Reduction and typing of class definitions with complex content

Peter Vanderbilt

pvanderbilt@acm.org

Abstract

This paper describes a reduction of, and an implied typing rule for, class definitions with complex content. A class with *complex content*, or more simply a "complex class," is one with virtual types and virtual nested complex objects, as well as regular data fields (including methods). A key feature is that the types of data fields can depend on the *values* of the (possibly nested) type fields.

A *reduction* is a syntactic mapping of a higher-level definition to a series of lower-level definitions. The lower-level definitions can be taken as the detailed, concrete meaning of the higher-level one. This reduction is type preserving in the sense that any type-correct high-level program is reduced to a type-correct lower-level one.

This reduction yields, among other things, a definition of a first-class *class object*, a runtime representation of a class that can be passed to functions and stored in data structures. The reduction also yields a definition of a *class object type* whose instances are the class objects of the subclasses of this class.

We take the approach that complex virtual fields have exact and general aspects. The *exact* aspect of a field describes its implementation, while its *general* aspect constrains the ways in which the field can be overridden. We describe how a class body maps to these aspects and how these aspects relate to typing. In particular, the reduction yields definitions of two types per class C: the *exact type*, $C@$, provides type-safe access to all fields, while the *general type*, $C\#$, supports subclass polymorphism.

This paper presents one experiment in the design of complex classes and contributes ideas rather than being a self-contained, mathematically formal specification.

Keywords: virtual types, reduction, class object, soundness, inheritance

1. Introduction

This paper describes one piece of research in a field related to static typing of nested classes. Our goal has been to come to a deep understanding of features such as virtual types [7, 9, 15], virtual classes [16], extensible mutual recursion [11, 18, 19] and what we call sibling extension. We have been working on three fronts simultaneously: (a) working through popular examples, (b) defining reductions and (c) defining a lower-level language and its typing logic. As will be discussed in subsection 1.1, a reduction maps a higher-level thing, such as a class, to terms in the lower-level language.

Given example high-level code, such as for extensible Subject/Observer [7, 13, 22], its concrete reduction yields code in the lower-level language that should be syntactically correct, type-correct and have the intended semantics. In this case, reductions are applied to the Subject and Observer classes and then to their extensible container. Getting all this to work out has been non-trivial, especially the typing. We have applied our reductions to many extensibility examples [3, 11, 12, 22, 24–26] and others of our own design. These examples require additional features that lead to new reductions. We do not yet have a "unified reduction" that handles everything, instead we have a number reductions, each handling a different experimental variant of a high-level language. This paper presents one such reduction.

1.1 Reduction

A key aspect of our methodology is to define reductions for classes and other higher-level constructs. For example, the following is a simplistic reduction for simple top-level class definitions:

$$\begin{array}{l}\textsf{class}\ C\ \{\ \textsf{const}\ \bar{x} : \bar{T} = \bar{e}\ \} \longrightarrow \\ \quad \textsf{type}\ C = \{\ \bar{x} : \bar{T}\ \}\ ; \\ \quad \textsf{func}\ C\_\textsf{new}\ ()\ :\ C = \mu(\textsf{self})\ \{\ \bar{x} = \bar{e}\ \}\end{array} \tag{1.1}$$

Without going into too much detail, to the left of the reduction arrow, $\longrightarrow$, is a class definition pattern and to its right is a series of type and function definition patterns that are taken as being equivalent. Symbols C, $\bar{x}$, $\bar{T}$ and $\bar{e}$ are all pattern variables appearing on both sides that are instantiated when the left-hand pattern is matched against an actual class definition. The μ operator signals recursion. Throughout this paper, over-bars indicate parallel lists, so for example $\bar{x}{:}\bar{T}$ is really $x^1{:}T^1;\ldots;x^n{:}T^n$ (for some n).

After the reduction arrow in (1.1), the first definition is of a type C, the type of class C's instances. The second definition is of a function, $C\_\textsf{new}$, that implements "**new** C". It has C as its return type because we require "**new** C" to return instances of C. Note that these definitions are built using lower-level facilities such as records, record types, functions and recursion and, so, give a concrete meaning to the higher-level class definition.

In section 2, we describe a considerably more complicated set of class definitions and give their reduction in section 3.

1.2 Type preservation

Consider the following simplistic typing rule for simple, top-level class definitions:

$$\frac{\Gamma \vdash \bar{T} :: *\qquad \Gamma;\ \textsf{self}{:}\{\bar{x}{:}\bar{T}\} \vdash \bar{e} : \bar{T}}{\Gamma \vdash \textsf{class}\ C\ \{\ \textsf{const}\ \bar{x} : \bar{T} = \bar{e}\ \} \rightsquigarrow} \tag{1.2}$$

EXAMPLE-CLASS-INTRO

The symbol Γ denotes the environment present before the definition of the class. The conclusion is a new form of judgement that uses a "yields" relation, "$\rightsquigarrow$", where "$F \rightsquigarrow \bar{L}$" means that the execution of definition F results in a state satisfying declarations $\bar{L}$. In this case, the following declarations describe the new bindings resulting from the class definition:

```
type C ↔ {x̄:T̄};                                        (1.2)
func C_new() : C
```

So basically the rule says that, given that $\bar{T}$ are types and given that $\bar{e}$ are of type $\bar{T}$ given that self (aka this) has the indicated type, then the class specified in the conclusion is type correct and its execution binds new type name "C" to $\{\bar{x}{:}\bar{T}\}$ and binds "C_new" to a function that yields objects of type C.

Now consider the reduction of a program $\mathcal{P}$ that is formed by replacing class definitions using (1.1) and replacing any "**new** C" by C_new(). This reduction is *type preserving* in the sense that, if $\mathcal{P}$ is type correct using the rule above, then the reduction of $\mathcal{P}$ is type correct. This follows from the observation that the first premise of the rule is exactly what is needed to make the type definition of (1.1) well-defined and the second premise ensures that its function definition is well-typed.

To some extent we can derive a typing rule from a reduction: by working backwards from what makes the lower-level program type-correct, we can determine the premises of a sound higher-level rule. In section 4, we describe a typing rule for the classes of section 2 that is derived from the reduction of section 3.

1.3 Class objects

Note that reduction (1.1) maps one definition to two. To make this one-to-one, we can redefine (1.1) to yield an object definition:

```
class C { const x̄ : T̄ = ē } ⟶                          (1.3)
  hconst C = μ(C) {
    type Instance = { x̄ : T̄ } ;
    func new () : C.Instance = μ(self) { x̄ = ē }
  }
```

The keyword **hconst** introduces a definition of a hybrid constant; it will be discussed further in subsection 1.6.

We call the resulting object, C, a *class object* because it is an object that represents the higher-level class to the left. It has a type field Instance that is the type of C's instances and a function, new, that creates such instances. We use C@ as an abbreviation for C.Instance, so the return type of new could be written C@.

Notice that object C has a type field and that the type of new depends on the value of that type. This is called a *dependent record* [15, 21], but we call it a *hybrid* for simplicity (because it is a cross between regular record and a type group [3, 7]).

An additional advantage of reduction (1.3), maybe the major one, is that class objects are first class in that they can be passed as parameters and stored in objects. For example, consider the following method in some partial class Iterable[T] that can be used to apply a given function to every element and collect the results in an instance of a given class:

```
func mapTo [R] (f:(T)→R, *C<<Buildable[R]) : C@        (1.4)
{
  const obj = C.new();
  foreach (e in self.elements()) { obj.add(f(e)) };
  return obj;
}
```

The second argument to mapTo is a class object, which supplies both the way to create the collection object (C.new) and the type of the result (C@, aka C.Instance). The star, *, before C indicates that C can appear in the function's parameter and return types (as it does in the latter). The "extends" relation, "<<", will be described in section 2.2.3. The following defines Buildable[R]:

```
interface Buildable [R] {                              (1.5)
  func add (R) : Void;
  static func new () : Self@
}
```

Self@ is the self-type.

Assume that class IntSet is such that IntSet<<Buildable[Int] and that a is an object implementing the Iterable interface. Then the following expression has the indicated type (for appropriate f):

```
a.mapTo[Int](f, IntSet) : IntSet@                      (1.6)
```

This return type indicates that an instance of IntSet is returned, which is good since IntSet.new() was used to create it.

The reduction of section 3 yields a definition of a class object like the one of (1.3), except that it is not complete in that the reduction also defines other entities separate from the class object.

1.4 Complex content

In order to handle virtual nested/inner classes and types, classes need to support fields containing classes and types. For example, if class C above is nested within class B, we would like to reduce C within the body of B and then reduce B. But when we reduce the inner class using (1.1), we get the following outer class:

```
class B {                                              (1.7)
  type C = { x̄ : T̄ } ;
  func C_new () : C = μ(cself) { x̄ = ē };
  /* other content, presumably referring to the above */
};
```

Although reduction (1.1) cannot handle the type field (and certain other technical issues), any reduction that can will result in hybrid instances, so B_new will be a hybrid-yielding function and B will be a type over hybrids. If B is then a member of further outer class A, then A_new will be a function that returns a hybrid that contains a hybrid type and a hybrid-yielding function; A will be a type over such things. The reduction becomes more complicated, and its nesting even worse, when features are added to handle standard features such as extension, overriding, abstract fields, class polymorphism and static fields.

We call such content *complex* and we end up needing complex content that includes data, hybrids with complex content, types over complex content, classes with complex content and functions from and to complex content. The reduction of section 3 handles all this, except that handling nested classes requires additional mechanisms.

1.5 Exact and general

Our approach is that a class and its type are different entities and that subclassing does *not* imply subtyping [5, 10]. In fact, we define *two types* per class C, an *exact type* C@ and a *general type* $C\#$, much like Bruce's types of the same name [2, 6]. Type C@ is the type of instances of class C while type $C\#$ is the type of instances of C and its subclasses. The following relationships are true of any class C and its subclass B:

```
new C : C@ <: C#                                       (1.8)
new B : B@ <: B# <: C#
```

However B@ does *not* subtype C@.

Having these two types per class solves certain problems with classes containing virtual complex fields: C@ provides safe access, while $C\#$ provides subclass polymorphism. In particular, given a variable c of type C@, there is full access to all members of c. Such access is safe because an instance of class B can not be assigned to a C@ variable.

For subclass polymorphism, the variable c would have kind $C\#$ so that it can be assigned instances of C or B or any other subclass. Type $C\#$ is such that methods whose types reference the self-type or virtual types (particularly in contra-variant positions) generally cannot be safely used, thus preserving type safety. However, regular methods are safe.

The notions of exact and general have been expanded to permeate all through the reduction. In particular, the class's field definitions are broken down into exact and general components. The exact components are collected together to implement the class and to define type $C@$. The general components are collected together to define type $C\#$ and the type of subclasses of C.

1.6 Kinds

In this subsection, we give an overview of the unusual aspects of our kind system. Our kinds are of the following form:

$$K ::= * \mid \epsilon \mid [\bar{K}]\Rightarrow K \mid *(K) \mid \{\bar{X}::\bar{K}\} \ . \tag{1.9}$$

Kind $*$ is the kind of types and $[\bar{K}_d]\Rightarrow K_r$ is the kind of functions from meta-expressions of kind $\bar{K}_d$ to those of kind K_r. Below we describe hybrids kinds, $\{\bar{X}::\bar{K}\}$, meta-type kinds, $*(K)$, and the "empty" kind, ϵ.

First, consider the following definition of a hybrid:

$$\textbf{hconst}\ \mathsf{x} = \{\ \textbf{const}\ \mathsf{k}{=}5;\ \textbf{type}\ \mathsf{T}{=}\mathsf{Int}\ \} \tag{1.10}$$

Note that the hybrid literal to the right of the equal sign has both terms and types in it and, so, it is neither purely a term nor purely a type. Similarly, x has both data-like aspects (x.k) and type-like aspects (x.T).

To handle this, we assign the kind $\{\mathsf{T}::*\}$ to x to indicate that x.T has kind $*$, which means that x.T is a valid type expression. Field k does not show up in the kind because it is not type-level. In general, a hybrid kind is a record-like construct of the form $\{\bar{X}::\bar{K}\}$ and a value with such a kind must have at least meta-level fields $\bar{X}$ of corresponding kinds $\bar{K}$.

Actually, the result of definition (1.10) is described by a declaration that assigns x both a kind and a type:

$$\mathsf{x} :: \{\ \mathsf{T}::*\ \} : \{\ \mathsf{k}{:}\mathsf{Int};\ \mathsf{T}{:}\{\{?{\leftrightarrow}\mathsf{Int}\}\}\ \} \tag{1.11}$$

The kind is as described above. The type "$\{\mathsf{k}{:}\mathsf{Int};\mathsf{T}{:}\{\{?{\leftrightarrow}\mathsf{Int}\}\}\}$" indicates that x has at least a field k of type Int and a field T of type $\{\{?{\leftrightarrow}\mathsf{Int}\}\}$, which is a type type whose instances are those types equal to Int. Note that our notions of "kind" and "type type" are different, where "kind" describes structure and "type type" describes relations to other types.

