JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 6, No. 2, Special Issue OOPS Track at SAC 2006, February 2007

Union Types for Object-Oriented Programming

Atsushi Igarashi, Graduate School of Informatics, Kyoto University, Japan
Hideshi Nagira, Graduate School of Informatics, Kyoto University, Japan

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 the 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—fields of the same name and methods with similar signatures—of given types. 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.

1 INTRODUCTION

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 the main mechanisms to promote code reuse. While inheritance enables one class to reuse the 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, the subtyping relation is mostly based on the 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 the benefits of subtyping. However, interfaces cannot be added once a class is defined, so

[1] A notable exception is wildcards [16] in Java 5.0.

Cite this document as follows: Atsushi Igarashi, Hideshi Nagira: *Union Types for Object-Oriented Programming*, in Journal of Object Technology, vol. 6, no. 2, Special Issue OOPS Track at SAC 2006, February 2007, pages 31–45,
http://www.jot.fm/issues/issues_2007_02/article3

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.

In this article, 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 the set union of some given types (viewed as sets of instances that belong to those types) and behaves as their least common supertype. 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 type 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 as elements.

Our contributions in this article can be summarized as follows:

- The proposal of union types for class-based object-oriented languages with a name-based type system; and
- A formalization of a core object-oriented language FJ$\vee$ with union types on top of Featherweight Java [9] with a proof of type soundness of FJ$\vee$.

We also informally discuss how union types will interact with other standard language mechanisms such as generics and method overloading.

The rest of the article 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 language features in Section 4, discuss related work in Section 5, and finally give concluding remarks in Section 6.

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.

A union type can be constructed from any two types `A` and `B` by combining them 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 union of the two sets. Since union types are, unlike class names, not associated with an implementation,

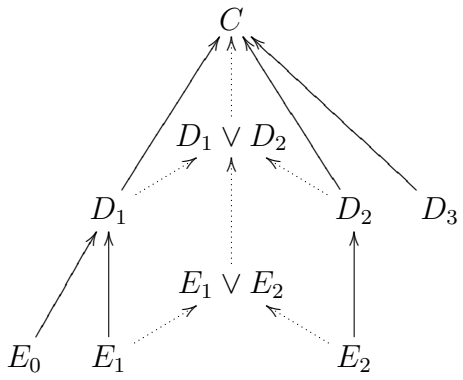


Figure 1: Union Types and Subtyping.

they cannot be used to instantiate objects. So, they are closer to interface types in 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∨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∨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∨B. So,

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

is also allowed. The least subtype property can be explained in terms of the types-as-sets interpretation above: A∨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. In this figure, C, D_i, and E_i are class names and solid arrows represent inheritance relations, which are also subtyping relations. For example, D_1 extends C and so D_1 is a subtype of C. Dotted arrows represent subtyping relations induced by union types: $D_1 \vee D_2$ is a supertype of D_1 and D_2 and also a subtype of C, which is also a common supertype of D_1 and D_2, but D_3 is not related to $D_1 \vee D_2$. Moreover, the union type constructor ∨ preserves subtyping relations of its summands. That is, $E_1 \vee E_2$, a union of subclasses of D_1 and D_2, is a subtype of $D_1 \vee D_2$, which is derived by the leastness condition.

Note that the subtyping relation here is not anti-symmetric as in usual object-oriented languages. In fact, there are two syntactically different types that are subtypes of each other. For example, A∨B and B∨A are syntactically different and subtypes of each other. A more interesting example is C∨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:
  instanceof(Jpg x) { x.draw(); }
  instanceof(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). We adopt the first-matching semantics; if two cases overlap, the first one has a precedence. Here, x and y are bound to the value of im but their static type information is more refined than Jpg∨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(Number 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. There are several plausible conditions when direct method invocation should be allowed. Perhaps, a most restrictive condition is to require the method signatures of the summands be exactly the same; a little relaxed is to require that argument types be compatible (and to allow return types to be different), as formalized in the conference version of this article [8]. Here, we will further relax the condition for the sake of flexibility in programming and allow argument types to be different, as long as each actual argument type is respectively a subtype of the corresponding formal of *both* method signatures. For example,

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

will be well typed because the actual argument type `Integer` is a subtype of both `Integer` and `Number`, which are formal argument types of `zoom()` of `Jpg` and `Gif`, respectively. When return types are object types (not `void`), they can be different as in field access; the type of the whole method invocation will be the union of those return types.

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`$\vee$`Gif` and, moreover, instances of other image formats cannot be mixed (unless they are subclasses of `Jpg` or `Gif`). Compared with a usual idiom of adapter classes to handle this kind of situations, one advantage of using union types is in subtyping—a union type is also a subtype of common supertypes of their summands, while adapter classes introduce new types, not necessarily related to common supertypes of summands.

