Matching *MyType* to Subtyping

Chieri Saito[a], Atsushi Igarashi[a]

[a] *Kyoto University*

Abstract

The notion of *MyType* has been proposed to promote type-safe reuse of binary methods and recently extended to mutually recursive definitions. It is well known, however, that *MyType* does not match with subtyping well. In the current type systems, type safety is guaranteed at the expenses of subtyping, hence dynamic dispatch. In this article, we propose two mechanisms, namely, *nonheritable methods* and `exact` *statements* to remedy the mismatch between *MyType* and subtyping. We rigorously prove their safety by modeling them in a small calculus.

Key words: binary methods, dynamic dispatch, exact types, MyType, subtyping

1. Introduction

1.1. Background

It is a well-known problem that it is difficult to safely reuse recursive classes (ones that refer to one another among them) by inheritance in simple type systems such as Java's (without generics). The classical problem is found in a self-recursive class, whose methods are expected to accept the objects from the *same* class as arguments. Such methods like `equals()` are called *binary methods* [5]. Ideally, the parameter types of binary methods covariantly change as the class is extended so that subclasses are again recursive. However, it is not safe to allow such covariant change without any restrictions and so Java and C++ disallow it. As a result, the parameter types of binary methods are fixed to particular class names and this mismatch is often sidestepped by using typecasting. The use of typecasting, however, is not a solution since it may fail at run time, if used carelessly. A similar problem occurs when two or more classes are involved: paradigmatic examples are found in the implementation of node and edge classes for graphs [13] and also in the expression problem [33].

Over the last two decades, there have been many proposals to solve the problem above; a common key idea is to provide types that can be used to write recursive dependencies among the interfaces of related classes and inheritance mechanisms that keep such dependencies. According to how dependencies are expressed in type systems, these proposals can be classified into two: one with dependent types [13, 14, 21, 22, 12, 25] and one without [3, 19, 30, 18, 6, 7, 9, 4, 29] [1]. This paper focuses on the latter, which is admittedly less expressive but simpler and usually expressive enough.

1.2. MyType and its Extensions

The notion of *MyType* [3] is proposed as a means to write the dependencies in self-recursive classes. Since *MyType* refers to the declaring class and changes its meaning covariantly as classes extend, it can be used for the parameter types of binary methods.

Email addresses: `saito@kuis.kyoto-u.ac.jp` (Chieri Saito), `igarashi@kuis.kyoto-u.ac.jp` (Atsushi Igarashi)

[1] In spite of what the title of the paper [19] suggests, Concord is not really equipped with dependent types in the traditional type-theoretic sense.

Although *MyType* in earlier proposals can express only self-recursion within a single class, it has been extended to more general settings: mutually recursive classes [30, 7, 9, 4], class hierarchies [19], and arbitrarily nested groups of classes [18]. A very recent extension deals with a generic class that refers to itself recursively but with different type instantiations [29].

1.3. Mismatch between MyType and Subtyping

It is also well known that if we gave the type system with *MyType* naively, type safety would be lost. The languages with *MyType* (e.g., $\mathcal{LOOM}$ [8], LOOJ [6], Concord [19], .FJ [30], and ˆFJ [18]) have sound type systems by imposing some restrictions on their type systems. These restrictions, however, sacrifice subtyping, hence dynamic dispatch, an important feature of object-oriented programming. There are two issues: (1) it is impossible to invoke binary methods unless the run-time types of the receivers are statically known; and (2) it is difficult to implement a method, with *MyType* being its return type, that returns a new object that is of the same type as the receiver, such as `clone()`, if objects have to be created by specifying a concrete class name (or type) [30, 18, 6, 8].

In summary, in the languages with *MyType*, programs, especially client code, are often tied to a specific implementation. We want to relax this restriction and make a more object-oriented style of programming possible, in the sense that dynamic dispatch is used in more occasions.

1.4. Contributions of the Article

In this article, we propose two mechanisms, namely `exact` *statements* and *nonheritable methods*, for languages with *MyType* to remedy the mismatch between *MyType* and subtyping.

To show that our proposed features are safe, we extend Featherweight Java [16], or FJ in short, with `exact` statements and nonheritable methods and prove its type soundness.

We summarize our technical contributions as follows:

- introduction of the new features, namely, `exact` statements and nonheritable methods;
- a formalization of the type system of these features; and
- a proof of type soundness.

In this article, we extend LOOJ [6], which is a typical language that has (the unextended) *MyType*. Nevertheless, similar techniques can be used for other languages with *MyType* and its extensions. In fact, we have shown that `exact` statements and nonheritable methods are useful for self type constructors [29].

This article is a revised version of the conference paper [28], with details of the proof of type soundness.

Structure of the Article. Section 2 examines the mismatch between *MyType* and subtyping after reviewing *MyType* in LOOJ. Section 3 proposes the two features to remedy the mismatch. Section 4 gives a formal core calculus for them and proves a type soundness property. Section 5 discusses related work and Section 6 concludes. Hereafter, we write `This`, as done in [18, 29], for *MyType*.

2. Mismatch between `This` and Subtyping

In this section, we briefly review `This` and exact types [6, 18, 29], which are required for the type system to be sound, through an example of extensible self-recursive classes and then examine their problems in the current type systems such as LOOJ's.

2.1. `This` and Exact Types

`This` represents the class in which `This` appears and its subclasses. Since the meaning of `This` changes along with class extension, `This` is used to write binary methods. Consider the following class definitions:

```
class C {
  int field1;
  boolean isEqual(This that){
    return this.field1 == that.field1;
  }
}
class D extends C {
  int field2;
  boolean isEqual(This that){
    return super.isEqual(that) && this.field2 == that.field2;
  }
}
```

Class `C` declares binary method `isEqual()` and its subclass `D` overrides it. `This` refers to class `C` in class `C` whereas `This` refers to class `D` in class `D`. So, the field access `that.field2` in `isEqual()` in class `D` is legal. If we wrote `isEqual()` with the parameter type being `C` as `boolean isEqual(C that){...}`, the field access would be illegal since `D` has to override the method with the same signature, but `C` does not have `field2`. `This` in the signature of `isEqual()` called via `super` in class `D` is interpreted as `D`—the signatures of super calls are resolved with the names of the subclasses that use `super`.

If we gave the type system naively, type safety would be lost. The naive rule is that the invocation of a binary method is well typed only if the argument type is a subtype of the parameter type which is obtained by replacing `This` by the receiver type. The following example illustrates this:

```
C c1, c2;
...
c1.isEqual(c2); // unsafe
```

This method invocation seems well typed because both parameter and receiver types are `C`, but it can fail at run time—if `c1` refers to an object of class `D` and `c2` refers to that of class `C`, then the overriding definition of `isEqual()` will be executed and the field access `that.field2` fails since `c2` does not have `field2`. The problem here is that the run-time signature of `isEqual()` can be different from the compile-time one.

Exact types [6, 18, 29] are introduced into LOOJ in order to guarantee the safe invocations of binary methods such as `isEqual()`. They are used to determine run-time types statically. An exact type `@C` represents the object of class `C` exactly, excluding its proper subclasses. In other words, `@C` assures that the run-time object is always an instance of class `C`. With the help of exact types, there is a restriction for safe invocation of binary methods, that is, "*the receivers of binary methods should have exact types.*" Consider the following code:

```
@C c3; C c4, c5;
c3.isEqual(c5); // 1: allowed
c4.isEqual(c5); // 2: not allowed
```

The first call is legal since the receiver has `@C`, an exact type, and the argument type `C` is a subtype of the parameter type `C`. (The parameter type is obtained by replacing `This` with the receiver's class name.) The second call is illegal since the receiver's type is not exact. If the second were allowed, the execution could get stuck since `c4` might refer to the object of class `D`, dispatching the overridden `isEqual()`, although `c5` might refer to that of class `C`, which does not have `field2`. Hereafter, we call types without `@` *inexact*. For example, `C` is an inexact type.

2.2. Problem Description

We examine the two problems that we tackle in this paper. Figure 1 shows our running example adapted from Bruce, Petersen and Fiech [8].

Assume that we develop singly-linked lists, where each node is represented by an object, and then develop doubly-linked lists by reusing the definition of the node for the singly-linked lists. Class `LinkedNode<E>`

```
class LinkedNode<E> {
  E elem;
  @This next;
  void insert(@This that){
    @This tmp=this.next;
    this.next = that;
    that.next = tmp;
  }
  void insertElem(E e){
    @This newNode=this.makeNode();
    newNode.elem = e;
    this.insert(newNode);
  }
  @This makeNode(){ ... }
}
class DoublyLinkedNode<E> extends LinkedNode<E> {
  @This previous;
  void insert(@This that){
    super.insert(that);
    that.previous = this;
    that.next.previous = that;
  }
  @This makeNode(){ ... }
}
```

Figure 1: `LinkedNode` and `DoublyLinkedNode` classes.

defines nodes for singly-linked lists. (It is not important that the class is parameterized [2] with the element type `E` for the field `elem` and we sometimes omit the argument for it.) The class has a field `next`, which points to the next node. The type of `next` is `@This`, referring to `LinkedNode` exactly (and not to its proper subclasses). Class `DoublyLinkedNode<E>` for doubly-linked lists is defined as an extension of `LinkedNode<E>`. This class has an extra field `previous` with the type `@This` to point its previous node. Note that the inherited field `next` now refers to `DoublyLinkedNode` in the class.

The use of type `@This` for `next` and `previous` brings the following property: a singly-linked list consists of only the objects of `LinkedNode`; a doubly-linked list consists of only those of `DoublyLinkedNode`. So, whether a list is singly or doubly linked, we at least know that the linked objects are from the *same* class.

2.2.1. Binary Methods to be Dispatched Must be Always Statically Determined

As mentioned before, binary methods should be invoked on the receivers of exact types. Obviously, the benefit of dynamic dispatch is lost due to this restriction. Consider the following client code:

```
LinkedNode<Integer> head;
head.next.insert(head); // ill-typed
```

The invocation above attempts to swap the head node and its next node, where we are not sure whether `head` is a singly or doubly-linked node. The type of `head.next` is inexact type `LinkedNode<Integer>`, but not `@LinkedNode<Integer>` even though the type of `next` is exact `@This`. This is because the receiver `head` is inexact. Since `insert()` is a binary method and `head.next` is not of exact type, this call is not allowed. However, this call could be executed *safely* because whether `head` refers to a singly or doubly-linked list, the run-time types of the receiver and argument are the same. The correct implementation would be dispatched, depending on the receiver type.

One might expect that the code above can be well typed by rewriting it into a parameterized method [8]:

```
<N extends LinkedNode<Integer>>void swap(@N head){
  head.next.insert(head);
}
```

Although the method declaration is well typed, its invocation `swap(head)` is not since there is no type that instantiates the type variable `N`. So, this is not the solution that we want.

2.2.2. Methods Cannot be Specialized to the Declaring Classes

In the current type system, the name of a class is not a subtype of `This` in the class since `This` changes its meaning by inheritance. The inconvenience coming from this restriction is typically seen in writing *factory methods* [15] or cloning methods.

In Figure 1, in `insertElem()`, the factory `makeNode()` is invoked to create a new node, which initializes a variable `newNode` with type `@This`. So, the return type of `makeNode()` must be `@This`. Naive definitions of `makeNode()` would be:

```
class LinkedNode<E> {
  ...
  @This makeNode(){
    return new LinkedNode<E>();
  } // ill-typed
}
class DoublyLinkedNode<E> extends LinkedNode<E> {
  ...
  @This makeNode(){
    return new DoublyLinkedNode<E>();
  } // ill-typed
}
```

Each method returns a new object created by the class name. However, both are ill-typed, since, for example, the type `@LinkedNode<E>` of the new object is not a subtype of `@This` in class `LinkedNode<E>`. If this method were inherited to `DoublyLinkedNode` and called on its object, `LinkedNode` would appear where `DoublyLinkedNode` is expected. (It is not allowed to give exact types `@LinkedNode<E>` and `@DoublyLinkedNode<E>` to these methods as their return types since `@DoublyLinkedNode<E>` is not a subtype of `@LinkedNode<E>`—the return types are not covariantly refined.)

The ill-typed example above shows a fundamental difference between the purpose of the return type `This` and that of object creation by a concrete class name: the use of `This` means that the method is reusable in or *polymorphic* over subclasses whereas object creation makes code *specialized* for that very class, that is, not reusable for its subclasses.

In general, due to the restriction, it is impossible to write a method such that its implementation is specialized to the declaring class and, at the same time, its interface is written by using `This`[2].

Although the methods of `makeNode()` above are ill-typed, they seem to return an object of the same type as the receiver correctly.

3. Our Proposals

In this section, we propose two language features, namely, `exact` *statements* (Section 3.1) and *nonheritable methods* (Section 3.2). These features solve the problems described in Sections 2.2.1 and 2.2.2, respectively. Figure 2 shows the complete definition of the classes in Figure 1. The new code is the same except that methods `makeNode()` are declared by using nonheritable methods.

3.1. `exact` *Statements*

We propose the typing feature `exact` *statements*[3] to enable the invocation of a binary method even if the run-time type of the receiver is not identified, while we keep the restriction that "binary methods should be called on exact types." We get our idea from the context of existential types [26]. We regard an inexact type `C` as $\exists$`X<:C.@X`, where `X` can be thought of a run-time class. With this feature, the expression of an inexact type is unpacked and made temporarily exact in a local scope.

The syntax of `exact` statements is:

[2]Such methods can be implemented by using the abstract factory pattern [15, 6], which will be discussed in Section 5.
[3]In the conference paper [28], they were called *local exactization*.

```
class LinkedNode<E> {
  E elem;
  @This next;
  void insert(@This that){
    @This tmp=this.next;
    this.next = that;
    that.next = tmp;
  }
  void insertElem(E e){
    @This newNode=this.makeNode();
    newNode.elem = e;
    this.insert(newNode);
  }
  nonheritable @This makeNode(){
    return new LinkedNode<E>();
  }
}
class DoublyLinkedNode<E> extends LinkedNode<E> {
  @This previous;
  void insert(@This that){
    super.insert(that);
    that.previous = this;
    that.next.previous = that;
  }
  nonheritable @This makeNode(){
    return new DoublyLinkedNode<E>();
  }
}
```

Figure 2: `LinkedNode` and `DoublyLinkedNode` classes with nonheritable methods.

```
exact <X extends C>(@X x = target) {
  ...
}
```

"`<X extends C>`" after the keyword `exact` declares a type variable `X` to be used in the type declarations for `x` and the body (inside the braces). The expression *`target`* is that of an inexact type. At compile time, the type variable `X` is assumed to extend the type of *`target`*, `C` here, and the type of `x` is an exact type `@X`. At run time, the body is executed where `x` is initialized to the value of *`target`*.