We use the kind $*(K)$ as the kind of meta-types whose instances are of kind K. For example, the type type $\{\{?{\leftrightarrow}\mathsf{Int}\}\}$ has kind $*(*)$ (a type of types) and the kind of the hybrid type appearing in (1.11) is $*(\{\mathsf{T}::*\})$ (a type over hybrids having a type field T).

For uniformity we choose to give *all* variables a kind and a type, where the special kind ϵ (the empty kind) indicates that the thing is term/data level, not type-level. Thus definition $\mathsf{n}{=}0$ results in declaration $\mathsf{n}::\epsilon{:}\mathsf{Int}$. We take $*(\epsilon)$ as equal to $*$. Also, fields of kind ϵ are ignored when creating a hybrid kind and the empty hybrid kind, $\{\}$, is equal to ϵ.

We use introductory keywords to give hints as to the kinds of things. The keywords **const** and **func** imply kind ϵ, **type** implies $*$, **hconst** and **hfunc** imply a hybrid kind and **meta** implies some non-ϵ kind, usually given explicitly.

1.7 Contributions

This paper describes a reduction of classes that have both member and static fields containing complex content as described in section 1.4, except nested classes. This reduction also yields a class object and a related class object type.

We use an underlying language that is similar in style to Java [14] and Scala [20] but having features of our own invention. The definition of this language and its logic is beyond the scope of this paper, but we describe key features as we go. It is meant as a vehicle for explanation, so that the features described here can be added to any language with similar expressivity. We assume the underlying language has a sound typing system.

In section 2, we describe the syntax and meaning of the class definitions that are the subject of the reduction and we describe the structure of class objects. In section 3, we define the reduction. In section 4, we define a typing rule that is derived from the reduction. In section 5, we describe several variations on the reduction, including its simplification when applied to simple classes, handling generic classes and support for self.Class. Finally, in section 6, we summarize this work and give some directions for future research. Appendix A contains a detailed justification of the typing of section 4.

2. Complex classes and class objects

In this section, we give the syntax of and restrictions on the classes that are the subject of the reduction of section 3 and we describe the expected results of such a class definition.

2.1 Syntax of Class Definitions

The following grammar defines the syntax of classes (as handled by this reduction):

$$J^C \ ::=\ \textbf{class}\ C\ (\bar{p}{:}\bar{T}_p)\ \textbf{extends}\ D(\bar{a})\ \{\ \bar{\mathcal{F}}\ \}\ . \tag{2.1}$$

Non-terminals C and D range over class identifiers, $\bar{p}$ over identifier lists (of constructor parameters), $\bar{T}_p$ over type lists of the same length as $\bar{p}$, $\bar{a}$ over expression lists and and $\bar{\mathcal{F}}$ over field definition lists (to be described shortly). Lists $\bar{p}$ and $\bar{T}_p$ are parallel, but $\bar{a}$ and $\bar{\mathcal{F}}$ are independent. The names defined by $\bar{\mathcal{F}}$ must be distinct from each other and the names of "new" fields (those not inherited) must be distinct from the names of D.

$\bar{\mathcal{F}}$ ranges over lists of class field definitions, each as described by the following grammar:

$$\begin{array}{rcl} \mathcal{F} & ::= & [\ \textbf{member} \mid \textbf{static}\ ]\ \mathcal{J}\ . \\ \mathcal{J} & ::= & [\ \textbf{virtual} \mid \textbf{refine}\ ]\ \mathcal{V} \\ & \mid & \textbf{abstract}\ L \\ & \mid & (\ \textbf{override} \mid \textbf{impl}\ )\ F\ . \end{array}$$

Non-terminal $\mathcal{V}$ will be described in subsection 2.1.1. Non-terminal L ranges over declarations and F over definitions, as discussed in section 2.1.2. The **virtual** keyword is the default and, so, can be omitted.

The **member** keyword means that the definition is for the instance, while **static** means that the definition is for the class object; if neither is specified, **member** is implied.

A field marked **virtual** is a new field that provides both general and exact aspects of a field. A field marked **refine** narrows the general declaration of an inherited field (one defined by some superclass) and provides a new implementation. An **abstract** field declares a new field without an implementation. An **override** changes the implementation of an inherited virtual field while an **impl** field implements an inherited abstract field. Other varieties of fields, such as **final**, **fragile** and **local**, are topics of future research.

If the class has any **abstract** fields, or inherits unimplemented abstract fields, it must be marked **partial**. A partial class does not generate instances, nor have a class object, as to be described shortly. In contrast, a *concrete* class is fully functional. Although we don't show it, an **interface** is a partial class where all the fields are implicitly **abstract**.

2.1.1 Virtual fields

The exact syntax of virtual fields, as denoted by $\mathcal{V}$, is a topic of future research, especially those for general hybrids and nested classes. However, for this reduction all we need is that complex field definitions reduce to one of the following three forms:

$$\begin{array}{rcl} \mathcal{V} & ::= & \textbf{const}\ x : T = e \\ & | & \textbf{meta}\ X :: K : Z = M \\ & | & \textbf{hconst}\ x :: K : T_g = (T_e)\ e\ . \end{array} \quad (2.2)$$

Non-terminals x and X range over identifiers, T and its subscripted versions over types, e over expressions, K over kinds, Z over meta-types and M over meta-expressions. Meta-expressions generalize type expressions and include types, hybrid types, type functions and type types. Two meta-types are "$\{\{?{\leftrightarrow}M\}\}$", which denotes the set of things that equal M, and "$\{\{?{<:}T\}\}$", which denotes the set of types that subtype T. Certain names are reserved and, so, can not be defined by the programmer; "Instance" and "Class" are two such names.

The **const** form defines a virtual data constant whose value is e which must be of type T, as must any override. The **meta** form defines a virtual type-level constant whose kind is K, whose value is M and whose overrides are constrained to satisfy meta-type Z. The **hconst** form defines a virtual hybrid constant whose name is x, whose kind is K, whose value is e which must be of (hybrid) type T_e and whose overrides must satisfy the type T_g. The two constant forms have variable forms, **var** and **hvar**, but we ignore them to make the reduction simpler.

Notice that in the **hconst** form, the exact type, T_e, is explicitly required, in addition to the general type. This is because, to simplify the logic, definitions in a recursion must be explicitly typed. In a practical system, T_e would be inferred from e most of the time.

Among higher-level virtual field definitions are the following, together with their reductions:

$$\begin{array}{l} \textbf{virtual func}\ x_f\ (\bar{x}_d : \bar{T}_d) : T_r = e \longrightarrow \\ \quad \textbf{virtual const}\ x_f : (\bar{T}_d){\rightarrow}T_r = \lambda(\bar{x}_d)e \\ \textbf{virtual type}\ X : Z = T \longrightarrow \\ \quad \textbf{virtual meta}\ X :: \textsf{*} : Z = T \\ \textbf{virtual type}\ {+}X = T \longrightarrow \\ \quad \textbf{virtual meta}\ X :: \textsf{*} : \{\{?{<:}T\}\} = T \end{array} \quad (2.3)$$

Kind "*" is the kind of types and $\{\{?{<:}T\}\}$ as described above. The **func** form is useful for defining methods. The first **type** form defines a virtual type field named X whose value is T and whose overrides are constrained to satisfy the type type Z. The second **type** form defines a virtual type field named X whose exact value is T and whose overrides must subtype T. Other forms of virtual field definitions can be added to the language by specifying how they reduce to one of the forms of (2.3).

2.1.2 Other fields

Fields marked **refine** have the same syntax as virtual fields, except that the kind annotations can be omitted, in which case they are determined in ways beyond the scope of this paper. The grammar of L for **abstract** fields is like (2.2) except that the equal sign and everything to its right is missing. The following is the grammar of F for **override** and **impl** fields:

$$\begin{array}{rcl} F & ::= & \textbf{const}\ x = e \\ & | & \textbf{meta}\ X :: K = M \\ & | & \textbf{hconst}\ x :: K : T_e = e\ . \end{array} \quad (2.4)$$

Again the kind and type annotations can be omitted in certain cases. For all these field varieties, there are also higher level forms with reductions similar to those of (2.3).

2.1.3 Self and self

Within a class definition, the special names self and Self refer to the future instance and class, respectively. In particular, if object x is created by "**new** C", then references to self within field implementations are bound to x and Self to C. As is traditional, member fields can be referenced directly, as m, but the canonical name self.m is used in the reduction. Additionally, the special names super and Super refer to records containing member and static fields (respectively) inherited from superclasses.

The special variables Self and self can appear free in field types. For example, a field could have declaration f:()→self.T, where T is a virtual type definition. This means that for object x of this class, the type of x.f is ()→x.T, which is a type that depends on the *value* of the T field of *instance* x. The implementation of field f, because it can be inherited, must work for self.T as defined in this class, as well as for all future definitions of self.T in subclasses. Thus inheritable code must be polymorphic over self and Self of appropriate types. The type Self@ is the type of instances of the future class denoted by Self and is referred to as the *self type* or Bruce's MyType [1, 4, 6, 8, 15, 17, 23].

Note that self and Self support extensible mutual recursion. Consider the following outline of a class, Graph:

```
class Graph {                                                    (2.5)
  static type +Edge = { const n1,n2 : Self.Node };
  static type +Node = { const edges : List[Self.Edge] };
  ...
}
```

The ellipsis denotes the major part of the class, which references these static types as Self.Edge and Self.Node. Note that each type refers to the other. The reduction will yield types such that the following assertions hold:

```
Graph.Edge ↔ { const n1,n2 : Graph.Node }                        (2.6)
Graph.Node ↔ { const edges : List[Graph.Edge] }
```

A subclass, say CGraph, will maintain the recursive relationship:

```
CGraph.Edge <: { const n1,n2 : CGraph.Node }                     (2.7)
CGraph.Node <: { const edges : List[CGraph.Edge] }
```

For the reduction of this paper, the variables super and Super can not appear in types, only in implementations. Also C itself cannot appear free in types nor implementations.

2.2 Result of class definitions

A class definition results in several entities, including various types, a class object and mechanisms for extension. In particular, the class definition of (2.1) creates at least the following entities:

- An implementation of "**new** C", the expression that creates instances of class C. It will be described in section 2.2.2.
- Two instance types C@ and $C\#$, as introduced in section 1.5. The structure of instance objects will be described in section 2.2.1 and C@ will be described in 2.2.2. The description of $C\#$ is deferred to section 3.4.4.
- A class object C, to be described in section 2.2.2.
- A class type $\{\{?{\ll}C\}\}$ whose instances are the class object of C and those of C's subclasses. It will be described further in section 2.2.3.
- Type constructors and functions used for extension. Their details will be given in section 3.

2.2.1 Structure of instance objects

The expression "**new** $C(...)$" creates instances of C. These instances contain those fields declared **member** in C or inherited from D (including those from D's super-classes). Member fields are accessed as $o.m$ for instance object o and member field name m.

2.2.2 Structure of class objects

The class object, as introduced in section 1.3, is a first-class object named C that contains at least the following fields:

- Those fields declared **static** in C or its superclasses. Static fields are accessed directly, as $C.s$ for static field name s.
- A type field named Instance that is the type of instances created by class C. We define type $C@$ to be syntactic sugar for the Instance type field:

$$C@ \triangleq C.\mathsf{Instance} \tag{2.8}$$

- A function new that creates instances of class C. The following syntactic sugar is used to access new:

$$\begin{array}{lcl} \textbf{new}\ C(\bar{e}) & \triangleq & C.\mathsf{new}(<\bar{e}>) \\ \textbf{new}\ C & \triangleq & C.\mathsf{new}(<>) \end{array} \tag{2.9}$$

  For technical reasons, the parameters, $\bar{e}$, are packaged as a tuple and passed to C.new. "**new** C" is an abbreviation for "**new** $C()$". Expressions of the form "**new** $C(\bar{e})$" return an object of type $C@$.

2.2.3 The extends relation

Class type $\{\{?<<C\}\}$ has as its instances the class object of C and those of C's subclasses. It is a hybrid type with declarations for C's static fields and type Instance. Its instances must be concrete classes.

We define the following syntactic sugar:

$$B << C \triangleq B : \{\{?<<C\}\} \tag{2.10}$$

Thus $<<$ can be viewed as a relation between classes such that $B<<C$ means that B is a concrete subclass of C. For a variable B bounded only by extends ($B<<C$), $B@$ is defined (as an alias for B.Instance by (2.8)), but none of $B\#$, $\{\{?<<B\}\}$ nor "**new** B" are defined; "**new** B" is undefined because B's initialization parameters are not known.

