Union Types for Object-Oriented Programming

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

Hideshi Nagira
Graduate School of Informatics
Kyoto University
Kyoto 606-8501 JAPAN
nagira@kuis.kyoto-u.ac.jp

ABSTRACT

We propose *union types* for statically typed class-based object-oriented languages as a means to enhance the flexibility of subtyping. As its name suggests, a union type can be considered a set union of instances of several types and behaves as their least common supertype. It also plays the role of an interface that 'factors out' commonality of given types—fields of the same name and methods with similar signatures. Union types can be useful for implementing heterogeneous collections and for grouping independently developed classes with similar interfaces, which has been considered difficult in languages like Java. To rigorously show the safety of union types, we formalize them on top of Featherweight Java and prove that the type system is sound.

Categories and Subject Descriptors

D.3.1 [**Programming 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

Java, language design, subtyping, type systems, union types

1. INTRODUCTION

Background. The design of good, reusable class libraries is known to be a very hard problem and, in mainstream object-oriented languages like Java and C++, inheritance and subtyping (and, more recently, generics) have been used as main mechanisms to promote code reuse. While inheritance enables one class to reuse implementation (declarations of instance variables and methods) of another class, subtyping is for *substitutability*—the property that if an object of one type can be used at a certain place, then another object of a subtype can be used at the same place, too. (Substitutability may be rephrased as *reusability of contexts*, in the sense that if some context is applicable to an object of one type then the same context is also applicable to any object of its subtype.) Thus, design concerns about inheritance and subtyping relations are somewhat different: it has to be taken into account, for inheritance, how new classes may reuse existing implementation and, for subtyping, how objects may be used in client code.

In the mainstream languages, however, subtyping relation is mostly based on inheritance relation[1]. It can happen that two classes used in similar contexts but with rather different implementations are placed apart in the class (inheritance) hierarchy, resulting in no useful supertype of those classes. Interfaces (as a programming construct) in Java are a solution to this problem: one can define a super-interface of classes of similar use, regardless of a given inheritance hierarchy, and enjoy benefits of subtyping. However, interfaces cannot be added once a class is defined, so library designers still have to do a lot of planning of their interface hierarchies *before* the library is shipped. This problem has been considered a significant limitation of type systems with declaration-based subtyping, as in Java.

Our Proposal—Union Types. In this paper, we propose *union types* to partially address the problem of the inability of adding supertypes to existing types (classes and interfaces). As its name suggests, a union type denotes a set union of given some types (viewed as sets of instances that belong to those types) and behave as a least common supertype of them. Since union types are composed from existing types, they give an ability to define a supertype even *after* a class hierarchy is fixed. Union types can be used not only by case analysis as in ML datatypes, but also by direct member access as ordinary types. In fact, given some types, their union type can be viewed as an interface that 'factors out' their common features, that is, the fields of the same name and methods with similar signatures.

We expect that union types can be useful for grouping independently developed classes with similar interfaces, by giving their supertype, and for implementing heterogeneous collections like lists where, say, strings and integers are mixed

[1]A notable exception is wildcards [13] in Java 5.0.

as elements.

Our contributions in this paper can be summarized as follows:

- The proposal of union types for class-based object-oriented languages with a name-based type system;
- A formalization of a core object-oriented language FJ$\vee$ with union types on top of Featherweight Java [7]; and
- A proof of type soundness of the core language.

Organization of The Paper. The rest of the paper is organized as follows. We first give an overview of union types and related constructs in Section 2 and then formalize FJ$\vee$ and prove its type soundness in Section 3. We discuss the interactions of union types with other common features such as generics and method overloading in Section 4, then discuss related work in Section 5 and finally give concluding remarks in Section 6. For brevity, we omit proofs of theorems; they will appear in a full version, available at `http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/papers/`.

2. UNION TYPES, PRIMER

In this section, we informally introduce union types and develop related constructs. As a first step, we focus only on core features typically found in class-based object-oriented languages, and defer the discussion on some other features in Section 4.2.

A union type can be constructed from any two types `A` and `B` by combining with $\vee$, written `A`$\vee$`B`. We often call `A` and `B` the *summands* of `A`$\vee$`B`. Intuitively, when types `A` and `B` denote some sets of instances, `A`$\vee$`B` denotes the set union of the two sets. Since union types are, unlike class names, not associated with implementation, they cannot be used to instantiate objects. So, they are closer to interface types of Java. We forbid another class (or interface, respectively) to 'implement' (or 'extend', respectively) union types—in the sense of Java—so as to ensure exhaustiveness of case analysis (see below), although there are non-trivial subtypes of union types other than their summands, as is discussed shortly.