The ill-typed invocation of `insert()` in Section 2.2.1 can be revised with `exact` as follows:

```
LinkedNode<Integer> head;
exact <X extends LinkedNode<Integer>>(@X x = head) {
  x.next.insert(x); // well-typed
}
```

The invocation is now well typed since the receiver `x.next` and argument `x` are of the same exact type `@X`.

In the body, the introduced type variable `X` is used as if it is an ordinary type. For example, we can write as follows:

```
exact <X extends LinkedNode<Integer>>(@X x = head) {
  @X n = x.makeNode();
  n.insert(n); // well-typed
}
```

Since the scope of the type variable `X` introduced by an `exact` statement is limited only within `{ }`, the type system will forget the fact that objects (for example, `x` and `x.next` in the example above) of type `@X` are instances of the same (but unknown) class. Consider the following code to get the last two nodes of `head` in the `exact` statement.

```
LinkedNode<Integer> n1, n2;
exact <X extends LinkedNode<Integer>>(@X x = head) {
  n1 = head.getLast();
  n2 = head.getLast();
}
```

The method `getLast()` returns the last node reached from the receiver through the link. Inside the `exact` statement, the return types of both calls of `getLast()` are the same `@X`. However, as the two nodes are assigned to `n1` and `n2`, which are declared outside the `exact` statement, the fact that they come from the same class is lost because the type of `n1` and `n2` is an inexact type. Note that there is no suitable exact type for `n1` and `n2` outside the `exact` statement.

We can actually avoid this unpleasant situation by using other advanced features such as generics [2] and wildcards [32].[4] First, we need a generic collection of exact types.[5]

```
class ExactList<X> {
  ...
  void add(@X x){ ... }
  @X get(int i) { ... }
}
```

`X` is a type parameter and the collection can contain only the objects of type `@X`. With this class, the code above is revised as follows:

```
ExactList<? extends LinkedNode<Integer>> s1;
exact <X extends LinkedNode<Integer>>(@X x = head) {
  ExactList<X> s2=new ExactList<X>();
  s2.add(head.getLast());
  s2.add(head.getLast());
  s1 = s2;
}
```

Inside the `exact` statement, an instance `s2` of `ExactList` is created with the type parameter being `X` and then the two last nodes are pushed into it. Finally, `s2` is assigned to `s1`, which has a wildcard type `ExactList<? extends LinkedNode<Integer>>`. It means a list of the instances of the same class, which is an unknown subclass of `LinkedNode`. So, even outside of the `exact` statement, the type system knows the two nodes in `s1` are the objects of the same class (even though it is not really known).

3.2. Nonheritable Methods

We propose the feature *nonheritable methods*[6] to allow a class name to be a subtype of `This` in that class under a certain condition. The key observation is that it is safe to allow a method whose signature contains `This` to have a specialized implementation as long as the specialized implementation is not reused in the subclasses. Nonheritable methods have the following characteristics:

1. a nonheritable method is not inherited to subclasses, in which it is impossible to call it by `super`;
2. the subclasses must override the method with the same signature;
3. `This` in the signature of a nonheritable method is replaced with the name of the declaring class when the method is typed; and
4. it is not necessary that methods overriding nonheritable methods are nonheritable.

[4] Dependent types [21, 23] can also be used to solve the problem. See Section 5.

[5] Our language design, which follows LOOJ's, does not allow type parameters to be instantiated with an exact type like `List<@Integer>`. So, we need another, very similar class definition, in which `@` is attached to type parameters.

[6] Do not confuse with NotInheritable in Visual Basic, a modifier for classes. It prevents the classes from being extended, corresponding to Java's `final`.

The first and second force each of a class and its subclasses to have its own implementation of the method. The third allows `This` and the class name to be compatible so that the method will be appropriate for the class. The fourth is mainly for convenience: we can stop the overriding chain in subclasses if we want.

In Figure 2, each of `LinkedNode` and `DoublyLinkedNode` implements the nonheritable method `makeNode()`, modified by `nonheritable`. Both are well typed since, for example, in `LinkedNode`, `This` in the return type `@This` is replaced with `LinkedNode` and its result `@LinkedNode` is a (trivial) supertype of the type `@LinkedNode` of the new object. If `DoublyLinkedNode` did not override `makeNode()`, it would be ill-typed and in fact should be as we have seen in Section 2.2.2.

Using `nonheritable`, the cloning method, which returns the shallow copy of the receiver object, can be defined as follows:

```
class LinkedNode<E> {
  ...
  nonheritable @This clone() {
    @This copy=new LinkedNode<E>();
    copy.next = this.next;
    return copy;
} }
class DoublyLinkedNode<E> extends LinkedNode<E> {
  ...
  nonheritable @This clone() {
    @This copy=new DoublyLinkedNode<E>();
    copy.next = this.next;
    copy.previous = this.previous;
    return copy;
} }
```

In the methods `clone()` above, objects are created by the class names and the pointers to the `next` and `previous` objects are copied. Note that in this definition the code `copy.next = this.next;` is not reused since nonheritable methods are not allowed to be called by `super` for type safety. Actually, if we wrote `clone()` in `DoublyLinkedNode` with `super.clone()` as below, calling the overridden `clone()` would result in throwing `NoSuchFieldException` because `copy` refers to the object of `LinkedNode`, which does not have `previous`.

```
class DoublyLinkedNode<E> extends LinkedNode<E> {
  ...
  nonheritable @This clone() {
    @This copy=super.clone();
    copy.previous = this.previous;
    return copy;
} }
```

More code reuse is possible by extracting class-specific behavior in `clone()` as a separate nonheritable method. In fact, using `makeNode()`, we can write `clone()`, independent from specific class names, as follows:

```
class LinkedNode<E> {
  ...
  @This clone() {
    @This copy=this.makeNode();
    copy.next = this.next;
    return copy;
} }
class DoublyLinkedNode<E> extends LinkedNode<E> {
  ...
  @This clone() {
    @This copy=super.clone();
    copy.previous = this.previous;
    return copy;
} }
```

Here, the code `copy.next = this.next;` is reused in `DoublyLinkedNode`. However, it is a responsibility of the programmers to make sure to override `clone()`, which is not `nonheritable`.

4. A Formal Core Calculus

In this section, we formalize the ideas described in the previous section as a small calculus based on Featherweight Java [16], a functional core of class-based object-oriented languages. What we model here includes `This`, exact types, `exact` statements, and nonheritable methods, as well as the usual features of calculi of the FJ family, that is, fields, methods, object creations and recursion by `this`. Since this calculus is functional, mutable fields are not formalized. Section 4.1 defines the syntax; Sections 4.2 and 4.3 define the type system; Section 4.4 defines the operational semantics. Finally, we show type soundness in Section 4.5.

4.1. Syntax

The abstract syntax of types, class declarations, method declarations, expressions, and values is given below. In method declarations, the modifier `nonheritable` is optional. The metavariables `C`, `D`, and `E` range over class names; `X` and `Y` range over type variables; `f` and `g` range over field names; `m` ranges over method names; `x` and `y` range over variables.

$\mathtt{G},\mathtt{H},\mathtt{I}$	::=	$\mathtt{X} \mid \mathtt{C}$	inexact types
$\mathtt{S},\mathtt{T},\mathtt{U}$	::=	$\mathtt{H} \mid \mathtt{@H}$	types
$\mathtt{L}$	::=	$\mathtt{class\ C\ extends\ C\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\mathtt{\}}$	class declarations
$\mathtt{M}$	::=	$[\mathtt{nonheritable}]\ \mathtt{T\ m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e;\ \}}$	method declarations
$\mathtt{d},\mathtt{e}$	::=	$\mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m(}\overline{\mathtt{e}}\mathtt{)} \mid \mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{)} \mid \mathtt{exact\ e\ as\ x,\ X\ in\ e}$	expressions
$\mathtt{v}$	::=	$\mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)}$	values

Following the custom of FJ, we put a line over a metavariable to denote a possibly empty sequence. Furthermore, we abbreviate pairs of sequences in a similar way, writing "$\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;}$" for "$\mathtt{T}_1\ \mathtt{f}_1\mathtt{;}\ldots\ \mathtt{T}_n\ \mathtt{f}_n\mathtt{;}$", where n is the length of $\overline{\mathtt{T}}$ and $\overline{\mathtt{f}}$. Sequences of field declarations, parameter names, and method definitions are assumed to contain no duplicate names. We write the empty sequence as $\bullet$ and concatenation of sequences using a comma. As in FJ, every class has a single constructor that takes initial values of all the fields and assigns them; we omit constructor declarations for simplicity. Also for simplicity, generics is not supported, unlike LOOJ. Typecasts are dropped since we aim at safe and extensible programming without typecasting, a possibly unsafe operation. We omit `super` calls for brevity.

An inexact type is either a type variable or a class name. A type is either an inexact type or an exact type, which is obtained by adding `@` to an inexact type. Since this language is expression-based, the body of a method is a single `return` statement, rather than a compound statement as in the examples in the previous section. An expression is either a variable, a field access, a method invocation, an object creation, or an `exact` *expression*, whose body is an expression. We assume that the set of variables includes the special variable `this`, which cannot be used as the name of a parameter to a method, and that the set of type variables includes the special type variable `This`.

We use a different syntax for `exact` expressions to simplify the notation. An `exact` statement

```
exact <X extends C>(@X x = target) { body }
```

corresponds to `exact` *target* `as x, X in` *body* here. (The upper bound of `X` will be determined by the type of *target*.) The variables `x` and `X` in `exact` $\mathtt{e}_0$ `as x, X in` $\mathtt{e}_1$ are bound in the body expression $\mathtt{e}_1$.

A class table CT is a finite mapping from class names `C` to class declarations `L` and is assumed to satisfy the following sanity conditions: (1) $CT(\mathtt{C}) = \mathtt{class\ C\ \ldots}$ for every $\mathtt{C} \in dom(CT)$; (2) $\mathtt{Object} \notin dom(CT)$; (3) for every class name `C` (except `Object`) appearing anywhere in CT, we have $\mathtt{C} \in dom(CT)$; and (4) there are no cycles in the inheritance relation induced by CT. A program is a pair $(CT, \mathtt{e})$. In what follows, we assume a *fixed* class table CT to simplify the notation.

4.2. Lookup Functions

We give functions to look up field or method definitions. The function *fields*(`C`) returns a sequence $\overline{\mathtt{T}}\ \overline{\mathtt{f}}$ of field names of the class `C` with their types. The function *mtype*(`m`, `C`) takes a method name and a class name as input and returns the corresponding method signature of the form $\overline{\mathtt{T}} \to \mathtt{T}_0$. They are defined by the

rules below. Unlike LOOJ, type substitution for `This` is not performed in the definitions—it is performed in the typing rules. So, the definitions are the same as those in FJ [16] except that *mtype* accounts the optional modifier `nonheritable`. Here, $\mathtt{m} \notin \overline{\mathtt{M}}$ means the method of name `m` does not exist in $\overline{\mathtt{M}}$.

$$\mathit{fields}(\mathtt{Object}) = \bullet \qquad \frac{\mathtt{class\ C\ extends\ D\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;...\}} \qquad \mathit{fields}(\mathtt{D}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}}{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}, \overline{\mathtt{T}}\ \overline{\mathtt{f}}}$$

$$\frac{\mathtt{class\ C\ extends\ D\ \{...}\overline{\mathtt{M}}\mathtt{\}} \quad [\mathtt{nonheritable}]\ \mathtt{T_0\ m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e;\ \}} \in \overline{\mathtt{M}}}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{T}} \to \mathtt{T_0}} \qquad \frac{\mathtt{class\ C\ extends\ D\{...}\overline{\mathtt{M}}\mathtt{\}} \quad \mathtt{m} \notin \overline{\mathtt{M}} \quad \mathit{mtype}(\mathtt{m}, \mathtt{D}) = \overline{\mathtt{T}} \to \mathtt{T_0}}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{T}} \to \mathtt{T_0}}$$

4.3. Type System

The main judgments of the type system consist of $\Delta \vdash \mathtt{S <: T}$ for subtyping, $\Delta \vdash \mathtt{T}$ ok for type well-formedness, and $\Delta; \Gamma \vdash \mathtt{e : T}$ for expression typing. Here, Δ is a *bound environment*, which is a finite mapping from type variables to their bounds, written $\overline{\mathtt{X}} \mathtt{<:} \overline{\mathtt{H}}$. The scope of a type variable in a bound environment is the following type variable declarations. Γ is a *type environment*, which is a finite mapping from variables to types, written $\overline{\mathtt{x}} \mathtt{:} \overline{\mathtt{T}}$. The concatenation Δ_1, Δ_2 of bound environments Δ_1 and Δ_2 is defined, when their domains are disjoint, as follows: $\mathit{dom}(\Delta_1, \Delta_2) = \mathit{dom}(\Delta_1) \cup \mathit{dom}(\Delta_2)$ and $(\Delta_1, \Delta_2)(\mathtt{X}) = \Delta_1(\mathtt{X})$ if $\mathtt{X} \in \mathit{dom}(\Delta_1)$ and $(\Delta_1, \Delta_2)(\mathtt{X}) = \Delta_2(\mathtt{X})$ if $\mathtt{X} \in \mathit{dom}(\Delta_2)$. The concatenation Γ_1, Γ_2 of type environments is defined similarly. We abbreviate a sequence of judgments: $\Delta \vdash \mathtt{S}_1 \mathtt{<:} \mathtt{T}_1, \ldots, \Delta \vdash \mathtt{S}_n \mathtt{<:} \mathtt{T}_n$ to $\Delta \vdash \overline{\mathtt{S}} \mathtt{<:} \overline{\mathtt{T}}$, and $\Delta \vdash \mathtt{T}_1$ ok, $\ldots, \Delta \vdash \mathtt{T}_n$ ok to $\Delta \vdash \overline{\mathtt{T}}$ ok, and $\Delta; \Gamma \vdash \mathtt{e}_1 \mathtt{:} \mathtt{T}_1, \ldots, \Delta; \Gamma \vdash \mathtt{e}_n \mathtt{:} \mathtt{T}_n$ to $\Delta; \Gamma \vdash \overline{\mathtt{e}} \mathtt{:} \overline{\mathtt{T}}$. We write $[\overline{\mathtt{H}}/\overline{\mathtt{X}}]$ for the simultaneous substitution of $\mathtt{H}_1$ for $\mathtt{X}_1$, ..., and of $\mathtt{H}_n$ for $\mathtt{X}_n$. We assume that $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}$ implies $\Delta \vdash \Gamma(\mathtt{x})$ ok for all $\mathtt{x} \in \mathit{dom}(\Gamma)$.