A key use of the extends relation is with class parameters. It appears in the example of mapTo, (1.4) of section 1.3, so that class parameter C ranges over classes that implement interface Buildable[R] as defined by (1.5).

3. Reduction

This section describes a reduction of classes with complex content, as introduced in section 1.1. Its input are those classes defined in the last section. Subsection 3.1 defines the basic structure of the reduction with forward references to its following subsections.

3.1 Definition of the reduction

The following defines a reduction of classes with complex fields as described in section 2.1:

class C $(\bar{p}{:}\bar{T}_p)$ **extends** $D(\bar{a})$ { $\bar{\mathcal{F}}$ } $\longrightarrow$ (3.1)
  let
    / rewrites (3.2) to (3.5) */*
  in
    / definitions (3.6) to (3.21) */*

Superclass D must have been reduced by this reduction. The rewrites of (3.2) to (3.5) will be defined in section 3.2 and definitions (3.6) to (3.21) will be defined in sections 3.3 to 3.5.

To the left of the reduction arrow, $\longrightarrow$, is a pattern that matches classes whose syntax is as given in section 2.1; each non-terminal of the grammar relates to a meta-variable which will match the appropriate part of the actual class definition to be reduced.

As we will see in section 3.2, (3.2) to (3.5) define a collection of rewrites of the form $\bar{\mathcal{P}}{\approx}\bar{\mathcal{E}}$. The phrase "*let* $\bar{\mathcal{P}}{\approx}\bar{\mathcal{E}}$ *in* $\mathcal{R}$" indicates that each meta-variable $\mathcal{E}^i$ in turn is rearranged to have the form matching pattern $\mathcal{P}^i$, instantiating any meta-variables in $\mathcal{P}^i$; after the matching, $\mathcal{R}$ (which incorporates those meta-variables) is the result of the reduction. The symbol $\approx$ denotes syntactic equality, but allowing rearrangement and canonicalization.

In particular, the rewrites of (3.2) to (3.5) cause meta-variable $\bar{\mathcal{F}}$ to be matched instantiating $\bar{\mathcal{J}}_s$ and $\bar{\mathcal{J}}_m$, which in turn are matched to instantiate a bunch of other meta-variables which are then used in the definitions of (3.6) to (3.21).

As described in section 3.3, (3.6) defines kind identifiers useful in the following definitions. As described in section 3.4, (3.7) to (3.17) define a collection of type constructors and types. As described in section 3.5, (3.18) to (3.20) define functions used for constructing class object C and its extensions and (3.21) defines class object C.

Naming conventions: Because there are so many entities yielded by the reduction, we use a naming convention. The class object is named "C", while auxiliary entities' names start with "$C\_$". After the underscore is either αK (kind), $\alpha\beta$Tδ (type) or αc (constructor function). α is either s (static), m (member), or nothing (when about the class object itself). β is either e (exact) or g (general). δ is zero or more letters c ("constructor") to indicate the level of parameterization. Types iTc ("initialization" type constructor) and hT ("hash" type) are also defined.

3.2 Rewrite field definitions

The first step of reduction (3.1) is to rewrite the class's body into a particular form:

$$\begin{array}{ll} \text{"}\textbf{static}\ \bar{\mathcal{J}}_s;\ \textbf{member}\ \bar{\mathcal{J}}_m\text{"} & \approx \bar{\mathcal{F}}; \\ \text{"}\textbf{exact}\ \bar{x}_{\alpha e} :: \bar{K}_{\alpha e} : \bar{T}_{\alpha e} = \bar{e}_\alpha; & \\ \quad \textbf{general}\ \bar{x}_{\alpha g} :: \bar{K}_{\alpha g} : \bar{T}_{\alpha g}\text{"} & \approx \bar{\mathcal{J}}_\alpha \end{array} \tag{3.2}$$

The presence of α subscripts in the second rewrite (the last two lines) indicates that it actually denotes two rewrites (one each for α=s and α=m).

The first part of (3.2) reorders the field definitions so that $\bar{\mathcal{J}}_s$ has the static definitions and $\bar{\mathcal{J}}_m$ has the member ones. The second part splits each field definition into up to seven components as described in subsections 3.2.1 and 3.2.2. This instantiates a collection of 14 meta-variables which are used in the reduction (sections 3.3 and later). Using $\alpha\in\{$s,m$\}$ and $\beta\in\{$e,g$\}$ to abbreviate, these meta-variables are: $\bar{x}_{\alpha\beta}$ (4 lists of field names), $\bar{K}_{\alpha\beta}$ (4 lists of kinds of $\bar{x}_{\alpha\beta}$), $\bar{T}_{\alpha\beta}$ (4 lists of types of $\bar{x}_{\alpha\beta}$) and $\bar{e}_\alpha$ (2 lists of implementations of $\bar{x}_{\alpha e}$). For each pair of α and β, the lists are parallel.

3.2.1 Rewrite virtual fields

As introduced in section 1.5, we take the approach that each virtual field definition has two main aspects: its exact definition and its general declaration. The *exact definition* of a field defines the field's implementation; it has the following form:

exact $x :: K : T = e$

Non-terminal x ranges over identifiers, K over kinds, T over types and e over expressions. This defines a field whose name is x and whose value is that resulting from e. K and T are kind and type *announcements*, which are intended properties of field x.

The *general declaration* specifies what must be true of this class and its extensions. It has the following form:

general $x :: K : T$

Again x ranges over identifiers, K over kinds and T over types. This specifies that the field x must be of type T in all extensions of this class.

For non-complex (data) virtual fields, the general and exact declarations are the same. The following equivalence shows how a simple data definition is represented in its two aspects:

$$\textbf{exact } x :: \epsilon : T = e;\ \textbf{general } x :: \epsilon : T \approx \textbf{virtual const } x : T = e \tag{3.3}$$

Recall that the symbol ϵ denotes the empty kind, the kind of pure data. Note that the exact and general types are the same.

The general form of a virtual meta-constant definitions is given as follows, with its equivalent:

$$\textbf{exact } X :: K : \{\{?\leftrightarrow M\}\} = M;\ \textbf{general } X :: K : Z \approx \textbf{virtual meta } X :: K : Z = M \tag{3.4}$$

Note that the exact type is one that captures the current value of M: $p.X{:}\{\{?\leftrightarrow M\}\}$ implies $p.X \leftrightarrow M$.

The rewrite for a hybrid constant is as follows:

$$\textbf{exact } x :: K : T_e = e;\ \textbf{general } x :: K : T_g \approx \textbf{virtual hconst } x :: K : T_g = (T_e)\ e \tag{3.5}$$

Recall that a virtual **hconst** has general and exact types included. Thus the exact part is generated from the exact type and implementation, while the general part is generated from the general type.

3.2.2 Rewrite other fields

Fields marked **refine** are rewritten just like **virtual** above, fields marked **abstract** rewrite to just **general** definitions (with no **exact** part) and fields marked **override** or **impl** rewrite to just **exact** definitions (with no **general** part). Some kind or type annotations may be missing in the source field, in which case they are determined in ways beyond the scope of this paper.

3.3 Define kinds

Since complex classes contain complex fields, type paths can involve them. The second step of the reduction is to define kinds to keep track of what paths are legal:

$$\textbf{kind } C\_\alpha\beta\mathsf{K} = D\_\alpha\beta\mathsf{K} \textbf{ with } \{\ \bar{x}_{\alpha\beta} :: \bar{K}_{\alpha\beta}\ \};\ \textbf{kind } C\_\beta\mathsf{K} = C\_\mathsf{s}\beta\mathsf{K} \textbf{ with } \{\ \mathsf{Instance} :: \mathsf{*}(C\_\mathsf{m}\beta\mathsf{K})\ \} \tag{3.6}$$

If there is no superclass, the **with** and text to its left are not present; otherwise, $D\_\alpha\beta\mathsf{K}$ are the kinds defined by D's version of (3.6). Operator "**with**" unions the field declarations of two hybrid kinds, where a declaration from the right argument overrides a same-named one from the left. To the right of **with** are hybrid kinds as introduced in section 1.6. Recall that data fields have kind ϵ and are ignored when creating a hybrid kind.

To save space, we continue the convention that α and β subscripts denote multiple definitions, one for each set of reasonable values of α and β. In this case, the first line denotes four definitions (of kinds C_seK, C_sgK, C_meK and C_mgK) and the second denotes two definitions (of C_eK and C_gK).

The first definition of 3.6 defines four kinds, $C\_\alpha\beta\mathsf{K}$, each inheriting from $D\_\alpha\beta\mathsf{K}$ with overriding kind declarations of appropriate fields defined in C. The second definition defines class object kinds, C_eK and C_gK, from the corresponding static kind plus a declaration of a type field, Instance, whose instances have the corresponding member field kinds.

3.4 Define types

The third step of reduction (3.1) is to define the types needed to annotate the constructor functions to come. First we define a few mechanisms that are needed for these definitions.

3.4.1 Mechanisms

Type constructors: Many of the entities to be defined are actually type *constructors*, (compile-time) functions from types to types. They are written using a definition of the form "$X[\bar{Y}::\bar{K}]{=}T$" which reduces to "$X{=}\lambda[\bar{Y}::\bar{K}]T$" in canonical form. When a function is over hybrids (as they all are since Self and self are hybrids), it operates on the (possibly nested) type fields of its arguments; the argument's data fields are not relevant. We use a "c" after "T" in the name to suggest "constructor".

To save space in the constructor definitions, we leave out the kind annotations: we take C_gK as the kind of parameter Self, C_eK as the kind of C and C_mgK as the kind of self.

The τ operator: We use our operator "τ" to construct dependent types. It parallels μ when constructing recursive hybrids (but is not type-level μ.). The following two rules reason about τ:

$$\frac{\Gamma;\ x : T \vdash e : T}{\Gamma \vdash \mu(x)e : \tau[x]T}\ \tau\text{-INTRO} \qquad \frac{\Gamma \vdash p : \tau[x]T}{\Gamma \vdash p : T\langle x{\mapsto}p\rangle}\ \tau\text{-ELIM}$$

Type T usually has x free; otherwise the τ isn't needed. In the second rule, p must be a path. Technically, the assertions of these rules are at the level of some kind K and the first rule should have the premise "$x{::}K \vdash T{::}\mathsf{*}(K)$" to ensure that T is well-kinded.

The first rule types a value-level recursion, $\mu(x)e$, where the type of e depends on the value of x – it arranges (via the second rule) that, if p is a path leading to an object created by $\mu(x)e$, then the type of p is $T\langle x{\mapsto}p\rangle$, which is T with p substituted for x.

Now we return to defining the type constructors and types produced by the reduction.

3.4.2 Instance-level types

$C_$iTc: First we define type C_iTc to be the tuple type containing $\bar{T}_p$, the parameter types as specified in the class definition:

$$\textbf{type } C\_\mathsf{iTc}\ [\mathsf{Self}] = <\bar{T}_p>; \tag{3.7}$$

Types $\bar{T}_p$ are allowed to reference Self, so C_iTc is parameterized by Self, as discussed above.

$C_\mathsf{s}\beta$Tc: Next we define two type constructors related to the static fields:

$$\textbf{htype } C\_\mathsf{s}\beta\mathsf{Tc}\ [\mathsf{Self}] = D\_\mathsf{s}\beta\mathsf{Tc}[\mathsf{Self}] \textbf{ with } \{\ \bar{x}_{s\beta} : \bar{T}_{s\beta}\ \}; \tag{3.8}$$

If there is no superclass, the **with** and text to its left are not present; otherwise, $D\_\mathsf{s}\beta\mathsf{Tc}$ are type constructors defined by D's version of (3.8). The **with** type operator, like its kind-level counterpart, yields a record type that has the union of the field declarations of its two arguments (which are record types), with those of its right-hand argument overriding same-named ones from the left. In this case **with** arranges that a declaration from C overrides the corresponding one inherited from its superclass, D.

Recall that the β subscript means that (3.8) denotes two definitions, one for each value of β (g and e). C_sgTc is the constructor for the general types of the static fields and C_seTc is the constructor for their exact types; recall that these are different for type and hybrid fields. They are parameterized by Self because Self can appear in $\bar{T}_{sg}$ and $\bar{T}_{se}$.

$C_\mathsf{m}\beta$Tcc: Next we define a pair of type constructors for the member fields:

$$\textbf{htype } C\_\mathsf{m}\beta\mathsf{Tcc}\ [\mathsf{Self}, \mathsf{self}] = D\_\mathsf{m}\beta\mathsf{Tcc}[\mathsf{Self}, \mathsf{self}] \textbf{ with } \{\ \bar{x}_{m\beta} : \bar{T}_{m\beta}\ \}; \tag{3.9}$$

As above, if there is no superclass, the **with** and text to its left are not present; otherwise, $D\_\mathsf{m}\beta\mathsf{Tcc}$ are defined by D's (3.9). Again there are two versions, for general and exact. And again they are parameterized by Self because Self can appear in $\bar{T}_{mg}$ and $\bar{T}_{me}$. However, they are also parameterized by self because it too can appear in these types. The "cc" after "T" in the name is meant to suggest "two argument constructor".