By (naively) viewing subtyping as set inclusion, `A`$\vee$`B` is a supertype of both `A` and `B`. Thus, supposing there are two classes `Jpg` and `Gif` implementing image objects, an assignment below is allowed:

```
Jpg∨Gif im = new Jpg("portrait.jpg");
```

Moreover, `Jpg`$\vee$`Gif` is a *least* supertype among supertypes of `A` and `B` in the sense that any common supertype of `A` and `B` is also a supertype of `A`$\vee$`B`. So,

```
Image x = im;  // assuming Jpg ad Gif extend Image
```

is also allowed. The leastness property can be explained in terms of the 'types-as-sets' interpretation above: `A`$\vee$`B` includes only instances that belong to `A` or `B` and nothing else, while other supertypes may include instances belonging to classes other than `A` and `B`. Figure 1 shows an example of a subtyping hierarchy. `C`, $\mathtt{D}_i$, and $\mathtt{E}_i$ are class names and solid arrows represent inheritance relations, which are also subtyping relations. For example, $\mathtt{D}_1$ extends `C` and so $\mathtt{D}_1$ is a subtype of `C`. Dotted arrows represent subtyping relations induced by union types: $\mathtt{D}_1\vee\mathtt{D}_2$ is a supertype of $\mathtt{D}_1$ and $\mathtt{D}_2$ and also a subtype of `C`, which is also a common supertype

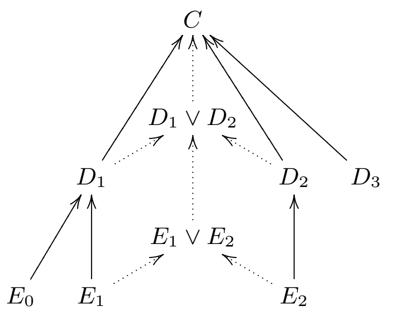

Figure 1: Union Types and Subtyping

of $\mathtt{D}_1$ and $\mathtt{D}_2$, but $\mathtt{D}_3$ is not related to $\mathtt{D}_1\vee\mathtt{D}_2$. Moreover, the union type constructor $\vee$ preserves subtyping relations of its summands. That is, $\mathtt{E}_1\vee\mathtt{E}_2$, a union of subclasses of $\mathtt{D}_1$ and $\mathtt{D}_2$, is a subtype of $\mathtt{D}_1\vee\mathtt{D}_2$, which is derived by the leastness condition.

Note that the subtyping relation here is not anti-symmetric as in usual object-oriented languages. There are two syntactically different types that are subtypes of each other. For example, `A`$\vee$`B` and `B`$\vee$`A` are syntactically different and subtypes of each other. A more interesting example is `C`$\vee$`D` and `C` when `C` is a superclass of `D`. We often call such types *compatible* types, which denote the same set of instances, though they are syntactically different.

We provide two kinds of operations on union types: *case analysis* and *direct member access*. Case analysis is a conditional construct that branches according to the run-time class of the value of an expression being tested. For example,

```
case im of (Jpg x) { x.draw(); }
        | (Gif y) { y.zoom(2); y.draw(); }
```

invokes method `draw()` if the value of `im` is an instance of `Jpg` (or one of its subclasses) or methods `zoom()` and `draw()` if it is of `Gif` (or one of its subclasses). Here, `x` and `y` are bound to the value of `im` but their static type information is more refined than `Jpg`$\vee$`Gif`. In this sense, it can be considered (at least, operationally) a combination of dynamic test of run-time types (`instanceOf`) and typecasts. So, it could be written:

```
if (im instanceOf Jpg) { Jpg x = (Jpg)im; x.draw(); }
else { Gif y = (Gif)im; y.zoom(2); y.draw(); }
```

One benefit of providing this combining construct is that the type system can check the *exhaustiveness* of branching conditions against the expression being tested. In fact, we require that the type of the test expression be a subtype of the union of the types appearing in the branches (in the example above, `Jpg` and `Gif`). This requirement will guarantee that either branch will be taken and its execution succeeds. On the other hand, the success of typecasts in the second code will not be guaranteed by standard type systems.

Direct member access allows field access directly on union types if its summands have fields of the same name. For example, consider concrete definitions of `Jpg` and `Gif`:

```
class Jpg extends Image {
  Integer hsize;  Integer ncolors;
  void zoom(Integer x) { ... }
}
```

```
class Gif extends Image {
  Integer hsize;  Byte ncolors;
  void zoom(Integer x) { ... }
}
```

Then, directly accessing field `hsize` on `im` (of type `Jpg∨Gif`) is allowed:

```
Integer i = im.hsize;
```

Moreover, even when field types are different, it is allowed to read from a field of a common name:

```
Integer∨Byte x = im.ncolors;
```

We can use union types again to type the result.

We need be a little more careful about method invocation since methods of the same name may have different signatures. Here, we will allow method invocation only when the names, the numbers of arguments, and the corresponding argument types all agree. Hence,

```
im.zoom(new Integer(100));
```

will be well typed. When return types are object types (not `void`), they can be different as in field access. We could relax the condition on argument types so that it is well typed as long as each actual argument type is a subtype of both of the corresponding formal argument types but such relaxation seems to have subtle interactions with overloading (see Section 4.2).

In this way, direct member access provides a much more concise way to write a simple member access than using case analysis, when summands have members of common names. By this mechanism, a union type can be considered a sort of interface type that 'factors out' common members from the summands. We expect that this mechanism would be useful when independently developed classes with similar functionality are combined. For example, `Jpg` and `Gif` might have been developed separately, there is no class like `Image`, and a common superclass of `Jpg` and `Gif` might have been only `Object`. Even in such a case, instances of these two classes can be handled together by using `Jpg∨Gif` and, moreover, instances of other image formats cannot be mixed (unless they are subclasses of `Jpg` or `Gif`).

