Matching *ThisType* to Subtyping

Chieri Saito
Graduate School of Informatics
Kyoto University
Kyoto 606-8501 JAPAN
saito@kuis.kyoto-u.ac.jp

Atsushi Igarashi
Graduate School of Informatics
Kyoto University
Kyoto 606-8501 JAPAN
igarashi@kuis.kyoto-u.ac.jp

ABSTRACT

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

Categories and Subject Descriptors

D.3.1 [**Programing Languages**]: Formal Definitions and Theory; D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Classes and objects; Polymorphism*; F.3.3 [**Logics and Meaning of Programs**]: Studies of Program Constructs—*Object-oriented constructs; Type structure*

General Terms

Design, Languages, Theory

Keywords

binary methods, dynamic dispatch, exact types, subtyping, ThisType

1. INTRODUCTION

Background. Language designs for statically-typed class-based object-oriented programming languages have been studied to promote code reuse by inheritance. The target of reuse has been scaling up from a class to a group of classes and even a class hierarchy so that reusable units (components) can have more complex structures. The key to achieving reuse is to keep intra-component dependencies through extension. There have been many proposals, in which there are two styles to write the dependencies, that is, one using dependent types [11, 12, 19, 20, 10, 22] and the other not [17, 24, 16, 5, 6, 9, 3]. The discussion in this paper focuses on the latter style.

Binary methods [4] is a familiar example showing that even a single class is difficult to be safely reused by inheritance if it has self-recursive references. A binary method is one whose parameter type is the same as the receiver type. Ideally, in a class definition, the parameter type has to change covariantly as the class extends so that subclasses refer to themselves. However, covariant change is disallowed in the languages such as C++ and Java for safety. As a result, the subclasses refer to its superclass and this gap is often solved by typecasting, a potentially unsafe operation.

***ThisType** and its Extensions.* The notion of *MyType* [2, 8, 7] is proposed for the languages with structural type systems to support type-safe reuse of binary methods. *MyType* represents the type of an object that it appears in and its meaning covariantly changes along with class extension, as desired. Later, it is adapted to GJ [1] with a nominal type system, resulting in the language LOOJ [5], in which *MyType* is called `ThisType`[1]. Subsequently `ThisType` is extended to mutually recursive classes [24, 6, 9, 3], class hierarchies [17], and arbitrarily nested groups of classes [16].

*Mismatch between **ThisType** and Subtyping.* It is well-known that `ThisType` and its extensions do not match with subtyping well. If we gave the type system naively, type safety would be lost. LOOJ guarantees type safety, but sacrifices subtyping, hence dynamic dispatch, an important feature of object-oriented programming.

A naive type system can be given by adding the following (informal) rules: (1) 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 `ThisType` by the receiver type; (2) a method declaration is typed in a class under the assumption that `ThisType` and the class name are compatible. However, such a type system breaks safety. The former rule does not guarantee safety of the invocation because the signature of the dispatched method can be different from the one obtained at compile-time. The latter, in particular the assumption that the class name is a

[1] In LOOJ, actually, `ThisType` represents the public interface of the class whereas `ThisClass` refers to the class type. Here, we use `ThisType` for `ThisClass`.

subtype of `ThisType`, does not guarantee safe inheritance because the meaning of `ThisType` changes along with class extension—the inherited method may not be well typed in subclasses.

The type systems of LOOJ and others have two restrictions to guarantee type safety:

1. a binary method can be invoked only on the receiver whose run-time type is statically known; and
2. when a method is typed in a class, the class name is not a subtype of `ThisType`.

LOOJ and others are equipped with *exact types* as a means to identify run-time types statically; binary methods can be invoked only on the receivers of exact types. As a result, it is impossible to dynamically dispatch binary methods. The latter makes it difficult to implement a method, with `ThisType` being its return type, that returns a new object that is of the same type as the receiver, such as `clone()`, since objects have to be created by specifying a concrete class name in LOOJ[2]. Giving the return type of `clone()` `ThisType` is crucial to realize that returned objects have the same type as the receiver whatever type the receiver has. In Java, returned objects must be casted explicitly since the return type of `clone()` is `Object`.