*C_**m**β**Tc**:* Next we define two one-argument type constructors for methods:

$$\textbf{htype}\ C\_\mathsf{m}\beta\mathsf{Tc}\ [\mathsf{Self}] = \tau[\mathsf{self}]\ C\_\mathsf{m}\beta\mathsf{Tcc}[\mathsf{Self},\ \mathsf{self}]; \quad (3.10)$$

These use the τ operator, described in section 3.4.1, to provide the appropriate type for the recursion used to create new within C_c as to be defined by (3.20). This means that, for example, $p{:}C$_mgTc[B] implies $p{:}C$_mgTcc[B,p].

3.4.3 Class-level types

*C_**eT**δ:* We define type C_eT to be the class-level exact type of C: it declares the static fields, the Instance type field and the function new, all using exact types. We define it in two steps:

$$\begin{array}{l}\textbf{htype}\ C\_\mathsf{eTc}\ [C] = C\_\mathsf{seTc}[C]\ \textbf{with}\ \{ \quad (3.11)\\ \quad \textbf{htype}\ \mathsf{Instance} \leftrightarrow C\_\mathsf{meTc}[C];\\ \quad \textbf{hfunc}\ \mathsf{new}\ (C\_\mathsf{iTc}[C]) : C\mathtt{@}\\ \};\end{array}$$

$$\textbf{htype}\ C\_\mathsf{eT} = \tau[C]\ C\_\mathsf{eTc}[C] \quad (3.12)$$

These definitions are present only if the class is concrete. The first definition defines a type constructor over parameter C. The second definition defines a type dependent in C; it uses τ (see 3.4.1) to provide a type appropriate for the recursion used to create C. Later we establish $C{:}C$_eT which implies $C{:}C$_eTc[C], which means that, when reasoning about C, we use C's exact types with C substituted for Self. Also, given (2.8), these definitions give the value of C@.

*C_**gT**δ:* We define type C_gT to be the class-level general type of C: its instances include the class object C and the class objects of all extensions of C. It defines the type "{{?<<C}}" introduced in section 2.2.3:

$$\{\{?{<<}C\}\} \triangleq C\_\mathsf{gT} \quad (3.13)$$

Thus $B{<<}C$ is equivalent to $B{:}C$_gT.

Again we define C_gT in two steps:

$$\begin{array}{l}\textbf{htype}\ C\_\mathsf{gTc}\ [\mathsf{Self}] = C\_\mathsf{sgTc}[\mathsf{Self}]\ \textbf{with}\ \{ \quad (3.14)\\ \quad \textbf{htype}\ \mathsf{Instance} <: C\_\mathsf{mgTc}[\mathsf{Self}]\\ \};\end{array}$$

$$\textbf{htype}\ C\_\mathsf{gT} = \tau[\mathsf{Self}]\ C\_\mathsf{gTc}[\mathsf{Self}]; \quad (3.15)$$

The second definition uses τ to arrange that if $B{<<}C$, then $B{:}C$_gTc[B], which means that, when reasoning about B, we use C's general types with B substituted for Self.

Note that while the exact declarations are used within C_eT, the general declarations are used within C_gT. This captures the notion that C (which is of type C_eT) has exactly the fields defined by class definition C, while subclasses of C (which are of type C_gT) may have overridden fields that satisfy only their general declarations.

3.4.4 The "hash" type

*C_**hT**:* We define type C_hT as the concrete realization of $C\#$:

$$C\# \triangleq C\_\mathsf{hT} \quad (3.16)$$

We use the following definition for C_hT:

$$\begin{array}{ll}\textbf{htype}\ C\_\mathsf{hT} = \cup[\mathsf{Self} << C]\ \mathsf{Self}\mathtt{@} & (3.17)\\ \qquad\qquad = \cup[\mathsf{Self} : C\_\mathsf{gT}]\ \mathsf{Self.Instance}\end{array}$$

The second equality is the first without syntactic sugar. We use $\cup$ to denote undiscriminated union: the type $\cup[x{:}D]T$ (where x is free in T) is the type of values, v, for which there is some value, say d, with type D such that v is an instance of type $T\langle x{\mapsto}d\rangle$ (which is T with d substituted for x).

Recall that C's general type, $C\#$, is intended to have instances of C and C's subclasses. Definition (3.17) defines $C\#$ to be exactly this: the union of the instances of all concrete subclasses of C. Thus if B is a concrete subclass of C, then its instances are in $C\#$ (by taking Self to be B). Also if v is in $C\#$, then there must be some concrete class, V, that extends C (so $V{<<}C$) such that v was created by V.new(...) (so $v{:}V$@).

3.5 Define constructors and class object

The final step of reduction (3.1) defines class-constructing functions and the class object:

$$\begin{array}{l}\textbf{hfunc}\ C\_\mathsf{sc}\ (\text{*}\mathsf{Self} : C\_\mathsf{gT}) : C\_\mathsf{seTc}[\mathsf{Self}] = \quad (3.18)\\ \quad \textbf{let}\ \mathsf{Super} = D\_\mathsf{sc}(\mathsf{Self})\\ \quad \textbf{in}\ \mathsf{Super}\ \textbf{with}\ \{\ \bar{x}_{se} = \bar{e}_s\ \};\\ \textbf{hfunc}\ C\_\mathsf{mc} \quad (3.19)\\ \quad (\text{*}\mathsf{Self} : C\_\mathsf{gT},\ {<}\bar{p}{>} : C\_\mathsf{iTc}[\mathsf{Self}],\ \text{*}\mathsf{self} : \mathsf{Self}\mathtt{@})\\ \quad : C\_\mathsf{meTcc}[\mathsf{Self},\ \mathsf{self}] =\\ \quad \textbf{let}\ \mathsf{super} = D\_\mathsf{mc}(\mathsf{Self},\ {<}\bar{a}{>},\ \mathsf{self})\\ \quad \textbf{in}\ \mathsf{super}\ \textbf{with}\ \{\ \bar{x}_{me} = \bar{e}_m\ \};\\ \textbf{hfunc}\ C\_\mathsf{c}\ (\text{*}C : C\_\mathsf{eT}) : C\_\mathsf{eTc}[C] = \quad (3.20)\\ \quad C\_\mathsf{sc}(C) + \{\\ \quad\quad \textbf{htype}\ \mathsf{Instance} = C\_\mathsf{meTc}[C];\\ \quad\quad \textbf{hfunc}\ \mathsf{new}\ (\mathsf{pt} : C\_\mathsf{iTc}[C]) : C\mathtt{@} =\\ \quad\quad\quad \mu(\mathsf{self})\ C\_\mathsf{mc}(C,\ \mathsf{pt},\ \mathsf{self})\\ \quad \};\\ \textbf{hconst}\ C : C\_\mathsf{eT} = \mu(C)\ C\_\mathsf{c}(C) \quad (3.21)\end{array}$$

Definitions (3.20) and (3.21) are present only if the class is concrete. If there is no superclass, the text from **let** to **with** (inclusive) is not present in the first two definitions; otherwise, D_sc and D_mc are functions defined by D's version of (3.18) and (3.19). The keyword **hfunc** introduces a hybrid-to-hybrid function. The value operator **with**, like its type-level counterpart described at (3.8), unions the fields of two records, with those of the right argument overriding those of the left. The pattern parameter $<\bar{p}>$ in C_mc matches a tuple argument and binds each component to the corresponding name of $\bar{p}$ (which, by (3.7), have types $\bar{T}_p$). The functions are all *dependent*, with a star, "*", before a parameter indicating that it can appear in the types of later parameters and in the return type. Lambda expressions (like those appearing in $\bar{e}_s$ and $\bar{e}_m$) capture the values of free variables, such as Self and self, by forming closures.

Description: Function C_sc (C's static constructor), defined by (3.18), creates C's static fields. It is parameterized by Self to provide access to the late-bound class object. The implementation of C_sc calls the superclass's static constructor, D_sc, to create the D's portion of the static fields, which it assigns to Super. C_sc then returns the record formed by executing C's static field definitions, $\bar{x}_s{=}\bar{e}_s$, and adding any non-overridden fields from Super. Expressions $\bar{e}_s$ can access Self and Super whose values are captured.

Function C_mc (C's member constructor), defined by (3.19), is parameterized by Self (as above), by a tuple containing constructor parameters $\bar{p}$ and by self (for access to the late bound object instance). The implementation of C_mc calls the superclass's member constructor, D_mc, to create the superclass's portion of the member fields, which it assigns to super. C_mc then returns the record formed by executing C's member field definitions, $\bar{x}_m{=}\bar{e}_m$, and adding any non-overridden fields from super. Expressions $\bar{e}_m$ can access Self, $\bar{p}$, self and super whose values are captured.

Function C_c (C's constructor), defined by (3.20), is used to create class object C. It combines the previous two functions to create the framework for the object C: it uses C_sc to create the static part, to which it adds a type named "Instance" and a function named "new." Type Instance is assigned the instance-level exact type. Function new takes the constructor parameters $\bar{p}$ and creates a recursive object using C_mc to create the member fields.

Finally, definition (3.21) creates the actual class object C by creating a recursive object populated by C_c.

Type annotations: As mentioned, the first two functions, C_sc and C_mc, are parameterized by Self to provide access to the late-bound class object. Parameter Self is given the class-level general type C_gT (so Self<<C) because these functions are used for C and its extensions.

Function C_mc also has parameters <$\bar{p}$> and self. Pattern parameter <$\bar{p}$> is assigned a type so that $\bar{p}$:$\bar{T}_p$, as given in the class definition. Parameter self is given type Self@ to express that it is an instance of the future class denoted by Self.

The return types of C_sc and C_mc are dependent exact types, C_seTc[Self] and C_meTcc[Self,self], respectively. They are exact because those types most accurately characterize the results of the functions. These return types are parameterized by Self because what the functions create can depend on Self (as Self can be free in the type expressions of $\bar{e}_s$ and $\bar{e}_m$). The return type of C_mc is also parameterized by self for analogous reasons.

In function C_c, parameter C is given the exact type, C_eT, because C:C_eT is needed to imply C@↔C_meTc[C] which is needed when typing new; use of the exact type is acceptable because C_c is not used for extension. The return type of new is C@ as expected and the return type of C_c is C_eTc[C] by design.

The object C is annotated with type C_eT by design.

4. Typing

This section presents a typing rule for the classes described in section 2. In subsection 4.1 we describe the premises of the rule and in 4.2, we give the rule. The conclusion of this rule uses a judgement of our invention of the form "$\Gamma \vdash J^C \rightsquigarrow \bar{L}$" which is read, "in environment Γ, definition J^C creates bindings which are described by declarations $\bar{L}$." In subsection 4.3, we describe certain useful assertions that follow from these declarations.

This typing rule is designed to make the reduction of section 3 type preserving: any high-level class definition (J^C) satisfying the premises of the rule (in Γ) is reduced to lower-level definitions that are type-correct and that yield the specified declarations ($\bar{L}$). In fact, we mainly work backwards from the reduction: we type the lower-level definitions of sections 3.3 through 3.5 and keep track of what conditions are required and what declarations are produced and use them to form the premises and conclusion of the rule.

Type-dot: We use an operator, "type dot" which is a raised dot "·", to extract the type of a field from a record (or hybrid) type: {...,x:T,...}·x is T. Essentially, type-dot treats a record type like a record of types. Section 4.3.8 describes how type-dot relates to key reduction-defined types.

4.1 Premises for well-formed classes

The type checking of a class definition proceeds in essentially three stages: (1) the kinds are well-defined, (2) the types are well-kinded and satisfy certain subtype relations, and (3) the expressions are well-typed. The following subsections describe each stage.

In the premises below, the meta-variable Γ denotes the environment outside the class definition. An environment is a list of declarations used to track the entities denoted by variables currently in scope.

4.1.1 Premises for well-defined kinds

Because the reduction defines kind variables (see (3.6) of section 3.3), it is possible that a programmer could use a kind variable that is not defined. The following premise requires that all kinds appearing in the body of C as rewritten by (3.2) are well-defined:

$$\Gamma \vdash \bar{K}_{\alpha\beta} :: : \tag{4.1}$$

We use "K::" as a judgement meaning that kind K is well-defined. We use the convention that over-bars appearing in the condition denote a series of sub-conditions, one for each element of the list. Also, we continue the convention that α and β subscripts are expanded to multiple conditions, which, in this case, are for $\bar{K}_{se}$, $\bar{K}_{sg}$, $\bar{K}_{me}$ and $\bar{K}_{mg}$.