We think Java-style interfaces and union types are complementary, rather than conflicting, mechanisms. On the one hand, explicitly declared interfaces are useful to abstract out class implementations and also for documentation purposes: an interface gives not only method signatures but also more semantic (or behavioral) concerns of its implementing classes, like "method `sort()` should really do sorting" (if not enforced by programming languages). On the other hand, union types are more useful to give a posteriori interfaces for legacy or third-party classes, over which programmers do not always have control.

3. FJ∨: FORMAL MODEL OF UNION TYPES

In this section, we formalize union types in a small calculus FJ∨ as an extension of Featherweight Java (FJ) [7], which is a functional core of class-based object-oriented languages. Thus, we model only a minimal set of features: classes, inheritance, fields, virtual method invocation, `this`, and, of course, union types. What are not modeled include, among others, field shadowing, overloading, and `super`. Since case analysis can be substituted for typecasts, we have dropped typecasts in FJ∨. The definition of FJ∨ is summarized in Figure 2.

3.1 Syntax

The abstract syntax of FJ∨ is as follows:

$$\begin{array}{lcl}
\mathtt{L} & ::= & \mathtt{class\ C\ extends\ C\ \{\overline{T}\ \overline{f};\ K\ \overline{M}\}} \\
\mathtt{T,S,U} & ::= & \mathtt{C} \mid \mathtt{T{\vee}T} \\
\mathtt{K} & ::= & \mathtt{C(\overline{T}\ \overline{f})\{super(\overline{f});\ this.\overline{f}=\overline{f};\}} \\
\mathtt{M} & ::= & \mathtt{T\ m(\overline{T}\ \overline{x})\{\ return\ e;\ \}} \\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m(\overline{e})} \mid \mathtt{new\ C(\overline{e})} \\
 & \mid & \mathtt{(case\ e\ of\ (T\ x)\ e\ \mid\ (T\ x)\ e)}
\end{array}$$

Here, the metavariables `C`, `D`, and `E` range over class names; `f` and `g` range over field names; `m` ranges over method names; and `x` and `y` range over variables.

We put an over-line for a possibly empty sequence. Furthermore, we abbreviate pairs of sequences in a similar way, writing "$\mathtt{\overline{C}\ \overline{f}}$" for "$\mathtt{C_1\ f_1,\ldots,C_n\ f_n}$", where n is the length of $\mathtt{\overline{C}}$ and $\mathtt{\overline{f}}$, and "$\mathtt{this.\overline{f}=\overline{f};}$" as shorthand for "$\mathtt{this.f_1=f_1;\ldots;this.f_n=f_n;}$" and so on. Sequences of type variables, field declarations, variables, and method declarations are assumed to contain no duplicate names. We write the empty sequence as • and denote concatenation of sequences using a comma.

A class declaration `L` consists of its name, its superclass, field declarations, a constructor, and methods. A type `T` (`S` or `U`) is either a class name or a union type $\mathtt{T_1{\vee}T_2}$; only class names can be used to instantiate objects, so they play the role of run-time types of objects. As in FJ, a constructor `K` is given in a stylized syntax and just takes initial (and final) values for the fields and assigns them to corresponding fields. The body of a method `M` is a single `return` statement since the language is functional. An expression `e` is either a variable, field access, method invocation, object creation, or case analysis. 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. A case analysis expression $\mathtt{case\ e_0\ of\ (T_1\ x_1)\ e_1\ \mid\ (T_2\ x_2)\ e_2}$ first evaluates $\mathtt{e_0}$ to an object `new C(...)`, and execute $\mathtt{e_1}$ if the object is a subtype of $\mathtt{T_1}$ or $\mathtt{e_2}$ if the object is not a subtype of $\mathtt{T_1}$ but $\mathtt{T_2}$, with $\mathtt{x}_i$ being bound to the object. In the execution of a well-typed program, the second subtype check can be omitted, thanks to the type system that checks the exhaustiveness of the case analysis.

A class table CT is a mapping from class names to class declarations. A program is a pair $(CT, \mathtt{e})$ of a class table and an expression. To lighten the notation in what follows, we always assume a *fixed* class table CT. As in FJ, we assume that `Object` has no members and its definition does *not* appear in the class table. We also assume other usual sanity conditions on CT: $CT(\mathtt{C}) = \mathtt{class\ C\ \ldots}$ for every $\mathtt{C} \in dom(CT)$; for every class name `C` (except `Object`) appearing anywhere in CT, we have $\mathtt{C} \in dom(CT)$; and there are no cycles in the transitive closure of `extends` relation. Given these conditions, we can identify a class table with a sequence of class declarations in an obvious way. Thus, in what follows, we write simply `class C ...` to mean $CT(\mathtt{C}) = \mathtt{class\ C\ \ldots}$.