In summary, in LOOJ and the similar languages, programs, especially client code, are often tied to a specific implementation. We wish to relax it and realize more "object-oriented" programming in the sense that we can use dynamic dispatch in more occasions.

Contributions of the Paper. In this paper, we propose two mechanisms, namely, *local exactization* and *nonheritable methods* for the languages with `ThisType` and exact types such as LOOJ. The former allows binary methods to be invoked even if the run-time types of the receivers are not identified. The latter allows a class name to be a subtype of `ThisType` under a certain condition.

To rigorously show that our proposing features are safe, we formalize them on top of Featherweight Java [14], or FJ in short, and prove its type soundness. Although FJ models a minimal set of features for class-based languages and is not equipped with a grouping mechanism, for example class nesting, our proposals are easily adaptable to the extensions of `ThisType` in the languages with grouping mechanisms.

We summarize our technical contributions as follows:

- introduction of the new features, namely, local exactization and nonheritable methods; and
- a formalization of a sound core language with these features.

Rest of This Paper. Section 2 examines the mismatch between `ThisType` and subtyping after reviewing `ThisType` and exact types. 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. Section 6 concludes. Hereafter, we write `This`, as done in [16], for `ThisType`.

[2]Such methods can be implemented by using abstract factory pattern [13] as seen in [5], which will be discussed in Section 5.

2. MISMATCH BETWEEN *THIS* AND SUBTYPING

In this section, we examine the problems of `This` and exact types found in the current type systems such as LOOJ's after briefly reviewing them through an example of extensible self-recursive classes, written in Java-like syntax.

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

Class `C` declares the 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 argument type being `C`, the field access would be illegal since `D` has to override the method with the same signature, but `C` does not have `field2`.

Exact types [16, 5] are used to guarantee the safe invocations of binary methods. 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`. So, the condition for safe invocation of binary methods (mentioned in Introduction) can be rephrased "*the receivers of binary methods should have exact types.*" Consider the following code:

```
@C c1; C c2, c3;
c1.isEqual(c2); // 1: allowed
c2.isEqual(c3); // 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 `c2` might refer to the object of class `D`, dispatching the overridden `isEqual()`, although `c3` might refer to that of class `C`, which does not have `field2`. Hereafter, we call types without `@` *inexact.*

2.2 Problem Description

We examine the two problems that we tackle in this paper. Figure 1 shows our running example adapted from [7].

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>` defines nodes for singly-linked lists. (It is not important that the class is parameterized with the element type `E` for the field `elem` and we sometimes omit the argument for it.) The class has a field

```
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**

`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 the objects of `DoublyLinkedNode`. So, whether a list is singly or doubly linked, we at least know that the linked objects in the list are those of a *same* type.

2.2.1 Binary Methods are Always Statically Dispatched

As mentioned before, binary methods should be invoked on the receivers of exact types. This restriction prevents dynamic dispatch of binary methods. 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. The type of `head.next` is `LinkedNode<Integer>`, the same as that of `head`, in LOOJ. 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.

Bruce et al. [7] claim that the code above could be well typed by rewriting it into a parameterized method as follows:

```
<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 systems, 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* [13] or cloning methods.

In Figure 1, in `insertElem()`, the factory `makeNode()` is invoked to create a new node. The return type of `makeNode()` must be `@This` so as to guarantee that the new node has exactly the same type as the receiver `this` whatever class's instance `this` refers to. 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.

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[3]: 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`.

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 the two language features, namely, *local exactization* (Section 3.1) and *nonheritable methods* (Section 3.2). Each solves the problem described in Section 2.2.1 and Section 2.2.2, respectively. Figure 2 shows the complete definition of the classes in Figure 1 using the latter proposal.

3.1 Local Exactization

We propose the typing feature *local exactization* 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 [23]. 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.

[3]LOOJ allows object creation only with a concrete class name (not including `This`), just as Java.

```
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

The syntax of local exactization is:

```
exact ... as x, X in { ... }
```

The statement begins with `exact`, followed by the expression to be exactized. The variable `x` and type variable `X` are bound in the body enclosed by braces. In the body, `x` has type `@X` and `X` is bounded by the type of the target expression. At run-time, the body is executed where `x` is initialized to the value of the target expression.