We define Γ_k to be an environment that contains the six declarations resulting from the kind definitions (3.6) of section 3.3:

$$\begin{array}{l}\Gamma_k \triangleq \\ \quad \textbf{kind}\ C\_\alpha\beta\mathsf{K} \leftrightarrow D\_\alpha\beta\mathsf{K}\ \textbf{with}\ \{\ \bar{x}_{\alpha\beta} :: \bar{K}_{\alpha\beta}\ \}\ ; \\ \quad \textbf{kind}\ C\_\beta\mathsf{K} \leftrightarrow C\_\mathsf{s}\beta\mathsf{K}\ \textbf{with}\ \{\ \mathsf{Instance} :: *(C\_\mathsf{m}\beta\mathsf{K})\ \}\end{array} \tag{4.2}$$

Condition (4.1) ensures that Γ_k is a well-formed environment.

4.1.2 Premises for well-formed types

The following premise requires that the initialization parameter types, $\bar{T}_p$, are well-kinded:

$$\begin{array}{l}\Gamma;\ \Gamma_k;\ \Gamma_i \vdash \bar{T}_p :: * \\ \quad \textit{where}\ \Gamma_i \triangleq \textbf{hconst}\ \mathsf{Self} :: C\_\mathsf{gK};\end{array} \tag{4.3}$$

This says that each type T_p^l must be a simple type (of kind *).

The following premise requires that each static field type $T_{s\beta}^i$ has instances of the corresponding kind $K_{s\beta}^i$:

$$\begin{array}{l}\Gamma;\ \Gamma_k;\ \Gamma_s \vdash \bar{T}_{s\beta} :: *(\bar{K}_{s\beta}) \\ \quad \textit{where}\ \Gamma_s \triangleq \textbf{hconst}\ \mathsf{Self} :: C\_\mathsf{gK};\end{array} \tag{4.4}$$

Recall that β ranges over e or g (exact and general), so this denotes two sets of sub-conditions. Here the over-bars are parallel as usual, so this is a collection of conditions, $T_{s\beta}^i$::*($K_{s\beta}^i$).

The following premise requires that each member field type $T_{m\beta}^j$ is over instances of the corresponding kind $K_{m\beta}^j$:

$$\begin{array}{l}\Gamma;\ \Gamma_k;\ \Gamma_m \vdash \bar{T}_{m\beta} :: *(\bar{K}_{m\beta}) \\ \quad \textit{where}\ \Gamma_m \triangleq \textbf{hconst}\ \mathsf{Self} :: C\_\mathsf{gK}; \\ \qquad\qquad \textbf{hconst}\ \mathsf{self} :: C\_\mathsf{mgK}\end{array} \tag{4.5}$$

We define Γ_t to contain the declarations resulting from the type definitions of section 3.4:

$$\begin{array}{l}\Gamma_t \triangleq \textit{/* declarations resulting from (3.7) to (3.17) */} \\ \quad \textit{/* with the following kinds: */} \\ \qquad \begin{array}{lll} C\_\mathsf{iTc} & :: [C\_\mathsf{gK}] \Rightarrow & * \\ C\_\mathsf{m}\beta\mathsf{Tcc} & :: [C\_\mathsf{gK},\ C\_\mathsf{mgK}] \Rightarrow & *(C\_\mathsf{m}\beta\mathsf{K}) \\ C\_\alpha\beta\mathsf{Tc} & :: [C\_\mathsf{gK}] \Rightarrow & *(C\_\alpha\beta\mathsf{K}) \\ C\_\beta\mathsf{Tc} & :: [C\_\mathsf{gK}] \Rightarrow & *(C\_\beta\mathsf{K}) \\ C\_\beta\mathsf{T} & :: & *(C\_\beta\mathsf{K}) \\ C\_\mathsf{hT} & :: & *(C\_\mathsf{mgK}) \end{array}\end{array} \tag{4.6}$$

The declaration resulting from a definition of the form X=T is X:{{?↔T}}, which we also write as X↔T, and the declaration resulting from $X[\bar{Y}]$=T (which is really X=λ[$\bar{Y}$]T) is X:[$\bar{Y}$]⇒{{?↔T}}, which we also write as $X[\bar{Y}]$↔T. Also each declaration should have a kind component as given above.

For example, type constructor definition (3.10) results in the following canonical-form declaration:

$$\begin{array}{l}C\_\mathsf{mgTc} :: [C\_\mathsf{gK}] \Rightarrow *(C\_\mathsf{mgK}) \\ \qquad\quad : [\mathsf{Self}] \Rightarrow \{\{?\leftrightarrow\tau[\mathsf{self}]C\_\mathsf{mgTcc}[\mathsf{Self},\ \mathsf{self}]\}\}\end{array}$$

Environment Γ_t as defined by (4.6) is well defined by premises (4.3) to (4.5) and so Γ_t may appear to the left of ⊢ in the premises of the following section.

The following two premises ensure that the exact types subtype the corresponding general ones, which ensures C<<C. First, the exact types of static fields must subtype the corresponding general ones:

$$\Gamma;\ \Gamma_k;\ \Gamma_t;\ \Gamma_{sd} \vdash \bar{T}_{se} <: C\_\mathsf{sgTc[Self]}{\cdot}\bar{x}_{se} \tag{4.7}$$
where $\Gamma_{sd} \triangleq$ **hconst** Self :: C_gK

The type expression C_sgTc[Self]$\cdot x^i_{se}$ denotes the type of field x^i_{se} in the static general type, C_sgTc[Self]; this field type may have been defined in C or inherited. The raised "type-dot" operator was discussed at the start of this section.

Second, the member's exact types must subtype the corresponding general ones:

$$\Gamma;\ \Gamma_k;\ \Gamma_t;\ \Gamma_{md} \vdash \bar{T}_{me} <: C\_\mathsf{mgTcc[Self,self]}{\cdot}\bar{x}_{me} \tag{4.8}$$
where $\Gamma_{md} \triangleq$ **hconst** Self :: C_gK ; **hconst** self :: C_mgK

Note that for **const** fields of (2.2), the premises resulting from (4.7) or (4.8) are trivial as both types are the same. On the "**type** $+X{=}T$" form, the reduction yields a premise with "$T{:}\{\{?{<:}T\}\}$," which is also trivial (as T subtypes itself).

Technically, since the exact and general kinds can be different, there must also be sub-kind conditions $\bar{K}_{\alpha e}{<::}C\_\alpha\mathsf{gK}{\cdot}\bar{x}_{\alpha e}$, which we will not discuss further.

The following two premises ensure that a refinement does not mess up C_gT$<:D$_gT. First, if static field x^i_{sg} is marked **refine**, then its general type must be a subtype of the one that it overrides (due to **with** in C_sgTc of (3.8)):

$$\Gamma;\ \Gamma_k;\ \Gamma_t;\ \Gamma_{sd} \vdash T^i_{sg} <: D\_\mathsf{sgTc[Self]}{\cdot}x^i_{sg} \tag{4.9}$$
where $\Gamma_{sd} \triangleq$ **hconst** Self :: C_gK

Second a member field x^j_{mg} marked **refine** has a similar condition:

$$\Gamma;\ \Gamma_k;\ \Gamma_t;\ \Gamma_{md} \vdash T^j_{mg} <: D\_\mathsf{mgTcc[Self,self]}{\cdot}x^j_{mg} \tag{4.10}$$
where $\Gamma_{sd} \triangleq$ **hconst** Self :: C_gK **hconst** self :: C_mgK

Of course, if there are no overrides then these conditions are vacuous. Kind-level refinement conditions, $K^i_{\alpha g}{<::}D\_\alpha\mathsf{gK}{\cdot}x^i_{\alpha g}$, are also needed.

4.1.3 Premises for well-typed expressions

The following premise ensures that $\bar{a}$, the initialization arguments for superclass D, are well-typed:

$$\Gamma;\ \Gamma_k;\ \Gamma_t;\ \Gamma_i \vdash {<}\bar{a}{>} :: \epsilon : D\_\mathsf{iTc[Self]} \tag{4.11}$$
where $\Gamma_i \triangleq$ **hconst** Self :: C_gK : C_gT; **const** $<\bar{p}>$: C_iTc[Self]

D_iTc is D's version of C_iTc which is defined by (3.7). $\bar{p}$ and $\bar{a}$ appear in the class definition (see (3.1)) and $\bar{p}$ may be free in $\bar{a}$. If there is no superclass, this premise is omitted.

The following premise ensures that the implementations of the static fields satisfy their exact types:

$$\Gamma;\ \Gamma_k;\ \Gamma_t;\ \Gamma_s \vdash \bar{e}_s :: \bar{K}_{se} : \bar{T}_{se} \tag{4.12}$$
where $\Gamma_s \triangleq$ **hconst** Self :: C_gK : C_gT; **hconst** Super :: D_seK : D_seTc[Self]

If there is no superclass, the clause related to Super is omitted. Note that Self:C_gT is equivalent to the more intuitive Self$<<C$ but we use the lower-level version to be more consistent with other clauses.[1]

The member fields get a similar treatment, except that the self variable is also handled:

$$\Gamma;\ \Gamma_k;\ \Gamma_t;\ \Gamma_m \vdash \bar{e}_m :: \bar{K}_{me} : \bar{T}_{me} \tag{4.13}$$
where $\Gamma_m \triangleq$ **hconst** Self :: C_gK : D_gT; **const** $<\bar{p}>$: C_iTc[Self]; **hconst** self :: C_mgK : Self@; **hconst** super :: D_meK : D_meTcc[Self, self]

If there is no superclass, the clause related to super is omitted.

Note that for each **meta** field of (2.2), the associated premise due to (4.12) or (4.13) comes down to an assertion of the form "$M{:}\{\{?{\leftrightarrow}M\}\}$," which is trivial (as M equals itself). Thus this premise is only relevant to **const** and **hconst** fields.

Conditions (4.11) to (4.13) (and (4.7) and (4.8)) ensure that the the class constructors and object of section 3.5 are well-typed. We define environment Γ_c to contain the declarations resulting from (3.18) to (3.21):

$$\begin{array}{l}\Gamma_c \triangleq \\ \quad \textbf{hfunc}\ C\_\mathsf{sc}\ (\mathsf{*Self} : C\_\mathsf{gT}) : C\_\mathsf{seTc[Self]}\ ; \\ \quad \textbf{hfunc}\ C\_\mathsf{mc}\ (\mathsf{*Self} : C\_\mathsf{gT},\ C\_\mathsf{iTc[Self]},\ \mathsf{*self} : \mathsf{Self@}) \\ \qquad : C\_\mathsf{meTcc[Self,\ self]}\ ; \\ \quad \textbf{hconst}\ C :: C\_\mathsf{eK} : C\_\mathsf{eT}\end{array} \tag{4.14}$$

The last line is not present if the class is partial.

4.2 Typing rule

The following is our typing rule for concrete class definitions with complex content:

CLASS-INTRO

$$\frac{J^C \triangleq \text{"}\textbf{class}\ C\ (\bar{p}{:}\bar{T}_p)\ \textbf{extends}\ D(\bar{a})\ \{\ \bar{\mathcal{F}}\ \}\text{"} \qquad \begin{array}{cc}\text{(3.2) - (3.5)} & \text{(4.1) - (4.14)}\end{array}}{\Gamma \vdash J^C \rightsquigarrow \Gamma_k\ ;\ \Gamma_t\ ;\ \Gamma_c}$$

The meta-variables are the non-terminals described in section 2.1, those introduced by the rewrites of section 3.2 and Γ_k, Γ_t and Γ_c.

The first premise defines J^C to be the class that is the subject of the reduction, (3.1). The second set of premises are the rewrites of (3.2) to (3.5), which relate $\bar{\mathcal{F}}$ to meta-variables $\bar{\mathcal{J}}_\alpha$ and then to $\bar{x}_\alpha$, $\bar{K}_\alpha$, $\bar{T}_{\alpha\beta}$ and $\bar{e}_\alpha$ (for $\alpha{\in}\{\mathsf{s},\mathsf{m}\}$ and $\beta{\in}\{\mathsf{e},\mathsf{g}\}$). Premise (4.1) ensures that the kinds $\bar{K}_\alpha$ are well-defined, (4.2) is the definition of Γ_k, (4.3) to (4.5) ensure that the types $\bar{T}_p$ and $\bar{T}_{\alpha\beta}$ are well-kinded, (4.6) is the definition of Γ_t, (4.7) and (4.8) ensure that the exact types are subtypes of the general ones, (4.9) and (4.10) ensure that the general types of fields marked **refine** are subtypes of the ones of the superclass, (4.11) to (4.13) ensure that expressions $\bar{a}$ and $\bar{e}_\alpha$ are well-typed, and (4.14) is the definition of Γ_c. The conclusion is a new kind of judgement (involving the relation "$\rightsquigarrow$") which states that, in environment Γ, class definition J^C results in a state where the declarations of Γ_k, Γ_t and Γ_c are valid. This rule is justified in section A.2.

