Batak Java: Version Numbered Object-oriented Language to Solve Dependency Hell

Tokyo Institute of Technology, School of Science,
Department of Information Science

Luthfan Anshar Lubis

15B15953

Academic Supervisor

Hidehiko Masuhara

February 28, 2019

Contents

Chapter 1

Introduction

Version is an important, innate property of a computer program. Most often than not, aside from their names, the programs we use are always identified by their versions. Many computer programs and softwares will, to meet the demand for functions and better capabilities, be produced with newer versions as they are updated over time. Over the years, the need to keep track and maintain version updates has spawned the creation of version control systems, such as Git and SVM.

With software update however comes version incompatibility, which is a very common problem in software development. This problem is especially common in the case where interdependent softwares are developed and updated independently and asynchronously. One problem that we can examine is called *dependency hell*. Dependency hell is incompatibility or conflict that may occur when there are different versions of libraries being used, directly or indirectly. In most existing programming languages, two definitions of the same function or class cannot be distinguished when they have the same names and signatures.

Figure 1.1 illustrates an example of dependency hell. Suppose that the libraries `Base`, `Education` and `Health` are all provided by different sources and developed independently. At the beginning, both `Education` and `Health` depend on `Base`, while `Municipal` depends on both `Education` and `Health`. Before the update, `Municipal` runs without any issue. Afterwards however, a second version of `Base` was introduced, and subsequently used by `Health`. Here, although `Base` does not have a direct relationship with `Base`, due to its dependency relation with both `Education` and `Health` which now use different version of `Base`, error will occur in `Municipal` because two different definitions of the same library are mixed together. This type of dependency

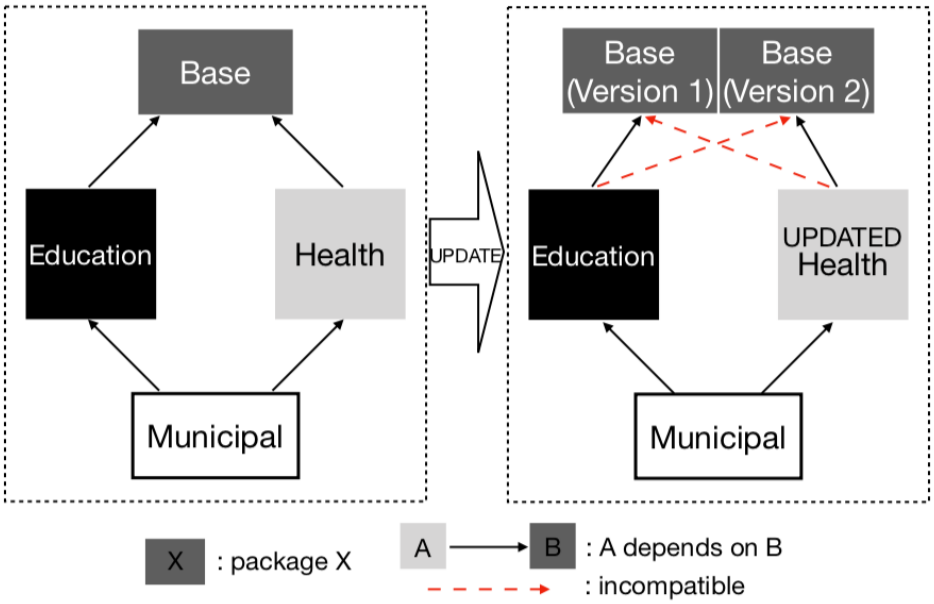

Figure 1.1: Example of dependency hell

hell is called *diamond dependency* due to its diamond-like shape.

Another example of dependency hell is conflicting dependencies that occur between multiple parts of a program. Suppose we have an application `app1` using the first version of `Base` and a second application using the second version of `Base`. To run the whole application we will require both applications, but due to conflicting dependencies, the application will not run properly.

One concrete example of this problem can be found in frameworks and programs requiring ASM. ASM is a Java bytecode manipulation and analysis framework[1]. A program is built using Groovy and Spring framework, both depending on ASM version 3. Later on Groovy was updated to use ASM version 4, as a consequence of this, the original program ran into an issue because version 3 and version 4 are not compatible with each other[2].

Even considering the importance of the notion of version, there are not that many languages created for the purpose of addressing this issue. Preceding studies which has addressed this issue before includes Variational

[1] ASM, https://asm.ow2.io/

[2] https://blog.jayway.com/2013/04/12/solving-asm-conflicts-after-upgrading-to-groovy-2-1/

Programming [2] and VL [7]. However, both of these researches focus on functional programming. There is not yet a research working on the solution for object-oriented programming, where issues occuring from version changes may propagate through inheritance.

A very simple way to deal with the issue in Figure 1.1 is by renaming all the class declarations in either the first or second version of `Base` so there is no name conflicts. However, using this solution means we lose the relationship between one version with another. In other words we are creating new definitions with different names every time.

Thus to solve this problem, we introduce the idea of version number as an attribute of package (versioned package), to discern a type in a particular version from another. By doing so we can avoid conflict from occuring and increase flexibility. We propose Batak Java, an object-oriented language based on a minimal core calculus for Java system called Featherweight Java [5], equipped with versioned package and package interface.

We expect the following capabilities from Batak Java:

- The feature of version numbering gives the users freedom to choose which version of a particular library they want to use.
- In general, same named classes taken from different version are treated as separate types, therefore it is possible for users to mix different version within the attributes of a class and an expression.
- Package interface to manage which behavior (constructor, method) of a class the user can keep public.
- A type system which soundness has been proven.

Chapter 2

Background

Featherweight Java

Batak Java is based on Featherweight Java, a minimal core calculus that models Java's type system. Featherweight Java is minimal in the way that most of the features normally found in Java, such as assignment, interfaces, overloading, side effects and abstract method are omitted. In the calculus of Featherweight Java we can only find classes, methods, fields, inheritance, and dynamic typecasts. Featherweight Java provides only five forms of expression: object creation, method invocation, field access, casting and variables. The compactness of Featherweight Java's calculus makes it easy to conduct rigorous proofs for any extensions and variations that we may add into the object-oriented language. In our case we will be adding versioned package and package interface into this calculus.

Although the compactness entails the inability to do various computations with the calculus, encoding another calculus, such as the lambda calculus within Featherweight Java is not difficult.

In Featherweight Java, a program consists of class definitions and an expression to be evaluated. Listing 2.1 shows an example of class definitions in Featherweight Java.

```
class A extends Object {
  A() { super(); }
}

class B extends Object {
  Object single;

  B(Object single){
    super();
```

```
      this.single = single;
    }
  }

  class Pair extends Object {
    A fst;
    B snd;

    Pair(A fst, B snd){
      super();
      this.fst = fst;
      this.snd = snd;
    }

    Pair setfst(A newfst){
      return new Pair(newfst, this.snd);
    }
  }
```

Listing 2.1: Class declarations in Featherweight Java

As can be seen above, class definition in Featherweight Java is strict. Every class must declare its superclass even if it is only `Object`. The constructor must include all the fields, including those of the superclass. The special name `super` calls the constructor of the superclass. `super` must always be called even if the superclass is only `Object`.

As previously mentioned, a program consists of class definitions and an expression to be evaluated. Based on Listing 2.1, below is an example of field access expression. $\rightarrow$ here denotes evaluation.

`new Pair(new A(), new B(new Object())).fst` $\rightarrow$ `new A()`

While method invocation in Featherweight Java looks like below. $\mapsto$ here denotes substitution.

$$\texttt{new Pair(new B(new Object()), new A()).setfst(new A())}$$
$$\rightarrow \begin{bmatrix}\texttt{newfst} \mapsto \texttt{new A}(), \\ \texttt{this} \mapsto \texttt{new Pair}(\texttt{newB}(\texttt{newObject}(), \texttt{newA}()))\end{bmatrix}$$
$$\texttt{new Pair(newfst, this.snd)}$$

Example of Dependency Hell

Using the example of `Base`, `Education`, `Health` and `Municipal` introduced shortly in the first chapter, we will show a concrete dependency hell problem.

Below we will show a simple example of the implementation of these libraries written in Java. Figure 2.1 shows the class diagram.

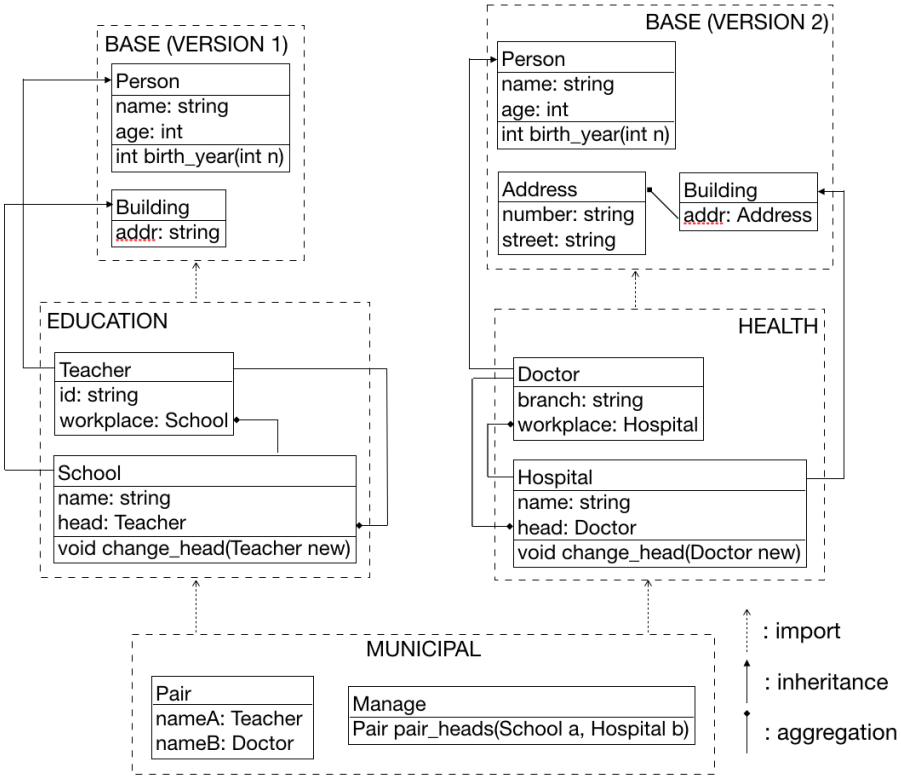

Figure 2.1: Class diagram of the libraries

```
// The first version of Base
package Base

class Person {
  String name;
  int age;

  Person(String name, int age){
    this.name = name;
    this.age = age;
  }

  int birth_year(int current_year){
    return current_year;
```

```
  }
}

class Building {
  String addr;

  Building(String addr){
    this.addr = addr;
  }
}
```

Listing 2.2: First version of package `Base`

```
// The second version of Base
package Base
class Person {
  String name;
  int age;

  Person(String name, int age){
    this.name = name;
    this.age = age;
  }

  int birth_year(int current_year){
    return current_year;
  }
}

class Address {
  String street;
  String number;

  Address(String street, String number){
    this.street = street;
    this.number = number;
  }
}

class Building {
  Address addr;

  Building(Address addr){
    this.addr = addr;
  }
}
```

Listing 2.3: Second version of package `Base`

From the Figure 1.1, we assumed that we have two different versions of

`Base`. Here, we assume that the difference lies in the introduction of the class `Address` and its usage in the class `Building`, while both versions have identical class `Person`.

```
// The library/package Education imports
// the first version of Base
package Education imports Base

class Teacher extends Person {
  String id;
  School workplace;

  Teacher(String name, int age, String id, School workplace){
    super(name, age);
    this.id = id;
    this.workplace = workplace;
  }
}

class School extends Building {
  String name;
  Teacher head;

  School(String addr, String name, Teacher head){
    super(addr);
    this.name = name;
    this.head = head;
  }

  void change_head(Teacher new_head){
    this.head = new_head;
  }
}
```

Listing 2.4: Package `Education`

As the Figure 2.1 illustrates, the package `Education` uses the first version of `Base`. In this package we declared the class `Teacher` which is a subclass of `Person` from the first version of `Base` and `School` which is a subclass of `Building`, also from the first version of `Base`.

```
// This package Health imports the second version of Base
package Health imports Base

class Doctor extends Person {
  String branch;
  Hospital workplace;

  Doctor(String name,
```

```
          int age,
          String branch,
          Hospital workplace)
  {
    super(name, age);
    this.branch = branch;
    this.workplace = workplace;
  }
}

class Hospital extends Building {
  String name;
  Doctor head;

  Hospital(Address addr, String name, Doctor head){
    super(addr);
    this.name = name;
    this.head = head;
  }

  void change_head(Doctor new_head){
    this.head = new_head;
  }
}
```

Listing 2.5: Package `Health`

The package `Health` uses the second version of `Base`. In this package, class `Doctor` is a subclass of `Person` and `Hospital` is a subclass of `Building`.

```
package Municipal imports Education, Health

class Pair extends Object {
  Teacher nameA;
  Doctor nameB;

  Pair(Teacher nameA, Doctor nameB){
    this.nameA = nameA;
    this.nameB = nameB;
  }
}

class Manage extends Object {
  Manage(){ }

  Pair pair_heads(School a, Hospital b){
    return new Pair(a.head, b.head);
  }
```

```
}
```

Listing 2.6: Package `Municipal`

In `Municipal`, we use both `Education` and `Health`. However, since `Education` and `Health` both use a different version of `Base`, multiple definitions of `Base` ended mixing in `Municipal`. This results in error because we do not have the ability to decide which `Base` to use for each object.

Problems in ASM

ASM, a framework used to manipulate Java bytecode is one example which we can use to show the problem of dependency hell.

From comparing ASM 3.X and ASM 4.X we can find several differences such as the removal of class `ClassAdapter` and conversion of `ClassVisitor`, `AnnotationVisitor`, etc. which were interfaces in ASM 3.X into abstract classes in later versions. However, these changes are not as disrupting as the update which happened in ASM 5.X.

As mentioned in the introduction, the method `VisitMethodInsn` in ASM 4.X requires only 4 arguments but it requires 5 arguments in the ASM 5.X. The ASM program quoted in Listing 2.7 and 2.8 are written in full Java.

```
public abstract class MethodVisitor {
  protected final int api;
  protected MethodVisitor mv;
      ...
  public void visitMethodInsn(int opcode, String owner,
  String name, String desc) {
        if (mv != null) {
            mv.visitMethodInsn(opcode, owner, name, desc);
        }
    }
      ...
}
```

Listing 2.7: `MethodVisitor` in ASM 4.X

```
public abstract class MethodVisitor {
  protected final int api;
  protected MethodVisitor mv;
    ...
    public void visitMethodInsn(int opcode, String owner,
    String name, String desc, boolean itf) {
        if (api < Opcodes.ASM5) {
          ...
        }
```

```
            if (mv != null) {
                mv.visitMethodInsn(opcode, owner, name, desc,
                itf);
            }
        }
           ...
}
```

Listing 2.8: `MethodVisitor` in ASM 5.X

If we were the direct user of ASM, this problem can easily be dealt with by changing parts of our own program. The dependency hell problem however, occurs when a user's program requiring multiple libraries, where each package is using different version of a different package. If we put ASM in the previous section, then it is equivalent to having ASM 4.X as the first version of `Base` and ASM 5.X as the second version of `Base`. Conflict will immediately occurs as two different versions of the same package cannot coexist in the program. Unfortunately, programming languages we have right now cannot solve this.