Before concluding this section, let us compare our union types with union types in other languages. The notion of union types in programming languages can usually 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 [2, 13] does not (and uses subtyping and subsumption, instead). A language with tagged unions is equipped with a case analysis construct 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, 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).

Although we criticized Java-style interface types in the introduction, we think Java-style interfaces 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∨: A FORMAL MODEL OF UNION TYPES

In this section, we formalize union types in a small calculus FJ∨ as an extension of Featherweight Java (FJ) [9], 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. FJ∨ does not model, among other features, field shadowing, overloading, and `super`. Since case analysis can be substituted for typecasts, we have dropped typecasts in FJ∨.

Syntax

The abstract syntax of FJ∨ is as follows:

`L`	::=	`class C extends C {`$\overline{\texttt{T}}$ $\overline{\texttt{f}}$`; K` $\overline{\texttt{M}}$`}`	*classes*
`T, S, U`	::=	`C` | `T`∨`T`	*types*
`K`	::=	`C(`$\overline{\texttt{T}}$ $\overline{\texttt{f}}$`){super(`$\overline{\texttt{f}}$`); this.`$\overline{\texttt{f}}$`=`$\overline{\texttt{f}}$`;}`	*constructors*
`M`	::=	`T m(`$\overline{\texttt{T}}$ $\overline{\texttt{x}}$`){ return e; }`	*methods*
`e`	::=	`x` | `e.f` | `e.m(`$\overline{\texttt{e}}$`)` | `new C(`$\overline{\texttt{e}}$`)`	
	|	`(case e of (T x) e | (T x) e)`	*expressions*

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 "$\overline{\texttt{C}}$ $\overline{\texttt{f}}$" for "$\texttt{C}_1$ $\texttt{f}_1$`,...,`$\texttt{C}_n$ $\texttt{f}_n$", where n is the length of $\overline{\texttt{C}}$ and $\overline{\texttt{f}}$, and "`this.`$\overline{\texttt{f}}$`=`$\overline{\texttt{f}}$`;`" as shorthand for "`this.`$\texttt{f}_1$`=`$\texttt{f}_1$`;...;this.`$\texttt{f}_n$`=`$\texttt{f}_n$`;`" 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 $T_1 \vee T_2$; 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, delegate the initialization of the fields inherited from the superclass (super($\overline{f}$);), and assigns the rest of them to the new fields (this.$\overline{f}$=$\overline{f}$;). As we will see, typing rules enforce this behavior. 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 case e_0 of (T_1 x_1) e_1 | (T_2 x_2) e_2 (in which the syntax is slightly changed from the last section for brevity) first evaluates e_0 to an object new C(...), and execute e_1 if the object is a subtype of T_1 or e_2 if the object is not a subtype of T_1 but T_2, with 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. Here, x_1 and x_2 are bound variables in e_1 and e_2, respectively.

A class table CT is a mapping from class names to class declarations. A program is a pair (CT, 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: (1) $CT(C) =$ class C ... for every $C \in dom(CT)$; (2) for every class name C (except Object) appearing anywhere in CT, we have $C \in dom(CT)$; and (3) 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 $CT(C) =$ class C ....

Lookup functions

As in FJ, we use auxiliary functions, defined in Figure 2, to look up field and method definitions: *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 of T.

The definitions of *fields*(C) and *mbody*(m, C), which will be used to define the operational semantics of FJ∨, 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 $\overline{x}$.e, in which $\overline{x}$ are the formal parameters and e is the method body to be evaluated. Here, m $\notin \overline{M}$ means the method of name m does not exist in $\overline{M}$.

Field lookup:

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

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

Field type lookup:

$$\frac{\mathit{fields}(\texttt{C}) = \overline{\texttt{T}}\ \overline{\texttt{f}}}{\mathit{ftype}(\texttt{f}_i, \texttt{C}) = \texttt{T}_i} \qquad \frac{\mathit{ftype}(\texttt{f}, \texttt{T}_1) = \texttt{U}_1 \qquad \mathit{ftype}(\texttt{f}, \texttt{T}_2) = \texttt{U}_2}{\mathit{ftype}(\texttt{f}, \texttt{T}_1 \vee \texttt{T}_2) = \texttt{U}_1 \vee \texttt{U}_2}$$

Method body lookup:

$$\frac{\texttt{class C extends D \{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \qquad \texttt{S}_0\ \texttt{m(}\overline{\texttt{S}}\ \overline{\texttt{x}}\texttt{)\{ return e; \}} \in \overline{\texttt{M}}}{\mathit{mbody}(\texttt{m}, \texttt{C}) = \overline{\texttt{x}}\texttt{.e}}$$