4.3 Conclusion-implied typing assertions

The typing rule above says that the low-level declarations of Γ_k, Γ_t and Γ_c (as defined by (4.2), (4.6) and (4.14)) are valid in the state resulting from the definition of class C as given by J^C. This section describes certain useful, higher-level assertions that follow from Γ_k, Γ_t and Γ_c. This gives a sense of how a typing algorithm would reason about the result of a class definition. We defer to section A.3 the detailed arguments that establish these assertions.

In some assertions to come, we use the "type dot" operator, as described at the start of this section, to extract the type of a field from a record (or hybrid) type. Section 4.3.8 describes how type-dot relates to key reduction-defined types.

For partial classes, the following assertions are meaningless since partial classes have no class object C, nor "**new** C", nor C@: (4.15), (4.16), (4.18), (4.19), (4.25) and (4.26).

[1] In other research, I use "C_gT°" as the type of Self, where the little circle indicates that the value of Self is a pointer to uninitialized storage. This means that Self can be stored, but not accessed, until after the recursive construction is complete.

4.3.1 Reasoning about "new"

The following assertions show that "**new** C" has C's exact type:

$$(\textbf{new}\ C(\bar{e})) : C@\ \textit{given}\ <\bar{e}> : C\_\mathsf{iTc}[C] \quad (4.15)$$
$$(\textbf{new}\ C) : C@\ \textit{given}\ <> : C\_\mathsf{iTc}[C]$$

4.3.2 Reasoning about member fields

For $p{:}C@$, the type of a member field, $p.m$, can be determined by extracting it from the member exact type:

$$p : C@ \implies p.m : C\_\mathsf{meTcc}[C,p]{\cdot}m \quad (4.16)$$

Type $C\_\mathsf{meTcc}[C,p]{\cdot}m$ can be simplified by (4.34). If p appears free in the simplified type, p must be a path, otherwise it can be any expression.

For $p{:}C\#$, the type of a member field, $p.m$, can be determined by extracting it from the member general type:

$$p : C\# \implies p.m : \cup[\mathsf{Self}{<<}C]\ C\_\mathsf{mgTcc}[\mathsf{Self},p]{\cdot}m \quad (4.17)$$

Notice that the type involves a union, coming from the definition of $C\#$, (3.17). Type $C\_\mathsf{mgTcc}[\mathsf{Self},p]{\cdot}m$ can be simplified using (4.34) and, if the simplified type does not have Self free in it, the simplified type is the type of $p.m$; otherwise $p.m$ is not well typed.

4.3.3 Reasoning about static fields

The type of a static field, $C.s$, can be determined by extracting it from the exact type for statics:

$$C.s : C\_\mathsf{seTc}[C]{\cdot}s \quad (4.18)$$

To reason further about this, we can use (4.35).

4.3.4 Reasoning about subtypes

A key relationship is that the exact type subtypes the general type which subtypes the superclass's (D's) general type:

$$C@ <: C\# \quad (4.19)$$
$$C\# <: D\# \quad (4.20)$$

4.3.5 Reasoning about extensions of C

When a class object, B, extends C, we can use the following assertions to reason about class object B and class B's instances:

$$B << C \implies \textit{// each of the following:}$$
$$p : B@ \implies p.m : C\_\mathsf{mgTcc}[B,p]{\cdot}m \quad (4.21)$$
$$B.s : C\_\mathsf{sgTc}[B]{\cdot}s \quad (4.22)$$
$$B@ <: C\# \quad (4.23)$$
$$B{<<}D \quad (4.24)$$

D is any superclass of C. The types appearing rightmost in (4.21) and (4.22) can be simplified by (4.34) and (4.35), respectively. For the first two assertions, if B appears free in the simplified type, B must be a path, otherwise it can be any expression; similarly, in (4.21), p must be a path if it appears free in the simplified type.

4.3.6 Reasoning about superclasses of C

The following assertions say that class C extends itself and its superclass, D:

$$C << C \quad (4.25)$$
$$C << D \quad (4.26)$$

4.3.7 Reasoning about Self and self

When typing the body of a class definition, the premises described in section 4.1.3 must be met, which requires reasoning about Self and self. This section mentions some useful implications following from the assumptions of those rules. Assume the class is named C and its superclass is D.

First, premises (4.11) to (4.13) have Self:C_gT as an assumption, which implies the following assertions:

$$\mathsf{Self}.s : C\_\mathsf{sgTc}[\mathsf{Self}]{\cdot}s \quad (4.27)$$
$$\mathsf{Self} << \mathsf{C} \quad (4.28)$$
$$\mathsf{Self}@ <: C\# \quad (4.29)$$

Condition (4.12) also involves Super which can be reasoned about using the following assertion:

$$\mathsf{Super}.s : D\_\mathsf{seTc}[\mathsf{Self}]{\cdot}s \quad (4.30)$$

Condition (4.13) involves self and super which can be reasoned about using the following assertions:

$$\mathsf{self} : \mathsf{Self}@ \quad (4.31)$$
$$\mathsf{self}.m : C\_\mathsf{mgTcc}[\mathsf{Self},\mathsf{self}]{\cdot}m \quad (4.32)$$
$$\mathsf{super}.m : D\_\mathsf{meTcc}[\mathsf{Self},\mathsf{self}]{\cdot}m \quad (4.33)$$

4.3.8 "Looking up" field types

Many of the assertions above involve reasoning about a type extracted from one of the constructor types, $C\_\mathsf{m}\beta\mathsf{Tcc}$ or $C\_\mathsf{s}\beta\mathsf{Tc}$, applied to certain parameters. These meta-expressions can be simplified (evaluated) by the following rules:

$$C\_\mathsf{m}\beta\mathsf{Tcc}[M,p] \cdot x \leftrightarrow \textit{if } x \textit{ is } x^j_m \quad (4.34)$$
$$\quad \textit{then } T^j_{m\beta}\langle\mathsf{Self}{\mapsto}M\rangle\langle\mathsf{self}{\mapsto}p\rangle$$
$$\quad \textit{else } D\_\mathsf{m}\beta\mathsf{Tcc}[M,p]{\cdot}x$$
$$C\_\mathsf{s}\beta\mathsf{Tc}[M] \cdot x \leftrightarrow \textit{if } x \textit{ is } x^i_s \quad (4.35)$$
$$\quad \textit{then } T^i_{s\beta}\langle\mathsf{Self}{\mapsto}M\rangle$$
$$\quad \textit{else } D\_\mathsf{s}\beta\mathsf{Tc}[M]{\cdot}x$$

These are based on definitions (3.9) and (3.8), respectively. Recall that a superscripted meta-variable denotes the named component of the meta-variable with a bar, so for example, x^j_m denotes the j-th component of $\bar{x}_m$. The meta-variables $\bar{x}_\alpha$ and $\bar{T}_{\alpha\beta}$ are those defined by the rewrites of (3.2). Recall that $T\langle X{\mapsto}M\rangle$ denotes T with free occurrences of X replaced by M. To reason about $D\_\mathsf{m}\beta\mathsf{Tcc}$ or $D\_\mathsf{s}\beta\mathsf{Tc}$, we would use (4.34) or (4.35) except relative to the reduction of D. If there is no superclass, the *else* clauses are undefined.

For example, to simplify $C\_\mathsf{meTcc}[C,p]{\cdot}m$ of (4.16), if m matches the j-th identifier of $\bar{x}_m$ from (3.2), the result is the j-th type of $\bar{T}_{me}$ (also from (3.2)) with Self replaced by C and self by p; otherwise the process continues up the inheritance hierarchy.

5. Variations

This section describes certain variations on the reduction of section 3 and its typing as discussed in section 4. In subsection 5.1, we discuss how the reduction is modified to handle generic classes. In subsection 5.2, we discuss a reduction that includes a member field, Class, that equals Self.

5.1 Generic Classes

In this section, we describe polymorphic classes, also known as "generic classes" or just "generics". In the first subsection, we describe how the reduction is modified to handle generic classes and, in the second, we describe how such a class is extended.

5.1.1 Generic class definitions

A generic class definition is of the following form:

$$\textbf{class}\ C\ [\bar{X}{::}\bar{K}_x]\ (\bar{p}{:}\bar{T}_p)\ \textbf{extends}\ D(\bar{a})\ \{\ \bar{\mathcal{F}}\ \} \quad (5.1)$$

Non-terminal $\bar{X}$ ranges over identifier lists and $\bar{K}_x$ ranges over kind lists of the same length; the remaining non-terminals are as described at (3.1). The variables of $\bar{X}$ can appear in $\bar{T}_p$, D, $\bar{a}$ and $\bar{\mathcal{F}}$. Compared to (2.1), this has the polymorphic parameter specification "$[\bar{X}{::}\bar{K}_x]$" after the class name.

As a further abuse of syntax, we extend the special operators:

$$C[\bar{X}]@ \quad \triangleq C[\bar{X}].\mathsf{Instance} \tag{5.2}$$
$$\textbf{new}\ C[\bar{X}](\bar{e}) \quad \triangleq C[\bar{X}].\mathsf{new}(<\bar{e}>) \tag{5.3}$$
$$\{\{?<<C[\bar{X}]\}\} \quad \triangleq C\_\mathsf{gT}[\bar{X}] \tag{5.4}$$
$$C[\bar{X}]\# \quad \triangleq C\_\mathsf{hT}[\bar{X}] \tag{5.5}$$

These replace (2.8), (2.9), (3.13) and (3.16), respectively.

To handle generic classes, we modify the reduction as follows:

- In section 3.3, additional kinds are generated for generic class objects:

$$\textbf{kind}\ C\_\beta\mathsf{Kg} = [\bar{K}_x]\Rightarrow C\_\beta\mathsf{K} \tag{5.6}$$

- In section 3.4, each type or type constructor definition (3.7) to (3.17) has "$[\bar{X}::\bar{K}_x]$" added as the first parameter (pushing other parameters right). In canonical form, this adds $\lambda[\bar{X}::\bar{K}_x]$ to the front of each type constructor.
- In sections 3.4 to 4.3, any reference to such a type constructor has "$[\bar{X}]$" immediately after it.
- In section 3.5, each constructor function is made polymorphic in "$[\bar{X}::\bar{K}_x]$", as is the class constant itself.
- In sections 3.5 to 4.3, any call to a constructor function has "$[\bar{X}]$" as its first polymorphic argument. Any reference to the class is as "$C[\bar{X}]$".

The typing section is modified as follows:

- In section 4.1.2, "$\bar{K}_x$ :::" is included in (4.1) and the definition of Γ_k in (4.2) includes the declaration resulting from (5.6).
- Each kinding condition, (4.3) to (4.5), has "**meta** $\bar{X}::\bar{K}_x$" as an additional first assumption.
- The definition of Γ_t in (4.6) has "$[\bar{K}_x]\Rightarrow$" at the start of each kind to indicate that each type or type constructor has an additional parameter.
- In section 4.1.3, each typing requirement, (4.7) to (4.13), has "**meta** $\bar{X}::\bar{K}_x$" as an additional first assumption. Note that $\mathsf{Self}{:}C\_\mathsf{gT}[\bar{X}]$, while $\mathsf{self}{:}\mathsf{Self@}$ as previously. In other words, Self is not generic as it denotes a future subclass of $C[\bar{X}]$.
- The definition of Γ_c in (4.14) has "$[\bar{X}::\bar{K}_x]$" added to the start of each parameter list to indicate that each function is additionally polymorphic. Also, the class constant is polymorphic:

$$C\ [\bar{X}::\bar{K}_x] :: C\_\mathsf{eK} : C\_\mathsf{eT}[\bar{X}]$$
$$C :: C\_\mathsf{eKg} : [\bar{X}]\Rightarrow C\_\mathsf{eT}[\bar{X}]$$

  The second is the canonical form of the first, given (5.6).
- Each assertion, A, of section 4.3 is wrapped in a universal quantification, $\forall[\bar{X}::\bar{K}_x]A'$, where A' is A with references to C and friends modified to add $[\bar{X}]$ as described above.

5.1.2 Extending a generic class

When a class extends a generic superclass, its definition is of the following form:

$$\textbf{class}\ C\ \ldots\ \textbf{extends}\ D[\bar{A}](\bar{a})\ \{\ \ldots\ \} \tag{5.7}$$

Types $\bar{A}$ are generic arguments to D. To handle a generic superclass, we modify the reduction as follows:

- Any reference to one of D's auxiliary types has "$[\bar{A}]$" as an additional first argument.
- Any reference to one of D's constructor functions, D_sc or D_mc, also has "$[\bar{A}]$" as the first argument.
- In section 4.1.2, there is an additional condition to ensure that the "calls" of D's type constructors are well-kinded:

$$\Gamma \vdash \bar{A} :: \bar{K}_x$$

  The kinds $\bar{K}_x$ are those from D.