This problem can also further propagate the more dependencies involved in the development. Suppose other programs require the user's program, then that program will also fall into dependency hell.

Chapter 3

Programming with Version

In this chapter we will introduce an example on how we program with version.

We introduce versioned package and package interface in Batak Java. With versioned package, the version number is explicitly declared within the program and becomes an attribute of package. This also means that a type consists of not only class name and its package name, but also version number. To separate one particular version from another, after declaring the package name and package interface, we create version declarations with the class declarations in them, in the form of `version n {...}`. Each version declaration must have a corresponding package interface.

Interface or package interface in Batak Java does not have the same role as interface in Java. Package interface is the window into a package, consisting of information about declared classes with their constructors and methods, which can be seen by any importing package. The package interface of a class is written in the form of `interface n {...}`.

Example

The example introduced here is the same from the previous chapter, consisting of four different libraries: `Base`, `Education`, `Health` and `Municipal`. `Base` package consists of two versions, where the first version is used by `Education` and the second version is used by `Health`. `Municipal` then uses both `Education` and `Health` in its implementation, to show how differing version can work within the proposed system.

To grasp an easier understanding of the version numbering, the example introduced below is not written in the formal calculus (different mainly in

the way the class name is written).

```
package Base

interface 1 {
  Person {
    birth_year :: (Object) -> Object;
    new :: (Object, Object) -> Person;
  }
  Building {
    new :: (Object) -> Building;
  }
}

interface 2 {
  Person {
    birth_year :: (Object) -> Object;
    new :: (Object, Object) -> Object;
  }
  Address {
    new :: (Object, Object) -> Address;
  }
  Building {
    new :: (Address) -> Building;
  }
}
```

Listing 3.1: Package interface of `Base`

Within the interface we can find the superclass for each declared class and the methods allowed to be used by any importing package. From the above example, there are two interfaces, each corresponding to a different version of `Base`. From the interface of version 1, we can see that any importing package may use the method `birth_year` which takes a value and returns a value of class `Object`. The special name `new` represents the constructor of the class, with class `Person` requiring two values of class `Object` to instantiate and class `Building` requiring one value of class `Object`. If `new` is not declared within the interface, any importing package is not allowed to create a new object of that class, which in a sense is similar to `internal` in Java. In the interface of version 2, we can see the main differences to version with the existence of the class `Address` and its usage as the field of class `Building`.

```
version 1 {
  class Person {
    String name;
    int age;

    Person(String name, int age){
```

```
      this.name = name;
      this.age = age;
    }

    int birth_year(int current_year){
      return current_year;
    }
  }

  class Building {
    String addr;

    Building(String addr){
      this.addr = addr;
    }
  }
}

version 2 {
  class Person {
    String name;
    int age;

    Person(String name, int age){
      this.name = name;
      this.age = age;
    }

    int birth_year(int current_year){
      return current_year;
    }
  }

  class Address {
    String street;
    String number;

    Address(String street, String number){
      this.street = street;
      this.number = number;
    }
  }

  class Building {
    Address addr;

    Building(Address addr){
      this.addr = addr;
    }
```

```
  }
}
```

Listing 3.2: Version and class declarations of `Base`

From Listing 3.2, two versions of the package `Base` are declared. Both are separated by `version 1 {...}` and `version 2 {...}`. In both versions, the class `Person` is declared without any difference. In the second version, we declare a new class `Address`. In the first version, the field of the class `Building` consists of only an `addr` of type `String`, while in the second version we use the newly declared class `Address` as the field of the class `Building`.

```
package Education imports Base

interface 1 {
  Teacher <: [1]Person {
    new :: (Object, Object, Object, School) -> Teacher;
  }
  School <: [1]Building {
    change_head :: (Teacher) -> School;
    new :: (Object, Object, Teacher) -> School;
  }
}

version 1 {
  class Teacher extends [1]Person {
    String id;
    School workplace;

    Teacher(String name,
            int age,
            String id,
            School workplace)
    {
      super(name, age);
      this.id = id;
      this.workplace = workplace;
    }
  }

  class School extends [1]Building {
    String name;
    Teacher head;

    School(String addr, String name, Teacher head){
      super(addr);
      this.name = name;
      this.head = head;
```

```
    }

    void change_head(Teacher new_head){
      this.head = new_head;
    }
  }
}
```

Listing 3.3: Package `Education` with version

The package `Education` is Listing 3.3 uses the above defined package `Base` by importing it as usual. As before, we first declare an interface, since here we have only `version 1 {...}`, we only require `interface 1 {...}`. The declaration of class `Teacher` is written as `class Teacher extends [1]Person` to signify that it extends the first version of `Person`. The same goes for class `School` which extends the first version of the class `Building`. In the interface, the information on inheritance is declared together with the class name connected with `<:` to denote subclass relation.

```
package Health imports Base

interface 1 {
  Doctor <: [2]Person {
    promotion :: (Object) -> Surgeon;
    new :: (Object, Object, Object, Hospital) -> Doctor
  }
  Surgeon <: Doctor {
    new :: (Object, Object, Object, Hospital, Object)
         -> Surgeon;
  }
  Hospital <: [2]Building {
    change_head :: (Doctor) -> Hospital;
    new :: (Address, Object, Doctor) -> Hospital;
  }
}

version 1 {
  class Doctor extends [2]Person {
    String branch;
    Hospital workplace;

    Doctor(String name,
           int age,
           String branch,
           Hospital workplace)
    {
      super(name, age);
      this.branch = branch;
```

```
      this.workplace = workplace;
    }

    Surgeon promotion(int hours){
      return new Surgeon(this.name, this.age,
      this.branch, this.workplace, hours);
    }
  }

  class Surgeon extends Doctor {
    int total_hours;

    Surgeon(String name,
            int age,
            String branch,
            Hospital workplace,
            int total_hours)
    {
      super(name, age, branch, workplace);
      this.total_hours = total_hours;
    }
  }

  class Hospital extends [2]Building {
    String name;
    Doctor head;

    Hospital(Address addr,
             String name,
             Doctor head)
    {
      super(addr);
      this.name = name;
      this.head = head;
    }

    void change_head(Doctor new_head){
      this.head = new_head;
    }
  }
}
```

Listing 3.4: Package `Health` with version

Similar to `Education` package, the `Health` package defined above also imports `Base`. It differs in the way that the classes in this package extend the second version of the classes in `Base`.

It should be noted that both `Education` and `Health` do not import a specific version of `Base`. The package imports the whole `Base` and then

selectively choose which version it wishes to use. It means that one of the classes in the package may use the first version of `Base` and another class may use the second version of `Base`.

```
package Municipal imports Education, Health

interface 1 {
  Pair {
    new :: (Teacher, Doctor) -> Pair;
  }
  Manage <: Object {
    pair_heads :: (School, Hospital) -> Pair;
    new :: () -> Manage;
  }
}

version 1 {
  class Pair extends Object {
    Teacher nameA;
    Doctor nameB;

    Pair(Teacher nameA, Doctor nameB){
      this.nameA = nameA;
      this.nameB = nameB;
    }
  }

  class Manage extends Object {
    Manage(){}

    Pair pair_heads(School a, Hospital b){
      return new Pair(a.head, b.head);
    }
  }
}
```

Listing 3.5: Package `Municipal` with version

The `Municipal` package imports `Education` and `Health`, both of which utilize different versions of `Base` in its implementation. First, there will be no conflict arising from the existence of two different `Person` or `Building`, because they have already been distinguished with differing version number.

In the method `pair_buildings` in the class `Manage`, we can see how the class `School` and `Hospital` can still interact with each other in the same expression `return new Pair(a.head, b.head);` even though both classes extend a different class `Building`, where `School` extends the first version and `Hospital` extends the second version.

Chapter 4

Calculus

4.1 Syntax

Formal Syntax

Package declaration

$$P ::= \texttt{package}\ N\ \texttt{imports}\ \overline{Q}\ \{\overline{IL}\ \overline{V}\}$$

Interface declaration

$$IL ::= \texttt{interface}\ n\ \{\overline{IC}\}$$

Version declaration

$$V ::= \texttt{version}\ n\ \{\overline{CL}\}$$

Class declaration

$$CL ::= \texttt{class}\ C\ \texttt{extends}\ DV\ \{\overline{CV}\ \overline{f};\ K\ \overline{M}\}$$

Constructor declaration

$$K ::= C(\overline{DV}\ \overline{g}, \overline{CV}\ \overline{f})\ \{\texttt{super}(\overline{g}),\ \texttt{this}.\overline{f} = \overline{f}\}$$

Method declaration

$$M ::= CV\ m(\overline{CV}\ \overline{x})\{\ \texttt{return}\ t;\}$$

Class name

$$CV, DV, EV ::= [n]N.C \mid C$$

Interface's class

$$IC ::= C <: DV\{\overline{IM}, \texttt{new} :: \overline{CV} \to C\}$$

Interface's method

$$IM ::= m :: \overline{CV} \to CV$$

Expression

$$e ::= t\ \texttt{in version}\ n\ \texttt{of package}\ N$$

Terms

$$t ::= x \mid t.f \mid t.m(\bar{t}) \mid \texttt{new}\ [n]N.C(\bar{t}) \mid (CV)\ t \mid \texttt{new Object()}$$

Values

$$v ::= \texttt{new}\ [n]N.C(\overline{v}) \mid \texttt{new Object()}$$

Subtyping

Reflexivity

$$CV <: CV$$

Transitivity

$$\frac{CV <: DV \qquad DV <: EV}{CV <: EV}$$

Inheritance (Same Package)

$$\frac{CT([n]N.C) = \texttt{class}\ C\ \texttt{extends}\ D...}{[n]N.C <: [n]N.D}$$

Inheritance (Different Package)

$$\frac{CT([n]N.C) = \texttt{class}\ C\ \texttt{extends}\ [m]Q.D...}{[n]N.C <: [m]Q.D}$$

The concrete syntax of Batak Java is given above. The metavariable N and Q range over package names; n ranges over version numbers; C, D range over class names without version and package name; f, g range over field names; m ranges over method names; t ranges over terms; x ranges over variable names. `this` is a special reserved variable that is implicitly bound in every method declaration.

$\overline{Q}$ is shorthand for a possibly empty sequence $Q_1, ..., Q_n$, similarly also with $\overline{CV}$, $\overline{CL}$, $\overline{t}$, and others. Pairs of sequences such as "$\overline{IL}\ \overline{V}$" signifies "$IL_1\ V_1, ..., IL_n\ V_n$" where n is the length of $\overline{IL}$ and $\overline{V}$. Sequences of field declarations, argument names and method declarations in the same version of the same package are assumed to contain no duplicate names. Concatenation is denoted by a comma.

The syntax is mostly similar with Featherweight Java. However, because Batak Java introduces versioned package, the differences with the syntax of Featherweight Java can be found in the existence of package declaration and version declaration. Consequently, a program in Batak Java is made of package declarations and an expression to be evaluated.

Due to the additional attribute of package, we also require a proper way to express type/class. A class with version number and package name is written as $[n]N.C$ where n denotes version number, N denotes package name and C denotes class name. The other distinguishing aspect is the interface declaration which has a different semantic compared to full Java.

The package declaration $P ::= \texttt{package}\ N\ \texttt{imports}\ \overline{Q}\ \{\overline{IL}\ \overline{V}\}$ creates a new package N that imports a set of packages $\overline{Q}$. For every version declaration V declared in the package, there must be a corresponding interface declaration IL. Importing $\overline{Q}$ implies that within the package N, every classes and their behaviors declared within $\overline{Q}$'s interfaces can be used in package N.

The interface declaration $IL ::= \texttt{interface}\ n\ \{\overline{IC}\}$ introduces the interface for classes declared in version n within the package. n denotes the

version number corresponding to this interface. Unlike full Java, interface in Batak Java specifies which behaviors (superclass, constructor and methods) can be viewed by an importing package.

The version declaration $V ::= \texttt{version}\ n\ \{\overline{CL}\}$ introduces the version n of the package. The class declarations $\overline{CL}$ signify that $\overline{CL}$ are defined within version n.

The class declaration $CL ::= \texttt{class}\ C\ \texttt{extends}\ DV\ \{\overline{CV}\ \overline{f};\ K\ \overline{M}\}$ introduces a class named C with superclass DV. The new class C has fields $\overline{f}$ with types $\overline{CV}$, a constructor K and a set of methods $\overline{M}$. The class name C must also be distinct from every class defined within the same package and version. As mentioned above, DV may be written in the form of class name as D, only if C and D are declared in the same package and version. In other cases, DV must be written in the form of $[n]N.D$. DV also cannot be a class from a different version of the same package. The fields defined in C will be added to the fields which are already declared in DV and its superclasses; they should have distinct names from the superclasses' fields. The methods of C may override methods from DV or we can also create new methods special to the class C.

The constructor declaration $K ::= C(\overline{DV}\ \overline{g}, \overline{CV}\ \overline{f})\ \{\texttt{super}(\overline{g}), \texttt{this}.\overline{f} = \overline{f}\}$ represents the initialization of fields of C. $\overline{g}$ and their types $\overline{DV}$ must correspond with all fields of the superclass of C; $\overline{f}$ and their types $\overline{CV}$ must correspond with fields that were introduced in the class declaration of C. $\texttt{super}(\overline{g})$ is a special function that calls the constructor of the superclass of C, while $\texttt{this}.\overline{f} = \overline{f}$ will initialize the fields declared in C. In other words, all the fields must be initialized during the creation of an object of class C.

The method declaration $M ::= CV\ m(\overline{CV}\ \overline{x})\{\ \texttt{return}\ t;\}$ introduces a method named m with a result type CV and arguments $\overline{x}$ of types $\overline{CV}$. In the method body return t, `this` and $\overline{x}$ are bound.

The class name CV is defined in two forms: $[n]N.C$ and C. $[n]N.C$ represents class C in the version n of package N. We can only use the simple class name C in two cases, which are during declaration of a new class and extending a class from the same package and version, both in interface declaration and class declaration. Apart from those two cases, the calculus only allows CV to be written in the form of $[n]N.C$.