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

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

3.2 Nonheritable Methods

We propose the feature *nonheritable methods*[4] 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—they have to *rewrite*[5] the method with the same signature;

2. `This` in the signature of a nonheritable method is replaced with the name of the declaring class when the method is typed;

3. it is not necessary that the rewritten methods are nonheritable.

The first forces each class to have its own implementation. The second allows `This` and the class name to be compatible so that the method will be specialized for the class. The third is mainly for convenience: we can stop the rewriting 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 supertype of the type `@LinkedNode` of the new object. If `DoublyLinkedNode` did not rewrite `makeNode()`, it would be ill-typed.

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

[5]We do not say "override" since the subclasses do not know the implementation of the nonheritable method. For the same reason, it is impossible to call it by `super`.

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 [14], a functional core of class-based object-oriented languages. 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, $\oslash$ is read `nonheritable` and $[\oslash]$ means the modifier $\oslash$ 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. The symbols $\triangleleft$ and $\uparrow$ are read `extends` and `return`, respectively.

$$
\begin{array}{rcll}
\mathtt{H} & ::= & \mathtt{X} \mid \mathtt{C} & \text{inexact types} \\
\mathtt{S},\mathtt{T},\mathtt{U} & ::= & \mathtt{H} \mid \mathtt{@H} & \text{types} \\
\mathtt{L} & ::= & \mathtt{class}\ \mathtt{C} \triangleleft \mathtt{C}\{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\} & \text{classes} \\
\mathtt{M} & ::= & [\oslash]\ \mathtt{T}\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\uparrow \mathtt{e};\} & \text{methods} \\
\mathtt{d},\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m}(\overline{\mathtt{e}}) \mid \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}) & \text{expressions} \\
 & & \mid \mathtt{exact}\ \mathtt{e}\ \mathtt{as}\ \mathtt{x},\ \mathtt{X}\ \mathtt{in}\ \mathtt{e} & \\
\mathtt{v} & ::= & \mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}}) & \text{values}
\end{array}
$$

Following the custom of FJ, we put an over-line for a possibly empty sequence. Furthermore, we abbreviate pairs of sequences in a similar way, writing "$\overline{\mathtt{T}}\ \overline{\mathtt{f}};$" for "$\mathtt{T}_1\ \mathtt{f}_1;\ldots\ \mathtt{T}_n\ \mathtt{f}_n;$", 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.

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 `exact` is a single expression, rather than a statement as in the previous section. 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`.

A class table CT is a finite mapping from class names `C` to class declarations `L`. A program is a pair (CT, `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 $\mathit{fields}(\mathtt{C})$ returns a sequence $\overline{\mathtt{T}}\ \overline{\mathtt{f}}$ of field names of the class `C` with their types. The function $\mathit{mtype}(\mathtt{m}, \mathtt{C})$ 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 [14] except that *mtype* accounts the optional modifier $\oslash$. Here, $\mathtt{m} \notin \overline{\mathtt{M}}$ means the method of name `m` does not exist in $\overline{\mathtt{M}}$.

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

$$\frac{\mathtt{class\ C \lhd D\ \{...\overline{M}\}} \quad [\oslash]\ \mathtt{T_0\ m(\overline{T}\ \overline{x})\{\uparrow e;\ \}} \in \overline{\mathtt{M}}}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{T}} \to \mathtt{T}_0} \qquad \frac{\mathtt{class\ C \lhd D\{...\overline{M}\}} \quad \mathtt{m} \notin \overline{\mathtt{M}} \quad \mathit{mtype}(\mathtt{m}, \mathtt{D}) = \overline{\mathtt{T}} \to \mathtt{T}_0}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{T}} \to \mathtt{T}_0}$$