4.3.1. Bounds of Types

The function $\mathit{bound}_\Delta(\mathtt{H})$, defined below, takes an inexact type as input and returns a class name `C`, which is the least upper bound of the input type.

$$\mathit{bound}_\Delta(\mathtt{X}) = \mathit{bound}_\Delta(\Delta(\mathtt{X})) \qquad \mathit{bound}_\Delta(\mathtt{C}) = \mathtt{C}$$

If the input is a type variable, the function is recursively applied to the output, which, again, can be a type variable.

4.3.2. Subtyping

The subtyping judgment $\Delta \vdash \mathtt{S <: T}$, read as "`S` is a subtype of `T` under Δ," is defined below. This relation is the reflexive and transitive closure of the `extends` relation with the rule that an exact type is a subtype of its inexact version. Note that $\mathtt{@C} \not<: \mathtt{@D}$ even if `class C extends D{..}`. So, this subtyping relation is actually *matching* [8] since $\mathtt{C <: D}$ does not always mean that an object of class `C`, which is of type `@C`, is substitutable for one of class `D`, which is of type `@D`.

$$\Delta \vdash \mathtt{T <: T} \qquad \Delta \vdash \mathtt{X <:} \Delta(\mathtt{X}) \qquad \Delta \vdash \mathtt{@H <: H}$$

$$\frac{\mathtt{class\ C\ extends\ D\{...}\}}{\Delta \vdash \mathtt{C <: D}} \qquad \frac{\Delta \vdash \mathtt{S <: T} \quad \Delta \vdash \mathtt{T <: U}}{\Delta \vdash \mathtt{S <: U}}$$

4.3.3. Type Well-formedness

The type well-formedness judgment $\Delta \vdash \mathtt{T}$ ok, read as "`T` is a well-formed type under Δ," is defined below. `Object` and class names in *CT* are well formed. Type `X` is well formed if it is in the domain of Δ. Finally, an exact type `@H` is well formed if so is its inexact version `H`.

$$\Delta \vdash \mathtt{Object}\ \text{ok} \qquad \frac{\mathtt{class\ C\ extends\ D\{..}\}}{\Delta \vdash \mathtt{C}\ \text{ok}} \qquad \frac{\mathtt{X} \in \mathit{dom}(\Delta)}{\Delta \vdash \mathtt{X}\ \text{ok}} \qquad \frac{\Delta \vdash \mathtt{H}\ \text{ok}}{\Delta \vdash \mathtt{@H}\ \text{ok}}$$

4.3.4. Expression Typing

The typing judgment for expressions is of the form $\Delta;\Gamma \vdash \mathtt{e}:\mathtt{T}$, read as "under bound environment Δ and type environment Γ, expression $\mathtt{e}$ has type $\mathtt{T}$," defined by the following rules:

$$\Delta;\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \qquad \text{(T-Var)}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{@H}_0 \qquad \mathit{fields}(\mathit{bound}_\Delta(\mathtt{H}_0)) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{f}_i : [\mathtt{H}_0/\mathtt{This}]\mathtt{T}_i} \qquad \text{(T-Field-E)}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{H}_0 \qquad \mathit{fields}(\mathit{bound}_\Delta(\mathtt{H}_0)) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \Delta, \mathtt{This}{<:}\mathtt{H}_0 \vdash \mathtt{T}_i{<:}\mathtt{T} \qquad \Delta \vdash \mathtt{T}\ \mathrm{ok}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{f}_i : \mathtt{T}} \qquad \text{(T-Field-I)}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{@H}_0 \qquad \mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{H}_0)) = \overline{\mathtt{T}}\to\mathtt{T}_0 \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \qquad \Delta \vdash \overline{\mathtt{U}} <: [\mathtt{H}_0/\mathtt{This}]\overline{\mathtt{T}}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : [\mathtt{H}_0/\mathtt{This}]\mathtt{T}_0} \qquad \text{(T-Invk-E)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{H}_0 \qquad \mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{H}_0)) = \overline{\mathtt{T}}\to\mathtt{T}_0 \qquad \overline{\mathtt{T}} \text{ do not contain } \mathtt{This} \\ \Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \qquad \Delta \vdash \overline{\mathtt{U}} <: \overline{\mathtt{T}} \qquad \Delta, \mathtt{This}{<:}\mathtt{H}_0 \vdash \mathtt{T}_0{<:}\mathtt{T} \qquad \Delta \vdash \mathtt{T}\ \mathrm{ok}\end{array}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{T}} \qquad \text{(T-Invk-I)}$$

$$\frac{\Delta \vdash \mathtt{C}_0\ \mathrm{ok} \qquad \mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \qquad \Delta \vdash \overline{\mathtt{U}} <: [\mathtt{C}_0/\mathtt{This}]\overline{\mathtt{T}}}{\Delta;\Gamma \vdash \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{e}}) : \mathtt{@C}_0} \qquad \text{(T-New)}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_1 : \mathtt{H} \qquad \Delta, \mathtt{X}{<:}\mathtt{H}; \Gamma, \mathtt{x} : \mathtt{@X} \vdash \mathtt{e}_0 : \mathtt{U}_0 \qquad \Delta, \mathtt{X}{<:}\mathtt{H} \vdash \mathtt{U}_0{<:}\mathtt{T}_0 \qquad \Delta \vdash \mathtt{T}_0\ \mathrm{ok}}{\Delta;\Gamma \vdash \mathtt{exact}\ \mathtt{e}_1\ \mathtt{as}\ \mathtt{x},\ \mathtt{X}\ \mathtt{in}\ \mathtt{e}_0 : \mathtt{T}_0} \qquad \text{(T-Exact-I)}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_1 : \mathtt{@H} \qquad \Delta;\Gamma, \mathtt{x} : \mathtt{@H} \vdash \mathtt{e}_0 : \mathtt{U}_0}{\Delta;\Gamma \vdash \mathtt{exact}\ \mathtt{e}_1\ \mathtt{as}\ \mathtt{x},\ \mathtt{X}\ \mathtt{in}\ \mathtt{e}_0 : \mathtt{U}_0} \qquad \text{(T-Exact-E)}$$

There are two rules for field accesses and method invocations, respectively, depending on weather the receivers are of exact or inexact types. The rule T-Field-E is for field accesses on exact types. In this case the type of field access $\mathtt{e}_0.\mathtt{f}_i$ is obtained by looking up field declarations from the bound of $\mathtt{H}_0$ and then substituting $\mathtt{H}_0$ for `This` in the type $\mathtt{T}_i$ corresponding to $\mathtt{f}_i$. The rule T-Field-I is for receivers of inexact types. In this rule, the type of the whole expression is a "`This`-free" supertype $\mathtt{T}$ of $\mathtt{T}_i$ so that `This` does not escape from its scope (that is, the receiver's class). The last premise ensures that $\mathtt{T}$ does not contain `This`. (As we will see in method typing, when the expression appears in a method, Δ already contains `This` in its domain. So, strictly speaking, implicit renaming of type variables—that is, `This`—in the judgments is assumed here, to avoid confusing the two kinds of `This`: one in Δ for the class in which this expression appears and the other from the receiver's class.)

The rule T-Invk-E for method invocations on exact types is defined similarly to T-Field-E: the method type is retrieved from the receiver's type; then, it is checked if the types of actual arguments are subtypes of those of the formal parameters. The rule T-Invk-I, for invocations on inexact types, is defined similarly to T-Field-I. The condition that $\overline{\mathtt{T}}$ do not contain `This` expresses that binary methods cannot be invoked on inexact types.

The rule T-New says that the type of a `new` expression is the *exact* type of the class being instantiated.

There are two rules for `exact` expressions, depending on whether the type of expression $\mathtt{e}_1$ is inexact $\mathtt{H}$ or exact `@H`. The rule T-Exact-I means that if $\mathtt{e}_1$ is of type $\mathtt{H}$, the body expression $\mathtt{e}_0$ is typed under Δ extended by $\mathtt{X}{<:}\mathtt{H}$ and Γ extended by $\mathtt{x}:\mathtt{@X}$. Note that variable $\mathtt{x}$ is of type `@X`, an exact type. Since the result type $\mathtt{U}_0$ may contain the type variable $\mathtt{X}$, the type of the whole expression is obtained by taking a supertype $\mathtt{T}_0$ of $\mathtt{U}_0$ so that $\mathtt{T}_0$ does not contain $\mathtt{X}$, preventing $\mathtt{X}$ from escaping. The rule T-Exact-E, which will not be used in ordinary programs, is required to show the subject reduction property since an expression

of inexact type eventually reduces to one (typically a value) of an exact type at run time. In this rule, the body expression is typed under Δ unextended and Γ extended by `x:@H` since there is no proper subtype of `@H`.

We show the typing examples of field accesses on exact and inexact types. Assume that *fields*(`LinkedNode`) contains `@This next`. The following derivation tree is an example of field access on an exact type:

$$\frac{\Delta;\Gamma \vdash \texttt{n} : \texttt{@LinkedNode} \quad \texttt{@This next} \in \mathit{fields}(\mathit{bound}_\Delta(\texttt{LinkedNode}))}{\Delta;\Gamma \vdash \texttt{n.next} : \texttt{@LinkedNode}(= [\texttt{LinkedNode}/\texttt{This}]\texttt{@This})}\ \text{T-Field-E}$$

The next tree $\mathcal{D}_1$ is one on an inexact type:

$$\frac{\Delta;\Gamma \vdash \texttt{n:LinkedNode} \qquad \mathcal{D}_2 \qquad \Delta, \texttt{This<:LinkedNode} \vdash \texttt{@This<:LinkedNode} \qquad \mathcal{D}_3}{\Delta;\Gamma \vdash \texttt{n.next:LinkedNode}}\ \text{T-Field-I}$$

where $\mathcal{D}_2$ is `@This next` $\in \mathit{fields}(\mathit{bound}_\Delta(\texttt{LinkedNode}))$ and $\mathcal{D}_3$ is $\Delta \vdash$ `LinkedNode` ok.

By inserting `exact`, the member access on inexact `LinkedNode` above can be derived by the combination of rules T-Field-E and T-Exact-I, as the following tree $\mathcal{D}_4$ shows:

$$\frac{\Delta;\Gamma \vdash \texttt{n} : \texttt{LinkedNode} \quad \mathcal{D}_5 \quad \Delta, \texttt{X<:LinkedNode} \vdash \texttt{@X<:LinkedNode} \quad \Delta \vdash \texttt{LinkedNode ok}}{\Delta;\Gamma \vdash \texttt{exact n as x, X in x.next} : \texttt{LinkedNode}}\ \text{T-Exact-I}$$

where $\mathcal{D}_5$ is

$$\frac{\Delta, \texttt{X<:LinkedNode};\Gamma, \texttt{x} : \texttt{@X} \vdash \texttt{x} : \texttt{@X} \quad \mathit{fields}(\mathit{bound}_{\Delta,\texttt{X<:LinkedNode}}(\texttt{X})) = \mathit{fields}(\texttt{LinkedNode})}{\Delta, \texttt{X<:LinkedNode};\Gamma, \texttt{x} : \texttt{@X} \vdash \texttt{x.next} : \texttt{@X}(= [\texttt{X}/\texttt{This}]\texttt{@This})}\ \text{T-Field-E}$$

In the conclusion, `exact` is inserted before accessing the field. By comparing the derivations $\mathcal{D}_1$ and $\mathcal{D}_4$, one can find that the rules T-Field-I/T-Invk-I can be considered as the combination of T-Field-E/T-Invk-E and T-Exact-I, in which `exact` expressions are dropped from the conclusions.

4.3.5. Closing of Types

In the rule T-Exact-I, $\texttt{T}_0$ is not unique for given `X`, `H`, $\texttt{U}_0$ and Δ. It is easy to give a set of rules to determine a minimal `X`-free supertype of $\texttt{U}_0$. We introduce the judgment *closing of types*, written $\texttt{S} \Downarrow_{\texttt{X<:H}} \texttt{T}$, meaning that "`T` is a minimal supertype of `S` without `X`". We also say that "type `S` is closed to `T` under `X<:H`" [17]. In T-Exact-I, $\texttt{U}_0 \Downarrow_{\texttt{X<:H}} \texttt{T}_0$ can be used for $\Delta, \texttt{X<:H} \vdash \texttt{U}_0 <: \texttt{T}_0$ and $\Delta \vdash \texttt{T}_0$ ok. Similarly, $\texttt{T}_i \Downarrow_{\texttt{X<:H}} \texttt{T}$ for $\Delta, \texttt{X<:H} \vdash \texttt{T}_i <: \texttt{T}$ and $\Delta \vdash$ `T` ok in T-Field-I and $\texttt{T}_0 \Downarrow_{\texttt{X<:H}} \texttt{T}$ for $\Delta, \texttt{X<:H} \vdash \texttt{T}_0 <: \texttt{T}$ and $\Delta \vdash$ `T` ok in T-Invk-I. Separating this judgment from typing rules makes the proof of subject reduction more concise.

The rules for closing of types are as follows:

$$\frac{\texttt{T} \neq \texttt{X} \qquad \texttt{T} \neq \texttt{@X}}{\texttt{T} \Downarrow_{\texttt{X<:H}} \texttt{T}} \qquad \texttt{X} \Downarrow_{\texttt{X<:H}} \texttt{H} \qquad \texttt{@X} \Downarrow_{\texttt{X<:H}} \texttt{H}$$

The left rule says that if type `T` does not contain type variable `X`, the result is the same `T`. The other rules say that both `X` and `@X` close to `H` under `X<:H`. Note that `@X` does not close to `@H` since the subtyping relation that holds is `@X<:X<:H`, but `@X` $\not<:$ `@H`.