5.2 Member Class = Self

Consider the following partial definition of class Object:

```
partial class Object {                                   (5.8)
    final hconst Class = Self;
}
```

This definition assigns to the member field "Class" the class object itself, as denoted by Self. While methods already have access to the class object via Self, having Class in each instance means that Self is directly nameable from outside the class. This has some interesting properties for typing, such as (5.13) and (5.15) below. Our "self.Class" is like "this.class" of [18].

Consider Object's reduction. There may be more fields to Object but let us ignore them for now. Since the class object is complex, its kind is non-trivial:

$$\textbf{kind}\ \mathsf{Object\_s}\beta\mathsf{K} = \{\ \}\ ; \tag{5.9}$$
$$\textbf{kind}\ \mathsf{Object\_m}\beta\mathsf{K} = \{\ \mathsf{Class} :: \mathsf{Object\_}\beta\mathsf{K}\ \}\ ;$$
$$\textbf{kind}\ \mathsf{Object\_}\beta\mathsf{K} = \{\ \mathsf{Instance} :: *(\mathsf{Object\_m}\beta\mathsf{K})\ \}$$

Note that the last two kind definition pairs are mutually recursive. Whether kind-level recursion is well-defined is a topic of future research, but for now, let us assume it is. Since the class is partial, there is no class object nor its related types and constructors.

The member constructor, Object_mc, defines the class field:

$$\textbf{hfunc}\ \mathsf{Object\_mc}\ (\mathsf{Self},\ \mathsf{self}) = \{\ \mathsf{Class} = \mathsf{Self}\ \}; \tag{5.10}$$

The exact type captures this fact, as does the general type since the field is final:

$$\textbf{htype}\ \mathsf{Object\_meTcc}[\mathsf{Self},\ \mathsf{self}] = \{\ \mathsf{Class} \leftrightarrow \mathsf{Self}\ \} \tag{5.11}$$
$$\textbf{htype}\ \mathsf{Object\_mgTcc}[\mathsf{Self},\ \mathsf{self}] = \{\ \mathsf{Class} \leftrightarrow \mathsf{Self}\ \}$$

Since every class, C, extends Object, C inherits the assignment of (5.10) into C_mc and inherits the equalities of (5.11) into $C\_\mathsf{m}\beta\mathsf{Tcc}$. They can not be overridden since the field is final. This means that C's member types are of the following form:

$$\textbf{htype}\ C\_\mathsf{meTcc}[\mathsf{Self},\mathsf{self}] \leftrightarrow \{\ \mathsf{Class} \leftrightarrow \mathsf{Self};\ \ldots\ \} \tag{5.12}$$
$$\textbf{htype}\ C\_\mathsf{mgTcc}[\mathsf{Self},\mathsf{self}] \leftrightarrow \{\ \mathsf{Class} \leftrightarrow \mathsf{Self};\ \ldots\ \}$$

The ellipsis denotes the substantive, class-specific content of these types.

One consequence of this is the following:

$$p : \mathsf{Object}\# \implies p : p.\mathsf{Class@} \tag{5.13}$$

This follows from (A.12) and (5.12) to imply $p.\mathsf{Class}\leftrightarrow\mathsf{Self}$ which implies $p.\mathsf{Class@}\leftrightarrow\mathsf{Self@}$. Assuming all objects inherit from Object, (5.13) is true for all.

Another implication of (5.12) uses (4.16) to give the following assertions, useful for reasoning about $C@$:

$$p : C@ \implies \tag{5.14}$$
$$p.\mathsf{Class} \leftrightarrow C$$
$$p.\mathsf{Class}.s : C\_\mathsf{seTc}[C] \cdot s$$
$$p.m : C\_\mathsf{meTcc}[C,p] \cdot m$$

The last implication is the same as (4.16).

The following implications can be used to reason about the $C\#$ type:

$$p : C\# \implies \tag{5.15}$$
$$p.\mathsf{Class} << C$$

$p.\mathsf{Class}.s : C\_\mathsf{sgTc}[p.\mathsf{Class}] \cdot s$
$p.m : C\_\mathsf{mgTcc}[p.\mathsf{Class},p] \cdot m$

Note that, in contrast to (4.17), none of these involves a union, which makes them easier to handle. Essentially the free Self of the union is replaced by $p.\mathsf{Class}$. Cases that were unsound before still are because $p.\mathsf{Class}$ is not equal to any specific class.

The assertions of (5.15) can be justified by the following reasoning related to the properties of $C\#$:

$$p : C\# \implies \exists[\mathsf{Self}{<<}C]\ p : \mathsf{Self@}\ \textit{// by (A.12)} \qquad (5.16)$$
$$\mathsf{Self}{<<}C \wedge p{:}\mathsf{Self@}$$
$$\implies p : C\_\mathsf{mgTcc}[\mathsf{Self},p]\ \textit{// by (4.21)}$$
$$\implies p.\mathsf{Class} \leftrightarrow \mathsf{Self}\ \textit{// by (5.12)}$$
$$\implies p : C\_\mathsf{mgTcc}[p.\mathsf{Class},p]$$

The last one follows from the first two by substituting $p.\mathsf{Class}$ for Self.

6. Summary and future research

6.1 Summary

In this paper, we described a reduction that takes class definitions with complex content to a series of lower-level definitions that precisely define the higher-level class. We also described a typing rule for such classes whose soundness follows from the reduction.

6.2 Future work

A key area of future research is to publish the lower level language and its logic. Although much of it is worked out, there are still some rough areas, especially related to hybrid paths and hybrid functions.

Future research on the reduction of this paper will allow recursive references to C (and to $C@$, $C\#$ and $\{\{?{<<}C\}\}$). Further we wish to allow what we call *sibling* or *current* reference, where we define a special variable "def" that provides access to implementations of the current class, not the future one as accessed by self. The def of a class becomes super of its subclass and both can appear in types. With def, we should be able to reduce mutually-recursive nested classes.

This reduction currently generates a class object with only a subset of the overall class functionality, as much of the reduction is outside the class object. Another area of future research is to design a reduction that puts everything in one class object. This may require extensions to the lower-level constructs and logic. This may also lead to reductions of mutually-recursive nested classes.

References

[1] Martín Abadi and Luca Cardelli. On subtyping and matching. *ACM Trans. Program. Lang. Syst.*, 18(4):401–423, 1996.

[2] Kim B. Bruce. *Foundations of Object-Oriented Programming Languages: Types and Semantics*. MIT Press, Cambridge, Massachusetts, 2002.

[3] Kim B. Bruce. Some challenging typing issues in object-oriented languages. *Electronic Notes in Computer Science*, 82(8), 2003.

[4] Kim B. Bruce, Jon Crabtree, Thomas P. Murtagh, Robert van Gent, Allyn Dimock, and Robert Muller. Safe and decidable type checking in an object-oriented language. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 29–46, New York, NY, USA, 1993. ACM Press.

[5] Kim B. Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good "match" for object-oriented languages. In *ECOOP '97 Proceedings, LNCS 1241*, pages 104–127. Springer-Verlag, 1997. Extended abstract.

[6] Kim B. Bruce and J. Nathan Foster. Looj: Weaving loom into java. To appear in *Proceedings of ECOOP 2004*, 2004.

[7] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 523–549, London, UK, 1998. Springer-Verlag.

[8] Kim B. Bruce, Angela Schuett, Robert van Gent, and Adrian Fiech. Polytoil: A type-safe polymorphic object-oriented language. *ACM Trans. Program. Lang. Syst.*, 25(2):225–290, 2003.

[9] Kim B. Bruce and Joseph Vanderwaart. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. In *Proceedings of MFPS '99*, 1999.

[10] William R. Cook, Walter Hill, and Peter S. Canning. Inheritance is not subtyping. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 125–135, New York, NY, USA, 1990. ACM Press.

[11] Erik Ernst. Family polymorphism. In Jørgen Lindskov Knudsen, editor, *Proceedings ECOOP 2001*, LNCS 2072, pages 303–326, Heidelberg, Germany, 2001. Springer-Verlag.

[12] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 270–282, New York, NY, USA, 2006. ACM Press.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[14] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Prentice Hall, 3rd edition, 2005.

[15] Atsushi Igarashi and Benjamin C. Pierce. Foundations for virtual types. In *European Conference on Object-Oriented Programming (ECOOP), Lisbon, Portugal*, June 1999. Also in informal proceedings of the *Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 1999. Full version in *Information and Computation*, 175(1): 34–49, May 2002.

[16] O. L. Madsen and B. Moller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 397–406, New York, NY, USA, 1989. ACM.

[17] Adriaan Moors, Frank Piessens, and Martin Odersky. Safe type-level abstraction in scala. In *Workshop Proceedings on Foundations of Object-Oriented Languages (FOOL)*, New York, NY, USA, 2008. ACM.

[18] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 99–115, New York, NY, USA, 2004. ACM Press.

[19] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: nested intersection for scalable software composition. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 21–36, New York, NY, USA, 2006. ACM Press.

[20] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[21] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proc. ECOOP'03*, Springer LNCS, July 2003.

[22] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 41–57, New York, NY, USA, 2005. ACM Press.

[23] Mads Torgersen. Virtual types are statically safe. In *Proceedings of the 5th Workshop on Foundations of Object-Oriented Languages (FOOL)*, San Diego, CA, 1998.

[24] Mads Torgersen. The expression problem revisited – four new solutions using generics. In *Proceedings of the 18th European Conference on Object-Oriented Programming*, June 2004.

[25] Philip Wadler. The expression problem. Message to *Java-Genericity* email list, November 1998.

[26] Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. In *FOOL 12: The twelfth international workshop on Foundations of programming languages*. ACM SIGPLAN, January 2005.

A. Justification

This section is a fairly technical justification of the claims made in the previous sections. Subsection A.1 shows that the code produced by the reduction of section 3 is type safe, given the preconditions of section 4.1. Subsection A.2 justifies the typing rule of section 4.2. Finally, subsection A.3 justifies the typing assertions of section 4.3.

This justification assumes the class C is concrete. If it is not, the exact types (and kinds) will not have the same width as the corresponding general ones and, so, the subtype relations of (A.3) must be handled more carefully (as must the sub-kind relations of (A.1)).

A.1 The reduction-produced code is type safe

In this section we show that the code produced by the reduction is type safe and results in the declarations of (4.2), (4.6) and (4.14) given the preconditions of section 4.1.

We assume that the superclass, D, is reduced by this same reduction and inductively has equivalent declarations and properties. Thus we assume that Γ (the environment before the definition of J^C) contains D's version of the declarations of (4.2), (4.6) and (4.14).

Kind definitions: The kind definitions of $C\_\alpha\beta\mathsf{K}$ in (3.6) are well-defined because $D\_\alpha\beta\mathsf{K}$ are well-defined by induction and because (4.1) ensures that the parts to the right of **with** are well-defined. The well-formedness of kinds $C\_\beta\mathsf{K}$ follows. Thus the definitions of (3.6) result in the declarations of Γ_k, defined by (4.2).

These kinds have the following properties:

$$\begin{array}{lllll} C\_\mathsf{seK} & <:: & C\_\mathsf{sgK} & <:: & D\_\mathsf{sgK} \\ C\_\mathsf{meK} & <:: & C\_\mathsf{mgK} & <:: & D\_\mathsf{mgK} \\ C\_\mathsf{eK} & <:: & C\_\mathsf{gK} & <:: & D\_\mathsf{gK} \end{array} \tag{A.1}$$

We use "K_1<::K_2" to indicate that K_1 is a "sub-kind" of K_2, which implies that any value having kind K_1 also has kind K_2. These follow from the definitions of (3.6) and subkind conditions mentioned in section 4.1.2.

Type definitions are well-kinded: The type definitions of section 3.4 can be given the kinds as specified in (4.6) by observing the following:

- Precondition (4.3) ensures that definition (3.7) gives C_iTc the kind specified in (4.6).
- Definitions (3.8) and (3.9) give $C\_\mathsf{s}\beta\mathsf{Tc}$ and $C\_\mathsf{m}\beta\mathsf{Tcc}$ the kinds specified in (4.6) by reasoning involving (A.1), (4.4) and (4.5), and the definitions of $C\_\alpha\beta\mathsf{K}$ in (3.6).
- The definitions of (3.10) give $C\_\mathsf{m}\beta\mathsf{Tc}$ the kinds specified in (4.6) by basic kinding of τ.
- Definitions of (3.11) and (3.14) give $C\_\beta\mathsf{Tc}$ the kinds specified in (4.6) by construction and the definition of $C\_\beta\mathsf{K}$ in (3.6). In C_eTc, the definition of new, being data-only, does not factor into the kind; its return type, $C@$, is a type by C::C_eK.
- Definitions of (3.12) and (3.15) give $C\_\beta\mathsf{T}$ the kinds specified in (4.6) by properties of τ.
- The definition of (3.17) gives C_hT the kind specified in (4.6) by properties of union and {{?<<C}}::*(C_gK) and that Self::C_gK implies Self@::*(C_mgK).

