properties of the translation

Type preservation

The translation of a typed program in the source calculus is a program with the same type in the target calculus

Semantic preservation

problems:

Properties of the translation

Type preservation

The translation of a typed program in the source calculus is a program with the same type in the target calculus

Semantic preservation

problems:

- some casts are still unreduced in the translated terms (for instance, they appear in the body of λ-expressions)

Properties of the translation

Type preservation

The translation of a typed program in the source calculus is a program with the same type in the target calculus

Semantic preservation

problems:

- some casts are still unreduced in the translated terms (for instance, they appear in the body of λ-expressions)
- the translation of a value is not a value, in general

Properties of the translation

Type preservation

The translation of a typed program in the source calculus is a program with the same type in the target calculus

Semantic preservation

problems:

- some casts are still unreduced in the translated terms (for instance, they appear in the body of λ-expressions)
- the translation of a value is not a value, in general

solution: an equivalence relation on terms which

Properties of the translation

Type preservation

The translation of a typed program in the source calculus is a program with the same type in the target calculus

Semantic preservation

problems:

- some casts are still unreduced in the translated terms (for instance, they appear in the body of λ-expressions)
- the translation of a value is not a value, in general

solution: an equivalence relation on terms which

- ignores casts on terms different from λ-expressions

Properties of the translation

Type preservation

The translation of a typed program in the source calculus is a program with the same type in the target calculus

Semantic preservation

problems:

- some casts are still unreduced in the translated terms (for instance, they appear in the body of λ-expressions)
- the translation of a value is not a value, in general

solution: an equivalence relation on terms which

- ignores casts on terms different from λ-expressions
- uses casts on λ-expressions as target types

Properties of the translation

Type preservation

The translation of a typed program in the source calculus is a program with the same type in the target calculus

Semantic preservation

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

implies

$$(\!\![\mathsf{t}_1]\!\!) \longrightarrow^*_T \mathsf{t}'_2 \longrightarrow^*_T \ldots \mathsf{t}'_i \longrightarrow^*_T \ldots$$

where $(\!\![\mathsf{t}_i]\!\!) \approx \mathsf{t}'_i,\ i > 1$

Conclusions

We showed that intersections as first-class types

Conclusions

We showed that intersections as first-class types

- exploit the use of interfaces (with default methods) as pieces of code that can be differently mixed to express compound types

Conclusions

We showed that intersections as first-class types

- exploit the use of interfaces (with default methods) as pieces of code that can be differently mixed to express compound types
- avoid interface pollution and enable clean code evolution

Conclusions

We showed that intersections as first-class types

- exploit the use of interfaces (with default methods) as pieces of code that can be differently mixed to express compound types
- avoid interface pollution and enable clean code evolution
- minimise the use of type casts, enforcing type safety statically

Conclusions

We showed that intersections as first-class types

- exploit the use of interfaces (with default methods) as pieces of code that can be differently mixed to express compound types
- avoid interface pollution and enable clean code evolution
- minimise the use of type casts, enforcing type safety statically
- maximise code reuse by exploiting the polytype nature of λ-expressions

Conclusions

We showed that intersections as first-class types

- exploit the use of interfaces (with default methods) as pieces of code that can be differently mixed to express compound types
- avoid interface pollution and enable clean code evolution
- minimise the use of type casts, enforcing type safety statically
- maximise code reuse by exploiting the polytype nature of λ-expressions

We gave a compilation of typed programs in *Java with deconfined intersection types* into *Java* typed programs preserving typing and semantics

Future work

- translate functional interfaces with many abstract methods into functional interfaces with exactly one abstract method

Future work

- translate functional interfaces with many abstract methods into functional interfaces with exactly one abstract method
- investigate a light notion of traits for Java that can express combinations of traits as intersection types

Future work

- translate functional interfaces with many abstract methods into functional interfaces with exactly one abstract method
- investigate a light notion of traits for Java that can express combinations of traits as intersection types
- and compile this notion of traits into Java

Take home message

Thank you


Ευχαριστώ
Gracias
Merci
Спасибо
Thank you
Danke
Merci
Danke
Dziękuję
Gracias
Gracias
Спасибо
متشكرم