3.2 Subtyping

The subtype relation `S <: T` includes the reflexive transitive closure of inheritance relation as in Java. The last three rules together mean that a union type `T∨U` is a least upper bound of `T` and `U`. As mentioned in the last section, the subtype relation is not anti-symmetric: for example, if `class C extends D {...}`, then both `C∨D <: D` and

Syntax:

$$
\begin{array}{lcl}
\mathtt{L} & ::= & \mathtt{class\ C\ extends\ C\ \{\overline{T}\ \overline{f};\ K\ \overline{M}\}} \\
\mathtt{T,S,U} & ::= & \mathtt{C} \mid \mathtt{T{\vee}T} \\
\mathtt{K} & ::= & \mathtt{C(\overline{T}\ \overline{f})\{super(\overline{f});\ this.\overline{f}{=}\overline{f};\}} \\
\mathtt{M} & ::= & \mathtt{T\ m(\overline{T}\ \overline{x})\{\ return\ e;\ \}} \\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m(\overline{e})} \mid \mathtt{new\ C(\overline{e})} \\
 & \mid & \mathtt{(case\ e\ of\ (T\ x)\ e\ \mid\ (T\ x)\ e)} \\
\mathtt{v} & ::= & \mathtt{new\ C(\overline{v})}
\end{array}
$$

Subtyping:

$$
\mathtt{T <: T} \qquad \frac{\mathtt{S <: T} \qquad \mathtt{T <: U}}{\mathtt{S <: U}} \qquad \frac{\mathtt{class\ C\ extends\ D\ \{\ldots\}}}{\mathtt{C <: D}}
$$

$$
\mathtt{S <: S{\vee}T} \qquad \mathtt{T <: S{\vee}T} \qquad \frac{\mathtt{S <: U} \qquad \mathtt{T <: U}}{\mathtt{S{\vee}T <: U}}
$$

Field lookup:

$$
\mathit{fields}(\mathtt{Object}) = \bullet
$$

$$
\frac{\mathtt{class\ C\ extends\ D\ \{\overline{T}\ \overline{f};\ K\ \overline{M}\}} \qquad \mathit{fields}(\mathtt{D}) = \mathtt{\overline{S}\ \overline{g}}}{\mathit{fields}(\mathtt{C}) = \mathtt{\overline{S}\ \overline{g}}, \mathtt{\overline{T}\ \overline{f}}}
$$

Field type lookup:

$$
\frac{\mathit{fields}(\mathtt{C}) = \mathtt{\overline{T}\ \overline{f}}}{\mathit{ftype}(\mathtt{f}_i, \mathtt{C}) = \mathtt{T}_i}
$$

$$
\frac{\mathit{ftype}(\mathtt{f}, \mathtt{T}_1) = \mathtt{U}_1 \qquad \mathit{ftype}(\mathtt{f}, \mathtt{T}_2) = \mathtt{U}_2}{\mathit{ftype}(\mathtt{f}, \mathtt{T}_1{\vee}\mathtt{T}_2) = \mathtt{U}_1{\vee}\mathtt{U}_2}
$$

Method body lookup:

$$
\frac{\begin{array}{c}\mathtt{class\ C\ extends\ D\ \{\overline{T}\ \overline{f};\ K\ \overline{M}\}} \\ \mathtt{S_0\ m(\overline{S}\ \overline{x})\{\ return\ e;\ \}} \in \mathtt{\overline{M}}\end{array}}{\mathit{mbody}(\mathtt{m}, \mathtt{C}) = \mathtt{\overline{x}.e}}
$$

$$
\frac{\begin{array}{c}\mathtt{class\ C\ extends\ D\ \{\overline{T}\ \overline{f};\ K\ \overline{M}\}} \qquad \mathtt{m} \notin \mathtt{\overline{M}} \\ \mathit{mbody}(\mathtt{m}, \mathtt{D}) = \mathtt{\overline{x}.e}\end{array}}{\mathit{mbody}(\mathtt{m}, \mathtt{C}) = \mathtt{\overline{x}.e}}
$$

Method type lookup:

$$
\frac{\begin{array}{c}\mathtt{class\ C\ extends\ D\ \{\overline{T}\ \overline{f};\ K\ \overline{M}\}} \\ \mathtt{S_0\ m(\overline{S}\ \overline{x})\{\ return\ e;\ \}} \in \mathtt{\overline{M}}\end{array}}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \mathtt{\overline{S}}{\to}\mathtt{S}_0}
$$

$$
\frac{\begin{array}{c}\mathtt{class\ C\ extends\ D\ \{\overline{T}\ \overline{f};\ K\ \overline{M}\}} \qquad \mathtt{m} \notin \mathtt{\overline{M}} \\ \mathit{mtype}(\mathtt{m}, \mathtt{D}) = \mathtt{\overline{S}}{\to}\mathtt{S}_0\end{array}}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \mathtt{\overline{S}}{\to}\mathtt{S}_0}
$$

$$
\frac{\mathit{mtype}(\mathtt{m}, \mathtt{T}_1) = \overline{\mathtt{S}}_1{\to}\mathtt{U}_1 \qquad \mathit{mtype}(\mathtt{m}, \mathtt{T}_2) = \overline{\mathtt{S}}_2{\to}\mathtt{U}_2 \qquad \overline{\mathtt{S}}_1 \cong \overline{\mathtt{S}}_2}{\mathit{mtype}(\mathtt{m}, \mathtt{T}_1{\vee}\mathtt{T}_2) = \overline{\mathtt{S}}_1{\to}\mathtt{U}_1{\vee}\mathtt{U}_2}
$$

Expression typing:

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

$$
\frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \mathit{ftype}(\mathtt{f}, \mathtt{T}_0) = \mathtt{T}}{\Gamma \vdash \mathtt{e}_0.\mathtt{f}_i : \mathtt{T}} \qquad \text{(T-Field)}
$$

$$
\frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \mathit{mtype}(\mathtt{m}, \mathtt{T}_0) = \mathtt{\overline{T}}{\to}\mathtt{T} \\ \Gamma \vdash \mathtt{\overline{e}} : \mathtt{\overline{S}} \qquad \mathtt{\overline{S}} <: \mathtt{\overline{T}}\end{array}}{\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\mathtt{\overline{e}}) : \mathtt{T}} \qquad \text{(T-Invk)}
$$

$$
\frac{\mathit{fields}(\mathtt{C}) = \mathtt{\overline{T}\ \overline{f}} \qquad \Gamma \vdash \mathtt{\overline{e}} : \mathtt{\overline{S}} \qquad \mathtt{\overline{S}} <: \mathtt{\overline{T}}}{\Gamma \vdash \mathtt{new\ C(\overline{e})} : \mathtt{C}} \qquad \text{(T-New)}
$$

$$
\frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \mathtt{T}_0 <: \mathtt{S}_1{\vee}\mathtt{S}_2 \\ \Gamma, \mathtt{x}_1{:}\mathtt{S}_1 \vdash \mathtt{e}_1 : \mathtt{T}_1 \qquad \Gamma, \mathtt{x}_2{:}\mathtt{S}_2 \vdash \mathtt{e}_2 : \mathtt{T}_2\end{array}}{\Gamma \vdash \mathtt{case\ e_0\ of\ (S_1\ x_1)\ e_1\ \mid\ (S_2\ x_2)\ e_2} : \mathtt{T}_1{\vee}\mathtt{T}_2} \qquad \text{(T-Case)}
$$

Method and class typing:

$$
\frac{\begin{array}{c}\mathtt{\overline{x}} : \mathtt{\overline{T}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e}_0 : \mathtt{S}_0 \qquad \mathtt{S}_0 <: \mathtt{T}_0 \\ \mathtt{class\ C\ extends\ D\ \{\ldots\}} \\ \text{if } \mathit{mtype}(\mathtt{m}, \mathtt{D}) = \mathtt{\overline{U}}{\to}\mathtt{U}_0,\ \text{then } \mathtt{\overline{T}} \cong \mathtt{\overline{U}} \text{ and } \mathtt{T}_0 <: \mathtt{U}_0\end{array}}{\mathtt{T_0\ m(\overline{T}\ \overline{x})\{\ return\ e_0;\ \}} \text{ ok in } \mathtt{C}} \qquad \text{(T-Meth)}
$$

$$
\frac{\begin{array}{c}\mathtt{K} = \mathtt{C(\overline{S}\ \overline{g},\ \overline{T}\ \overline{f})\{super(\overline{g});\ this.\overline{f}{=}\overline{f};\}} \\ \mathit{fields}(\mathtt{D}) = \mathtt{\overline{S}\ \overline{g}} \qquad \mathtt{\overline{M}} \text{ ok in } \mathtt{C}\end{array}}{\mathtt{class\ C\ extends\ D\ \{\overline{T}\ \overline{f};\ K\ \overline{M}\}} \text{ ok}} \qquad \text{(T-Class)}
$$

Reduction:

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

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

$$
\frac{\mathtt{C} <: \mathtt{T}_1}{\begin{array}{c}\mathtt{case\ new\ C(\overline{d})\ of\ (T_1\ x_1)e_1\ \mid\ (T_2\ x_2)e_2} \\ \longrightarrow [\mathtt{new\ C(\overline{d})}/\mathtt{x}_1]\mathtt{e}_1\end{array}} \qquad \text{(R-Case1)}
$$

$$
\frac{\mathtt{C} \not<: \mathtt{T}_1 \qquad \mathtt{C} <: \mathtt{T}_2}{\begin{array}{c}\mathtt{case\ new\ C(\overline{d})\ of\ (T_1\ x_1)e_1\ \mid\ (T_2\ x_2)e_2} \\ \longrightarrow [\mathtt{new\ C(\overline{d})}/\mathtt{x}_2]\mathtt{e}_2\end{array}} \qquad \text{(R-Case2)}
$$

Figure 2: Definition of FJ$\vee$

$\mathtt{D} <: \mathtt{C}\vee\mathtt{D}$. The former relation can be derived by:

(1)	$\mathtt{C} <: \mathtt{D}$	since C extends D
(2)	$\mathtt{D} <: \mathtt{D}$	by reflexivity
(3)	$\mathtt{C}\vee\mathtt{D} <: \mathtt{D}$	by (1), (2), and the last rule.

In what follows, we write $\mathtt{S} \cong \mathtt{T}$ if $\mathtt{S} <: \mathtt{T}$ and $\mathtt{T} <: \mathtt{S}$; $\overline{\mathtt{S}} <: \overline{\mathtt{T}}$ as an abbreviation of $\mathtt{S}_1 <: \mathtt{T}_1, \ldots, \mathtt{S}_n <: \mathtt{T}_n$; and $\overline{\mathtt{S}} \cong \overline{\mathtt{T}}$ as an abbreviation of $\mathtt{S}_1 \cong \mathtt{T}_1, \ldots, \mathtt{S}_n \cong \mathtt{T}_n$.

3.3 Lookup functions

As in FJ, we use auxiliary functions to look up field and method definitions. They include *fields*(C) to enumerate field names of class C with their types, *ftype*(f, T) to look up the type of field f that type T has, *mbody*(m, C) to look up the body of method m in class C, and *mtype*(m, T) to look up the signature of method m that type T has.