The interface class $IC ::= C <: DV\{\overline{IM}, \texttt{new} :: \overline{CV} \to C\}$ adds a class named C into the interface. $C <: DV$ signifies that class C is a subclass of DV. DV can written in the form of simple class name as D only if it is also declared in the same package and version. In other cases, DV must be written in the form of $[n]N.D$. DV also cannot be a class from a different version of the same package. $\overline{Y}$ are the methods with their types, while `new`

represents the constructor of the class C and $\overline{CV}$ in $\overline{CV} \rightarrow C$ represents the fields' types of C. Declaring all methods and the constructor of C in its interface is not obligatory.

The interface method $IM ::= m :: \overline{CV} \rightarrow CV$ adds the method named m into the set of methods that an importing package may use. $\overline{CV}$ are the types of arguments of the method and CV is its result type.

Expression is written as t `in version` n `of package` N. This means that the term t will be evaluated on the basis that it exists with version n of package N.

Terms consist of x for variable, $t.f$ for field access, $t.m(\bar{t})$ for method invocation, `new` $[n]N.C(\bar{t})$ for object creation, $(CV)\ t$ for casting and `new Object()` to create an object of class `Object`. `new Object()` needs to be separately defined because it does not belong to any particular package or version and takes the role of the superclass for every class declared.

Values consist of object creation `new` $[n]N.C(\overline{v})$ and `new Object()`.

A package table PT is a mapping from package name N to its package declaration P. An interface table IT is a mapping from a class name $[n]N.C$ to its interface declaration IL. A version table VT is a mapping from a pair of version number and package name $[n]N$ to the sequence of class $\overline{CL}$ declared in that particular version and package. A class table CT is a mapping from a class name $[n]N.C$ to class declaration CL. The definition of the class `Object` cannot be found in class table. A program consists of the tuple (PT, IT, VT, CT, e) of a package table, interface table, version table, class table and an expression e to be evaluated.

Package table is assumed to satisfy some conditions: (**1**) $PT(N) =$ `package` N `imports` ... for every $N \in dom(PT)$; (**2**) for every version declaration V in $PT(N)$, there is a corresponding interface declaration IL with the same version number in $PT(N)$, e.g., version 2 {...} has a corresponding interface 2 {...}; (**3**) there are no cycles in import relation induced by PT.

From Featherweight Java, class table is assumed to satisfy some conditions: (**1**) $CT([n]N.C) =$ class $[n]N.C$ for every $[n]N.C \in dom(CT)$; (**2**) `Object` $\notin dom(CT)$; (**3**) for every class name $[n]N.C$ (except `Object`) in CT, we also have $[n]N.C \in dom(CT)$; (**4**) there are no cycles in subtype relation induced by CT.

The subtype relation can be figured out from the class table, separated into two cases, superclass in the same package and superclass in a different package. $CV <: DV$ signifies that CV is the subtype of DV. The subtyping is reflexive and transitive. The formal definition for subtyping is shown above.

Auxillary Definitions

Field Lookup $\left[fields(CV) = \overline{CV}\ \overline{f}\right]$

Object

$$fields(\texttt{Object}) = \bullet$$

Superclass is Object

$$\frac{\begin{array}{c} VT([n]N) = \overline{CL} \text{ where } \texttt{class}\ C\ \texttt{extends}\ ... \in \overline{CL} \\ CT([n]N.C) = \texttt{class}\ C\ \texttt{extends}\ \texttt{Object}\ \{\overline{CV}\ \overline{f};\ K\ \overline{M}\} \end{array}}{fields([n]N.C) = \overline{CV}\ \overline{f}}$$

Superclass in the same package

$$\frac{\begin{array}{c} VT([n]N) = \overline{CL} \text{ where } \texttt{class}\ C\ \texttt{extends}\ ... \in \overline{CL} \\ CT([n]N.C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{CV}\ \overline{f};\ K\ \overline{M}\} \\ fields([n]N.D) = \overline{DV}\ \overline{g} \end{array}}{fields([n]N.C) = \overline{DV}\ \overline{g}, \overline{CV}\ \overline{f}}$$

Superclass in a different package

$$\frac{\begin{array}{c} VT([n]N) = \overline{CL} \text{ where } \texttt{class}\ C\ \texttt{extends}\ ... \in \overline{CL} \\ CT([n]N.C) = \texttt{class}\ C\ \texttt{extends}\ [v]Q.D\ \{\overline{CV}\ \overline{f};\ K\ \overline{M}\} \quad N \neq Q \\ PT(N) = \texttt{package}\ N\ \texttt{imports}\ \overline{N}\ \{...\} \quad Q \in \overline{N} \\ ifields([v]Q.D) = \overline{DV}\ \overline{g} \end{array}}{fields([n]N.C) = \overline{DV}\ \overline{g}, \overline{CV}\ \overline{f}}$$

Interface Field Lookup $\left[ifields(CV) = \overline{CV}\ \overline{f}\right]$

$$\frac{IT([n]N.C) = C <: DV\ \{\overline{Y};\ \texttt{new} :: (\overline{CV}) \to C\}}{ifields([n]N.C) = \overline{CV}\ \overline{f}}$$

Method Type Lookup $\left[mtype(m, CV) = \overline{BV} \to BV\right]$

Method defined in the class

$$\frac{VT([n]N) = \overline{CL} \text{ where } \texttt{class}\ C\ \texttt{extends}\ ... \in \overline{CL} \quad CT([n]N.C) = \texttt{class}\ C\ \texttt{extends}\ DV\ \{\overline{CV}\ \overline{f};\ K\ \overline{M}\} \quad BV\ m(\overline{BV}\ \overline{x})\{\ \texttt{return}\ t;\} \in \overline{M}}{mtype(m, [n]N.C) = \overline{BV} \to BV}$$

Method defined in the superclass belonging to the same package

$$\frac{VT([n]N) = \overline{CL} \text{ where } \texttt{class}\ C\ \texttt{extends}\ ... \in \overline{CL} \quad CT([n]N.C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{CV}\ \overline{f};\ K\ \overline{M}\} \quad m \text{ is not defined in } \overline{M}}{mtype(m, [n]N.C) = mtype(m, [n]N.D)}$$

Method defined in the superclass belonging to a different package

$$\frac{VT([n]N) = \overline{CL} \text{ where } \texttt{class}\ C\ \texttt{extends}\ ... \in \overline{CL} \quad CT([n]N.C) = \texttt{class}\ C\ \texttt{extends}\ [v]Q.D\ \{\overline{CV}\ \overline{f};\ K\ \overline{M}\} \quad N \neq Q \quad m \text{ is not defined in } \overline{M} \quad PT(N) = \texttt{package}\ N\ \texttt{imports}\ \overline{N}\ \{...\} \quad Q \in \overline{N}}{mtype(m, [n]N.C) = imtype(m, [v]Q.D)}$$

Interface Method Type Lookup $\left[imtype(m, CV) = \overline{BV} \to BV\right]$

Method found in the interface

$$\frac{IT([n]N.C) = C <: DV\ \{\overline{Y};\ \texttt{new} :: \overline{CV} \to C\} \quad m :: \overline{BV} \to BV \in \overline{Y}}{imtype(m, [n]N.C) = \overline{BV} \to BV}$$

Method in the superclass belonging to the same package

$$\frac{IT([n]N.C) = C <: D\ \{\overline{Y};\ \texttt{new} :: \overline{CV} \to C\} \quad \text{Type of } m \text{ is not defined in } \overline{Y}}{imtype(m, [n]N.C) = imtype(m, [n]N.D)}$$

Method in the superclass belonging to a different package

$$\frac{IT([n]N.C) = C <: [v]Q.D\ \{\overline{Y};\ \texttt{new} :: \overline{CV} \to C\} \quad N \neq Q \quad \text{Type of } m \text{ is not defined in } \overline{Y} \quad PT(N) = \texttt{package}\ N\ \texttt{imports}\ \overline{N}\ \{...\} \quad Q \in \overline{N}}{imtype(m, [n]N.C) = imtype(m, [v]Q.D)}$$

Method Body Lookup $\left[mbody(m, CV) = (\overline{x}, t)\right]$

Method defined in the class

$$\frac{VT([n]N) = \overline{CL} \text{ where } \texttt{class}\ C\ \texttt{extends}\ ... \in \overline{CL} \quad CT([n]N.C) = \texttt{class}\ C\ \texttt{extends}\ DV\ \{\overline{CV}\ \overline{f};\ K\ \overline{M}\} \quad BV\ m(\overline{BV}\ \overline{x})\{\ \text{return}\ t;\} \in \overline{M}}{mbody(m, [n]N.C) = (\overline{x}, t)}$$

Method defined in the superclass belonging to the same package

$$\frac{VT([n]N) = \overline{CL} \text{ where } \texttt{class}\ C\ \texttt{extends}\ ... \in \overline{CL} \quad CT([n]N.C) = \texttt{class}\ C\ \texttt{extends}\ DV\ \{\overline{CV}\ \overline{f};\ K\ \overline{M}\} \quad m \text{ is not defined in } \overline{M}}{mbody(m, [n]N.C) = mbody(m, [n]N.D)}$$

Method defined in the superclass belonging to a different package

$$\frac{VT([n]N) = \overline{CL} \text{ where } \texttt{class}\ C\ \texttt{extends}\ ... \in \overline{CL} \quad CT([n]N.C) = \texttt{class}\ C\ \texttt{extends}\ [v]Q.D\ \{\overline{CV}\ \overline{f};\ K\ \overline{M}\} \quad N \neq Q \quad m \text{ is not defined in } \overline{M} \quad PT(N) = \texttt{package}\ N\ \texttt{imports}\ \overline{N}\ \{...\} \quad Q \in \overline{N}}{mtype(m, [n]N.C) = imbody(m, [v]Q.D)}$$

Method Body Lookup (Different Package) $\left[imbody(m, CV) = (\overline{x}, t)\right]$

Method found in the interface

$$\frac{IT([n]N.C) = C <: DV\ \{\overline{Y};\ \texttt{new} :: \overline{CV} \to C\} \quad m :: \overline{BV} \to BV \in \overline{Y}}{imbody(m, [n]N.C) = mbody(m, [n]N.C)}$$

Method in the superclass belonging to the same package

$$\frac{IT([n]N.C) = C <: D\ \{\overline{Y};\ \texttt{new} :: \overline{CV} \to C\} \quad \text{Type of } m \text{ is not defined in } \overline{Y}}{imbody(m, [n]N.C) = imbody(m, [n]N.D)}$$

Method in the superclass belonging to a different package

$$\frac{IT([n]N.C) = C <: [v]Q.D\ \{\overline{Y};\ \texttt{new} :: \overline{CV} \to C\} \quad N \neq Q \quad \text{Type of } m \text{ is not defined in } \overline{Y} \quad PT(N) = \texttt{package}\ N\ \texttt{imports}\ \overline{N}\ \{...\} \quad Q \in \overline{N}}{imbody(m, [n]N.C) = imbody(m, [v]Q.D)}$$

Valid Method Overriding

$$\frac{mtype(m, DV) = \overline{DV} \to DV_0 \text{ implies } \overline{CV} = \overline{DV} \text{ and } CV_0 = DV_0}{override(m, DV, \overline{CV} \to CV_0)}$$

$$\frac{imtype(m, DV) = \overline{DV} \to DV_0 \text{ implies } \overline{CV} = \overline{DV} \text{ and } CV_0 = DV_0}{ioverride(m, DV, \overline{CV} \to CV_0)}$$

For the typing and reduction rules, we also require a few auxillary definitions as given above.

The fields of a class $fields(CV)$ is a sequence of $\overline{CV}\ \overline{f}$ pairs. f denotes the field name and CV is its type. `Object` is a special class without any field, denoted by $\bullet$. Aside from $fields(\texttt{Object})$, fields of a class is always written with version and package in the form of $fields([n]N.C)$. Since every class extends a different class, finding $fields([n]N.C)$ depends on whether the superclass of CV belongs in the same package or not. In the case of superclass in the same package, the superclass is declared in a simple class name D and must also be declared in the same package and version. In the case of extending a class from a different package, we first check whether the other package has been imported or not, then we use $ifields([v]Q.D)$. Unlike $fields(CV)$ which checks the class table, $ifields(CV)$ goes directly into the package interface to check the fields. $ifields(CV)$ checks that (`new` :: ...) is included in the interface table.

The type of a method m in class CV is written as $mtype(m, CV)$. It is a pair, written as $\overline{BV} \to BV$, of a sequence of argument types $\overline{BV}$ and

a result type BV. If the method m is not defined within the class $[n]N.C$, as in the case of field lookup, we have to separate the method type lookup into two different cases, superclass in the same package and superclass in different package. In the former, it is straightforward recursion. In the latter, we first need to check the package table before performing an interface method type lookup. Unlike $mtype(m, CV)$ which checks the class table, the interface method type lookup $imtype(m, CV)$ checks only package interface. Therefore, a package only interacts with the interface of other packages in type checking. A similar check is performed as in the case of $mtype(m, CV)$ but only by looking through the interface table.

Similarly the body of m in class CV is written as $mbody(m, CV)$ consisting of $\overline{x}, t$ pair of a sequence of argument types $\overline{x}$ and term t. The method body lookup is performed in a similar manner as method type lookup that branches into interface method body lookup in the case of superclass from a different package.

Additionally, since `Object` doesn't have methods, $mtype(m,$ `Object`$)$, $imtype(m,$ `Object`$)$, $mbody(m,$ `Object`$)$ and $imbody(m,$`Object`$)$ are undefined.

$override(m, DV, \overline{CV} \to CV_0)$ determines whether a method m with argument types $\overline{CV}$ and result type CV_0 can be declared in the subclass of DV which belongs to the same package and version. $ioverride(m, DV, \overline{CV} \to CV_0)$ determines whether a method m can be defined in the subclass of DV which belongs to a different package. If a method with the same name is declared in the superclass, then its signature must be identical.