4.3.6. Method Typing

The typing judgment for method declarations is written $\texttt{C} \vdash \texttt{M}$ ok. There are two rules, T-Method for usual methods and T-NHMethod for nonheritable methods. In each rule, the last premise is to check if the method correctly overrides (if it does) the method of the same name in the superclass with the same signature. A further explanation is given only for the latter since the former is straightforward. In premises, `This` that appears in the signature is replaced with the class name `C` as well as `this` is of type `@C` (= [`C`/`This`]`@This`) in the expression typing judgment. As a result, the method declaration is safe only for the declaring class and would be unsafe if its subclasses inherited it.

$$\frac{\Delta = \{\texttt{This<:C}\} \quad \Gamma = \{\overline{\texttt{x}} : \overline{\texttt{T}}, \texttt{this} : \texttt{@This}\} \quad \Delta;\Gamma \vdash \texttt{e}_0 : \texttt{U}_0 \quad \Delta \vdash \texttt{U}_0 <: \texttt{T}_0 \quad \Delta \vdash \texttt{T}_0, \overline{\texttt{T}} \text{ ok} \quad \texttt{class C extends D\{...\}} \quad \mathit{mtype}(\texttt{m}, \texttt{D}) = \overline{\texttt{S}} \to \texttt{S}_0 \text{ implies } (\overline{\texttt{S}}, \texttt{S}_0) = (\overline{\texttt{T}}, \texttt{T}_0)}{\texttt{C} \vdash \texttt{T}_0\ \texttt{m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\texttt{\{ return e}_0\texttt{; \}} \text{ ok}} \text{ (T-Method)}$$

$$\frac{\Gamma = \{\overline{\texttt{x}} : \overline{\texttt{T}}, \texttt{this} : \texttt{@This}\} \quad \emptyset;[\texttt{C/This}]\Gamma \vdash \texttt{e}_0 : \texttt{U}_0 \quad \emptyset \vdash \texttt{U}_0 <: [\texttt{C/This}]\texttt{T}_0 \quad \emptyset \vdash [\texttt{C/This}](\texttt{T}_0, \overline{\texttt{T}}) \text{ ok} \quad \texttt{class C extends D\{...\}} \quad \mathit{mtype}(\texttt{m}, \texttt{D}) = \overline{\texttt{S}} \to \texttt{S}_0 \text{ implies } (\overline{\texttt{S}}, \texttt{S}_0) = (\overline{\texttt{T}}, \texttt{T}_0)}{\texttt{C} \vdash \texttt{nonheritable T}_0\ \texttt{m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\texttt{\{ return e}_0\texttt{; \}} \text{ ok}} \text{ (T-NHMethod)}$$

4.3.7. Class Typing

The typing judgment for class declarations is written $\vdash \texttt{L}$ ok. The rule T-Class checks if the field types are well formed and if the method declarations are ok, as done in FJ. The introduction of nonheritable methods requires an additional check to make sure that all the nonheritable methods in the superclass are overridden. Here, $\texttt{m} \in \overline{\texttt{M}}$ means that the method of name m exists in $\overline{\texttt{M}}$.

$$\frac{\texttt{C} \vdash \overline{\texttt{M}} \text{ ok} \quad \texttt{This<:C} \vdash \overline{\texttt{T}}, \texttt{D} \text{ ok} \quad \text{if } \texttt{D} \neq \texttt{Object} \text{ and } \texttt{class D extends E\{..}\overline{\texttt{M}}'\texttt{\}}, \text{ then for each } \texttt{nonheritable U}_0\ \texttt{m}(\overline{\texttt{U}}\ \overline{\texttt{x}})\texttt{\{..\}} \in \overline{\texttt{M}}', \texttt{m} \in \overline{\texttt{M}}}{\vdash \texttt{class C extends D\{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{;}\ \overline{\texttt{M}}\texttt{\}} \text{ ok}} \text{ (T-Class)}$$

A program $(CT, \texttt{e})$ is ok if all definitions in CT are ok and $\emptyset;\emptyset \vdash \texttt{e} : \texttt{T}$ for some T.

4.4. Operational Semantics

The operational semantics is given by the reduction relation of the form $\texttt{e} \longrightarrow \texttt{e}'$, read "expression e reduces to $\texttt{e}'$ in one step." We require another lookup function $\mathit{mbody}(\texttt{m}, \texttt{C})$ for method body with formal parameters, written $\overline{\texttt{x}}.\texttt{e}$, of given method and class names.

The reduction rules are given below. We write $[\overline{\texttt{d}}/\overline{\texttt{x}}, \texttt{e}/\texttt{y}]$ for the capture-avoiding simultaneous substitution of $\texttt{d}_1$ for $\texttt{x}_1$, ..., of $\texttt{d}_n$ for $\texttt{x}_n$, and of e for y. The rule R-Exact means that when the target of an exact expression is a new expression, the body $\texttt{e}_0$ is evaluated where x is bound to $\texttt{new C(}\overline{\texttt{e}}\texttt{)}$ and X is bound to C. Note that the application of type substitution $[\texttt{C/X}]$ to $\texttt{e}_0$ is omitted since no type variables appear in expressions. Similarly, [C/This] is omitted in R-Invk. The reduction rules may be applied at any point in an expression, so we also need the congruence rules. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

$$\frac{\texttt{class C extends D\{...}\overline{\texttt{M}}\texttt{\}} \quad [\texttt{nonheritable}]\ \texttt{T}_0\ \texttt{m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\texttt{\{ return e}_0\texttt{; \}} \in \overline{\texttt{M}}}{\mathit{mbody}(\texttt{m}, \texttt{C}) = \overline{\texttt{x}}.\texttt{e}_0} \text{ (MB-Class)}$$

$$\frac{\texttt{class C extends D \{...}\overline{\texttt{M}}\texttt{\}} \quad \texttt{m} \notin \overline{\texttt{M}} \quad \mathit{mbody}(\texttt{m}, \texttt{D}) = \overline{\texttt{x}}.\texttt{e}_0}{\mathit{mbody}(\texttt{m}, \texttt{C}) = \overline{\texttt{x}}.\texttt{e}_0} \text{ (MB-Super)}$$

$$\frac{\mathit{fields}(\texttt{C}) = \overline{\texttt{T}}\ \overline{\texttt{f}}}{\texttt{new C(}\overline{\texttt{e}}\texttt{).f}_i \longrightarrow \texttt{e}_i} \text{ (R-Field)}$$

$$\frac{\mathit{mbody}(\texttt{m}, \texttt{C}) = \overline{\texttt{x}}.\texttt{e}_0}{\texttt{new C(}\overline{\texttt{e}}\texttt{).m(}\overline{\texttt{d}}\texttt{)} \longrightarrow [\overline{\texttt{d}}/\overline{\texttt{x}}, \texttt{new C(}\overline{\texttt{e}}\texttt{)/this}]\texttt{e}_0} \text{ (R-Invk)}$$

$$\frac{}{\texttt{exact new C(}\overline{\texttt{e}}\texttt{) as x, X in e}_0 \longrightarrow [\texttt{new C(}\overline{\texttt{e}}\texttt{)/x}]\texttt{e}_0} \text{ (R-Exact)}$$

$$\frac{\texttt{e}_0 \longrightarrow \texttt{e}_0{}'}{\texttt{e}_0\texttt{.f} \longrightarrow \texttt{e}_0{}'\texttt{.f}} \text{ (RC-Field)}$$

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'}{\mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) \longrightarrow \mathtt{e}_0{}'.\mathtt{m}(\overline{\mathtt{e}})} \quad \text{(RC-Invk-Recv)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i{}'}{\mathtt{e}_0.\mathtt{m}(\ldots,\mathtt{e}_i,\ldots) \longrightarrow \mathtt{e}_0.\mathtt{m}(\ldots,\mathtt{e}_i{}',\ldots)} \quad \text{(RC-Invk-Arg)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i{}'}{\mathtt{new\ C}(\ldots,\mathtt{e}_i,\ldots) \longrightarrow \mathtt{new\ C}(\ldots,\mathtt{e}_i{}',\ldots)} \quad \text{(RC-New-Arg)}$$

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'}{\mathtt{exact\ e}_0\ \mathtt{as\ x,\ X\ in\ e}_1 \longrightarrow \mathtt{exact\ e}_0{}'\ \mathtt{as\ x,\ X\ in\ e}_1} \quad \text{(RC-Exact)}$$

$$\frac{\mathtt{e}_1 \longrightarrow \mathtt{e}_1{}'}{\mathtt{exact\ e}_0\ \mathtt{as\ x,\ X\ in\ e}_1 \longrightarrow \mathtt{exact\ e}_0\ \mathtt{as\ x,\ X\ in\ e}_1{}'} \quad \text{(RC-Exact-Body)}$$

4.5. Properties

The type system is sound with respect to the operational semantics, as expected. Type soundness is proved in the standard manner via subject reduction and progress [35, 16]. Here, we show only the statements. See Appendix A for the required lemmas and proof sketches of the theorems.

Theorem 1 (Subject Reduction). *If* $\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{e} \longrightarrow \mathtt{e}'$*, then* $\Delta;\Gamma \vdash \mathtt{e}' : \mathtt{T}'$*, for some* $\mathtt{T}'$ *such that* $\Delta \vdash \mathtt{T}' <: \mathtt{T}$.

Proof. See Appendix A. □

Theorem 2 (Progress). *If* $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{e}$ *is not a value, then* $\mathtt{e} \longrightarrow \mathtt{e}'$*, for some* $\mathtt{e}'$.

Proof. See Appendix A. □

Theorem 3 (Type Soundness). *If* $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{e} \longrightarrow^* \mathtt{e}'$ *with* $\mathtt{e}'$ *a normal form, then* $\mathtt{e}'$ *is a value* $\mathtt{v}$ *with* $\emptyset;\emptyset \vdash \mathtt{v} : \mathtt{T}'$ *and* $\Delta \vdash \mathtt{T}' <: \mathtt{T}$.

Proof. Immediate from Theorems 1 and 2. □

5. Related Work

The first two subsections discuss the work related to `exact` statements whereas the next five discuss that to nonheritable methods. The last subsection discusses how to simulate nonheritable methods and `exact` statements in current Java.

5.1. Existential Types in Java

In our type system, inexact types are treated as existential. Java is already equipped with a kind of existential types for generics [2], called wildcard types [32], derived from variant parametric types [17]. While type arguments for a parameterized class are abstracted in wildcard types in Java, run-time classes are abstracted in inexact types in our language. Cameron and Drossopoulou [10] formalize the calculus ∃FJ, in which all types are considered existential as our inexact types.

Pizza [24], one of the earliest proposals of adding generics to Java, also has existential types (only internally, though, in the sense that programmers cannot write down existential types in their programs). The unpacking construct is integrated into the `switch` statement, which makes branches by pattern matching; here, a pattern to test the run-time class of an object may contain type variables, which stand for (existential) type arguments to the generic class of the matching object.

5.2. Dependent Type Systems

In the languages [13, 21, 22, 12, 25] using dependent types, it is possible to dynamically dispatch binary methods since it is easy to check that both receiver and argument depend on the same object. In Jx [21], `x.class` is a dependent type, meaning the run-time type of the object that `x` refers to. For example, the return type of `makeNode()` would be given `this.class`, which means that the run-time type of the receiver. For type safety, the variable before `.class` must be *immutable* as in the following example using Jx syntax:

```
final LinkedNode<Integer> head = ...;
head.class n1 = head.makeNode();
head.class n2 = head.makeNode();
n1.insert(n2);
```

Here, `head` is immutable by the modifier `final`. So, the invocation of `insert()` is legal. However, if `head` were mutable, the type `head.class` would be illegal since an assignment on `head` between the declarations of `n1` and `n2` would be possible, resulting in the unsafe method invocation. In our type system, immutability would not be required even if the language had side-effects.

5.3. Object Creation with Abstract Types

In C#, in parameterized classes, a type parameter can be instantiated with no arguments if it has a constraint `new()`. For example:

```
class C<E> where E : new() {
  void method(){ ... new E(); ... } // allowed
}
```

In the code above, `E`'s object can be created. For type safety, the type argument for `E` is legal only if it has a constructor with no parameters.

This idea can be adapted to the context of `This`, as can be seen in BETA [20], Concord [19] and an early version of LOOJ: if each of a class and its subclasses has a constructor with no parameters, it is safe to create a new object by `new This()` in that class. Our proposal of nonheritable methods can simulate the idea of `new This()` by declaring factory methods with the return type of `@This` such as those in Figure 2. Other differences are: (1) nonheritable methods allow arbitrary code specialized for the declaring class, not only object creation; (2) they give a better control on the duty imposed in subclassing: overriding nonheritable methods in subclasses does not require the overridden methods to be nonheritable so that further subclassing can be free from overriding, whereas object creation with type variables requires *all* classes to override the constructors if constructors are not inherited.

In Jx [21] and J& [22], object creation with dependent types such as `new n.class(...)`, with arguments, is allowed if the type of `n` has a corresponding constructor. In Jx, constructors are inherited to subclasses, unlike Java. If `final` fields are added to a subclass, all the constructors inherited must be overridden in the subclass so that the `final` fields are initialized.

5.4. Abstract Factory Pattern

Even when nonheritable methods are not supported, factory methods and `clone()` can be implemented by using the abstract factory pattern [15, 6], as follows:

```
interface Factory<T> {
  @T create();
}
class LinkedNodeFactory<E> implements Factory<LinkedNode<E>>{
  @LinkedNode<E> create(){
    return new LinkedNode<E>(this);
  }
}
class LinkedNode<E> {
  Factory<This> factory;

  LinkedNode(Factory<This> f){
```

```
    this.factory = f;
  }

  @This makeNode() { return factory.create(); }
}
```

In this programming style, a factory class must be defined for *each* of class `LinkedNode` and its extensions. This is similar to the fact that all subclasses must override nonheritable methods. In a nutshell, nonheritable methods are the trick that allows object creations written in factory classes such as `LinkedNodeFactory` to be put into factory methods such as `makeNode()`.

5.5. Template Specialization

Template specialization in C++ allows us to give a definition for the template instantiated with a certain type. For example, `Vector<boolean>` is defined in isolation from the definition of template `Vector<T>` so as to be space-efficient by using bit operations. It has a similarity with our nonheritable methods in that we can give a definition specialized to a certain instantiation of type variables (in our case, `This`).

5.6. Explicit Self Typing in Scala

In Scala [25], we can give a class an arbitrary self type explicitly. For type safety, it is checked if a class being instantiated in a `new` expression is a subtype of the self type of the class. This mechanism is similar to our nonheritable methods in that the given self type is different from the default. A difference is that the given self type in our proposal is limited to the class name whereas it is arbitrary in Scala. Another difference is that in our proposal the `nonheritable` modifier affects a single method whereas Scala's explicit self typing affects a whole class definition.

5.7. Special Typing Scheme for Methods that Return New Instances

Winter [34] proposes a feature similar to nonheritable methods for Abadi and Cardelli's object calculus **O-3** with self types and matching [1]. In his proposal, there are two kinds of self types: one is **Self**, which corresponds to our `This`, and the other is called **This**.[7] When **This** is used in a method, the method is typed under the condition that **This** and the class type are equal just as in nonheritable methods. So, methods like `clone()` would return **This** rather than **Self**.