Thus the type definitions of (3.7) to (3.17) result in the declarations captured by Γ_t in (4.6).

Implied type relations: Consider the following argument for establishing that C's general class type subtypes D's:

$$\begin{array}{ll} C\_\mathsf{sgTc[Self]} & <: D\_\mathsf{sgTc[Self]} \\ C\_\mathsf{mgTcc[Self,self]} & <: D\_\mathsf{mgTcc[Self,self]} \\ C\_\mathsf{mgTc[Self]} & <: D\_\mathsf{mgTc[Self]} \\ C\_\mathsf{gTc[Self]} & <: D\_\mathsf{gTc[Self]} \\ C\_\mathsf{gT} & <: D\_\mathsf{gT} \end{array} \tag{A.2}$$

Assume Self::C_gK and self::C_mgK as necessary; they can be widened to D_K and D_mK by (A.1). The first two follow from the refinement conditions (4.9) and (4.10) and from the first definitions of (3.8) and (3.9). The third follows from the second and properties of τ subtyping. The first and third assertions, with definition (3.14) and D's version of (3.14), imply the fourth, which implies the last (with τ subtyping, (3.15) and D's (3.15)). There are no corresponding assertions for the exact types because there are cases for which C's exact type does not subtype D's.

Consider the following argument for establishing that C's exact class type subtypes its general type:

$$\begin{array}{ll} C\_\mathsf{seTc[Self]} & <: C\_\mathsf{sgTc[Self]} \\ C\_\mathsf{meTcc[Self,self]} & <: C\_\mathsf{mgTcc[Self,self]} \\ C\_\mathsf{meTc[Self]} & <: C\_\mathsf{mgTc[Self]} \\ C\_\mathsf{eTc[Self]} & <: C\_\mathsf{gTc[Self]} \\ C\_\mathsf{eT} & <: C\_\mathsf{gT} \end{array} \tag{A.3}$$

The first assertion follows from D's version of the first assertion, typing condition (4.7) and definitions (3.8). The second follows from D's version of it, typing condition (4.8) and definitions (3.9). The second implies the third by definitions (3.10) and τ subtyping. The first and third assertions, together with definitions (3.11) and (3.14), imply the fourth. The fourth, with definitions (3.12) and (3.15) and τ subtyping, imply the last.

Typing the class object construction: Now consider the constructor functions and class object defined in section 3.5.

The following argument shows that constructor function C_sc, as defined by (3.18), has its specified type:

- Assume Self:C_gT. By (A.2), Self:D_gT.
- Since D_sc was created by D's version of (3.18), the expression D_sc(Self) has type $D\_\mathsf{seTc[Self]}$ which is the type assigned to Super.
- Typing requirement (4.12) ensures that expression $\{\bar{x}_s{=}\bar{e}_s\}$ has type $\{\bar{x}_s{:}\bar{T}_{se}\}$.
- Expression "Super **with** $\{\bar{x}_s{=}\bar{e}_s\}$" has type "$D\_\mathsf{seTc[Self]}$ **with** $\{\bar{x}_s{:}\bar{T}_{se}\}$" which equals $C\_\mathsf{seTc[Self]}$ by definition (3.8).
- Thus the body of C_sc has type $C\_\mathsf{seTc[Self]}$ which matches the specified return type of C_sc.

The following argument shows that constructor function C_mc, as defined by (3.19), has its specified type:

- Assume Self:C_gT, <$\bar{p}$>:$C\_\mathsf{iTc[Self]}$ and self:Self@.
- Since D_mc is defined by D's version of (3.19), the expression D_mc(Self,<$\bar{a}$>,self) is type safe (by (A.2), (4.11) and self:Self@) and has type $D\_\mathsf{meTcc[Self,self]}$ which is the type assigned to super.
- Typing requirement (4.13) ensures that expression $\{\bar{x}_m{=}\bar{e}_m\}$ has type $\{\bar{x}_m{:}\bar{T}_{me}\}$.

- Expression "super **with** $\{\bar{x}_m{=}\bar{e}_m\}$" has the type that equals $C\_\mathsf{meTcc}[\mathsf{Self},\mathsf{self}]$ by definition (3.9).
- Thus the body of C_mc has type $C\_\mathsf{meTcc}[\mathsf{Self},\mathsf{self}]$ which matches the specified return type of C_mc.

The following argument shows that function C_c, as defined by (3.20), has its specified type:

- Assume $C{:}C\_\mathsf{eT}$. $C{:}C\_\mathsf{gT}$ by (A.3) and $C{:}C\_\mathsf{eTc}[C]$ by (3.12).
- The expression $C\_\mathsf{sc}(C)$ has type $C\_\mathsf{seTc}[C]$ (by basic properties of dependent functions).
- The declaration resulting from "**type** $\mathsf{Instance} = C\_\mathsf{mTc}[C]$" is "**type** $\mathsf{Instance} \leftrightarrow C\_\mathsf{mTc}[C]$".
- Function new has type $(C\_\mathsf{iTc}[C])\rightarrow C@$ by the following:
  - Assume $\mathsf{pt}{:}C\_\mathsf{iTc}[C]$.
  - Let $T_R \triangleq C\_\mathsf{meTcc}[C,\mathsf{self}]$ (to save space).
  - Expression $\mu(\mathsf{self})C\_\mathsf{mc}(C,\mathsf{pt},\mathsf{self})$ has type $\tau[\mathsf{self}]T_R$ by properties of τ and the following:
    - Assume $\mathsf{self}{:}T_R$.
    - $C{:}C\_\mathsf{gT}$ and $\mathsf{pt}{:}C\_\mathsf{iTc}[C]$ by assumptions.
    - $\mathsf{self}{:}C@$ by $C{:}C\_\mathsf{eT}$, (3.11), (3.10) and properties of τ.
    - Thus $C\_\mathsf{mc}(C,\mathsf{pt},\mathsf{self}) : T_R$ by the type of C_mc.
  - $\tau[\mathsf{self}]T_R$ equals $C\_\mathsf{meTc}[C]$ by the second definition of (3.10) which equals $C@$ by $C{:}C\_\mathsf{eTc}[C]$.
- The return type of C_c is $C\_\mathsf{eTc}[C]$ by definition (3.11).

The following argument shows that class object C, as defined by (3.21), has its specified type:

- Assign the recursion variable, C, the type $C\_\mathsf{eTc}[C]$ which is also the return type of C_c.
- The expression $C\_\mathsf{c}(C)$ is trivially well-typed.
- Thus the recursion has type $\tau[C]C\_\mathsf{eTc}[C]$ which equals C_eT by (3.12).
- C has type C_eT.

Thus the definitions of (3.18), (3.19), (3.20) and (3.21) result in the declarations captured by Γ_c in (4.14).

A.2 The typing rule is consistent with the reduction

In this section, we justify the typing rule CLASS-INTRO of section 4.2. We use the following metarule (which we claim is reasonable):

RED-TYPING-INTRO

$$\frac{J_H \longrightarrow \mathit{let}\ \bar{\mathcal{P}}{\approx}\bar{\mathcal{E}}\ \mathit{in}\ \bar{J}_L \qquad \dfrac{\bar{C}}{\Gamma \vdash \bar{J}_L \rightsquigarrow \bar{L}_L}}{\dfrac{\bar{\mathcal{P}}{\approx}\bar{\mathcal{E}} \qquad \bar{C}}{\Gamma \vdash J_H \rightsquigarrow \bar{L}_L}}$$

Basically this meta-rule says that

- if high-level definition J_H is equivalent to a series of low-level definitions $\bar{J}_L$ where the meta-variables of J_H are mapped to those of $\bar{J}_L$ by $\bar{\mathcal{P}}{\approx}\bar{\mathcal{E}}$,
- and if there is a meta-level proof that low-level definitions $\bar{J}_L$ result in a state satisfying low-level declarations $\bar{L}_L$ given preconditions $\bar{C}$,
- then the following rule is valid: high-level definition J_H results in a state satisfying the low-level declarations $\bar{L}_L$ given meta-variable mapping $\bar{\mathcal{P}}{\approx}\bar{\mathcal{E}}$ and preconditions $\bar{C}$.

In this case, J_H is the class definition of (3.1), $\bar{\mathcal{P}}{\approx}\bar{\mathcal{E}}$ is given by (3.2) to (3.5), $\bar{J}_L$ is (3.6) to (3.21), $\bar{C}$ is (4.1) to (4.14) and $\bar{L}_L$ is "$\Gamma_k;\Gamma_t;\Gamma_c$" (where $\bar{C}$ contains definitions of Γ_k, Γ_t and Γ_c to lists of those lower-level declarations resulting from (3.6) to (3.21)).

The reduction of section 3 matches the first premise of RED-TYPING-INTRO. The reasoning of section A.1 matches the second premise, because it systematically argues that, given premises $\bar{C}$, each definition of $\bar{J}_L$ is well-typed and results in the corresponding declaration of $\bar{L}_L$ (so $\bar{J}_L \rightsquigarrow \bar{L}_L$). Typing rule CLASS-INTRO of section 4.2 matches the conclusion of RED-TYPING-INTRO and, so, we consider it justified.

A.3 The assertions of 4.3 follow from the rule's conclusion

According to typing rule RED-TYPING-INTRO, the declarations of "$\Gamma_k;\ \Gamma_t;\ \Gamma_c$" (as defined by (4.2), (4.6) and (4.14)) describe the state after definition J^C of class C. In this subsection, we justify that the assertions given in section 4.3 follow from these declarations. First we make some general observations.

The following is the core assertion about class object C:

$$C : C\_\mathsf{eT} \tag{A.4}$$

This is the last declaration of (4.14). This implies, by definition (3.12), $C{:}C\_\mathsf{eTc}[C]$ which, by (3.11), has several direct implications:

$$C : C\_\mathsf{seTc}[C] \tag{A.5}$$
$$C.\mathsf{Instance} \leftrightarrow C\_\mathsf{meTc}[C] \tag{A.6}$$
$$C.\mathsf{new} : (C\_\mathsf{iTc}[C])\rightarrow C@ \tag{A.7}$$

Since $C@$ is defined to be $C.\mathsf{Instance}$ by (2.8), (A.6) is equivalent to the following:

$$C@ \leftrightarrow C\_\mathsf{meTc}[C] \tag{A.8}$$

Recall that $B{<<}C$ is equivalent to $B{:}\{\{?{<<}C\}\}$ by (2.10) which is equivalent to $B{:}C\_\mathsf{gT}$ by (3.13), which, by definitions (3.15) and (3.14), has the following direct implications:

$$B << C \implies$$
$$B : C\_\mathsf{sgTc}[B]\ \textit{and} \tag{A.9}$$
$$B@ <: C\_\mathsf{mgTc}[B] \tag{A.10}$$

(A.10) uses definition (2.8).

Recall from (3.16) and (3.17) that $C\#$ is defined to be a union:

$$C\# \leftrightarrow \cup[\mathsf{Self} << C]\ \mathsf{Self}@ \tag{A.11}$$

A direct consequence of (A.11) is the following assertion:

$$p{:}C\# \implies \exists[\mathsf{Self} << C]\ .\ p{:}\mathsf{Self}@ \tag{A.12}$$

Now we justify in detail each assertion of section 4.3.

- The assertions of (4.15) are implied by (2.9) and (A.7).
- Assertion (4.16) is implied by (A.8) and the definition of C_meTc, (3.10).
- Assertion (4.17) is implied by (A.11), (A.10) and (3.10).
- Assertion (4.18) is implied by (A.5).
- Assertion (4.19) follows from (A.11) with $\mathsf{Self}{=}C$ and (4.25).
- Assertion (4.20) is equivalent to $C\_\mathsf{hT}{<:}D\_\mathsf{hT}$, which follows from (A.11) and D's (A.11), obvious properties of unions and (A.2).
- Assertion (4.21) follows from (A.10).
- Assertion (4.22) follows from (A.9).

- Assertion (4.23) follows (A.11) with $\mathsf{Self}{=}B$.
- Assertion (4.24) follows from (A.2), (3.13) and D's (3.13).
- Assertion (4.25) is implied by (A.4), (A.3) and definition (3.13).
- Assertion (4.26) is implied by (A.4), (A.3), (A.2) and (3.13).
- Assertions (4.27) to (4.29) follow from $\mathsf{Self}{:}C\_\mathsf{gT}$ (the assumption) and from (4.22), (3.13) and (4.23), respectively.
- Assertion (4.30) follows from the assumption about Super in (4.12).
- Assertions (4.31) to (4.33) follow from the assumptions of condition (4.13); (4.32) uses (4.21).