4.2 Reduction

Exp-Red

$$\frac{t_0 \to t_0'}{t_0 \texttt{ in version } n \texttt{ of package } N \to t_0' \texttt{ in version } n \texttt{ of package } N}$$

E-ProjNewS

$$\frac{fields([n]N.C) = \overline{CV}\ \overline{f} \qquad sl = [n]N}{\big(\texttt{new } [n]N.C(\overline{v})\big).f_i \to v_i}$$

E-ProjNewD

$$\frac{ifields([n]N.C) = \overline{CV}\ \overline{f} \qquad sl = [v]Q}{\big(\texttt{new}\ [n]N.C(\overline{v})\big).f_i \to v_i}$$

E-InvkNewS

$$\frac{mbody(m, [n]N.C) = (\overline{x}, t_0) \qquad sl = [n]N}{\big(\texttt{new}\ [n]N.C(\overline{v})\big).m(\overline{u}) \to \big[\overline{x} \mapsto \overline{u},\ \texttt{this} \mapsto \big(\texttt{new}\ [n]N.C(\overline{v})\big)\big]t_0}$$

E-InvkNewD

$$\frac{imbody(m, [n]N.C) = (\overline{x}, t_0) \qquad sl = [v]Q}{\big(\texttt{new}\ [n]N.C(\overline{v})\big).m(\overline{u}) \to \big[\overline{x} \mapsto \overline{u},\ \texttt{this} \mapsto \big(\texttt{new}\ [n]N.C(\overline{v})\big)\big]t_0}$$

E-Field

$$\frac{t_0 \to t_0'}{t_0.f \to t_0'.f}$$

E-InvkRecv

$$\frac{t_0 \to t_0'}{t_0.m(\overline{t}) \to t_0'.m(\overline{t})}$$

E-InvkArg

$$\frac{t_i \to t_i'}{v_0.m(\overline{v}, t_i, \overline{t}) \to v_0.m(\overline{v}, t_i', \overline{t})}$$

E-NewArg

$$\frac{t_i \to t_i'}{\texttt{new}\ [n]N.C(\overline{v}, t_i, \overline{t}) \to \texttt{new}\ [n]N.C(\overline{v}, t_i', \overline{t})}$$

E-CastNew

$$\frac{[n]N.C <: DV}{(DV)\ \texttt{new}\ [n]N.C(\overline{v}) \to \texttt{new}\ [n]N.C(\overline{v})}$$

E-Cast

$$\frac{t_0 \to t_0'}{(CV)\ t_0 \to (CV)\ t_0'}$$

The evaluation rules are in small-step semantics. Expression simply reduces into the term it contains through Exp-Red. The reduction relation for terms is in the form of $t \to t'$ which reads as "term t reduces to term t' in one step." The reduction rules are mostly identical to the evaluation rules of Featherweight Java with the difference lies in separating E-ProjNewS from E-ProjNewD as well as E-InvkNewS from E-InvkNewD. E-ProjNewS is used on object created from class declared in the same package, while E-ProjNewD is used for object created from class imported from a different package.

The evaluation rules are given above. There are three reduction rules: E-ProjNewS and E-ProjNewD for field access, E-InvkNewS and E-InvkNewD for method invocation and E-CastNew for casting. The notation $\left[\overline{x} \to \overline{u},\ \texttt{this} \to \left(\texttt{new}\ [n]N.C(\overline{v})\right)\right]$ denotes substitution of x_1 with $u_1, ..., x_n$ with u_n and `this` with `new` $[n]N.C(\overline{v})$. The five remaining rules of E-Field, E-InvkRecv, E-InvkArg, E-NewArg and E-Cast are congruence rules.

4.3 Typing

Exp-Typ

$$\frac{sl = [n]N,\ \Gamma \vdash t_0 : CV}{\Gamma \vdash t_0\ \texttt{in version}\ n\ \texttt{of package}\ N : CV}$$

T-Var

$$\frac{x : CV \in \Gamma}{sl,\ \Gamma \vdash x : CV}$$

T-FieldS

$$\frac{sl = [n]N,\ \Gamma \vdash t_0 : [n]N.C_0 \qquad fields([n]N.C_0) = \overline{CV}\ \overline{f}}{sl = [n]N,\ \Gamma \vdash t_0.f_i : CV_i}$$

T-FieldD

$$\frac{sl = [v]Q,\ \Gamma \vdash t_0 : [n]N.C_0 \qquad ifields([n]N.C_0) = \overline{CV}\ \overline{f}}{sl = [v]Q,\ \Gamma \vdash t_0.f_i : CV_i}$$

T-InvkS

$$\frac{sl = [n]N,\ \Gamma \vdash t_0 : [n]N.C_0 \quad mtype(m, [n]N.C_0) = \overline{DV} \to DV \quad sl = [n]N,\ \Gamma \vdash \overline{t} : \overline{CV} \quad \overline{CV} <: DV}{sl = [n]N,\ \Gamma \vdash t_0.m(\overline{t}) : DV}$$

T-InvkD

$$\frac{sl = [v]Q,\ \Gamma \vdash t_0 : [n]N.C_0 \quad imtype(m, [n]N.C_0) = \overline{DV} \to DV \quad sl = [v]Q,\ \Gamma \vdash \overline{t} : \overline{CV} \quad \overline{CV} <: DV}{sl = [v]Q,\ \Gamma \vdash t_0.m(\overline{t}) : DV}$$

T-NewS

$$\frac{fields([n]N.C) = \overline{DV}\ \overline{f} \quad sl = [n]N,\ \Gamma \vdash \overline{t} : \overline{CV} \quad \overline{CV} <: \overline{DV}}{sl = [n]N,\ \Gamma \vdash \texttt{new}\ [n]N.C(\overline{t}) : [n]N.C}$$

T-NewD

$$\frac{PT(Q) = \texttt{package}\ Q\ \texttt{imports}\ \overline{N}\ \{\cdots\} \text{ where } N \in \overline{N} \quad ifields([n]N.C) = \overline{DV}\ \overline{f} \quad sl = [v]Q,\ \Gamma \vdash \overline{t} : \overline{CV} \quad \overline{CV} <: \overline{DV}}{sl = [v]Q,\ \Gamma \vdash \texttt{new}\ [n]N.C(\overline{t}) : [n]N.C}$$

T-UCast

$$\frac{sl,\ \Gamma \vdash t_0 : DV \quad DV <: CV}{sl,\ \Gamma \vdash (CV)\ t_0 : CV}$$

T-DCast

$$\frac{sl,\ \Gamma \vdash t_0 : DV \quad CV <: DV \quad CV \neq DV}{sl,\ \Gamma \vdash (CV)\ t_0 : CV}$$

T-SCast

$$\frac{sl, \Gamma \vdash t_0 : DV \qquad CV \not<: DV \qquad DV \not<: CV \quad \text{stupid warning}}{sl, \Gamma \vdash (CV)\ t_0 : CV}$$

T-Sub

$$\frac{sl, \Gamma \vdash t : CV \qquad CV <: DV}{sl, \Gamma \vdash t : DV}$$

Constructor typing

$$\frac{\begin{array}{c} CT([n]N.C) = \texttt{class}\ C\ \texttt{extends}\ DV\ \{\overline{CV}\ \overline{f} ...\} \\ fields([n]N.C) = \overline{DV}\ \overline{g}, \overline{CV}\ \overline{f} \\ {}^{\forall}[v]Q.D \in \overline{CV} \cup DV \text{ where } Q = N,\ v = n \\ {}^{\forall}[v]Q.D \in \overline{CV} \cup DV \text{ where } Q \neq N, \\ PT(N) = \texttt{package}\ N\ \texttt{imports}\ \overline{N}\ \{\cdots\} \text{ where } Q \in \overline{N} \end{array}}{C(\overline{DV}\ \overline{g}, \overline{CV}\ \overline{f})\{\texttt{super}(\overline{g});\ \texttt{this}.\overline{f} = \overline{f};\} \text{ OK in } [n]N.C}$$

Method typing

Method-I

$$\frac{\begin{array}{c} sl = [n]N, \overline{x} : \overline{EV}, \texttt{this} : [n]N.C \vdash t_0 : EV_0 \qquad EV_0 <: EV \\ CT([n]N.C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{...\} \\ override(m, [n]N.D, \overline{EV} \to EV) \\ {}^{\forall}[v]Q.D \in \overline{EV} \cup EV \text{ where } Q = N,\ v = n \\ {}^{\forall}[v]Q.D \in \overline{EV} \cup EV \text{ where } Q \neq N, \\ PT(N) = \texttt{package}\ N\ \texttt{imports}\ \overline{N}\ \{\cdots\} \text{ where } Q \in \overline{N} \end{array}}{EV\ m(\overline{EV}\ \overline{x})\{\text{ return } t_0;\} \text{ OK in } [n]N.C}$$

Method-II

$$sl = [n]N,\, \overline{x} : \overline{EV},\, \texttt{this} : [n]N.C \vdash t_0 : EV_0 \qquad EV_0 <: EV$$
$$CT([n]N.C) = \texttt{class}\ C\ \texttt{extends}\ [v]Q.D\ \{...\}$$
$$PT(N) = \texttt{package}\ N\ \texttt{imports}\ \overline{N}\ \{...\} \quad Q \in \overline{N}$$
$$ioverride(m, [v]Q.D, \overline{EV} \to EV)$$
$$^{\forall}[v]Q.D \in \overline{EV} \cup EV \text{ where } Q = N,\, v = n$$
$$^{\forall}[v]Q.D \in \overline{EV} \cup EV \text{ where } Q \neq N,$$
$$\frac{PT(N) = \texttt{package}\ N\ \texttt{imports}\ \overline{N}\ \{\cdots\} \text{ where } Q \in \overline{N}}{EV\ m(\overline{EV}\ \overline{x})\{\text{ return } t_0;\} \text{ OK in } [n]N.C}$$

Class typing

$$\frac{\begin{array}{c} K \text{ OK in } [n]N.C \\ \overline{M} \text{ OK in } [n]N.C \end{array}}{\texttt{class}\ C\ \texttt{extends}\ DV\ \{\overline{CV}\ \overline{f}; K\ \overline{M}\} \text{ OK in version } n \text{ of package } N}$$

Interface well-definedness

Interface's method

$$\frac{mtype(m, [n]N.C) = \overline{BV} \to BV}{m :: \overline{BV} \to BV \text{ OK in interface of } [n]N.C}$$

Interface's class

$$\frac{\begin{array}{c} CT([n]N.C) = \texttt{class}\ C\ \texttt{extends}\ DV\ \{...\} \\ fields([n]N.C) = \overline{CV}\ \overline{f} \\ \overline{Y} \text{ OK in interface of } [n]N.C \end{array}}{C <: DV\ \{\overline{Y}; \texttt{new} :: \overline{CV} \to C\} \text{ OK in interface of } [n]N}$$

$$\frac{\begin{array}{c} CT([n]N.C) = \texttt{class}\ C\ \texttt{extends}\ DV\ \{...\} \\ fields([n]N.C) = \overline{CV}\ \overline{f} \\ \overline{Y} \text{ OK in interface of } [n]N.C \end{array}}{C <: DV\ \{\overline{Y}\} \text{ OK in interface of } [n]N}$$

The typing rules for expression, terms, constructor declarations, method declarations, class declarations and interface declarations are shown above. A typing environment Γ is a finite mapping from variables to types, written as $\overline{x} : \overline{CV}$. *sl* is short for *source location* which denotes the context (version number and package name) in which a term is supposed to be evaluated. The context *sl* for term typing is extracted from the expression through the Exp-Typ rule. Typing judgement for terms has the form $sl, \Gamma \vdash t : CV$ which reads as "in the typing environment Γ and context *sl*, the term t has type CV." The term typing rules are identical to the term typing in Featherweight Java only with CV replacing C and the introduction of *sl*. The typing rule T-FieldS and T-FieldD, only differ in the lookup used, in which T-FieldS uses $fields(CV)$ whereas T-FieldD uses $ifields(CV)$. The same applies for T-InvkS and T-InvkD, as well as for T-NewS and T-NewD.

The significant differences with Featherweight Java lie in the typing judgment for constructor, method, class and interface declarations. The typing judgment for constructor declarations has the form K OK in $[n]N.C$ which reads "constructor declaration K is ok when it occurs within class $[n]N.C$." The main point of the typing is to check whether the types $\overline{CV}$ of the fields $\overline{f}$ and the superclass DV are conflicting with version n of package N or not. Not conflicting implies that if there is any class among $\overline{CV}$ which comes from the same package N, then it must come from the same version n, because it would be naturally strange to use a class defined in the different version of the package. If the type of the field or superclass is from a different package, we check $PT(N)$ to see if the package has been imported or not. This is done by the last three lines before the conclusion. This also shows that the fields of a class can be quite flexible as we can even introduce two fields from a single different package but differing in version number. At a glance it seems counterintuitive to perform field lookup on $[n]N.C$. However, in the case of the class having a superclass belonging to a different package, we need to also check the package table N along with the interface table of DV to see whether the constructor is available or not, and field lookup does exactly this.

The typing judgment for method declarations has the form M OK in $[n]N.C$ which reads "method declaration M is ok when it occurs within class $[n]N.C$." We use the term typing on the body of the method, where the free variables are the arguments of the method and the special variable this with type $[n]N.C$. In case of overriding, if a method with the same name exists within the superclass, then its type must be the same. As before, there are two different cases for method typing: superclass in the same package and a different package. The similar point in both cases is that we need to ensure

that the argument types and the result type of the method are not conflicting with package and version where it is declared. The first case simply uses $override(m, [n]N.D, \overline{EV} \rightarrow EV)$ for overriding. The second case however requires a check on the package table and uses $ioverride(m, [v]Q.D, \overline{EV} \rightarrow EV)$ which directly sees the content of the interface of the $[v]Q.D$.

The typing judgment for class declaration has the form CL OK in version n of package N which reads "class declaration CL is ok when it occurs within version n of package N." We simply combine the constructor typing and methods typing to arrive at the conclusion that the class declaration is also acceptable.

The well-definedness of interface is inductively checked from the methods to the class. The typing judgement for interface's method has the form IM OK in interface of $[n]N.C$ which reads "the interface's method IM is ok when it occurs within the class C in the interface of version n of package N." It simply checks whether the interface method corresponds with *mtype* or not.

The typing judgement for interface's class has the form IC OK in interface of $[n]N.C$ which reads "the interface's class IC is ok when it occurs within the interface of version n of package N." It checks whether the interface's methods are OK and the fields correspond with the constructor (`new`). As mentioned before, by not including constructor or a certain method in the interface, any importing package would not have access to object creation or a method of that particular class.

A version declaration is well defined, if every class declarations within it are well-defined. The same also applies to interface declarations, where an interface declaration is well-defined if all the interface's classes within it are well-defined.

4.4 Soundness

4.4.1 Progress

Theorem [Progress] Suppose $e = t$ `in version` n `of package` N is a well-typed expression. Then either (1) t is a value, (2) there is some t' with $t \rightarrow t'$, or (3) t contains (CV) `new` $DV(\overline{v})$ where $DV \not<: CV$ in which the evaluation will get stuck.

Proof. By induction on the typing of derivation of t.

Case T-Var

Impossible for t to be typed with T-Var without any typing environment.

Case T-FieldS

Suppose that t is typed with the T-FieldS rule,

$$\frac{sl = [n]N \vdash t_0 : [n]N.C_0 \qquad fields([n]N.C_0) = \overline{CV}\ \overline{f}}{sl = [n]N \vdash t_0.f_i : CV_i}\ \text{(T-FieldS)}$$

from the induction hypothesis then either there exists t_0' such that $t_0 \to t_0'$, t_0 is a value, or t_0 stuck because it contains (CV) `new` $DV(\overline{v})$ where $DV \not<: CV$.

If t_0 stuck then t also stuck because it contains (CV) `new` $DV(\overline{v})$ where $DV \not<: CV$.