An interesting difference is that a method whose signature contains **This** does not have to be overridden in subclasses. A subclass can inherit such a method but, in this case, **This** in its signature is replaced with the object type from the superclass and, as a result, a subclass may lose matching with its superclass. In our proposal, nonheritable methods must be overridden in the subclasses so that the inheritance relation always implies matching, which is called subtyping in our setting (see Section 4.3.2).

5.8. Simulating Nonheritable Methods and `exact` *Statements in Current Java*

There is a simulation technique [7, 31, 27] of `This` by using generics [2] and F-bounded polymorphism [11], with which current Java is equipped. Within this programming style, nonheritable methods can be easily simulated. Moreover, Java has wildcards, a kind of existential types, and *wildcard capture*, corresponding to the operation to open existential types. With them, `exact` statements can be also simulated. First, we rewrite the class definitions in Figure 2, as follows:

```
abstract class LinkedNode<E, Self extends LinkedNode<E,Self>> {
  E elem;
  Self next;
  void insert(Self that){ ... }
  void insertElem(E e){ ... }
  abstract Self makeNode();
  abstract Self getThis();
```

[7] These keywords are typeset in the typewriter font in Winter [34]; we change the font to avoid confusion with our types.

```
}
abstract class DoublyLinkedNode<E, Self extends DoublyLinkedNode<E,Self>>
         extends LinkedNode<E,Self> {
  Self previous;
  void insert(Self that){ ... that.previous = getThis(); ... }
}
```

Each class has one more type parameter `Self`, which simulates `This`. The upper bound of `Self` is the declaring class and covariantly refined when the class is inherited so that `Self` in subclasses refer to the subclasses. The method `makeNode()` is `abstract`—it is impossible to create and return a new object of the declaring class since `Self` is not a supertype of the class names, for example, `LinkedNode<E,Self>` $\not<:$ `Self`. Giving `abstract` method `getThis()` is a technique [7] to give `this` the correct (self) types, in this case `Self`. We give the fixed point classes for these classes, as follows:

```
class LinkedNodeFix<E> extends LinkedNode<E, LinkedNodeFix<E>>{
  LinkedNodeFix<E> makeNode(){ return new LinkedNodeFix<E>(); }
  LinkedNodeFix<E> getThis(){ return this; }
}
class DoublyLinkedNodeFix<E> extends DoublyLinkedNode<E, DoublyLinkedNodeFix<E>>{
  DoublyLinkedNodeFix<E> makeNode(){ return new DoublyLinkedNodeFix<E>(); }
  DoublyLinkedNodeFix<E> getThis(){ return this; }
}
```

Above, the concrete `makeNode()` and `getThis()` are given. Note that these classes are not in the inheritance relation. So, `makeNode()` is not inherited from the former to the latter, as nonheritable methods.

The fixed point classes play the role of object generators. More precisely, class `LinkedNodeFix<E>` corresponds to class `LinkedNode<E>` in Figure 2 and `DoublyLinkedNodeFix<E>` to `DoublyLinkedNode<E>`. In terms of types, `LinkedNodeFix<E>` corresponds to exact type `@LinkedNode<E>` and `DoublyLinkedNodeFix<E>` to `@DoublyLinkedNode<E>`. Wildcard types can correspond to inexact types. `LinkedNode<Integer,?>`, for example, corresponds to `LinkedNode<Integer>`, which represents a common supertype of fixed point classes `LinkedNodeFix< Integer>`, `DoublyLinkedNodeFix<Integer>` and so on. Similarly, wildcard type `DoublyLinkedNode<Integer,?>` corresponds to `DoublyLinkedNode<Integer>`. Note that there is a subtyping relation between these wildcard types, i.e., `DoublyLinkedNode<Integer,?>` is a subtype of `LinkedNode<Integer,?>`.

Now, we simulate `exact` statements, in particular the example at the end of Section 3.1. First, we assume:

```
LinkedNode<Integer,?> head;
```

Then, we need to separate the `exact` statement into a method declaration and its invocation since wildcard capture only works when methods are invoked, as follows:

```
static <X extends LinkedNode<Integer, X>> void body(LinkedNode<Integer,X> tmp){
  X x=tmp.getThis();
  x.next.insert(x);
}

body(head);
```

The method `body()` has a parameter `tmp` whose type is `LinkedNode<Integer,X>`, in which `X` will capture wildcards. Inside the method, `tmp`'s type `LinkedNode<Integer,X>` is coerced to a narrower type `X`, which is an abstraction of `tmp`'s run-time type, through method `getThis()`. The method invocation `body(head)` is well typed, as desired.

6. Conclusion

We propose two mechanisms, namely, `exact` statements and nonheritable methods for the languages with `This` and exact types such as LOOJ. The features remedy the mismatch between `This` and subtyping. As a result, programming relying on dynamic dispatch becomes possible in the presence of `This`.

Although the proposed mechanisms enhance programming with `This`, it is cumbersome to choose correct types, insert `exact` statements, or specify the `nonheritable` modifier in writing a program. For example, in writing class `C`, we have four choices, i.e., `@C`, `C`, `@This`, and `This`, for variables that would have been given type `C` in plain Java. We are developing an inference algorithm to suggest nonheritable annotations and correct types for what seems self-recursive references.

Other future work is to generalize the proposals for the extensions of `This` with grouping mechanisms [30, 18]. The integration with generics has been discussed in the context of self type constructors [29], in which `exact` statements are slightly extended and are formalized in $\mathrm{FGJ}_{\mathsf{stc}}$.

Acknowledgments

Comments from anonymous reviewers help improve the presentation of the present article. We would like to thank members of the Kumiki project for fruitful discussions on this subject. This work was supported in part by Grant-in-Aid for Scientific Research No. 18200001 and Grant-in-Aid for Young Scientists (B) No. 18700026 from MEXT of Japan. Saito is a research fellow of the Japan Society for the Promotion of Science for Young Scientists.

A. Proof of Theorems 1 and 2

We sketch the proof of Theorems 1 and 2. The structure of the proof of subject reduction is similar to those for Featherweight Java and Featherweight GJ [16]. So, we first prove various substitution lemmas with other auxiliary lemmas. The notable change from FJ's proof can be found in Lemma 12 to deal with nonheritable methods.

Lemma 1 (Weakening). *Assume that* $\mathtt{X} \notin dom(\Delta)$ *and* $\mathtt{x} \notin dom(\Gamma)$*, then*

1. *If* $\Delta \vdash \mathtt{S} <: \mathtt{T}$*, then* $\Delta, \mathtt{X} <: \mathtt{H} \vdash \mathtt{S} <: \mathtt{T}$.
2. *If* $\Delta \vdash \mathtt{S}$ ok*, then* $\Delta, \mathtt{X} <: \mathtt{H} \vdash \mathtt{S}$ ok.
3. *If* $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}$*, then* $\Delta, \mathtt{X} <: \mathtt{H}; \Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\Delta; \Gamma, \mathtt{x} : \mathtt{S} \vdash \mathtt{e} : \mathtt{T}$.

Proof. Each can be proved by straightforward induction on the derivation of $\Delta \vdash \mathtt{S} <: \mathtt{T}$, $\Delta \vdash \mathtt{S}$ ok and $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}$, respectively. □

Lemma 2. *Let* $\Delta = \Delta_1, \mathtt{X} <: \mathtt{I}, \Delta_2$*. If* $\Delta \vdash \mathtt{H}$ ok *and* $\Delta_1 \vdash \mathtt{G} <: \mathtt{I}$*, then* $\Delta_1, [\mathtt{G}/\mathtt{X}]\Delta_2 \vdash bound_{\Delta_1, [\mathtt{G}/\mathtt{X}]\Delta_2}([\mathtt{G}/\mathtt{X}]\mathtt{H}) <: bound_{\Delta}(\mathtt{H})$.

Proof. Case analysis on `H`. □

Lemma 3. *If* $\Delta \vdash \mathtt{S} <: \mathtt{T}$ *and* $fields(bound_{\Delta}(\mathtt{T})) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$*, then* $fields(bound_{\Delta}(\mathtt{S})) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}, \overline{\mathtt{U}}\ \overline{\mathtt{g}}$ *for some* $\overline{\mathtt{U}}$ *and* $\overline{\mathtt{g}}$.

Proof. By induction on the derivation of $\Delta \vdash \mathtt{S} <: \mathtt{T}$. □

Lemma 4. *If* $\Delta \vdash \mathtt{S} <: \mathtt{T}$ *and* $mtype(\mathtt{m}, bound_{\Delta}(\mathtt{T})) = \overline{\mathtt{T}} \to \mathtt{T}_0$*, then* $mtype(\mathtt{m}, bound_{\Delta}(\mathtt{S})) = \overline{\mathtt{T}} \to \mathtt{T}_0$.

Proof. By induction on the derivation of $\Delta \vdash \mathtt{S} <: \mathtt{T}$. □

Lemma 5 (Type Substitution Preserves Subtyping). *If* $\Delta_1, \mathtt{X} <: \mathtt{H}, \Delta_2 \vdash \mathtt{S} <: \mathtt{T}$ *and* $\Delta_1 \vdash \mathtt{G}$ ok *and* $\Delta_1 \vdash \mathtt{G} <: \mathtt{H}$*, then* $\Delta_1, [\mathtt{G}/\mathtt{X}]\Delta_2 \vdash [\mathtt{G}/\mathtt{X}]\mathtt{S} <: [\mathtt{G}/\mathtt{X}]\mathtt{T}$.

Proof. By induction on the derivation of $\Delta_1, \mathtt{X} <: \mathtt{H}, \Delta_2 \vdash \mathtt{S} <: \mathtt{T}$. □

Lemma 6 (Type Substitution Preserves Type Well-formedness). *If* $\Delta_1, \mathtt{X} <: \mathtt{H}, \Delta_2 \vdash \mathtt{T}$ ok *and* $\Delta_1 \vdash \mathtt{G}$ ok *and* $\Delta_1 \vdash \mathtt{G} <: \mathtt{H}$*, then* $\Delta_1, [\mathtt{G}/\mathtt{X}]\Delta_2 \vdash [\mathtt{G}/\mathtt{X}]\mathtt{T}$ ok.

Proof. By induction on the derivation of $\Delta_1, \mathtt{X} <: \mathtt{H}, \Delta_2 \vdash \mathtt{T}$ ok. □

Lemma 7. *If* $\Delta \vdash \mathtt{C}$ ok *and* $\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$*, then* $\Delta, \mathtt{This}{<:}\mathtt{C} \vdash \overline{\mathtt{T}}$ ok.

Proof. By induction on the derivation of $\Delta \vdash \mathtt{C}$ ok using Lemma 6. □

Lemma 8. *If* $\Delta \vdash \mathtt{C}$ ok *and* $\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{T}} \to \mathtt{T}_0$*, then* $\Delta, \mathtt{This}{<:}\mathtt{C} \vdash \overline{\mathtt{T}}, \mathtt{T}_0$ ok.

Proof. By induction on the derivation of $\Delta \vdash \mathtt{C}$ ok using Lemma 6. □

Lemma 9 (Well-typed Term is of Well-formed Type). *If* $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}$*, then* $\Delta \vdash \mathtt{T}$ ok.

Proof. By induction on the derivation of $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}$ using Lemmas 6, 7 and 8. □

Lemma 10 (Type Substitution Preserves Typing). *If* $\Delta_1, \mathtt{X}{<:}\mathtt{I}, \Delta_2; \Gamma \vdash \mathtt{e}{:}\mathtt{T}$ *and* $\Delta_1 \vdash \mathtt{H}$ ok *and* $\Delta \vdash \mathtt{H}{<:}\mathtt{I}$*, then* $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2; [\mathtt{H}/\mathtt{X}]\Gamma \vdash \mathtt{e}{:}\mathtt{S}$ *for some* $\mathtt{S}$ *such that* $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2 \vdash \mathtt{S}{<:}[\mathtt{H}/\mathtt{X}]\mathtt{T}$.

Proof. By induction on the structure of $\mathtt{e}$.

Case: $\mathtt{e} = \mathtt{x}$ or $\mathtt{e} = \mathtt{new\ C}(\overline{\mathtt{e}})$

Easy.

Case: $\mathtt{e} = \mathtt{e}_0.\mathtt{f}_i$

Case analysis on the last rule used.

Subcase T-Field-E**:** $\Delta_1, \mathtt{X}{<:}\mathtt{I}, \Delta_2; \Gamma \vdash \mathtt{e}_0{:}@\mathtt{H}_0$ $\quad \mathit{fields}(\mathit{bound}_{\Delta_1, \mathtt{X}<:\mathtt{I}, \Delta_2}(\mathtt{H}_0)) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$
$\mathtt{T} = [\mathtt{H}_0/\mathtt{This}]\mathtt{T}_i$

By the induction hypothesis, $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2; [\mathtt{H}/\mathtt{X}]\Gamma \vdash \mathtt{e}_0{:}\mathtt{S}_0$ for some $\mathtt{S}_0$ such that $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2 \vdash \mathtt{S}_0{<:}[\mathtt{H}/\mathtt{X}]@\mathtt{H}_0$. It is clear that $\mathtt{S}_0 = [\mathtt{H}/\mathtt{X}]@\mathtt{H}_0$ $(= @[\mathtt{H}/\mathtt{X}]\mathtt{H}_0)$. By Lemma 2, we have $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2 \vdash \mathit{bound}_{\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2}([\mathtt{H}/\mathtt{X}]\mathtt{H}_0){<:}\ \mathit{bound}_{\Delta_1, \mathtt{X}<:\mathtt{I}, \Delta_2}(\mathtt{H}_0)$. By Lemma 3 and T-Field-E, $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2; [\mathtt{H}/\mathtt{X}]\Gamma \vdash \mathtt{e}_0.\mathtt{f}_i : [[\mathtt{H}/\mathtt{X}]\mathtt{H}_0/\mathtt{This}]\mathtt{T}_i$. Since $\mathtt{T}_i$ does not contain $\mathtt{X}$, $[[\mathtt{H}/\mathtt{X}]\mathtt{H}_0/\mathtt{This}]\mathtt{T}_i = [\mathtt{H}/\mathtt{X}][\mathtt{H}_0/\mathtt{This}]\mathtt{T}_i$. Letting $\mathtt{S} = [\mathtt{H}/\mathtt{X}][\mathtt{H}_0/\mathtt{This}]\mathtt{T}_i$ finishes the case.