On one hand, the definitions of *fields*(C) and *mbody*(m, C), which will be used to define the operational semantics of FJ$\vee$, are straightforward and essentially the same as those in FJ: the former collects all the field declarations with their types from C and its superclasses; and the latter looks for the definition of m by ascending the inheritance chain and returns $\overline{\mathtt{x}}.\mathtt{e}$, in which $\overline{\mathtt{x}}$ are the formal parameters and e is the method body to be evaluated. Note that they take only class names as an input because there is no instance of a union type.

On the other hand, *ftype*(f, T) and *mtype*(m, T) are key functions to realize direct member access on union types in typing. They take types as an argument because the receiver *expression* of a field/method access may be of a union type. There are two rules for *ftype*(f, T). In the case where the type of a field f in class C is retrieved, the result of *fields*(C) is used. When a field f of an expression of a union type $\mathtt{T}_1\vee\mathtt{T}_2$ is accessed, the types of f for the summands are retrieved; if both retrievals succeed, their union is the result type, as described in the last section. For *mtype*(m, T), there are three rules. The first two rules, in which T is a class name, are essentially the same as *mbody*(m, C) except that this function returns the argument types $\overline{\mathtt{S}}$ and return type $\mathtt{S}_0$ in the form of $\overline{\mathtt{S}}\rightarrow\mathtt{S}_0$. The last rule is similar to the second rule of *ftype*(f, T): it is checked that both summands have the method m with the compatible argument types and, if so, the return type is the union of the two return types from the summands.

As mentioned in the last section, if the restriction on the argument types is to be relaxed, *mtype*$(\mathtt{m}, \mathtt{C}_1\vee\cdots\vee\mathtt{C}_n)$ would collect all signatures of m from $\mathtt{C}_i$; the typing rule for method invocations would check if actual argument types match all the possible formal parameter types.

3.4 Typing

A type judgment for an expression is of the form $\Gamma \vdash \mathtt{e} : \mathtt{T}$, read "in the type environment Γ expression e has type T." Here, 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 $\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_1, \ldots, \Gamma \vdash \mathtt{e}_n : \mathtt{T}_n$ to $\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}$.

Thanks to lookup functions, typing rules are simple and essentially the same as FJ. For example, in T-Invk, the method type is retrieved by using the type of the receiver $\mathtt{e}_0$, and it is checked that actual argument types are respectively subtypes of the corresponding formal. The rule T-Case for case analysis may be worth explaining: since each branch covers the case where the value of $\mathtt{e}_0$ is an instance of (a subtype of) $\mathtt{S}_i$, the type of the test expression $\mathtt{e}_0$ must be a subtype of the union of $\mathtt{S}_1$ and $\mathtt{S}_2$. In each branch, $\mathtt{x}_i$ is bound to the object after being tested, thus it can be assumed to have type $\mathtt{S}_i$.

A judgment of method typing is of the form M ok in C, read "method definition M is well formed in class C", derived by T-Meth. It is checked that the given method body expression is well typed under the assumption that formal arguments are of declared types and that this is of C, in which the method is defined. It also checks that the signature of an overriding method is compatible with the overridden; as in Java 5.0, we allow covariant overriding of return types.

Finally, a judgment of class typing is of the form L ok and derived by T-Class, which checks that field types agree with the constructor definition and that all methods are well formed.

3.5 Operational Semantics

The operational semantics is given by the reduction relation of the form $\mathtt{e} \longrightarrow \mathtt{e}'$, read "expression e reduces to expression e′ in one step." Here, we write $[\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{e}/\mathtt{y}]\mathtt{e}_0$ for an expression obtained from $\mathtt{e}_0$ by replacing $\mathtt{x}_1$ with $\mathtt{d}_1, \ldots, \mathtt{x}_n$ with $\mathtt{d}_n$, and y with e. There are four reduction rules, one for field access, one for method invocation, and two for case expressions, of which the last two are new. The rule R-Case1 means that if the first test (whether C is a subtype of T) succeeds, the first branch is taken; and the rule R-Case2 is for the other case. These rules show that the first branch has a precedence over the second when two types overlap. Note that the test $\mathtt{C} <: \mathtt{T}_2$ could be omitted since the type system guarantees that it succeeds; the inclusion of this condition makes the type soundness theorem easier to state. 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. In what follows, we write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

3.6 Type Soundness

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 [14, 7], which are also proved similarly to FJ. (Recall, in the statement of Theorems 2 and 3, that values are defined by: $\mathtt{v} ::= \mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})$, where $\overline{\mathtt{v}}$ can be empty.)

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

Proof. By induction on the derivation of $\mathtt{e} \longrightarrow \mathtt{e}'$. Key lemmas are:

- a lemma that says *ftype*$(\cdot,\cdot)$ is covariant in the second argument, that is, if *ftype*$(\mathtt{f},\mathtt{T}) = \mathtt{U}$ and $\mathtt{T}'{<:}\mathtt{T}$ then, there exists $\mathtt{U}'$ such that *ftype*$(\mathtt{f},\mathtt{T}') = \mathtt{U}'$ and $\mathtt{U}'{<:}\mathtt{U}$;
- a similar lemma for *mtype*$(\cdot,\cdot)$; and
- a substitution lemma that says, if $\Gamma, \overline{\mathtt{x}}{:}\overline{\mathtt{T}} \vdash \mathtt{e} : \mathtt{T}$ and $\Gamma \vdash \overline{\mathtt{d}} : \overline{\mathtt{S}}$ with $\overline{\mathtt{S}} <: \overline{\mathtt{T}}$, then there exists $\mathtt{S}_0$ such that $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e} : \mathtt{S}_0$ and $\mathtt{S} <: \mathtt{T}$.