If t_0 is a value, then from the definition of value, t_0 is in the form of `new` $[n]N.C(\overline{v})$. Using E-ProjNewS,

$$\frac{fields([n]N.C) = \overline{CV}\ \overline{f} \qquad sl = [n]N}{\big(\texttt{new}\ [n]N.C(\overline{v})\big).f_i \to v_i}\ \text{(E-ProjNewS)}$$

we can see that there exists $t' = v_i$. t_0 cannot of the form `new Object()` because it does not have any fields.

If t_0 is not a value, then by using E-Field,

$$\frac{t_0 \to t_0'}{t_0.f_i \to t_0'.f_i}\ \text{(E-Field)}$$

there exists $t' = t_0'.f_i$

Case T-FieldD

Suppose that t is typed with the T-FieldD rule,

$$\frac{sl = [v]Q \vdash t_0 : [n]N.C_0 \qquad ifields([n]N.C_0) = \overline{CV}\ \overline{f}}{sl = [v]Q \vdash t_0.f_i : CV_i}\ \text{(T-FieldD)}$$

from the induction hypothesis then either there exists t_0' such that $t_0 \to t_0'$, t_0 is a value, or t_0 stuck because it contains (CV) `new` $DV(\overline{v})$ where $DV \not<: CV$.

If t_0 stuck then t also stuck because it contains (CV) `new` $DV(\overline{v})$ where $DV \not<: CV$.

If t_0 is a value, then from the definition of value, t_0 is in the form of `new` $[n]N.C(\overline{v})$. Using E-ProjNewD,

$$\frac{ifields([n]N.C) = \overline{CV}\ \overline{f} \qquad sl = [v]Q}{\big(\texttt{new}\ [n]N.C(\overline{v})\big).f_i \to v_i}\ \text{(E-ProjNewD)}$$

we can see that there exists $t' = v_i$. t_0 cannot of the form `new Object()` because it does not have any fields.

If t_0 is not a value, then by using E-Field,

$$\frac{t_0 \to t_0'}{t_0.f_i \to t_0'.f_i} \text{ (E-Field)}$$

there exists $t' = t_0'.f_i$

Case T-InvkS

Suppose that t is typed with the T-InvkS rule,

$$\frac{\begin{array}{c} sl = [n]N \vdash t_0 : [n]N.C_0 \\ mtype(m, [n]N.C_0) = \overline{DV} \to DV \\ sl = [n]N \vdash \bar{t} : \overline{CV} \quad \overline{CV} <: DV \end{array}}{sl = [n]N \vdash t_0.m(\bar{t}) : \overline{DV}} \text{ (T-InvkS)}$$

from the induction hypothesis then both t_0 and $\bar{t}$ are values or reducible into t_0' and $\bar{t}'$ or any of t_0 and $\bar{t}$ stuck because it contains (CV) `new` $DV(\overline{v})$ where $DV \not<: CV$.

If either t_0 or any of the terms $\bar{t}$ stuck, then t also stuck because it contains (CV) `new` $DV(\overline{v})$ where $DV \not<: CV$.

If both t_0 and $\bar{t}$ are all values then because t is well-typed and we know that $mtype(m,$ Object$)$ is undefined, then t_0 must be of the form `new` $[n]N.C(\overline{v})$. Then by using E-InvkNewS,

$$\frac{mbody(m, [n]N.C) = (\overline{x}, t_0) \qquad sl = [n]N}{\big(\texttt{new } [n]N.C(\overline{v})\big).m(\bar{t}) \to \big[\overline{x} \mapsto \bar{t}, \texttt{this} \mapsto \big(\texttt{new } [n]N.C(\overline{v})\big)\big]\, t_0} \text{ (E-InvkNewS)}$$

there exists $t' = \big[\overline{x} \to \bar{t}, \texttt{this} \to t_0\big]\, s_0$

If t_0 is a value and $^{\exists} t_j \in \bar{t}$ which is not a value, suppose $\bar{t} = \overline{t_a}, t_j, \overline{t_b}$, then from

$$\frac{t_j \to t_j'}{t_0.m(\overline{t_a}, t_j, \overline{t_b}) \to t_0.m(\overline{t_a}, t_j', \overline{t_b})} \text{ (E-InvkArg)}$$

there exists $t' = t_0.m(\overline{t'})$

If t_0 is not a value then from

$$\frac{t_0 \to t_0'}{t_0.m(\bar{t}) \to t_0'.m(\bar{t})} \text{ (E-InvkRecv)}$$

there exists $t' = t'_0.m(\overline{t})$

Case T-InvkD

Suppose that t is typed with the T-InvkD rule,

$$\frac{\begin{array}{c} sl = [v]Q \vdash t_0 : [n]N.C_0 \\ imtype(m, [n]N.C_0) = \overline{DV} \to DV \\ sl = [v]Q \vdash \overline{t} : \overline{CV} \quad \overline{CV} <: DV \end{array}}{sl = [v]Q \vdash t_0.m(\overline{t}) : \overline{DV}} \text{ (T-InvkD)}$$

from the induction hypothesis then both t_0 and $\overline{t}$ are values or reducible into t'_0 and $\overline{t}'$ or any of t_0 and $\overline{t}$ stuck because it contains (CV) `new` $DV(\overline{v})$ where $DV \not<: CV$.

If either t_0 or any of the terms $\overline{t}$ stuck, then t also stuck because it contains (CV) `new` $DV(\overline{v})$ where $DV \not<: CV$.

If both t_0 and $\overline{t}$ are all values then because t is well-typed and we know that $mtype(m,$ Object$)$ is undefined, then t_0 must be of the form `new` $[n]N.C(\overline{v})$. Then by using E-InvkNewD,

$$\frac{imbody(m, [n]N.C) = (\overline{x}, t_0) \qquad sl = [v]Q}{\big(\texttt{new}\ [n]N.C(\overline{v})\big).m(\overline{t}) \to \big[\overline{x} \mapsto \overline{t},\ \texttt{this} \mapsto \big(\texttt{new}\ [n]N.C(\overline{v})\big)\big]t_0} \text{ (E-InvkNewD)}$$

there exists $t' = \big[\overline{x} \to \overline{t},\ \texttt{this} \to t_0\big]s_0$

If t_0 is a value and $^{\exists}t_j \in \overline{t}$ which is not a value, suppose $\overline{t} = \overline{t_a}, t_j, \overline{t_b}$, then from

$$\frac{t_j \to t'_j}{t_0.m(\overline{t_a}, t_j, \overline{t_b}) \to t_0.m(\overline{t_a}, t'_j, \overline{t_b})} \text{ (E-InvkArg)}$$

there exists $t' = t_0.m(\overline{t'})$

If t_0 is not a value then from

$$\frac{t_0 \to t'_0}{t_0.m(\overline{t}) \to t'_0.m(\overline{t})} \text{ (E-InvkRecv)}$$

there exists $t' = t'_0.m(\overline{t})$

Case T-NewS

Suppose that t is typed with T-NewS rule,

$$\frac{fields([n]N.C) = \overline{DV}\ \overline{f} \qquad sl = [n]N \vdash \bar{t} : \overline{CV} \quad \overline{CV} <: \overline{DV}}{sl = [n]N \vdash \texttt{new}\ [n]N.C(\bar{t}) : [n]N.C}\ \text{(T-NewS)}$$

again from the induction hypothesis, $\bar{t}$ are either values or there exist $\bar{t}'$ such that $\bar{t} \to \bar{t}'$ or any of the terms $\bar{t}$ stuck because it contains (CV) $\texttt{new}\ DV(\overline{v})$ where $DV \not<: CV$.

If any of the terms $\bar{t}$ stuck then t also stuck because it contains (CV) $\texttt{new}\ DV(\overline{v})$ where $DV \not<: CV$.

If $\bar{t}$ are values, then $t = \texttt{new}\ [n]N.C(\bar{t})$ is also a value.

If $^{\exists} t_j \in \bar{t}$ which is not a value, then from

$$\frac{t_j \to t'_j}{\texttt{new}\ [n]N.C(\overline{v}, t_j, \bar{t}) \to \texttt{new}\ [n]N.C(\overline{v}, t'_j, \bar{t})}\ \text{(E-NewArg)}$$

there exists $t' = \texttt{new}\ [n]N.C(\overline{v}, t'_j, \bar{t})$

Case T-NewD

Suppose that t is typed with T-NewD rule,

$$\frac{PT(Q) = \texttt{package}\ Q\ \texttt{imports}\ \overline{N}\ \{\cdots\} \text{ where } N \in \overline{N} \qquad ifields([n]N.C) = \overline{DV}\ \overline{f} \qquad sl = [v]Q \vdash \bar{t} : \overline{CV} \quad \overline{CV} <: \overline{DV}}{sl = [v]Q \vdash \texttt{new}\ [n]N.C(\bar{t}) : [n]N.C}\ \text{(T-NewD)}$$

again from the induction hypothesis, $\bar{t}$ are either values or there exist $\bar{t}'$ such that $\bar{t} \to \bar{t}'$ or any of the terms $\bar{t}$ stuck because it contains (CV) $\texttt{new}\ DV(\overline{v})$ where $DV \not<: CV$.

The proof is similar as in the case of T-NewS.

Case T-UCast

Suppose that t is typed with T-UCast rule,

$$\frac{sl \vdash t_0 : DV \quad DV <: CV}{sl \vdash (CV)\ t_0 : CV}\ \text{(T-UCast)}$$

from the induction hypothesis, t_0 is either a value or there exists t'_0 such that $t_0 \to t'_0$ or t_0 stuck because it contains (CV) $\texttt{new}\ DV(\overline{v})$ where $DV \not<: CV$.

If t_0 stuck then t also stuck because it contains (CV) $\texttt{new}\ DV(\overline{v})$ where $DV \not<: CV$.

If t_0 is a value, then from

$$\frac{t_0 : DV \quad DV <: CV}{(CV)\ t_0 \to t_0}\ \text{(E-CastNew)}$$

there exists $t' = t_0$

If t_0 is not a value, then from

$$\frac{t_0 \to t_0'}{(CV)\ t_0 \to (CV)\ t_0'}\ \text{(E-Cast)}$$

there exists $t' = (CV)\ t_0'$

Case T-DCast

Suppose that t is typed with T-DCast rule,

$$\frac{sl \vdash t_0 : DV \qquad CV <: DV \qquad CV \neq DV}{sl \vdash (CV)\ t_0 : CV}\ \text{(T-DCast)}$$

from the induction hypothesis, t_0 is either a value or there exists t_0' such that $t_0 \to t_0'$ or t_0 stuck because it contains (CV) `new` $DV(\overline{v})$ where $DV \not<: CV$.

If t_0 stuck then t also stuck because it contains (CV) `new` $DV(\overline{v})$ where $DV \not<: CV$.

If t_0 is a value, then t_0 can only progress in the form of `new` $[n]N.C(\overline{v})$. Since `new` $[n]N.C(\overline{v})$ is well-typed, $[n]N.C$ must be a subtype of both CV and DV such that

$$\frac{[n]N.C <: CV}{(CV)\ \text{new}\ [n]N.C(\overline{v}) \to \texttt{new}\ [n]N.C(\overline{v})}\ \text{(E-CastNew)}$$

there exists $t' = t_0$

If $[n]N.C$ is not the subtype of CV then the expression stuck.

If t_0 is not a value, then from

$$\frac{t_0 \to t_0'}{(CV)\ t_0 \to (CV)\ t_0'}\ \text{(E-Cast)}$$

there exists $t' = (CV)\ t_0'$

Case T-SCast

Suppose that t is typed with T-SCast rule,

$$\frac{sl \vdash t_0 : DV \qquad CV \not<: DV \qquad DV \not<: CV \qquad \text{stupid warning}}{sl \vdash (CV)\ t_0 : CV}\ \text{T-SCast}$$

from the induction hypothesis, exists t_0' such that $t_0 \to t_0'$ or t_0 stuck because it contains (CV) `new` $DV(\overline{v})$ where $DV \not<: CV$.

If t_0 stuck then t also stuck because it contains (CV) `new` $DV(\overline{v})$ where $DV \not<: CV$.

For t to be well-typed, t_0 cannot be a value. Then from

$$\frac{t_0 \to t_0'}{(CV)\ t_0 \to (CV)\ t_0'}\ \text{(E-Cast)}$$

there exists $t' = (CV)\ t_0'$

Case T-Sub

Suppose that t is typed with T-Sub rule,

$$\frac{sl \vdash t : CV \qquad CV <: DV}{sl \vdash t : DV}$$

then it follows directly from the induction hypothesis, $t \to t'$ or t stuck because it contains (CV) `new` $DV(\overline{v})$ where $DV \not<: CV$.

4.4.2 Preservation

Lemma [Term Substitution Preserves Typing] If $sl, \Gamma, x : AV \vdash t : CV$ and $sl, \Gamma \vdash d : AV$ then $sl, \Gamma \vdash [d/x]t : CV$

Proof. On the derivation of the statement $sl, \Gamma, x : AV \vdash t : CV$

Case T-Var

Suppose $t = z$ and $z : CV \in \Gamma, x : AV$

If $z = x$, then $sl, \Gamma \vdash [d/x]z = x : AV$, which is part of the assumptions. It also implies that $CV = AV$.

If $z \neq x$, then $sl, \Gamma \vdash [d/x]z = z : CV$.

Case T-FieldS

$$t = t_0.f_i : CV_i$$
$$sl = [n]N, \Gamma, x : AV \vdash t_0 : [n]N.C_0$$
$$fields([n]N.C_0) = \overline{CV}\ \overline{f}$$

From the induction hypothesis, then there exists $[n]N.C_0$ such that $sl = [n]N, \Gamma \vdash [d/x]t_0 : [n]N.C_0$. Then from T-FieldS

$$\frac{sl = [n]N,\, \Gamma \vdash [d/x]t_0 : [n]N.C_0 \qquad fields([n]N.C_0) = \overline{CV}\ \overline{f}}{sl = [n]N,\, \Gamma \vdash ([d/x]t_0).f_i : CV_i} \text{ (T-FieldS)}$$

where $[d/x](t_0.f_i)$ is exactly $([d/x]t_0).f_i$.

Case T-FieldD

$$t = t_0.f_i : CV_i$$
$$sl = [v]Q,\, \Gamma,\, x : AV \vdash [n]N.C_0 : CV_0$$
$$ifields([n]N.C_0) = \overline{CV}\ \overline{f}$$

Similarly as in the case of T-FieldS, from the induction hypothesis, then there exists $[n]N.C_0$ such that $sl = [v]Q,\, \Gamma \vdash [d/x]t_0 : [n]N.C_0$. Then from T-FieldD

$$\frac{sl = [v]Q,\, \Gamma \vdash [d/x]t_0 : [n]N.C_0 \qquad ifields([n]N.C_0) = \overline{CV}\ \overline{f}}{sl = [v]Q,\, \Gamma \vdash ([d/x]t_0).f_i : CV_i} \text{ (T-FieldD)}$$

where $[d/x](t_0.f_i)$ is exactly $([d/x]t_0).f_i$.