Subcase T-Field-I**:** $\Delta_1, \mathtt{X}{<:}\mathtt{I}, \Delta_2; \Gamma \vdash \mathtt{e}_0{:}\mathtt{H}_0$ $\quad \mathit{fields}(\mathit{bound}_{\Delta_1, \mathtt{X}<:\mathtt{I}, \Delta_2}(\mathtt{H}_0)) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$
$\Delta_1, \mathtt{X}{<:}\mathtt{I}, \Delta_2, \mathtt{This}{<:}\mathtt{H}_0 \vdash \mathtt{T}_i{<:}\mathtt{T}$ $\quad \Delta_1, \mathtt{X}{<:}\mathtt{I}, \Delta_2 \vdash \mathtt{T}$ ok

By the induction hypothesis, $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2; [\mathtt{H}/\mathtt{X}]\Gamma \vdash \mathtt{e}_0{:}\mathtt{S}_0$ for some $\mathtt{S}_0$ such that $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2 \vdash \mathtt{S}_0{<:}[\mathtt{H}/\mathtt{X}]\mathtt{H}_0$. By Lemma 5, we have $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2, \mathtt{This}{<:}[\mathtt{H}/\mathtt{X}]\mathtt{H}_0 \vdash [\mathtt{H}/\mathtt{X}]\mathtt{T}_i{<:}[\mathtt{H}/\mathtt{X}]\mathtt{T}$. By Lemma 2, we have $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2 \vdash \mathit{bound}_{\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2}([\mathtt{H}/\mathtt{X}]\mathtt{H}_0){<:}\ \mathit{bound}_{\Delta_1, \mathtt{X}<:\mathtt{I}, \Delta_2}(\mathtt{H}_0)$. Case analysis on $\mathtt{S}_0$ whether $\mathtt{S}_0 = \mathtt{G}_0$ or $\mathtt{S}_0 = @\mathtt{G}_0$. In both cases, it is easy to show that $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2 \vdash \mathit{bound}_{\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2}(\mathtt{G}_0){<:}\ \mathit{bound}_{\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2}([\mathtt{H}/\mathtt{X}]\mathtt{H}_0)$ and thus $\mathit{fields}(\mathit{bound}_{\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2}(\mathtt{G}_0)) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}, \overline{\mathtt{U}}\ \overline{\mathtt{g}}$ for some $\overline{\mathtt{U}}$ and $\overline{\mathtt{g}}$ by Lemma 3.

Subsubcase: $\mathtt{S}_0 = \mathtt{G}_0$

By Lemmas 1.(3) and 5, $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2, \mathtt{This}{<:}\mathtt{G}_0 \vdash [\mathtt{H}/\mathtt{X}]\mathtt{T}_i{<:}[\mathtt{H}/\mathtt{X}]\mathtt{T}$. By Lemmas 1.(2) and 6, we have $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2 \vdash [\mathtt{H}/\mathtt{X}]\mathtt{T}$ ok. By T-Field-I, letting $\mathtt{S} = [\mathtt{H}/\mathtt{X}]\mathtt{T}$ finishes the case.

Subsubcase: $\mathtt{S}_0 = @\mathtt{G}_0$

By Lemma 5, we have $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2 \vdash [\mathtt{G}_0/\mathtt{This}][\mathtt{H}/\mathtt{X}]\mathtt{T}_i{<:}[\mathtt{G}_0/\mathtt{This}][\mathtt{H}/\mathtt{X}]\mathtt{T}$. Note that $[\mathtt{G}_0/\mathtt{This}][\mathtt{H}/\mathtt{X}]\mathtt{T} = [\mathtt{H}/\mathtt{X}]\mathtt{T}$. By T-Field-E, letting $\mathtt{S} = [\mathtt{G}_0/\mathtt{This}][\mathtt{H}/\mathtt{X}]\mathtt{T}_i$ finishes the case.

Case: $\mathtt{e} = \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})$

Similar to the case $\mathtt{e} = \mathtt{e}_0.\mathtt{f}_i$ by using Lemma 4.

Case: $\mathtt{e} = \mathtt{exact\ e}_1\ \mathtt{as\ y,\ Y\ in\ e}_0$

There are two cases depending on the last rules to derive $\Delta_1, \mathtt{X}{<:}\mathtt{H}, \Delta_2; \Gamma \vdash \mathtt{e}{:}\mathtt{T}$. Here, we show only the case of T-Exact-I since the other of T-Exact-E is easy. By T-Exact-I, we have $\Delta_1, \mathtt{X}{<:}\mathtt{I}, \Delta_2; \Gamma \vdash \mathtt{e}_1{:}\mathtt{H}_1$ and $\Delta_1, \mathtt{X}{<:}\mathtt{I}, \Delta_2, \mathtt{Y}{<:}\mathtt{H}_1; \Gamma, \mathtt{y} : @\mathtt{Y} \vdash \mathtt{e}_0{:}\mathtt{T}_0$ and $\Delta_1, \mathtt{X}{<:}\mathtt{I}, \Delta_2, \mathtt{Y}{<:}\mathtt{H}_1 \vdash \mathtt{T}_0{<:}\mathtt{U}_0$ and $\Delta_1, \mathtt{X}{<:}\mathtt{I}, \Delta_2 \vdash \mathtt{U}_0$ ok and $\mathtt{T} = \mathtt{U}_0$. By the induction hypothesis, we have that $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2; [\mathtt{H}/\mathtt{X}]\Gamma \vdash \mathtt{e}_1{:}\mathtt{S}_1$ for some $\mathtt{S}_1$ such that $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2 \vdash \mathtt{S}_1{<:}[\mathtt{H}/\mathtt{X}]\mathtt{H}_1$, and that $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2, \mathtt{Y}{<:}[\mathtt{H}/\mathtt{X}]\mathtt{H}_1; [\mathtt{H}/\mathtt{X}]\Gamma, \mathtt{y}{:}@\mathtt{Y} \vdash \mathtt{e}_0{:}\mathtt{T}_0'$ for some $\mathtt{T}_0'$ such that $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2, \mathtt{Y}{<:}[\mathtt{H}/\mathtt{X}]\mathtt{H}_1 \vdash \mathtt{T}_0'{<:}[\mathtt{H}/\mathtt{X}]\mathtt{T}_0$. Case analysis on whether $\mathtt{S}_1$ is inexact or exact.

Subcase: $\mathtt{S}_1 = \mathtt{G}_1$

By Lemma 1.(3) and the induction hypothesis, $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2, \mathtt{Y}{<:}\mathtt{S}_1; [\mathtt{H}/\mathtt{X}]\Gamma,\ \mathtt{y}:@\mathtt{Y} \vdash \mathtt{e}_0 : \mathtt{T}_0''$ for some $\mathtt{T}_0''$ such that $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2,\ \mathtt{Y}{<:}\mathtt{S}_1 \vdash \mathtt{T}_0''{<:}\mathtt{T}_0'$. By Lemmas 1.(1) and 5, $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2,\ \mathtt{Y}{<:}\mathtt{S}_1 \vdash \mathtt{T}_0'{<:}[\mathtt{H}/\mathtt{X}]\mathtt{T}_0$ and $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2,\ \mathtt{Y}{<:}\mathtt{S}_1 \vdash [\mathtt{H}/\mathtt{X}]\mathtt{T}_0{<:}[\mathtt{H}/\mathtt{X}]\mathtt{U}_0$. By S-Trans, $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2,\ \mathtt{Y}{<:}\mathtt{S}_1 \vdash \mathtt{T}_0''{<:}[\mathtt{H}/\mathtt{X}]\mathtt{U}_0$. By T-Exact-I, $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2; [\mathtt{H}/\mathtt{X}]\Gamma \vdash \mathtt{exact}\ \mathtt{e}_1\ \mathtt{as}\ \mathtt{y},\ \mathtt{Y}\ \mathtt{in}\ \mathtt{e}_0 : [\mathtt{H}/\mathtt{X}]\mathtt{U}_0$. By Lemma 6, $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2 \vdash [\mathtt{H}/\mathtt{X}]\mathtt{U}_0$ ok. Letting $\mathtt{S} = [\mathtt{H}/\mathtt{X}]\mathtt{U}_0$ finishes the case.

Subcase: $\mathtt{S}_1 = @\mathtt{G}_1$

By the induction hypothesis, $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2; [\mathtt{H}/\mathtt{X}]\Gamma,\ \mathtt{y}:@\mathtt{G}_1 \vdash \mathtt{e}_0 : \mathtt{T}_0''$ for some $\mathtt{T}_0''$ such that $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2 \vdash \mathtt{T}_0''{<:}[\mathtt{G}_1/\mathtt{Y}]\mathtt{T}_0'$. By Lemma 5, $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2, \vdash [\mathtt{G}_1/\mathtt{Y}]\mathtt{T}_0'{<:}[\mathtt{G}_1/\mathtt{Y}][\mathtt{H}/\mathtt{X}]\mathtt{T}_0$ and $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2, \vdash [\mathtt{G}_1/\mathtt{Y}][\mathtt{H}/\mathtt{X}]\mathtt{T}_0{<:}[\mathtt{G}_1/\mathtt{Y}][\mathtt{H}/\mathtt{X}]\mathtt{U}_0$. Since $[\mathtt{G}_1/\mathtt{Y}][\mathtt{H}/\mathtt{X}]\mathtt{U}_0 = [\mathtt{H}/\mathtt{X}]\mathtt{U}_0$, by S-Trans $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2, \vdash \mathtt{T}_0''{<:}[\mathtt{H}/\mathtt{X}]\mathtt{U}_0$. By T-Exact-E, $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2; [\mathtt{H}/\mathtt{X}]\Gamma \vdash \mathtt{exact}\ \mathtt{e}_1\ \mathtt{as}\ \mathtt{y},\ \mathtt{Y}\ \mathtt{in}\ \mathtt{e}_0 : \mathtt{T}_0''$. Letting $\mathtt{S} = \mathtt{T}_0''$ finishes the case. □

Lemma 11 (Term Substitution Preserves Typing). *If* $\Delta; \Gamma, \overline{\mathtt{x}}:\overline{\mathtt{T}} \vdash \mathtt{e}:\mathtt{T}$ *and* $\Delta; \Gamma \vdash \overline{\mathtt{d}}:\overline{\mathtt{S}}$ *and* $\Delta \vdash \overline{\mathtt{S}}{<:}\overline{\mathtt{T}}$*, then* $\Delta; \Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}:\mathtt{S}$ *for some* $\mathtt{S}$ *such that* $\Delta \vdash \mathtt{S}{<:}\mathtt{T}$.

Proof. By induction on the structure of $\mathtt{e}$.

Case: $\mathtt{e} = \mathtt{x}$ or $\mathtt{e} = \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})$

Easy.

Case: $\mathtt{e} = \mathtt{e}_0.\mathtt{f}_i$

Case analysis on the last rule used.

Subcase T-Field-E**:** $\Delta; \Gamma, \overline{\mathtt{x}}:\overline{\mathtt{T}} \vdash \mathtt{e}_0 : @\mathtt{H}_0$ $\quad \mathit{fields}(\mathit{bound}_\Delta(\mathtt{H}_0)) = \overline{\mathtt{U}}\ \overline{\mathtt{f}}$ $\quad \mathtt{T} = [\mathtt{H}_0/\mathtt{This}]\mathtt{U}_i$.

By the induction hypothesis, we have $\Delta; \Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0 : \mathtt{S}_0$ for some $\mathtt{S}_0$ such that $\Delta \vdash \mathtt{S}_0{<:}@\mathtt{H}_0$. It is easy to show $\mathtt{S}_0 = @\mathtt{H}_0$. By T-Field-E, $\Delta; \Gamma \vdash ([\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0).\mathtt{f}_i : [\mathtt{H}_0/\mathtt{This}]\mathtt{U}_i$. Letting $\mathtt{S} = \mathtt{T}$ finishes the case.

Case T-Field-I**:** $\Delta; \Gamma, \overline{\mathtt{x}}:\overline{\mathtt{T}} \vdash \mathtt{e}_0 : \mathtt{H}_0$ $\quad \mathit{fields}(\mathit{bound}_\Delta(\mathtt{H}_0)) = \overline{\mathtt{U}}\ \overline{\mathtt{f}}$
$\Delta, \mathtt{This}{<:}\mathtt{H}_0 \vdash \mathtt{U}_i{<:}\mathtt{T}$ $\quad \Delta \vdash \mathtt{T}$ ok

By the induction hypothesis, we have $\Delta; \Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0 : \mathtt{S}_0$ for some $\mathtt{S}_0$ such that $\Delta \vdash \mathtt{S}_0{<:}\mathtt{H}_0$. Case analysis on whether $\mathtt{S}_0 = \mathtt{G}_0$ or $\mathtt{S}_0 = @\mathtt{G}_0$. In both cases, it is easy show that $\mathtt{U}_i\ \ \mathtt{f}_i \in \mathit{fields}(\mathit{bound}_\Delta(\mathtt{G}_0))$ by Lemma 3.

Case: $\mathtt{S}_0 = \mathtt{G}_0$

By Lemmas 1.(1) and 5, $\Delta, \mathtt{This}{<:}\mathtt{G}_0 \vdash \mathtt{U}_i{<:}\mathtt{T}$. By T-Exact-I, letting $\mathtt{S} = \mathtt{T}$ finishes the case.

Case: $\mathtt{S}_0 = @\mathtt{G}_0$

By Lemma 5, $\Delta \vdash [\mathtt{G}_0/\mathtt{This}]{<:}\mathtt{T}$. By T-Exact-E, letting $\mathtt{S} = [\mathtt{G}_0/\mathtt{This}]\mathtt{U}_i$ finishes the case.

Case: $\mathtt{e} = \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})$

Similar to the case where $\mathtt{e} = \mathtt{e}_0.\mathtt{f}_i$.

Case: $\mathtt{e} = \mathtt{exact}\ \mathtt{e}_1\ \mathtt{as}\ \mathtt{y},\ \mathtt{Y}\ \mathtt{in}\ \mathtt{e}_0$

There are two cases depending on the last rules to derive $\Delta; \Gamma \vdash \mathtt{e}:\mathtt{T}$. Here, we show only the case of T-Exact-I since the other of T-Exact-E is easy. By T-Exact-I, we have $\Delta; \Gamma, \overline{\mathtt{x}}:\overline{\mathtt{T}} \vdash \mathtt{e}_1 : \mathtt{H}_1$ and $\Delta, \mathtt{Y}{<:}\mathtt{H}_1; \Gamma, \overline{\mathtt{x}}:\overline{\mathtt{T}}, \mathtt{y}:@\mathtt{Y} \vdash \mathtt{e}_0 : \mathtt{T}_0$ and $\Delta, \mathtt{Y}{<:}\mathtt{H}_1 \vdash \mathtt{T}_0{<:}\mathtt{U}_0$ and $\Delta \vdash \mathtt{U}_0$ ok and $\mathtt{T} = \mathtt{U}_0$. By the induction hypothesis, we have $\Delta; \Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_1 : \mathtt{S}_1$ for some $\mathtt{S}_1$ such that $\Delta \vdash \mathtt{S}_1{<:}\mathtt{H}_1$, and $\Delta, \mathtt{Y}{<:}\mathtt{H}_1; \Gamma, \mathtt{y}:@\mathtt{Y} \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0 : \mathtt{T}_0'$ for some $\mathtt{T}_0'$ such that $\Delta, \mathtt{Y}{<:}\mathtt{H}_1 \vdash \mathtt{T}_0'{<:}\mathtt{T}_0$. By S-Trans, $\Delta, \mathtt{Y}{<:}\mathtt{H}_1 \vdash \mathtt{T}_0'{<:}\mathtt{U}_0$. Case analysis on whether $\mathtt{S}_1$ is inexact or exact.