They are also proved by straightforward induction. □

Theorem 2 (Progress). *If* $\emptyset \vdash \mathtt{e} : \mathtt{T}$ *and* e *is not a value, then there exists* e′ *such that* $\mathtt{e} \longrightarrow \mathtt{e}'$.

Proof. Easy. □

Theorem 3 (Type Soundness). *If* $\emptyset \vdash$ `e : T` *and* `e` $\longrightarrow^*$ `e'` *with* `e'` *being a normal form, then* `e'` *is a value* `new C(`$\overline{\texttt{v}}$`)` *and* `C <: T`.

Proof. An easy consequence of Theorems 1 and 2. □

4. INTERACTIONS WITH OTHER LANGUAGE FEATURES

We briefly discuss interactions of union types with other common language features, found in Java. Although, generics seems fairy easy to combine, field shadowing and overloading have some subtleties.

4.1 Generics and Variant Parametric Types

Obviously, union types are useful to represent heterogeneous collections like lists where each element is a string or integer and it is natural to combine them with generics: a heterogeneous collection is nothing more than a generic collection class instantiated with a union type as the element type parameter. Here, we will argue that variant parametric types [8] (a.k.a. wildcards [13] in Java 5.0) give powerful subtyping.

First, let us briefly review the idea of variant parametric types. In general, one instantiation of a generic class is neither a subtype nor a supertype of a different instantiation of the same class. For example, the fact that `String` is a subtype of `Object` does not mean `List<String>` is a subtype of `List<Object>` since, in general, list elements may be updateable. (Covariant) parametric types, which are of the form `C<+T>`, allow covariant subtyping in the type argument position but do not allow any invocations of methods whose argument types include the type parameter of the class `List`: for example, `List<String>` *is* a subtype of `List<+Object>` but the type system prohibits the invocation of method `setcar()`, which takes `X`—a type parameter of `List`, on `List<+Object>`.

By using this typing mechanism, we can promote, by subtyping, a type of *homogeneous* lists of one type of elements to a type of *read-only heterogeneous* lists consisting of that element type and another. For example, `List<String>` and `List<Integer>` can both be regarded as subtypes of `List<+(String∨Integer)>` since `String <: String∨Integer` and `Integer <: String∨Integer`. Note that neither `List<String>` nor `List<Integer>` should be a subtype of `List<String∨Integer>` (without `+`): if it were allowed, a list of `String`s could be given type `List<String∨Integer>`, which allows to write both strings and integers as elements.

4.2 Field Shadowing and Overloading

Interaction with Java style field shadowing and overloading is more subtle, because there is an inherent conflict between the idea of direct member access and Java's strategy to statically fix the signature of a method or the type of a field to be accessed.

In Java, a subclass can declare a new field, whose name is the same as another in a superclass. In that case, the field in a superclass is hidden by the new field in a subclass and can be accessed only by using upcasting. For example, consider the code below:

```
class Foo { Integer f; ... }
class Bar extends Foo { String f; ... }
Bar bar = ...;
```

Then, `bar.f` accesses the field of `String` declared in `Bar` while `((Foo)bar).f` accesses that of `Integer` in `Foo`. So, static types determine which field to access at run-time.

The behavior of direct member access, however, depends on run-time types. For example, let us consider how the expression `((Bar∨Foo)foo).f` (where `foo` is a variable of `Foo`) should behave. This direct member access, considered merely an abbreviation of

```
case foo of (Bar x) x.f | (Foo y) y.f
```

returns a value of different fields depending on the run-time type of `foo`. We feel that this semantics is natural even though it is different from Java's strategy described above, because it reflects what programmers write explicitly in their code—`Bar∨Foo` can be considered a programmer's intention to distinguish two cases where `foo` is an instance of `Bar` or that of `Foo`. One subtlety of this semantics, though, is that compatible types are not really compatible: `((Foo)foo).f` and `((Foo∨Bar)foo).f` using compatible but syntactically different types in upcasting both always return an `Integer` (recall that the first branch has a precedence)! Another option to avoid all the issues raised above might be to 'canonicalize' the static type `Bar∨Foo` of `foo` into `Foo`, which is a syntactically simplest form among its compatible types, and regard `((Bar∨Foo)foo).f` (and `((Foo∨Bar)foo).f`) as equivalent to `((Foo)foo).f`. We think, however, this option is not very intuitive.

A similar argument applies to overloading in Java, which determines at compile-time the signature of the method to be invoked. For example, in the following code

```
class Foo { void m(Number x){...} }
class Bar extends Foo { void m(Integer x){...} }
Foo foo = ...;
foo.m(new Integer(10));
```

even when `foo` is an instance of `Bar` at run-time, the invoked method is the one defined in `Foo`. Similarly to shadowing, one would expect that, by using a compatible type `Bar∨Foo`,

```
((Bar∨Foo)foo).m(new Integer(10));
```

would invoke `m` defined in `Bar` when `foo` is an instance of `Bar`. So, compatible types `Foo` and `Bar∨Foo` would make different behavior. We have found, however, that the problem of compatible types disappears when direct method invocation requires all possibly invoked methods to have the compatible argument types as in FJ∨: under this rule, the above code will be illegal, as the most specific version of `m` in `Bar` and that of `m` in `Foo` have different argument types. So, the restriction on argument types in direct method invocation is not just for simplicity, rather to avoid subtle behavior due to union types.