Case T-InvkS

$$t = t_0.m(\overline{t}) : DV$$
$$sl = [n]N,\, \Gamma,\, x : AV \vdash t_0 : [n]N.C_0$$
$$mtype(m, [n]N.C_0) = \overline{DV} \to DV$$
$$sl = [n]n,\, \Gamma,\, x : AV \vdash \overline{t} : \overline{CV}$$
$$\overline{CV} <: \overline{DV}$$

From the induction hypothesis, then there exist $[n]N.C_0$ and $\overline{CV}$ such that $sl = [n]N,\, \Gamma \vdash [d/x]t_0 : [n]N.C_0$ and $sl = [n]N,\, \Gamma \vdash [d/x]\overline{t} : \overline{CV}$. Using T-InvkS

$$\frac{\begin{array}{c} sl = [n]N,\, \Gamma \vdash [d/x]t_0 : [n]N.C_0 \\ mtype(m, [n]N.C_0) : \overline{DV} \to DV \\ sl = [n]N,\, \Gamma \vdash [d/x]\overline{t} : \overline{CV} \\ \overline{CV} <: \overline{DV} \end{array}}{sl = [n]N\ \Gamma \vdash \big([d/x]t_0\big).m([d/x]\overline{t}) : DV} \text{ (T-InvkS)}$$

where $[d/x]\big(t_0.m(\overline{t})\big)$ is equivalent to $\big([d/x]t_0\big).m([d/x]\overline{t}) : DV$.

Case T-InvkD

$$t = t_0.m(\bar{t}) : DV$$
$$sl = [v]Q,\ \Gamma,\ x : AV \vdash t_0 : [n]N.C_0$$
$$imtype(m, [n]N.C_0) = \overline{DV} \to DV$$
$$sl = [v]Q,\ \Gamma,\ x : AV \vdash \bar{t} : \overline{CV}$$
$$\overline{CV} <: \overline{DV}$$

Similar as in the case of T-InvkS, from the induction hypothesis, then there exist $[n]N.C_0$ and $\overline{CV}$ such that $sl = [v]Q,\ \Gamma \vdash [d/x]t_0 : [n]N.C_0$ and $sl = [v]Q,\ \Gamma \vdash [d/x]\bar{t} : \overline{CV}$. Using T-InvkD

$$\frac{\begin{array}{c} sl = [v]Q,\ \Gamma \vdash [d/x]t_0 : [n]N.C_0 \\ imtype(m, [n]N.C_0) : \overline{DV} \to DV \\ sl = [v]Q,\ \Gamma \vdash [d/x]\bar{t} : \overline{CV} \\ \overline{CV} <: \overline{DV} \end{array}}{sl = [v]Q\ \Gamma \vdash \big([d/x]t_0\big).m([d/x]\bar{t}) : DV}\ \text{(T-InvkD)}$$

where $[d/x]\big(t_0.m(\bar{t})\big)$ is equivalent to $\big([d/x]t_0\big).m([d/x]\bar{t}) : DV$.

Case T-NewS

$$t = \texttt{new}\ [n]N.C(\bar{t}) : [n]N.C$$
$$fields([n]N.C) = \overline{DV}\ \overline{f}$$
$$sl = [n]N,\ \Gamma, x : AV \vdash \bar{t} : \overline{CV}$$
$$\overline{CV} <: \overline{DV}$$

From the induction hypothesis, then there exists $\overline{CV}$ such that $sl = [n]N,\ \Gamma \vdash [d/x]\bar{t} : \overline{CV}$. Using T-NewS

$$\frac{\begin{array}{c} sl = [n]N,\ \Gamma \vdash [d/x]\bar{t} : \overline{CV} \\ fields([n]N.C) = \overline{DV}\ \overline{f} \\ \overline{CV} <: \overline{DV} \end{array}}{sl = [n]N,\ \Gamma \vdash \texttt{new}\ [n]N.C([d/x]\bar{t}) : [n]N.C}\ \text{(T-NewS)}$$

where $[d/x]\big(\texttt{new}\ [n]N.C(\bar{t})\big)$ is equivalent to $\texttt{new}\ [n]N.C([d/x]\bar{t})$.

Case T-NewD

$$t = \texttt{new}\ [n]N.C(\bar{t}) : [n]N.C$$
$$PT(Q) = \texttt{package}\ Q\ \texttt{imports}\ \overline{N}\ \{\cdots\} \text{ where } N \in \overline{N}$$
$$ifields([n]N.C) = \overline{DV}\ \overline{f}$$
$$sl = [v]Q,\ \Gamma, x : AV \vdash \bar{t} : \overline{CV}$$
$$\overline{CV} <: \overline{DV}$$

From the induction hypothesis, then there exists $\overline{CV}$ such that $sl = [v]Q$, $\Gamma \vdash [d/x]\bar{t} : \overline{CV}$. Using T-NewD

$$\frac{\begin{array}{c}PT(Q) = \texttt{package}\ Q\ \texttt{imports}\ \overline{N}\ \{\cdots\} \text{ where } N \in \overline{N} \\ sl = [v]Q,\ \Gamma \vdash [d/x]\bar{t} : \overline{CV} \\ ifields([n]N.C) = \overline{DV}\ \overline{f} \\ \overline{CV} <: \overline{DV}\end{array}}{sl = [v]Q,\ \Gamma \vdash \texttt{new}\ [n]N.C([d/x]\bar{t}) : [n]N.C}\ \text{(T-NewD)}$$

where $[d/x]\big(\texttt{new}\ [n]N.C(\bar{t})\big)$ is equivalent to $\texttt{new}\ [n]N.C([d/x]\bar{t})$.

Case T-UCast

$$t = (CV)\ t_0 : CV$$
$$sl,\ \Gamma, x : AV \vdash t_0 : DV$$
$$DV <: CV$$

From the induction hypothesis, then there exists DV such that sl, $\Gamma \vdash [d/x]t_0 : DV$. Then from T-UCast

$$\frac{sl,\ \Gamma \vdash [d/x]t_0 : DV \qquad DV <: CV}{sl,\ \Gamma \vdash (CV)\ [d/x]t_0 : CV}\ \text{(T-UCast)}$$

where $[d/x]\big((CV)\ t_0\big)$ is equivalent to $(CV)\ [d/x]t_0$.

Case T-DCast

$$t = (CV)\ t_0 : CV$$
$$sl,\ \Gamma, x : AV \vdash t_0 : DV$$
$$CV <: DV$$
$$CV \neq DV$$

From the induction hypothesis, then there exists DV such that sl, $\Gamma \vdash [d/x]t_0 : DV$. Then from T-DCast

$$\frac{sl, \Gamma \vdash [d/x]t_0 : DV \qquad CV <: DV \qquad CV \neq DV}{sl, \Gamma \vdash (CV)\ [d/x]t_0 : CV}\ \text{(T-DCast)}$$

where $[d/x]\big((CV)\ t_0\big)$ is equivalent to $(CV)\ [d/x]t_0$.

Case T-SCast

$$\begin{gathered} t = (CV)\ t_0 : CV \\ sl, \Gamma, x : AV \vdash t_0 : DV \\ CV \not<: DV \\ DV \not<: CV \\ \text{stupid warning} \end{gathered}$$

From the induction hypothesis, then there exists DV such that $sl, \Gamma \vdash [d/x]t_0 : DV$. Then from T-SCast

$$\frac{sl, \Gamma \vdash [d/x]t_0 : DV \qquad CV \not<: DV \qquad DV \not<: CV \qquad \text{stupid warning}}{sl, \Gamma \vdash (CV)\ [d/x]t_0 : CV}$$

where $[d/x]\big((CV)\ t_0\big)$ is equivalent to $(CV)\ [d/x]t_0$.

Case T-Sub

$$\begin{gathered} sl, \Gamma \vdash t : DV \\ sl, \Gamma \vdash t : CV \\ CV <: DV \end{gathered}$$

Follows directly from the induction hypothesis,

$$\frac{sl, \Gamma \vdash [d/x]t : CV \qquad CV <: DV}{sl, \Gamma \vdash [d/x]t : DV}\ \text{(T-Sub)}$$

Theorem [Preservation] If $sl, \Gamma \vdash t : CV$ and $t \to t'$ then $sl, \Gamma \vdash t' : CV$

Proof. By induction on the derivation of $t : CV$

Case T-Var

$$\frac{t = x : CV}{x : CV \in \Gamma}\ \text{(T-Var)}$$

There is no derivation rule for $t = x$.

Case T-FieldS

$$\begin{gathered} t = t_0.f_i : CV_i \\ sl = [n]N,\, \Gamma \vdash t_0 : [n]N.C_0 \\ fields([n]N.C_0) = \overline{CV}\ \overline{f} \end{gathered}$$

$t \to t'$ can be derived from E-Field and E-ProjNewS.

- **E-Field** If $t \to t'$ is derived with E-Field, then t_0 is not a value and can be reduced into t'_0. Using this,

$$\frac{t_0 \to t'_0}{t_0.f_i \to t'_0.f_i}\ \text{(E-Field)}$$

we have $t' = t'_0.fi$.

Then from the induction hypothesis, we know that there exists $[n]N.C_0$ such that $sl = [n]N,\, \Gamma \vdash t'_0 : [n]N.C_0$. Applying this to the T-FieldS rule,

$$\frac{sl = [n]N,\, \Gamma \vdash t'_0 : [n]N.C_0 \qquad fields([n]N.C_0) = \overline{CV}\ \overline{f}}{sl = [n]N,\, \Gamma \vdash t'_0.f_i : CV_i}\ \text{(T-FieldS)}$$

we then obtain $sl = [n]N,\, \Gamma \vdash t'_0.f_i : CV_i$.

- **E-ProjNewS** If $t \to t'$ is derived with E-ProjNewS then t_0 is of the form new $[n]N.C(\overline{v})$. Using this,

$$\frac{fields([n]N.C) = \overline{CV}\ \overline{f} \qquad sl = [n]N}{\big(\texttt{new}\ [n]N.C(\overline{v})\big).f_i \to v_i}\ \text{(E-ProjNewS)}$$

v_i is the i-th field of $[n]N.C_0$, which means that $v_i : CV_i$. Thus $sl = [n]N,\, \Gamma \vdash \big(\texttt{new}\ [n]N.C(\overline{v})\big).f_i : CV_i$.

Case T-FieldD

$$\begin{gathered} t = t_0.f_i : CV_i \\ sl = [v]Q,\, \Gamma \vdash t_0 : [n]N.C_0 \\ ifields([n]N.C_0) = \overline{CV}\ \overline{f} \end{gathered}$$

$t \to t'$ can be derived from E-Field and E-ProjNewD.

- **E-Field** If $t \to t'$ is derived with E-Field, then t_0 is not a value and can be reduced into t'_0. Using this,

$$\frac{t_0 \to t'_0}{t_0.f_i \to t'_0.f_i}\ \text{(E-Field)}$$

  we have $t' = t'_0.fi$.

  Then from the induction hypothesis, we know that there exists $[n]N.C_0$ such that $sl = [v]Q,\ \Gamma \vdash t'_0 : [n]N.C_0$. Applying this to the T-FieldD rule,

$$\frac{sl = [v]Q,\ \Gamma \vdash t'_0 : [n]N.C_0 \qquad ifields([n]N.C_0) = \overline{CV}\ \overline{f}}{sl = [v]Q,\ \Gamma \vdash t'_0.f_i : CV_i}\ \text{(T-FieldD)}$$

  we then obtain $sl = [v]Q,\ \Gamma \vdash t'_0.f_i : CV_i$.

- **E-ProjNewD** If $t \to t'$ is derived with E-ProjNew then t_0 is of the form new $[n]N.C(\overline{v})$. Using this,

$$\frac{ifields([n]N.C) = \overline{CV}\ \overline{f} \qquad sl = [v]Q}{\big(\texttt{new}\ [n]N.C(\overline{v})\big).f_i \to v_i}\ \text{(E-ProjNewD)}$$

  v_i is the i-th field of $[n]N.C_0$, which means that $v_i : CV_i$. Thus $sl = [v]Q,\ \Gamma \vdash \big(\texttt{new}\ [n]N.C(\overline{v})\big).f_i : CV_i$.

Case T-InvkS

$$\begin{gathered} t = t_0.m(\overline{t}) : DV \\ sl = [n]N,\ \Gamma \vdash t_0 : [n]N.C_0 \\ mtype(m, [n]N.C_0) = \overline{DV} \to DV \\ sl = [n]N,\ \Gamma \vdash \overline{t} : \overline{CV} \\ \overline{CV} <: \overline{DV} \end{gathered}$$

$t \to t'$ can be derived from E-InvkRecv, E-InvkArg and E-InvkNewS.

- **E-InvkRecv** If $t \to t'$ is derived with E-InvkRecv, then t_0 is not value and there exists t'_0 such that $t_0 \to t'_0$. From applying E-InvkRecv,

$$\frac{t_0 \to t_0'}{t_0.m(\overline{t}) \to t_0'.m(\overline{t})}\ \text{(E-InvkRecv)}$$

we will obtain $t' = t_0'.m(\overline{t})$.

Then from the induction hypothesis, there exists $[n]N.C_0$ such that $sl = [n]N, \Gamma \vdash t_0' : [n]N.C_0$. By applying the T-InvkS rule,

$$\frac{\begin{array}{c} sl = [n]N, \Gamma \vdash t_0' : [n]N.C_0 \\ mtype(m, [n]N.C_0) = \overline{DV} \to DV \\ sl = [n]N, \Gamma \vdash \overline{t} : \overline{CV} \quad \overline{CV} <: \overline{DV} \end{array}}{sl = [n]N, \Gamma \vdash t_0'.m(\overline{t}) : DV}\ \text{(T-InvkS)}$$

we then obtain $sl = [n]N, \Gamma \vdash t_0'.m(\overline{t}) : DV$.

- **E-InvkArg** If $t \to t'$ is derived with E-InvkArg then there exists $\overline{t}'$ such that $\overline{t} \to \overline{t}'$. Suppose that $\overline{t} = \overline{v}, t_j, \overline{s}$ and $\overline{t}' = \overline{v}, t_j', \overline{s}$. From applying E-InvkArg

$$\frac{t_j \to t_j'}{t_0.m(\overline{v}, t_j, \overline{s}) \to t_0.m(\overline{v}, t_j', \overline{s})}\ \text{(E-InvkArg)}$$

we have $t' = t_0.m(\overline{v}, t_j', \overline{s})$.

Then from the induction hypothesis, then there exists $\overline{CV}$ such that $sl = [n]N, \Gamma \vdash \overline{t}' : \overline{CV}$. Similar to the case above, by applying the T-InvkS rule,

$$\frac{\begin{array}{c} sl = [n]N, \Gamma \vdash t_0 : CV_0 \\ mtype(m, CV_0) = \overline{DV} \to DV \\ sl = [n]N, \Gamma \vdash \overline{t}' : \overline{CV} \quad \overline{CV} <: \overline{DV} \end{array}}{sl = [n]N\ \Gamma \vdash t_0.m(\overline{t}') : DV}\ \text{(T-InvkS)}$$

we obtain $sl = [n]N, \Gamma \vdash t_0.m(\overline{t}') : DV$.