Subcase: $\mathtt{S}_1 = \mathtt{G}_1$

By Lemmas 1.(3) and 10, $\Delta, \mathtt{Y}{<:}\mathtt{G}_1; \Gamma, \mathtt{y} : @\mathtt{Y} \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0 : \mathtt{T}_0''$ for some $\mathtt{T}_0''$ such that $\Delta, \mathtt{Y}{<:}\mathtt{G}_1 \vdash \mathtt{T}_0''{<:}\mathtt{T}_0'$. By Lemmas 1.(1) and 5, $\Delta, \mathtt{Y}{<:}\mathtt{G}_1 \vdash \mathtt{T}_0'{<:}\mathtt{U}_0$. By S-Trans, $\Delta, \mathtt{Y}{<:}\mathtt{G}_1 \vdash \mathtt{T}_0''{<:}\mathtt{U}_0$. By T-Exact-I, $\Delta; \Gamma \vdash \mathtt{exact}\ [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_1\ \mathtt{as}\ \mathtt{y},\ \mathtt{Y}\ \mathtt{in}\ [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0 : \mathtt{U}_0$. Letting $\mathtt{S} = \mathtt{U}_0$ finishes the case.

Subcase: $\mathtt{S}_1 = \mathtt{@G}_1$

By Lemma 10, $\Delta; \Gamma, \mathtt{y} : \mathtt{@G}_1 \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0 : \mathtt{T}_0''$ for some $\mathtt{T}_0''$ such that $\Delta \vdash \mathtt{T}_0'' \mathtt{<:} [\mathtt{G}_1/\mathtt{Y}]\mathtt{T}_0'$. By Lemma 5, $\Delta \vdash [\mathtt{G}_1/\mathtt{Y}]\mathtt{T}_0' \mathtt{<:} [\mathtt{G}_1/\mathtt{Y}]\mathtt{U}_0$. Since $[\mathtt{G}_1/\mathtt{Y}]\mathtt{U}_0 = \mathtt{U}_0$, by S-TRANS, $\Delta \vdash \mathtt{T}_0'' \mathtt{<:} \mathtt{U}_0$. By T-EXACT-E, $\Delta; \Gamma \vdash \mathtt{exact}\ [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_1\ \mathtt{as\ y,\ Y\ in}\ [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0 : \mathtt{T}_0''$. Letting $\mathtt{S} = \mathtt{T}_0''$ finishes the case.

□

Lemma 12. *If* $mbody(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{x}}.\mathtt{e}_0$ *and* $mtype(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{T}} \to \mathtt{T}_0$ *where* $\Delta \vdash \mathtt{C}$ ok*, then either*

1. *there exist some* $\mathtt{D}$ *and* $\mathtt{U}_0$ *such that* $\emptyset \vdash \mathtt{C} \mathtt{<:} \mathtt{D}$ *and* $\mathtt{This} \mathtt{<:} \mathtt{D} \vdash \mathtt{D}, \mathtt{U}_0$ ok *and* $\mathtt{This} \mathtt{<:} \mathtt{D} \vdash \mathtt{U}_0 \mathtt{<:} \mathtt{T}_0$ *and* $\mathtt{This} \mathtt{<:} \mathtt{D};\ \overline{\mathtt{x}} : \overline{\mathtt{T}},\ \mathtt{this} : \mathtt{@This} \vdash \mathtt{e}_0 : \mathtt{U}_0$*, or*
2. $\mathtt{class\ C\ extends\ D}\{\mathtt{..}\overline{\mathtt{M}}\}$ *and* $\mathtt{nonheritable\ T}_0\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\mathtt{..}\} \in \overline{\mathtt{M}}$ *and* $\emptyset; \overline{\mathtt{x}} : [\mathtt{C}/\mathtt{This}]\overline{\mathtt{T}},\ \mathtt{this} : \mathtt{@C} \vdash \mathtt{e}_0 : \mathtt{U}_0$ *and* $\emptyset \vdash \mathtt{U}_0 \mathtt{<:} [\mathtt{C}/\mathtt{This}]\mathtt{T}_0$.

Proof. By induction on the derivation of $mbody(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{x}}.\mathtt{e}_0$.

Case MB-CLASS**:**

Case analysis on whether the method is nonheritable or not.

Subcase: $\mathtt{class\ C\ extends\ D}\{\mathtt{..}\overline{\mathtt{M}}\} \qquad \mathtt{T}_0\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e}_0\mathtt{;}\} \in \overline{\mathtt{M}}$

By T-METHOD, $\mathtt{This} \mathtt{<:} \mathtt{C}; \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{@This} \vdash \mathtt{e}_0 : \mathtt{S}_0$ for some $\mathtt{S}_0$ such that $\mathtt{This} \mathtt{<:} \mathtt{C} \vdash \mathtt{S}_0 \mathtt{<:} \mathtt{T}_0$. Letting $\mathtt{U}_0 = \mathtt{S}_0$ and $\mathtt{D} = \mathtt{C}$ finishes the case.

Subcase: $\mathtt{class\ C\ extends\ D}\{\mathtt{..}\overline{\mathtt{M}}\} \qquad \mathtt{nonheritable\ T}_0\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e}_0\mathtt{;}\} \in \overline{\mathtt{M}}$

By T-NHMETHOD, $\emptyset; \overline{\mathtt{x}} : [\mathtt{C}/\mathtt{This}]\overline{\mathtt{T}}, \mathtt{this} : \mathtt{@C} \vdash \mathtt{e}_0 : \mathtt{S}_0$ for some $\mathtt{S}_0$ such that $\emptyset \vdash \mathtt{S}_0 \mathtt{<:} [\mathtt{C}/\mathtt{This}]\mathtt{T}_0$. Letting $\mathtt{U}_0 = \mathtt{S}_0$ finishes the case.

Case MB-SUPER**:** $\mathtt{class\ C\ extends\ D}\{\mathtt{..}\overline{\mathtt{M}}\} \qquad \mathtt{m} \notin \overline{\mathtt{M}} \qquad mbody(\mathtt{m}, \mathtt{D}) = \overline{\mathtt{x}}.\mathtt{e}_0$

By the induction hypothesis, we have the following two cases:

Subcase: $\mathtt{This} \mathtt{<:} \mathtt{E} \vdash \mathtt{D} \mathtt{<:} \mathtt{E}, \mathtt{S}_0 \mathtt{<:} \mathtt{T}_0 \qquad \mathtt{This} \mathtt{<:} \mathtt{E} \vdash \mathtt{E}, \mathtt{S}_0$ ok $\qquad \mathtt{This} \mathtt{<:} \mathtt{E}; \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{@This} \vdash \mathtt{e}_0 : \mathtt{S}_0$

By Lemmas 1.(3) and 10, $\mathtt{This} \mathtt{<:} \mathtt{D}; \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{@This} \vdash \mathtt{e}_0 : \mathtt{S}_0'$ for some $\mathtt{S}_0'$ such that $\mathtt{This} \mathtt{<:} \mathtt{D} \vdash \mathtt{S}_0' \mathtt{<:} \mathtt{S}_0$. Letting $\mathtt{U}_0 = \mathtt{S}_0'$ finishes the case.

Subcase: $\mathtt{class\ D\ extends\ E}\{\mathtt{..}\overline{\mathtt{M}}'\} \qquad \mathtt{nonheritable\ T}_0\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\mathtt{..}\} \in \overline{\mathtt{M}}'$

Cannot happen since $\mathtt{m} \notin \overline{\mathtt{M}}$ contradicts $\mathtt{m} \in \overline{\mathtt{M}}$ in the derivation of $\vdash \mathtt{class\ C\ extends\ D}\{\mathtt{..}\overline{\mathtt{M}}\}$ ok by T-CLASS. □

A.1. Proof of Theorem 1

By induction on the derivation of $\mathtt{e} \longrightarrow \mathtt{e}'$ with a case analysis on the reduction rule used.

Case R-FIELD**:** $\mathtt{e} = \mathtt{new\ C}(\overline{\mathtt{e}}).\mathtt{f}_i \qquad fields(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \mathtt{e}' = \mathtt{e}_i$

By the rules T-FIELD-E and T-NEW, we have

$$\Delta; \Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{e}}) : \mathtt{@C} \qquad \Delta; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \qquad \Delta \vdash \overline{\mathtt{U}} \mathtt{<:} [\mathtt{C}/\mathtt{This}]\overline{\mathtt{T}} \qquad \mathtt{T} = [\mathtt{C}/\mathtt{This}]\mathtt{T}_i$$

In particular, $\Delta; \Gamma \vdash \mathtt{e}_i : \mathtt{U}_i$. Letting $\mathtt{T}' = \mathtt{U}_i$ finishes the case.

Case R-INVK**:** $\mathtt{e} = \mathtt{new\ C}(\overline{\mathtt{e}}).\mathtt{m}(\overline{\mathtt{d}}) \qquad mbody(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{x}}.\mathtt{e}_0 \qquad \mathtt{e}' = [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ C}(\overline{\mathtt{e}})/\mathtt{this}]\mathtt{e}_0$

By the rules T-INVK-E and T-NEW, we have

$$\begin{array}{lll} \Delta; \Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{e}}) : \mathtt{@C} & mtype(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{U}} \to \mathtt{U}_0 & \Delta; \Gamma \vdash \overline{\mathtt{d}} : \overline{\mathtt{S}} \\ \Delta \vdash \overline{\mathtt{S}} \mathtt{<:} [\mathtt{C}/\mathtt{This}]\overline{\mathtt{U}} & \mathtt{T} = [\mathtt{C}/\mathtt{This}]\mathtt{U}_0 & \Delta \vdash \mathtt{C}\ \text{ok} \end{array}$$

By Lemma 12, we have either

1. $\mathtt{This} \mathtt{<:} \mathtt{D}; \overline{\mathtt{x}} : \overline{\mathtt{U}}, \mathtt{this} : \mathtt{@This} \vdash \mathtt{e}_0 : \mathtt{S}_0$ such that $\mathtt{This} \mathtt{<:} \mathtt{D} \vdash \mathtt{C} \mathtt{<:} \mathtt{D}, \mathtt{S}_0 \mathtt{<:} \mathtt{U}_0$ and $\mathtt{This} \mathtt{<:} \mathtt{D} \vdash \mathtt{D}, \mathtt{S}_0$ ok. Without loss of generality, we can assume that the domain of Δ does not contain $\mathtt{This}$ and that the domain of Γ and the set of the variables $\overline{\mathtt{x}}$ and $\mathtt{this}$ are disjoint. By Lemmas 1.(3), 10 and 11, $\Delta; \Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ C}(\overline{\mathtt{e}})/\mathtt{this}]\ \mathtt{e}_0 : \mathtt{S}_0'$ for some $\mathtt{S}_0'$ such that $\Delta \vdash \mathtt{S}_0' \mathtt{<:} [\mathtt{C}/\mathtt{This}]\mathtt{S}_0$. By Lemmas 1.(1) and 5, $\Delta \vdash [\mathtt{C}/\mathtt{This}]\mathtt{S}_0 \mathtt{<:} [\mathtt{C}/\mathtt{This}]\mathtt{U}_0$. By S-TRANS, $\Delta \vdash \mathtt{S}_0' \mathtt{<:} [\mathtt{C}/\mathtt{This}]\mathtt{U}_0$. Letting $\mathtt{T}' = \mathtt{S}_0'$ finishes the case.

2. there exists $\mathtt{S}_0$ such that $\emptyset; \overline{\mathtt{x}} : [\mathtt{C}/\mathtt{This}]\overline{\mathtt{U}}, \mathtt{this}:@\mathtt{C} \vdash \mathtt{e}_0 : \mathtt{S}_0$ and $\emptyset \vdash \mathtt{S}_0 <: [\mathtt{C}/\mathtt{This}]\mathtt{U}_0$. Without loss of generality, we can assume that the domain of Γ and the set of the variables $\overline{\mathtt{x}}$ and $\mathtt{this}$ are disjoint. By Lemmas 1.(3) and 11, $\Delta; \Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ C}(\overline{\mathtt{e}})/\mathtt{this}]\mathtt{e}_0 : \mathtt{S}_0'$ for some $\mathtt{S}_0'$ such that $\Delta \vdash \mathtt{S}_0' <: \mathtt{S}_0$. By S-Trans, $\Delta \vdash \mathtt{S}_0' <: [\mathtt{C}/\mathtt{This}]\mathtt{U}_0$. Letting $\mathtt{T}' = \mathtt{S}_0'$ finishes the case.

Case R-Exact**:** $\mathtt{e} = \mathtt{exact\ new\ C}(\overline{\mathtt{e}})\ \mathtt{as\ x,\ X\ in\ e}_0 \qquad \mathtt{e}' = [\mathtt{new\ C}(\overline{\mathtt{e}})/\mathtt{x}]\mathtt{e}_0$

By the rules T-Exact-E and T-New, we have

$$\Delta; \Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{e}}) : @\mathtt{C} \qquad \Delta \vdash @\mathtt{C}\ \mathrm{ok} \qquad \Delta; \Gamma, \mathtt{x} : @\mathtt{C} \vdash \mathtt{e}_0 : \mathtt{U}_0 \qquad \mathtt{T} = \mathtt{U}_0$$

By Lemma 11, $\Delta; \Gamma \vdash \mathtt{e}' : \mathtt{U}_0'$ such that $\Delta \vdash \mathtt{U}_0' <: \mathtt{U}_0$. Letting $\mathtt{T}' = \mathtt{U}_0'$ finishes the case.

Case RC-Field**:** $\mathtt{e} = \mathtt{e}_0.\mathtt{f} \qquad \mathtt{e}' = \mathtt{e}_0'.\mathtt{f} \qquad \mathtt{e}_0 \longrightarrow \mathtt{e}_0'$

Case analysis on the typing rule used.