4.3 Type System

The main judgments of the type system consist of one $\Delta \vdash \mathtt{S} \mathrel{<:} \mathtt{T}$ for subtyping, one $\Delta \vdash \mathtt{T}$ ok for type well-formedness, and one $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}$ for expression typing. Here, Δ is a *bound environment*, which is a finite mapping from type variables to their bounds, written $\overline{\mathtt{X}} \mathrel{<:} \overline{\mathtt{H}}$; Γ is a *type environment*, which is a finite mapping from variables to types, written $\overline{\mathtt{x}} : \overline{\mathtt{T}}$. We abbreviate a sequence of judgments: $\Delta \vdash \mathtt{S}_1 \mathrel{<:} \mathtt{T}_1, \ldots, \Delta \vdash \mathtt{S}_n \mathrel{<:} \mathtt{T}_n$ to $\Delta \vdash \overline{\mathtt{S}} \mathrel{<:} \overline{\mathtt{T}}$; $\Delta \vdash \mathtt{T}_1$ ok, $\ldots, \Delta \vdash \mathtt{T}_n$ ok to $\Delta \vdash \overline{\mathtt{T}}$ ok, and $\Delta; \Gamma \vdash \mathtt{e}_1 : \mathtt{T}_1, \ldots, \Delta; \Gamma \vdash \mathtt{e}_n : \mathtt{T}_n$ to $\Delta; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}$.

Bound of Types. The function $\mathit{bound}_\Delta(\mathtt{H})$, defined below, takes an inexact type as input and returns a class name, 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.

Subtyping. The subtyping judgment $\Delta \vdash \mathtt{S} \mathrel{<:} \mathtt{T}$, 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 $\mathtt{@C} \not\mathrel{<:} \mathtt{@D}$ even if `class C◁D{..}`. So, this subtyping relation is actually "matching" [7] since $\mathtt{C} \mathrel{<:} \mathtt{D}$ 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`.

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

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

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 its inexact version `H` is.

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

Closing of Types. Before proceeding to expression typing, we define the judgment $\mathtt{S} \Downarrow_{\mathtt{X<:H}} \mathtt{T}$, read as "type `S` is closed to `T` under `X<:H`", for closing of types. The rules are defined below. This judgment is used in the typing rules for `exact` expressions to prevent the type variable introduced from escaping. The basic idea is to lift the type variable to its supertype so that the type variable does not appear in the result. The left rule says that if type `T` does not contain the 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 is $\mathtt{@X} \mathrel{<:} \mathtt{X} \mathrel{<:} \mathtt{H}$, but $\mathtt{@X} \not\mathrel{<:} \mathtt{@H}$. Here, $\mathit{fv}(\mathtt{T})$ returns the empty or a singleton set of type variables that appear in `T`.

$$\frac{\mathtt{X} \notin \mathit{fv}(\mathtt{T})}{\mathtt{T} \Downarrow_{\mathtt{X<:H}} \mathtt{T}} \qquad \mathtt{X} \Downarrow_{\mathtt{X<:H}} \mathtt{H} \qquad \mathtt{@X} \Downarrow_{\mathtt{X<:H}} \mathtt{H}$$

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 `e` has type `T`," defined below. The key rules are T-Field and T-Invk. Both rules restrict the receivers' ($\mathtt{e}_0$) types to be exact. Although this restriction is imposed for all field accesses and method invocations, not only for binary methods, the expressive power of the language is not lost since we have `exact` expressions. (We later give typing rules for field accesses and method invocations on inexact types. See below.) The rule T-Field means that 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 $\mathtt{f}_i$. Similarly, in T-Invk 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-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 whether the expression `e` to be exactized is of type `H` or `@H`. The rule T-Exact1 means that if `e` is of type `H`, the body expression $\mathtt{e}_0$ is typed under Δ extended by `X<:H` and Γ extended by `x:@X`. Note that variable `x` is of type `@X`, an exact type. Since the resultant type $\mathtt{U}_0$ may contain the type variable `X`, the type of the whole expression is obtained by closing $\mathtt{U}_0$ under `X<:H`, preventing `X` from escaping. We give the rule T-Exact2 for the case that the expression which will be exactized is *already* exact. This rule 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 $\mathtt{x{:}@H}$ since there is no proper subtype of $\mathtt{@H}$.