- **E-InvkNewS** If $t \to t'$ is derived with E-InvkNewS, then both t_0 and $\overline{t}$ are values. Suppose $t_0 = \text{new } [n]N.C_0(\overline{v})$ and $\overline{t} = \overline{u}$, by applying the E-InvkNew rule,

$$\frac{mbody(m, [n]N.C_0) = (\overline{x}, s_0) \qquad sl = [n]N}{\big(\texttt{new}\ [n]N.C_0(\overline{v})\big).m(\overline{u}) \to [\overline{x} \mapsto \overline{u}, \texttt{this} \mapsto \big(\texttt{new}\ [n]N.C_0(\overline{v})\big)]s_0}$$

we obtain $t' = [\overline{x} \mapsto \overline{u}, \texttt{this} \mapsto \big(\texttt{new}\ [n]N.C(\overline{v})\big)]s_0$.

Because t_0 is well-typed, then from method's well-definedness, we have $sl = [n]N, \overline{x} : \overline{DV}, \texttt{this} : [n]N.C_0 \vdash s_0 : DV$. Afterwards, by applying weakening and the substitution lemma, we will obtain $sl = [n]N\ \Gamma \vdash \big[\overline{u}/\overline{x}, \big(\texttt{new}\ [n]N.C_0(\overline{v})/\texttt{this}\big)\big]s_0 : DV$.

Case T-InvkD

$$\begin{gathered} t = t_0.m(\overline{t}) : DV \\ sl = [v]Q, \Gamma \vdash t_0 : [n]N.C_0 \\ mtype(m, [n]N.C_0) = \overline{DV} \to DV \\ sl = [v]Q, \Gamma \vdash \overline{t} : \overline{CV} \\ \overline{CV} <: \overline{DV} \end{gathered}$$

$t \to t'$ can be derived from E-InvkRecv, E-InvkArg and E-InvkNewD.

- **E-InvkRecv** If $t \to t'$ is derived with E-InvkRecv, then t_0 is not value and there exists t_0' such that $t_0 \to t_0'$. From applying E-InvkRecv,

$$\frac{t_0 \to t_0'}{t_0.m(\overline{t}) \to t_0'.m(\overline{t})}\ \text{(E-InvkRecv)}$$

we will obtain $t' = t_0'.m(\overline{t})$.

Then from the induction hypothesis, there exists $[n]N.C_0$ such that $sl = [v]Q, \Gamma \vdash t_0' : [n]N.C_0$. By applying the T-InvkD rule,

$$\frac{\begin{gathered} sl = [v]Q, \Gamma \vdash t_0' : [n]N.C_0 \\ imtype(m, [n]N.C_0) = \overline{DV} \to DV \\ sl = [v]Q, \Gamma \vdash \overline{t} : \overline{CV} \quad \overline{CV} <: \overline{DV} \end{gathered}}{sl = [v]Q, \Gamma \vdash t_0'.m(\overline{t}) : DV}\ \text{(T-InvkD)}$$

we then obtain $sl = [v]Q, \Gamma \vdash t_0'.m(\overline{t}) : DV$.

- **E-InvkArg** If $t \to t'$ is derived with E-InvkArg then there exists $\overline{t}'$ such that $\overline{t} \to \overline{t}'$. Suppose that $\overline{t} = \overline{v}, t_j, \overline{s}$ and $\overline{t}' = \overline{v}, t_j', \overline{s}$. From applying E-InvkArg

$$\frac{t_j \to t_j'}{t_0.m(\overline{v}, t_j, \overline{s}) \to t_0.m(\overline{v}, t_j', \overline{s})}\ \text{(E-InvkArg)}$$

we have $t' = t_0.m(\overline{v}, t'_j, \overline{s})$.

Then from the induction hypothesis, then there exists $\overline{CV}$ such that $sl = [v]Q,\ \Gamma \vdash \overline{t}' : \overline{CV}$. Similar to the case above, by applying the T-InvkS rule,

$$\frac{\begin{array}{c} sl = [v]Q,\ \Gamma \vdash t_0 : CV_0 \\ imtype(m, CV_0) = \overline{DV} \to DV \\ sl = [v]Q,\ \Gamma \vdash \overline{t}' : \overline{CV} \quad \overline{CV} <: \overline{DV} \end{array}}{sl = [v]Q\ \Gamma \vdash t_0.m(\overline{t}') : DV}\ \text{(T-InvkD)}$$

we obtain $sl = [v]Q,\ \Gamma \vdash t_0.m(\overline{t}') : DV$.

- **E-InvkNewD** If $t \to t'$ is derived with E-InvkNewD, then both t_0 and $\overline{t}$ are values. Suppose $t_0 = \text{new } [n]N.C_0(\overline{v})$ and $\overline{t} = \overline{u}$, by applying the E-InvkNew rule,

$$\frac{imbody(m, [n]N.C_0) = (\overline{x}, s_0) \qquad sl = [v]Q}{\big(\texttt{new}\ [n]N.C_0(\overline{v})\big).m(\overline{u}) \to [\overline{x} \mapsto \overline{u},\ \texttt{this} \mapsto \big(\texttt{new}\ [n]N.C_0(\overline{v})\big)]s_0}$$

we obtain $t' = [\overline{x} \mapsto \overline{u},\ \texttt{this} \mapsto \big(\texttt{new}\ [n]N.C(\overline{v})\big)]s_0$.

Because t_0 is well-typed, then from method's well-definedness, we have $sl = [v]Q,\ \overline{x} : \overline{DV},\ \texttt{this} : [n]N.C_0 \vdash s_0 : DV$. Afterwards, by applying weakening and the substitution lemma, we will obtain $sl = [v]Q\ \Gamma \vdash \big[\overline{u}/\overline{x}, \big(\texttt{new}\ [n]N.C_0(\overline{v})/\texttt{this}\big)\big]s_0 : DV$.

Case T-NewS

$$\frac{\begin{array}{c} t = \texttt{new}\ [n]N.C(\overline{t}) : [n]N.C \\ fields([n]N.C) = \overline{DV}\ \overline{f} \\ sl = [n]N,\ \Gamma \vdash \overline{t} : \overline{CV} \\ \overline{CV} <: \overline{DV} \end{array}}{}$$

$t \to t'$ can only be derived if $\overline{t}$ is not value, with the E-NewArg rule.

- **E-NewArg** As before, suppose $\overline{t} = \overline{v}, t_j, \overline{s}$ and $\overline{t}' = \overline{v}, t'_j, \overline{s}$, by applying E-NewArg,

$$\frac{t_j \to t'_j}{\texttt{new}\ [n]N.C(\overline{v}, t_j, \overline{s}) \to \texttt{new}\ [n]N.C(\overline{v}, t'_j, \overline{s})}\ \text{(E-NewArg)}$$

we obtain $t' = \texttt{new}\ [n]N.C(\overline{v}, t'_j, \overline{s})$.

Then from the induction hypothesis, there exists $\overline{CV}$ such that $sl = [n]N, \Gamma \vdash \overline{t}' : \overline{CV}$. By applying the T-NewS rule,

$$\frac{fields([n]N.C) = \overline{DV}\ \overline{f} \qquad sl = [n]N, \Gamma \vdash \overline{t}' : \overline{CV} \quad \overline{CV} <: \overline{DV}}{sl = [n]N, \Gamma \vdash \texttt{new}\ [n]N.C(\overline{t}') : [n]N.C}\ \text{(T-NewS)}$$

we have $sl = [n]N, \Gamma \vdash \texttt{new}\ [n]N.C(\overline{t}') : [n]N.C$.

Case T-NewD

$$t = \texttt{new}\ [n]N.C(\overline{t}) : [n]N.C$$
$$PT(Q) = \texttt{package}\ Q\ \texttt{imports}\ \overline{N}\ \{\cdots\} \text{ where } N \in \overline{N}$$
$$fields([n]N.C) = \overline{DV}\ \overline{f}$$
$$sl = [v]Q, \Gamma \vdash \overline{t} : \overline{CV}$$
$$\overline{CV} <: \overline{DV}$$

$t \to t'$ can only be derived if $\overline{t}$ is not value, with the E-NewArg rule.

- **E-NewArg** As before, suppose $\overline{t} = \overline{v}, t_j, \overline{s}$ and $\overline{t}' = \overline{v}, t'_j, \overline{s}$, by applying E-NewArg,

$$\frac{t_j \to t'_j}{\texttt{new}\ [n]N.C(\overline{v}, t_j, \overline{s}) \to \texttt{new}\ [n]N.C(\overline{v}, t'_j, \overline{s})}\ \text{(E-NewArg)}$$

we obtain $t' = \texttt{new}\ [n]N.C(\overline{v}, t'_j, \overline{s})$.

Then from the induction hypothesis, there exists $\overline{CV}$ such that $sl = [v]Q, \Gamma \vdash \overline{t}' : \overline{CV}$. By applying the T-NewD rule,

$$\frac{PT(Q) = \texttt{package}\ Q\ \texttt{imports}\ \overline{N}\ \{\cdots\} \text{ where } N \in \overline{N} \qquad ifields([n]N.C) = \overline{DV}\ \overline{f} \qquad sl = [v]Q, \Gamma \vdash \overline{t}' : \overline{CV} \quad \overline{CV} <: \overline{DV}}{sl = [v]Q, \Gamma \vdash \texttt{new}\ [n]N.C(\overline{t}') : [n]N.C}\ \text{(T-NewD)}$$

we have $sl = [v]Q, \Gamma \vdash \texttt{new}\ [n]N.C(\overline{t}') : [n]N.C$.

Case T-UCast

$$\begin{gathered} t = (CV)\ t_0 : CV \\ sl, \Gamma \vdash t_0 : DV \\ DV <: CV \end{gathered}$$

$t \to t'$ can be derived through E-Cast and E-CastNew.

- **E-Cast** First by applying E-Cast,

$$\frac{t_0 \to t_0'}{(CV)\ t_0 \to (CV)\ t_0'}\ \text{(E-Cast)}$$

  we obtain $t' = (CV)\ t_0'$.

  Then from induction hypothesis, there exists DV such that $\Gamma \vdash t_0' : DV$. By applying the T-UCast rule,

$$\frac{sl, \Gamma \vdash t_0' : DV \qquad DV <: CV}{sl, \Gamma \vdash (CV)\ t_0' : CV}\ \text{(T-UCast)}$$

  we have $sl, \Gamma \vdash (CV)\ t_0' : CV$.

- **E-CastNew** If $t \to t'$ is derived from E-CastNew, then t_0 is of the form new $[n]N.E(\overline{v})$. From applying E-CastNew,

$$\frac{[n]N.E <: CV}{(CV)\ \texttt{new}\ [n]N.E(\overline{v}) \to \texttt{new}\ [n]N.E(\overline{v})}\ \text{(E-CastNew)}$$

  we obtain $t' = \texttt{new}\ [n]N.E(\overline{v})$.

  Then by using T-Sub rule,

$$\frac{sl, \Gamma \vdash \texttt{new}\ [n]N.E(\overline{v}) : [n]N.E \qquad [n]N.E <: CV}{sl, \Gamma \vdash \texttt{new}\ [n]N.E(\overline{v}) : CV}\ \text{(T-Sub)}$$

  we obtain $sl, \Gamma \vdash \texttt{new}\ [n]N.E(\overline{v}) : CV$

Case T-DCast

$$\begin{gathered} sl, \Gamma \vdash (CV)\ t_0 : CV \\ sl, \Gamma \vdash t_0 : DV \\ CV <: DV \\ CV \neq DV \end{gathered}$$

Same as in the case of T-UCast, $t \to t'$ can be derived through E-Cast and E-CastNew.

- **E-Cast** By applying E-Cast,

$$\frac{t_0 \to t_0'}{(CV)\ t_0 \to (CV)\ t_0'}\ \text{(E-Cast)}$$

  we obtain $t' = (CV)\ t_0'$.

  Then from induction hypothesis, there exists DV such that $\Gamma \vdash t_0' : DV$. By applying the T-DCast rule,

$$\frac{sl, \Gamma \vdash t_0 : DV \qquad CV <: DV \qquad CV \neq DV}{sl, \Gamma \vdash (CV)\ t_0' : CV}\ \text{(T-DCast)}$$

  we have $\Gamma \vdash (CV)\ t_0' : CV$.

- **E-CastNew** If $t \to t'$ is derived from E-CastNew, then t_0 is of the form new $[n]N.E(\overline{v})$. From applying E-CastNew,

$$\frac{[n]N.E <: CV}{(CV)\ \texttt{new}\ [n]N.E(\overline{v}) \to \texttt{new}\ [n]N.E(\overline{v})}\ \text{(E-CastNew)}$$

  we obtain $t' = \texttt{new}\ [n]N.E(\overline{v})$.

  From the assumption that $CV <: DV$, it means that for the derivation to progress, $[n]N.E$ must be a subtype of CV and DV. By applying T-Sub,

$$\frac{sl, \Gamma \vdash \texttt{new}\ [n]N.E(\overline{v}) : [n]N.E \qquad [n]N.E <: CV}{sl, \Gamma \vdash \texttt{new}\ [n]N.E(\overline{v}) : CV}\ \text{(T-Sub)}$$

  we obtain $sl, \Gamma \vdash \texttt{new}\ [n]N.E(\overline{v}) : CV$.

Case T-SCast

$$\begin{gathered} t = (CV)\ t_0 : CV \\ sl, \Gamma \vdash t_0 : DV \\ CV \not<: DV \\ DV \not<: CV \\ \text{stupid warning} \end{gathered}$$

Similar to the two cases above, $t \to t'$ can be only be derived through E-Cast, it cannot be derived through E-CastNew because it contradicts with the assumption.

- **E-Cast** By applying E-Cast,

$$\frac{t_0 \to t_0'}{(CV)\ t_0 \to (CV)\ t_0'}\ \text{(E-Cast)}$$

we obtain $t' = (CV)\ t_0'$.

Then from induction hypothesis, there exists DV such that $\Gamma \vdash t_0' : DV$. By applying the T-SCast rule,

$$\frac{sl, \Gamma \vdash t_0' : DV \qquad CV \not<: DV \qquad DV \not<: CV \qquad \text{stupid warning}}{sl, \Gamma \vdash (CV)\ t_0' : CV}$$

we obtain $sl, \Gamma \vdash (CV)\ t_0' : CV$ with stupid warning.

Case T-Sub

$$\begin{array}{c} sl, \Gamma \vdash t : DV \\ sl, \Gamma \vdash t : CV \\ CV <: DV \end{array}$$

It follows directly from the induction hypothesis.

$$\frac{sl, \Gamma \vdash t' : CV \qquad CV <: DV}{sl, \Gamma \vdash t' : DV}\ \text{(T-Sub)}$$

we have $\Gamma \vdash t' : DV$.