Subcase T-Field-E**:** $\Delta; \Gamma \vdash \mathtt{e}_0 : @\mathtt{H}_0 \qquad \mathit{fields}(\mathit{bound}_\Delta(\mathtt{H}_0)) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \mathtt{T} = [\mathtt{H}_0/\mathtt{This}]\mathtt{T}_i$

By the induction hypothesis, $\Delta; \Gamma \vdash \mathtt{e}_0' : \mathtt{T}_0'$ for some $\mathtt{T}_0'$ such that $\Delta \vdash \mathtt{T}_0' <: @\mathtt{H}_0$. Since $\mathtt{T}_0' = @\mathtt{H}_0$, $\Delta; \Gamma \vdash \mathtt{e}_0'.\mathtt{f} : [\mathtt{H}_0/\mathtt{This}]\mathtt{T}_i$ by T-Field-E. Letting $\mathtt{T}' = [\mathtt{H}_0/\mathtt{This}]\mathtt{T}_i$ finishes the case.

Subcase T-Field-I**:** $\Delta; \Gamma \vdash \mathtt{e}_0 : \mathtt{H}_0 \qquad \mathit{fields}(\mathit{bound}_\Delta(\mathtt{H}_0)) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$
$\Delta, \mathtt{This} <: \mathtt{H}_0 \vdash \mathtt{T}_i <: \mathtt{T} \qquad \Delta \vdash \mathtt{T}\ \mathrm{ok}$

By the induction hypothesis, $\Delta; \Gamma \vdash \mathtt{e}_0' : \mathtt{T}_0'$ for some $\mathtt{T}_0'$ such that $\Delta \vdash \mathtt{T}_0' <: \mathtt{H}_0$. Case analysis on $\mathtt{T}_0'$.

Subsubcase: $\mathtt{T}_0' = \mathtt{G}_0$

By Lemmas 1.(3) and 5, we have $\Delta, \mathtt{This} <: \mathtt{G}_0 \vdash \mathtt{T}_i <: \mathtt{T}$. By T-Field-I, we have $\Delta; \Gamma \vdash \mathtt{e}_0.\mathtt{f} : \mathtt{T}$. Letting $\mathtt{T}' = \mathtt{T}$ finishes the case.

Subsubcase: $\mathtt{T}_0' = @\mathtt{G}_0$

By Lemma 3 and T-Field-E, we have $\Delta; \Gamma \vdash \mathtt{e}_0'.\mathtt{f} : [\mathtt{G}_0/\mathtt{This}]\mathtt{T}_i$. By Lemma 5 and the fact that $\mathtt{T}$ is $\mathtt{This}$-free, we have $\Delta; \Gamma \vdash [\mathtt{G}_0/\mathtt{This}]\mathtt{T}_i <: \mathtt{T}$. Letting $\mathtt{T}' = \mathtt{T}$ finishes the case.

Case RC-Invk-Recv**:**

Similar to the case RC-Field.

Case RC-Exact**:** $\mathtt{e} = \mathtt{exact\ e}_0\ \mathtt{as\ x,\ X\ in\ e}_1$
$\mathtt{e}' = \mathtt{exact\ e}_0'\ \mathtt{as\ x,\ X\ in\ e}_1$
$\mathtt{e}_0 \longrightarrow \mathtt{e}_0'$

Case analysis on the typing rule used.

Subcase T-Exact-I**:** $\Delta; \Gamma \vdash \mathtt{e}_0 : \mathtt{H}_0 \qquad \Delta, \mathtt{X} <: \mathtt{H}_0; \Gamma, \mathtt{x} : @\mathtt{X} \vdash \mathtt{e}_1 : \mathtt{U}_1$
$\Delta, \mathtt{X} <: \mathtt{H}_0 \vdash \mathtt{U}_1 <: \mathtt{S}_1 \qquad \Delta \vdash \mathtt{S}_1\ \mathrm{ok} \qquad \mathtt{T} = \mathtt{S}_1$

By the induction hypothesis, we have $\Delta; \Gamma \vdash \mathtt{e}_0' : \mathtt{T}_0$ for some $\mathtt{T}_0$ such that $\Delta \vdash \mathtt{T}_0 <: \mathtt{H}_0$. Case analysis on whether $\mathtt{T}_0$ is inexact or exact.

Subsubcase: $\mathtt{T}_0 = \mathtt{G}_0$

By Lemmas 1.(3) and 10, $\Delta, \mathtt{X} <: \mathtt{G}_0; \Gamma, \mathtt{x} : @\mathtt{X} \vdash \mathtt{e}_1 : \mathtt{U}_1'$ for some $\mathtt{U}_1'$ such that $\Delta, \mathtt{X} <: \mathtt{G}_0 \vdash \mathtt{U}_1' <: \mathtt{U}_1$. By Lemmas 1.(1) and 5, $\Delta, \mathtt{X} <: \mathtt{G}_0 \vdash \mathtt{U}_1 <: \mathtt{S}_1$. By S-Trans, $\Delta, \mathtt{X} <: \mathtt{G}_0 \vdash \mathtt{U}_1' <: \mathtt{S}_1$. By T-Exact-I, $\Delta; \Gamma \vdash \mathtt{exact\ e}_0'\ \mathtt{as\ x,\ X\ in\ e}_1 : \mathtt{S}_1$. Letting $\mathtt{T}' = \mathtt{S}_1$ finishes the case.

Subsubcase: $\mathtt{T}_0 = @\mathtt{G}_0$

By Lemma 10, $\Delta; \Gamma, \mathtt{x} : @\mathtt{G}_0 \vdash \mathtt{e}_1 : \mathtt{U}_1'$ for some $\mathtt{U}_1'$ such that $\Delta \vdash \mathtt{U}_1' <: [\mathtt{G}_0/\mathtt{X}]\mathtt{U}_1$. By T-Exact-E, $\Delta; \Gamma \vdash \mathtt{exact\ e}_0'\ \mathtt{as\ x,\ X\ in\ e}_1 : \mathtt{U}_1'$. By Lemma 5, $\Delta \vdash [\mathtt{G}_0/\mathtt{X}]\mathtt{U}_1 <: [\mathtt{G}_0/\mathtt{X}]\mathtt{S}_1 (= \mathtt{S}_1)$. By S-Trans $\Delta \vdash \mathtt{U}_1' <: \mathtt{S}_1$. Letting $\mathtt{T}' = \mathtt{U}_1'$ finishes the case.

Subcase T-Exact-E**:**

Easy.

Case RC-Invk-Arg, RC-New-Arg, RC-Exact-Body**:**

Easy.

A.2. Proof of Theorem 2

By induction on the derivation of $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{T}$ with a case analysis on the last rule used.

Case T-VAR**:**

Cannot happen.

Case T-FIELD-E**:** $\mathtt{e} = \mathtt{e}_0.\mathtt{f}_i \qquad \emptyset;\emptyset \vdash \mathtt{e}_0 : \mathtt{@C}_0 \qquad \mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$

If $\mathtt{e}_0 = \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{e}})$, then $\mathtt{e}_0 \longrightarrow \mathtt{e}_i$ by R-FIELD. Otherwise, immediate from the induction hypothesis and RC-FIELD.

Case T-FIELD-I**:** $\mathtt{e} = \mathtt{e}_0.\mathtt{f}_i \qquad \emptyset;\emptyset \vdash \mathtt{e}_0 : \mathtt{C}_0$

Immediate from the induction hypothesis and RC-FIELD since $\mathtt{e}_0$ is not a `new` expression.

Case T-INVK-E**:** $\mathtt{e} = \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})$

If there is a non-value subexpression, then the conclusion is immediate from the induction hypothesis and the rules RC-INV-RECV and RC-INV-ARG. Suppose that $\mathtt{e}_0$ and $\overline{\mathtt{e}}$ are values $\mathtt{v}_0$ and $\overline{\mathtt{v}}$, respectively. Then, by T-INVK-E,

$$\frac{\emptyset;\emptyset \vdash \mathtt{v}_0 : \mathtt{@C}_0 \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{U}} \to \mathtt{U}_0 \qquad \emptyset;\emptyset \vdash \overline{\mathtt{v}} : \overline{\mathtt{C}} \qquad \emptyset \vdash \overline{\mathtt{C}} <: [\mathtt{C}_0/\mathtt{This}]\overline{\mathtt{U}}}{\emptyset;\emptyset \vdash \mathtt{v}_0.\mathtt{m}(\overline{\mathtt{v}}) : [\mathtt{C}_0/\mathtt{This}]\mathtt{U}_0}$$

From premises of the rule, the lengths of $\overline{\mathtt{v}}$, $\overline{\mathtt{C}}$ and $\overline{\mathtt{U}}$ are the same. Since $\mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{U}} \to \mathtt{U}_0$, it is easy to show that $\mathit{mbody}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{x}}.\mathtt{e}_0{}'$ for some $\overline{\mathtt{x}}$ and $\mathtt{e}_0{}'$, where the lengths of $\overline{\mathtt{U}}$ and $\overline{\mathtt{x}}$ are the same. Then, by R-INVK, $\mathtt{v}_0.\mathtt{m}(\overline{\mathtt{v}}) \longrightarrow [\overline{\mathtt{v}}/\overline{\mathtt{x}}, \mathtt{v}_0/\mathtt{this}]\mathtt{e}_0{}'$.

Case T-INVK-I**:**

Similar to the case T-FIELD-I.

Case T-NEW**:** $\mathtt{e} = \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})$

If one of $\overline{\mathtt{e}}$ is not a value, apply the induction hypothesis with the rule RC-NEW-ARG. Otherwise e is a value.

Case T-EXACT-I**:** $\mathtt{e} = \mathtt{exact}\ \mathtt{e}_0\ \mathtt{as}\ \mathtt{x},\ \mathtt{X}\ \mathtt{in}\ \mathtt{e}_1 \qquad \emptyset;\emptyset \vdash \mathtt{e}_0 : \mathtt{C}_0$

Since $\mathtt{e}_0$ is of an inexact type, $\mathtt{e}_0$ is not a value. So, the conclusion is immediate from the induction hypothesis and the rule RC-EXACT.

Case T-EXACT-E**:** $\mathtt{e} = \mathtt{exact}\ \mathtt{e}_0\ \mathtt{as}\ \mathtt{x},\ \mathtt{X}\ \mathtt{in}\ \mathtt{e}_1 \qquad \emptyset;\emptyset \vdash \mathtt{e}_0 : \mathtt{@C}_0$

Similar to the case T-INVK-E.

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1998.

[2] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proc. of OOPSLA '98*, pages 183–200, 1998.

[3] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, April 1994.

[4] Kim B. Bruce. Some challenging typing issues in object-oriented languages. In *Proc. of WOOD'03*, volume 82 of *ENTCS*, 2003.

[5] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

[6] Kim B. Bruce and J. Nathan Foster. LOOJ: Weaving LOOM into Java. In *Proc. of ECOOP 2004*, volume 3086 of *LNCS*, pages 390–414, June 2004.

[7] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *Proc. of ECOOP '98*, volume 1445 of *LNCS*, pages 523–549, 1998.

[8] Kim B. Bruce, Leaf Petersen, and Adrian Fiech. Subtyping is not a good "match" for object-oriented languages. In *Proc. of ECOOP '97*, volume 1241 of *LNCS*, pages 104–127, June 1997.

[9] Kim B. Bruce and Joseph C. Vanderwaart. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. In *Proc. of MFPS XV*, volume 20 of *ENTCS*, 1999.

[10] Nicholas Cameron and Sophia Drossopoulou. On subtyping, wildcards, and existential types. In *Proc. of FTfJP (FTfJP'09)*, Genova, Italy, 2009.
[11] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proc. of ACM Conference on Functional Programming and Computer Architecture (FPCA'89)*, pages 273–280, London, England, September 1989. ACM Press.
[12] Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: A simple virtual class calculus. In *Proc. of AOSD'07*, pages 121–134, 2007.
[13] Erik Ernst. Family polymorphism. In *Proc. of ECOOP 2001*, volume 2072 of *LNCS*, pages 303–326, 2001.
[14] Erik Ernst. Higher-order hierarchies. In *Proc. of ECOOP 2003*, volume 2743 of *LNCS*, pages 303–328, 2003.
[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Sotfware*. Addison-Wesley, 1995.
[16] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–450, May 2001.
[17] Atsushi Igarashi and Mirko Viroli. Variant parametric types: A flexible subtyping scheme for generics. *TOPLAS*, 28(5):795–847, September 2006.
[18] Atsushi Igarashi and Mirko Viroli. Variant path types for scalable extensibility. In *Proc. of OOPSLA 2007*, 2007.
[19] Paul Jolly, Sophia Drossopoulou, Christopher Anderson, and Klaus Ostermann. Simple dependent types: Concord. In *Proc. of FTfJP 2004*, June 2004.
[20] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison Wesley, 1993.
[21] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *Proc. of OOPSLA '04*, pages 99–115, October 2004.
[22] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: Nested intersection for scalable software composition. In *Proc. of OOPSLA'06*, pages 21–36, 2006.
[23] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In Luca Cardelli, editor, *Proc. of ECOOP '03*, volume 2743 of *LNCS*, pages 201–224, Darmstadt, Germany, July 2003.
[24] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. of POPL*, 1997.
[25] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proc. of OOPSLA '05*, pages 41–57, 2005.
[26] Benjamin C. Pierce. *Existential Types*, chapter 24, pages 363–379. The MIT Press, 2002.
[27] Chieri Saito and Atsushi Igarashi. The essence of lightweight family polymorphism. *Journal of Object Technology*, 7(5):67–99, June 2008. Special Issue: Workshop on FTfJP 2007.
[28] Chieri Saito and Atsushi Igarashi. Matching thistype to subtyping. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1851–1858, New York, NY, USA, 2009. ACM.
[29] Chieri Saito and Atsushi Igarashi. Self type constructors. In *Proc. of OOPSLA2009*, New York, NY, USA, To appear. ACM.
[30] Chieri Saito, Atsushi Igarashi, and Mirko Viroli. Lightweight family polymorphism. *Journal of Functional Programming*, 18(03):285–331, May 2008.
[31] Mads Torgersen. The expression problem revisited: Four new solutions using generics. In *Proc. of ECOOP (ECOOP2004)*, volume 3086 of *LNCS*, pages 123–146, Oslo, Norway, June 2004.
[32] Mads Torgersen, Erik Ernst, Christian Plesner Hansen, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. *Journal of Object Technology*, 3(11):97–116, December 2004.
[33] Philip Wadler. The expression problem. Discussion on the Java-Genericity mailing list, 1998.
[34] Michael Winter. On problems in polymorphic object-oriented languages with self types and matching. *Fundamenta Informaticae*, 71(4):477–491, 2006.
[35] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.