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

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

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

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

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_1 : \mathtt{H} \qquad \Delta, \mathtt{X} \mathrel{<:} \mathtt{H}; \Gamma, \mathtt{x} : \mathtt{@X} \vdash \mathtt{e}_0 : \mathtt{U}_0 \qquad \mathtt{U}_0 \Downarrow_{\mathtt{X<:H}} \mathtt{T}_0}{\Delta;\Gamma \vdash \mathtt{exact}\ \mathtt{e}_1\ \mathtt{as}\ \mathtt{x},\ \mathtt{X}\ \mathtt{in}\ \mathtt{e}_0 : \mathtt{T}_0} \qquad \text{(T-Exact1)}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_1 : \mathtt{@H} \qquad \Delta;\Gamma, \mathtt{x} : \mathtt{@H} \vdash \mathtt{e}_0 : \mathtt{U}_0}{\Delta;\Gamma \vdash \mathtt{exact}\ \mathtt{e}_1\ \mathtt{as}\ \mathtt{x},\ \mathtt{X}\ \mathtt{in}\ \mathtt{e}_0 : \mathtt{U}_0} \qquad \text{(T-Exact2)}$$

We show the typing examples of field accesses on exact and inexact types. Assume that *fields*(`LinkedNode`) contains `@This next`. If $\Delta;\Gamma \vdash \mathtt{n} : \mathtt{@LinkedNode}$, then $\Delta;\Gamma \vdash \mathtt{n.next} : \mathtt{@LinkedNode} (= [\mathtt{LinkedNode}/\mathtt{This}]\mathtt{@This})$ by T-Field. If $\Delta;\Gamma \vdash \mathtt{n} : \mathtt{LinkedNode}$, exactization is required before accessing the field to be well typed: $\Delta;\Gamma \vdash \mathtt{exact\ n\ as\ x,\ X\ in\ x.next} : \mathtt{LinkedNode}$ by T-Exact1 and T-Field since $\mathit{fields}(\mathit{bound}_{\Delta,\mathtt{X<:LinkedNode}}(\mathtt{X})) = \mathit{fields}(\mathtt{LinkedNode})$ and $\Delta, \mathtt{X} \mathrel{<:} \mathtt{LinkedNode}; \Gamma, \mathtt{x} : \mathtt{@X} \vdash \mathtt{x.next} : \mathtt{@X}$ and $\mathtt{@X} \Downarrow_{\mathtt{X<:LinkedNode}} \mathtt{LinkedNode}$.

To avoid cumbersome exactization in accessing members on inexact types, we could give the following derived rules, which can be obtained by the combination of T-Field/T-Invk and T-Exact1.

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{H}_0 \qquad \mathit{fields}(\mathit{bound}_\Delta(\mathtt{H}_0)) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \\ \mathtt{T}_i \Downarrow_{\mathtt{This<:H}_0} \mathtt{T}\end{array}}{\Delta;\Gamma \vdash \mathtt{e}_0\mathtt{.f}_i : \mathtt{T}} \qquad \text{(T-Field')}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{H}_0 \qquad \mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{H}_0)) = \overline{\mathtt{T}}{\rightarrow}\mathtt{T}_0 \\ \overline{\mathtt{T}} \text{ does not contain } \mathtt{This} \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \\ \Delta \vdash \overline{\mathtt{U}} \mathrel{<:} \overline{\mathtt{T}} \qquad \mathtt{T}_0 \Downarrow_{\mathtt{This<:H}_0} \mathtt{T}\end{array}}{\Delta;\Gamma \vdash \mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{T}} \qquad \text{(T-Invk')}$$

Method Typing. The typing judgment for method declarations is written $\mathtt{C} \vdash \mathtt{M}$ 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 or rewrites (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` ($= [\mathtt{C}/\mathtt{This}]\mathtt{@This}$) 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.

$$\frac{\begin{array}{c}\Delta = \{\mathtt{This} \mathrel{<:} \mathtt{C}\} \qquad \Gamma = \{\overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{@This}\} \\ \Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{U}_0 \qquad \Delta \vdash \mathtt{U}_0 \mathrel{<:} \mathtt{T}_0 \qquad \Delta \vdash \mathtt{T}_0, \overline{\mathtt{T}} \text{ ok} \\ \mathtt{class}\ \mathtt{C} \triangleleft \mathtt{D}\{\ldots\} \\ \mathit{mtype}(\mathtt{m}, \mathtt{D}) = \overline{\mathtt{U}}{\rightarrow}\mathtt{U}_0 \text{ implies } (\overline{\mathtt{U}}, \mathtt{U}_0) = (\overline{\mathtt{T}}, \mathtt{T}_0)\end{array}}{\mathtt{C} \vdash \mathtt{T}_0\ \mathtt{m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{} \uparrow \mathtt{e}_0\mathtt{;\}}} \qquad \text{(T-Method)}$$