4.4.3 Type Soundness

Theorem [Type Soundness] If $sl \vdash t : CV$ then either, (1) t is a value or, (2) there exists t' such that $t \to t'$ and $sl \vdash t' : CV$ or, (3) t contains a subterm of the form (DV) `new` $CV(\overline{v})$ where $CV \not<: DV$.

Proof. Immediate from Progress and Preservation.

Chapter 5

Formal Examples

In this chapter, the examples provided in the earlier chapter will be reintroduced by using the formal calculus.

```
package Base {

interface 1 {
  Person <: Object {
    birth_year :: (Object) -> Object;
    new :: (Object, Object) -> Person;
  }
  Building <: Object {
    new :: (Object) -> Building;
  }
}

interface 2 {
  Person <: Object {
    birth_year :: (Object) -> Object;
    new :: (Object, Object) -> Object;
  }
  Address <: Object {
    new :: (Object, Object) -> Address;
  }
  Building <: Object {
    new :: ([1]Base.Address) -> Building;
  }
}

version 1 {
  class Person extends Object {
    Object name;
    Object age;
```

```
    Person(Object name, Object age){
      super();
      this.name = name;
      this.age = age;
    }

    Object birth_year(Object current_year){
      return current_year;
    }
  }

  class Building extends Object {
    Object addr;

    Building(Object addr){
      super();
      this.addr = addr;
    }
  }
}

version 2 {
  class Person extends Object {
    Object name;
    Object age;

    Person(Object name, Object age){
      super();
      this.name = name;
      this.age = age;
    }

    Object birth_year(Object current_year){
      return current_year;
    }
  }

  class Address extends Object{
    Object street;
    Object number;

    Address(Object street, Object number){
      super();
      this.street = street;
      this.number = number;
    }
  }

  class Building extends Object{
```

```
    [2]Base.Address addr;

    Building([2]Base.Address addr){
      super();
      this.addr = addr;
    }
  }
}}
```

Listing 5.1: Package `Base` written in the formal syntax

Comparing the formal `Base` and the informal `Base` which was introduced before, the first stark difference is the types available. Since Batak Java is based on a minimal language, primitive types such as character, integer and boolean are not defined within the language. To account for that, we use the class `Object` as replacement.

In Batak Java, every class must extend another class. Therefore classes such as `Person` and `Building` which do not own any other superclass in the informal example will have to explicitly extend the class `Object` in this example.

In the formal calculus, every class name beside `Object` has to be written in its full name (later extensions may allow other types such as integer, boolean, etc.). Such example can be found in the second version of the class `Building`, where the type of the field `addr` is written as `[1]Base.Address` instead of just `Address` and likewise the constructor is written in the same manner.

```
package Education imports Base {

interface 1 {
  Teacher <: [1]Base.Person{
    new :: (Object, Object, Object, [1]Education.School)
          -> Teacher;
  }
  School <: [1]Base.Building {
    change_head :: ([1]Education.Teacher)
          -> [1]Education.School;
    new :: (Object, Object, [1]Education.Teacher) -> School;
  }
}

version 1 {
  class Teacher extends [1]Base.Person {
    Object id;
    [1]Education.School workplace;

    Teacher(Object name,
```

```
        Object age,
        Object id,
        [1]Education.School workplace)
    {
      super(name, age);
      this.id = id;
      this.workplace = workplace;
    }
  }

  class School extends [1]Base.Building {
    Object name;
    [1]Education.Teacher head;

    School(Object addr,
        Object name,
        [1]Education.Teacher head)
    {
      super(addr);
      this.name = name;
      this.head = head;
    }

    [1]Education.School change_head(
      [1]Education.Teacher new_head)
    {
      return new [1]Education.Teacher(this.addr, this.name,
      new_head)
    }
  }
}}
```

Listing 5.2: Package `Education` written in the formal syntax

Similarly here we need to specify the version number and the package name while declaring a class extending a class from a different package. We need to write `class Teacher extends [1]Base.Person` to avoid conflict that may happen when two different packages have a similar named class. In the current calculus, the same also applies for result type of method, with the result type of method `change_head` written as `[1]Education.School`; also for type of field, we write `[1]Education.School` instead of `School` and `[1]Education.Teacher` instead of `Teacher`.

```
package Health imports Base {

interface 1 {
  Doctor <: [2]Base.Person{
    promotion :: (Object) -> [1]Health.Surgeon;
```

```
      new :: (Object, Object, Object, [1]Health.Hospital)
            -> Doctor
    }
    Surgeon <: Doctor {
      new :: (Object, Object, Object, [1]Health.Hospital, Object)
            -> Surgeon;
    }
    Hospital <: [2]Base.Building {
      change_head :: ([1]Health.Doctor) -> [1]Health.Hospital;
      new :: ([2]Base.Address, Object, [1]Health.Doctor)
            -> Hospital;
    }
  }

  version 1 {
    class Doctor extends [2]Base.Person {
      Object branch;
      [1]Health.Hospital workplace;

      Doctor(Object name,
          Object age,
          Object branch,
          [1]Health.Hospital workplace)
      {
        super(name, age);
        this.branch = branch;
        this.workplace = workplace;
      }

      [1]Health.Surgeon promotion(Object hours){
        return new Surgeon(this.name, this.age, this.branch,
        this.workplace, hours);
      }
    }

    class Surgeon extends Doctor {
      Object total_hours;

      Surgeon(Object name,
            Object age,
            Object branch,
            [1]Health.Hospital workplace,
            Object total_hours){
        super(name, age, branch, workplace);
        this.total_hours = total_hours;
      }
    }

    class Hospital extends [2]Base.Building {
```

```
    Object name;
    [1]Health.Doctor head;

    Hospital([2]Base.Address addr,
             Object name,
             [1]Health.Doctor head){
      super(addr);
      this.name = name;
      this.head = head;
    }

    [1]Health.Hospital change_head([1]Health.Doctor new_head){
      return new [1]Health.Hospital(this.addr, this.name,
      new_head);
    }
  }
}}
```

Listing 5.3: Package `Health` written in the formal syntax

The `Health` also changes only in the way the class and type names are written. Apart from that, everything remains identical with the examples in the preceding chapter.

```
package Municipal imports Education, Health {

interface 1 {
  Pair <: Object {
    new :: ([1]Education.Teacher, [1]Health.Doctor) -> Pair;
  }
  Manage <: Object {
    pair_heads :: ([1]Education.School, [1]Health.Hospital)
                        -> Pair;
    new :: () -> Manage;
  }
}

version 1 {
  class Pair extends Object {
    [1]Education.Teacher nameA;
    [1]Health.Doctor nameB;

    Pair([1]Education.Teacher nameA, [1]Health.Doctor nameB){
      super();
      this.nameA = nameA;
      this.nameB = nameB;
    }
  }

  class Manage extends Object {
```

```
      Manage(){
        super();
      }

      Pair pair_heads([1]Education.School a,
      [1]Health.Hospital b){
        return new Pair(a.head, b.head);
      }
    }
  }}
```

Listing 5.4: Package `Municipal` written in the formal syntax

In the `Municipal` package, we use both `Education` and `Health` defined above. Here also there is no different with the definitions made in the third chapter aside from the different usage of classes.

Below is an example of a field access using the above class definitions.

```
new [2]Base.Building(new [2]Base.Address(new Object(),
new Object)).addr
→ new [2]Base.Address(new Object(), new Object)
```

To declare a new object, aside from and instance of the class `Object`, the version and package name must always be specified. Following the syntax rules, every field needs to instantiated by the constructor, so we can see that the class `Building` from the second version of package `Base` has only one field. From the class declaration we also know that the field is `addr`, hence by using `.addr` we access that field.

Below is an example of method invocation involving multiple versions.

```
new [1]Municipal.Manage().pair_heads(
   new [1]Education.School(
      new Object(), new Object, new [1]Education.Teacher(...))
   new [1]Health.Hospital(
      new [2]Base.Address(...), new Object(),
      new [1]Health.Doctor(...)))
   ⎡a ↦ new [1]Education.School(...)  ⎤
→  ⎢b ↦ new [1]Health.Teacher(...)     ⎥ new Pair(a.head, b.head)
   ⎣this ↦ new [1]Municipal.Manage()  ⎦
```

Using the above definitions, `new [1]Education.Teacher(...)` uses the first version of `Base` and `new [1]Health.Doctor(...)` uses the second version of `Base`. The mechanism of method invocation in (the langauge) is

identical to Featherweight Java. The square bracket denotes the substitution, with `a` and `b` which are the actual formal parameters of the method, replaced with the actual parameters received. Then we also replace `this` with the receiver object `new [1]Municipal.Manage()`.

The last example is the cast expression.

```
(Doctor) new Surgeon(new Object(), new Object(), new Object(),
new [1]Health.Hospital(...), new Object())
→ new Surgeon(new Object(), new Object(), new Object(),
   new [1]Health.Hospital(...), new Object())
```

Suppose that `new Surgeon(...)` is already a value, we can easily check whether the cast will progress or not. Hence, because `Surgeon` is a subclass of the target cast `Doctor`, the reduction removes the cast. If not, the expression must be of the form `(CV) new DV(...)` where `DV` $\not<:$ `CV`, and the evaluation will stuck.

Chapter 6

Related Work

Family Polymorphism

Family polymorphism [3][6], a programming language feature that introduces the idea of family was a promising candidate as a solution to dependency hell. A family is a unit that shows the relationship between multiple objects. We intended to treat interdependent libraries as a family of libraries, or in other words a version of a set of libraries. Using a shorter version of the example from the preceding chapter, the example below will demonstrate the mechanism of a family.

```
class BaseFirst {
  class Person {
    String name;
    Person{String name}{
      this.name = name;
    }
  }
}

class BaseSecond {
  class Person {
    String name;
    int age;
    Person(String name, int age){
      this.name = name; this.age = age;
    }
  }
}
```

Listing 6.1: Family polymorphism for `Base`

The idea of family polymorphism was formalized in *virtual class calcu-*

lus[]. Using the virtual class calculus, we will treat package in Batak Java as a class. This calculus treats class as another attribute of a class, similar to field and method. Therefore, the inner class `Person` is an attribute of the class `BaseFirst`. Here, `BaseFirst` represents the first version of `Base`. Likewise, `BaseSecond` represents the second version of `Base`. Within the framework of family, the outer class `BaseFirst` can be seen as a family which member is `Person`.

```
class Education extends BaseFirst {
  class Teacher extends Person {
    String id;
    Teacher(String name, String id){
      super(name); this.id = id;
    }
  }
}

class Health extends BaseSecond {
  class Doctor extends Person {
    String branch;
    Doctor(String name, String branch){
      super(name); this.branch = branch
    }
  }
}
```

Listing 6.2: Family polymorphism for `Education` and `Health`

Here we declare `class Education extends BaseFirst` and also `class Health extends BaseSecond`. In a sense it is somewhat similar to `package Education imports Base` and `package Health imports Base` since extending a class in virtual class calculus is equal to inheriting all of its attributes, including inner classes. However, as has been pointed before, Batak Java does not import a particular version of a package, but imports the whole package and then choose the desired version. In contrast, the concept of family polymorphism is less flexible because a particular class has to extend a particular version. This means that anything related to `Education` can only make use of a class belonging to `BaseFirst`. This means we cannot suppose add another class to `Education` which extends `Person` from the other version.

```
class Municipal extends Education, Health {
  class Manage {
    Manage() {}

    boolean compare_building(out.School a, out.Hospital b){
      return a.name == b.name;
```

```
      }
    }
}
```

Listing 6.3: Family polymorphism for `Municipal`

The keyword `out` refer to the enclosing object of the class it belongs to. In the example above, `out` simply refers to the class `Municipal`; `out.School` therefore points to the class `School` within `Municipal`.

Another point that family polymorphism lacks is the incapability of discerning two similar named classes or methods. First we have `class Municipal extends Education, Health` which enables its inner classes to use the classes declared within `Education` and `Health`. Unfortunately we face another problem during the method invocation of `compare_building`. Both `out.School` and `out.Hospital` extend the class `Person` in its implementation. Family polymorphism in unable to discern the two `Person`s. Instead it linearizes the definitions, which eventually chooses only one of the two versions, a result which opposes our objective.

Version Lambda

This research is partly inspired by the VL which introduces the idea of version from the concept of context-oriented programming. Versions are considered contexts and so allow the existence of multiple versions of classes by using layers. Unlike Batak Java which assign classes with versions and so discern values through its class, the version lambda calculus assign version to values and definitions. The calculus is based on a coeffect system [1] based on the simply typed lambda calculus.

Variational Programming

The calculus for variational programming attempts to allow variations within programming. It argues that variations can be useful in software product lines. The calculus uses the concept of choice between left and right to give variation to values and also their types.

Chapter 7

Conclusion and Future Work

We propose Batak Java, an object-oriented programming language that allows the existence of similarly named classes within different versions through the introduction of version as attribute of package. With that capability, we showed how Batak Java can be used to write a program which uses multiple versions of the same library. We also showed that Batak Java can deal with an example of dependency hell, in the form of diamond dependency.

Our future work include introducing *multiple-versioned* type, instead of the current single-versioned type. With such feature, we can declare an object such as `new [1,2]Base.Person(...)`. This will greatly increase the flexibility of the language as it allows us to extract common properties from multiple versions at once. Although currently programmers do not program in such manner, the possibility of having such a feature may open up new way to program.

Afterwards we also need to introduce the concept of *version polymorphism*. Version polymorphism which allows the representation of an unbounded number of versions using a parameter. Doisng so will help simplify how the type of an object needs to be expressed.

Once we added version polymorphism into the calculus, another extension to the current calculus that we would also like to add in the future is union type [4]. Union type denotes a set of some given types. Suppose we have a package `Foo` with two version, where a method `X` has return type `Int` in the first version and return type `Float` in the second version. Another package which imports `Foo` and intends to use `X` will find difficulties to determine the result type unless it specifies which version it uses.

References

[1] A. Brunel, M. Gaboardi, D. Mazza, and S. Zdancewic. A core quantitative coeffect calculus. In *Programming Languages and Systems*, 2014.

[2] S. Chen, M. Erwig, and E. Walkingshaw. A calculus for variational programming. In *Proceedings of European Conference on Object-oriented Programming (ECOOP2016)*, June 2016.

[3] E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on principles of programming langauges*, 2006.

[4] A. Igarashi and H. Nagira. Union types for object-oriented programming. In *Proceedings of the 2006 ACM symposium on applied computing*, Apr. 2006.

[5] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, pages 396–450, May 2001.

[6] C. Saito, A. Igarashi, and M. Viroli. Lightweight family polymorphism. *Journal of Functional Programming*, pages 285–331, May 2008.

[7] Y. Tanabe, T. Aotani, and H. Masuhara. A context-oriented programming approach to dependency hell. In *10th International Workshop on Context-oriented Programming*, July 2018.