5. RELATED WORK

The notion of union types in programming languages can be classified into two categories: tagged (or disjoint) union types and untagged union types. The former, found for instance in ML's datatypes or Pascal's variant records, usually requires an explicit operation of tagging (or constructor applications) to form an expression of a union type, while the latter [1, 10] does not (and uses subtyping and subsumption, instead). A language with tagged unions is equipped with a construct for case analysis to use tagged values, while an untagged union can usually be used with only operations that are valid for both summands. Our union types can be considered a hybrid of the two kinds; thanks to the fact that every object is inherently tagged by the name of the class from which it was instantiated, explicit tagging is not needed to construct an expression of a union type and, furthermore, as we have shown, both case analysis and direct member access are supported. However, in our language, unlike ML datatypes, forming a union of the same type results in a compatible type of the original type, so it is not meaningful to perform case analysis on such a type (the first branch will always be taken).

More recently, untagged union types are studied in the context of programming languages for semi-structured data such as XML [3, 6]. Subtyping supporting "distributivity" of unions over record field types, exemplified as

$$\texttt{\{a:S,b:T\}}\vee\texttt{\{a:U,c:V\}} <: \texttt{\{a:S}\vee\texttt{U\}},$$

($\texttt{\{a:S,b:T\}}$ is a type for records that have a field `a` of type `S` and `b` of `T`), has inspired us in the development of our direct member access mechanism. The type system of Xtatic [4], an extension of C# with mechanisms for native XML processing, is equipped with regular expression types [6, 5], which include the union type constructor. In Xtatic, however, types for XML documents and those for objects are separated and so there are not union types for object types.

The mechanism called intertype declarations [12] to add supertypes to existing classes can be found in the AspectJ language [9], an aspect-oriented extension of Java. However, our direct access mechanism, which allows to access members of the same name but different types, provides more than just the ability to add supertypes.

6. CONCLUSION AND FUTURE WORK

We have discussed a possible introduction of union types for class-based object-oriented languages. Union types can be used to represent a group of classes by forming their supertype *after* those classes are defined. A union type allows direct member access on it by playing a role of an interface consisting of common features of classes. Also, `case` expressions provide exhaustive case analysis on the run-time types of objects; we believe that exhaustive case analysis would be useful even for a language without union types. We have also formalized the core of the type system on top of Featherweight Java and proved that the type system is sound.

Although we expect it is useful as it is, the mechanism of direct member access may be criticized that it heavily depends on member *name* equality, which can be purely coincidental. To remedy the situation, member renaming operations as found in the recent proposal of traits [11] may be combined.

We do not discuss implementation issues in this paper and leave them for future work. Straightforward implementation would be by *erasure* [2, 7]: a union type $\texttt{C}\vee\texttt{D}$ can be translated to a common superclass of `C` and `D` (or simply to `Object`); `case` and direct member access can be expressed in terms of `instanceOf` and downcasts. Efficient implementation of direct member access is an interesting research topic.

Acknowledgments

Comments from anonymous reviewers help improve the final presentation of the present paper. The first author 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 on Priority Areas Research No. 13224013 from MEXT of Japan (Igarashi).

7. REFERENCES

[1] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types: Syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.

[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 *Proceedings of the ACM OOPSLA'98*, pages 183–200, October 1998.

[3] Peter Buneman and Benjamin Pierce. Union types for semistructured data. In *Proceedings of the the International Database Programming Languages Workshop (DBPL7)*, September 1999.

[4] Vladimir Gapeyev, Michael Y. Levin, Benjamin C. Pierce, and Alan Schmitt. The Xtatic experience. In *Proceedings of the Wokrshop on Programming Language Technology for XML (PLAN-X)*, Long Beach, CA, January 2005.

[5] Vladimir Gapeyev and Benjamin C. Pierce. Regular object types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP2003)*, volume 2743 of *Lecture Notes in Computer Science*, pages 151–175, Darmstadt, Germany, July 2003. Springer-Verlag.

[6] Haruo Hosoya, Jérôme Voullion, and Benjamin Pierce. Regular expression types for XML. In *Proceedings of the ACM International Conference on Functional Programming (ICFP'00)*, pages 11–22, September 2000.

[7] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

[8] Atsushi Igarashi and Mirko Viroli. On variance-based subtyping for parametric types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP2002)*, volume 2374 of *Lecture Notes in Computer Science*, pages 441–469. Springer-Verlag, June 2002.

[9] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP2001)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[10] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, February 1991.

[11] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP2003)*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–174, Darmstadt, Germany, July 2003. Springer-Verlag.

[12] The AspectJ Team. The AspectJ programming guide. Available online at `http://www.eclipse.org/aspectj/doc/released/progguide/index.html`.

[13] Mads Torgersen, Christian Plesner Hansen, Peter von der Ahé, Erik Ernst, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. In *Proceedings of the ACM Symposium on Applied Computing (SAC2004)*, pages 1289–1296, March 2004.

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