$$\frac{\begin{array}{c}\Gamma = \{\overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{@This}\} \qquad \emptyset; [\mathtt{C}/\mathtt{This}]\Gamma \vdash \mathtt{e}_0 : \mathtt{U}_0 \\ \emptyset \vdash \mathtt{U}_0 \mathrel{<:} [\mathtt{C}/\mathtt{This}]\mathtt{T}_0 \qquad \emptyset \vdash [\mathtt{C}/\mathtt{This}](\mathtt{T}_0, \overline{\mathtt{T}}) \text{ ok} \\ \mathtt{class}\ \mathtt{C} \triangleleft \mathtt{D}\{\ldots\} \\ \mathit{mtype}(\mathtt{m}, \mathtt{D}) = \overline{\mathtt{U}}{\rightarrow}\mathtt{U}_0 \text{ implies } (\overline{\mathtt{U}}, \mathtt{U}_0) = (\overline{\mathtt{T}}, \mathtt{T}_0)\end{array}}{\mathtt{C} \vdash \oslash\ \mathtt{T}_0\ \mathtt{m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{} \uparrow \mathtt{e}_0\mathtt{;\}}} \qquad \text{(T-NHMethod)}$$

Class Typing. The typing judgment for class declarations is written $\vdash \mathtt{L}$ 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 rewritten. Here, $\mathtt{m} \in \overline{\mathtt{M}}$ means that the method of name `m` exists in $\overline{\mathtt{M}}$.

$$\frac{\begin{array}{c}\mathtt{C} \vdash \overline{\mathtt{M}} \text{ ok} \qquad \mathtt{This} \mathrel{<:} \mathtt{C} \vdash \overline{\mathtt{T}}, \mathtt{D} \text{ ok} \qquad \mathtt{class}\ \mathtt{D} \triangleleft \mathtt{E}\{..\overline{\mathtt{M}}'\} \\ \text{for each } \oslash\ \mathtt{U}_0\ \mathtt{m(}\overline{\mathtt{U}}\ \overline{\mathtt{x}}\mathtt{)\{..\}} \in \overline{\mathtt{M}}', \mathtt{m} \in \overline{\mathtt{M}}\end{array}}{\vdash \mathtt{class}\ \mathtt{C} \triangleleft \mathtt{D}\{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\} \text{ ok}} \qquad \text{(T-Class)}$$

A class table CT is ok, if all its definitions are ok.

4.4 Operational Semantics

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

The reduction rules are given below. We write $[\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{e}/\mathtt{y}]\mathtt{e}_0$ for the expression obtained from $\mathtt{e}_0$ by replacing $\mathtt{x}_1$ with $\mathtt{d}_1$, ..., $\mathtt{x}_n$ with $\mathtt{d}_n$, and `y` with `e`. The rule R-Exact means that when the expression being exactized is a `new` expression the body $\mathtt{e}_0$ is evaluated where `x` is bound to $\mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{)}$ and `X` is bound to `C`. Note that the application of type substitution $[\mathtt{C}/\mathtt{X}]$ to $\mathtt{e}_0$ is omitted since there are no type variables in expressions. Similarly, $[\mathtt{C}/\mathtt{This}]$ is omitted in R-Invk. The reduction rules may be applied at any point in an expression, so we also need the obvious congruence rules (if $\mathtt{e} \longrightarrow \mathtt{e}'$ then $\mathtt{e.f} \longrightarrow \mathtt{e}'\mathtt{.f}$, and the like), omitted here. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

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

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

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

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 [26, 14]. We omit the proofs of the theorems. We refer interested readers to `http://www.sato.kuis.kyoto-u.ac.jp/~saito/oops2009/` for the proofs.

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* $\Delta \vdash \mathtt{T}' \mathrel{<:} \mathtt{T}$.

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* $\mathtt{e}'$.

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* $\Delta \vdash \mathtt{T}' \mathrel{<:} \mathtt{T}$.

5. RELATED WORK

In this section, the first two subsections discuss the work related to local exactization whereas the other three discuss that to nonheritable methods.

Existential Types in Java. In our type system, inexact types are treated existential. Java is already equipped with a kind of existential types for generics [1], called wildcard types [25], derived from variant parametric types [15]. 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.

Pizza [21], 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.

Dependent Type Systems. In the languages [11, 19, 20, 10, 22] using dependent types, it is possible to dynamically dispatch binary methods since it is easy to check that both receiver and argument depend on a same object. In Jx [19], the return type of `makeNode()` would be given `this.class`, which means that the run-time type of the receiver. In general, `x.class` is a dependent type, meaning the run-time type of the object that `x` refers to. 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.

Object Creation with Abstract Types. In C#, in parameterized classes, objects can be created on the type parameter 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 [18] 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 raised in subclassing: rewriting nonheritable methods in subclasses does not require the rewritten methods to be nonheritable so that further subclassing can be free from rewriting, whereas object creation with type variables requires *all* classes to rewrite the constructors, which are not inherited in Java.

In Jx [19] and J& [20], object creation with dependent types such as `new n.class()` is allowed. Moreover, the arguments can be of arbitrary many and type 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.

Abstract Factory Pattern. Even though nonheritable methods are not supported, factory methods and `clone()` can be implemented by using abstract factory pattern [13], discussed in [5], as follows:

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

In this programming, a factory class must be defined for *each* of class `LinkedNode` and its extensions. This resembles that all subclasses must rewrite 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()`.

Template Specialization. Template specialization in C++ allows us to give a definition for the template instantiated with a certain type. For example, `Vector<bool>` 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`).

6. CONCLUSION

We propose two mechanisms, namely, local exactization and nonheritable methods for the languages with `This` and exact types. 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 local exactization, 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 and to integrate with generics. For generics, wildcards will be required to close arbitrary types (LOOJ, which is equipped with generics, avoids wildcards by posing a syntactic restriction): for example, when variable `c` is of type `C` and class `C` has a field `f` of type `List<This>`, the expression `c.f` should be of type `List<? extends C>`, but not `List<C>`.

Acknowledgments

Comments from anonymous reviewers help improve the final presentation of the present paper. 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 Graint-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.

7. REFERENCES

[1] 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.

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

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In W. Olthoff, editor, *Proc. of ECOOP '95*, volume 952 of *LNCS*, pages 27–51, August 1995.

[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] Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: A simple virtual class calculus. In *Proc. of AOSD'07*, pages 121–134, 2007.

[11] Erik Ernst. Family polymorphism. In *Proc. of ECOOP 2001*, volume 2072 of *LNCS*, pages 303–326, 2001.

[12] Erik Ernst. Higher-order hierarchies. In *Proc. of ECOOP 2003*, volume 2743 of *LNCS*, pages 303–328, 2003.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Sotfware*. Addison-Wesley, 1995.

[14] 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.

[15] Atsushi Igarashi and Mirko Viroli. Variant parametric types: A flexible subtyping scheme for generics. *TOPLAS*, 28(5):795–847, September 2006.

[16] Atsushi Igarashi and Mirko Viroli. Variant path types for scalable extensibility. In *Proc. of OOPSLA 2007*, 2007.

[17] Paul Jolly, Sophia Drossopoulou, Christopher Anderson, and Klaus Ostermann. Simple dependent types: Concord. In *Proc. of FTfJP 2004*, June 2004.

[18] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison Wesley, 1993.

[19] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *Proc. of OOPSLA '04*, pages 99–115, October 2004.

[20] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: Nested intersection for scalable software composition. In *Proc. of OOPSLA'06*, pages 21–36, 2006.

[21] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. of POPL*, 1997.

[22] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proc. of OOPSLA '05*, pages 41–57, 2005.

[23] Benjamin C. Pierce. *Existential Types*, chapter 24, pages 363–379. The MIT Press, 2002.

[24] Chieri Saito, Atsushi Igarashi, and Mirko Viroli. Lightweight family polymorphism. *Journal of Functional Programming*, 18(03):285–331, May 2008.

[25] 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.

[26] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.