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

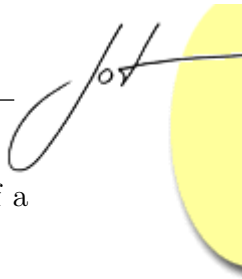
Method type lookup:

$$\frac{\texttt{class C extends D \{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \qquad \texttt{S}_0\ \texttt{m(}\overline{\texttt{S}}\ \overline{\texttt{x}}\texttt{)\{ return e; \}} \in \overline{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{C}) = \{\overline{\texttt{S}} \to \texttt{S}_0\}}$$

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

$$\frac{\mathit{mtype}(\texttt{m}, \texttt{T}_1) = \mathcal{S}_1 \qquad \mathit{mtype}(\texttt{m}, \texttt{T}_2) = \mathcal{S}_2}{\mathit{mtype}(\texttt{m}, \texttt{T}_1 \vee \texttt{T}_2) = \mathcal{S}_1 \cup \mathcal{S}_2}$$

Figure 2: FJ$\vee$ (1)—Lookup functions.



Note that they take only class names as an input because there is no instance of a union type.

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 $T_1 \vee T_2$ 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. There are three rules for *mtype*(m, T), which collects all signatures of m from the summands in T. 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 singleton set consisting of the method signature $\overline{T} \to T$, consisting of argument types $\overline{T}$ and a return type T. The last rule, which is similar to the second rule of *ftype*(f, T), collects signatures from the summands and returns their union. Here, $\mathcal{S}_1$ and $\mathcal{S}_2$ denote a set of signatures.

Typing

The subtype relation S <: T, which is defined in Figure 3, 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 D <: C∨D. The former relation can be derived by the following reasoning:

(1) C <: D since `C extends D`;

(2) D <: D by reflexivity; and

(3) C∨D <: D by (1), (2), and the last subtyping rule.

In what follows, we write S ≅ T if S <: T and T <: S; $\overline{S} <: \overline{T}$ as an abbreviation of $S_1 <: T_1, \ldots, S_n <: T_n$; and $\overline{S} \cong \overline{T}$ as an abbreviation of $S_1 \cong T_1, \ldots, S_n \cong T_n$.

A type judgment for an expression is of the form $\Gamma \vdash e : T$, 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 $\overline{x}:\overline{T}$. We abbreviate a sequence $\Gamma \vdash e_1 : T_1, \ldots, \Gamma \vdash e_n : T_n$ to $\Gamma \vdash \overline{e} : \overline{T}$.

Thanks to lookup functions, typing rules, presented also in Figure 3, are simple and the same as FJ except for T-Invk and T-Case, which require some explanation. In T-Invk, the set of signatures of possible methods is retrieved by using the type of the receiver e_0, and it is checked that actual argument types are respectively subtypes of the corresponding formal *for any signature*. The type of the whole expression is the union of possible return types, which we write $\bigvee_{(\overline{T} \to T \in \mathcal{S})} T$. The

Subtyping:

$$\texttt{T} <: \texttt{T} \qquad \frac{\texttt{S} <: \texttt{T} \qquad \texttt{T} <: \texttt{U}}{\texttt{S} <: \texttt{U}} \qquad \frac{\texttt{class C extends D \{...\}}}{\texttt{C} <: \texttt{D}}$$

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

Expression typing:

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

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

$$\frac{\Gamma \vdash \texttt{e}_0 : \texttt{T}_0 \qquad \mathit{mtype}(\texttt{m}, \texttt{T}_0) = \mathcal{S} \qquad \Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{S}} \qquad \text{for any } (\overline{\texttt{T}}\to\texttt{T} \in \mathcal{S}), \overline{\texttt{S}} <: \overline{\texttt{T}}}{\Gamma \vdash \texttt{e}_0.\texttt{m}(\overline{\texttt{e}}) : \bigvee_{(\overline{\texttt{T}}\to\texttt{T}\in\mathcal{S})} \texttt{T}} \qquad \text{(T-Invk)}$$

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

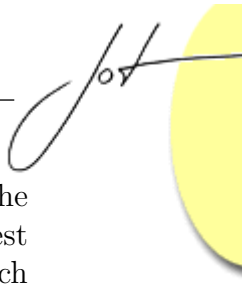
$$\frac{\Gamma \vdash \texttt{e}_0 : \texttt{T}_0 \qquad \texttt{T}_0 <: \texttt{S}_1\vee\texttt{S}_2 \qquad \Gamma, \texttt{x}_1{:}\texttt{S}_1 \vdash \texttt{e}_1 : \texttt{T}_1 \qquad \Gamma, \texttt{x}_2{:}\texttt{S}_2 \vdash \texttt{e}_2 : \texttt{T}_2}{\Gamma \vdash \texttt{case e}_0 \texttt{ of } (\texttt{S}_1\ \texttt{x}_1)\ \texttt{e}_1\ \texttt{|}\ (\texttt{S}_2\ \texttt{x}_2)\ \texttt{e}_2 : \texttt{T}_1\vee\texttt{T}_2} \qquad \text{(T-Case)}$$

Method and class typing:

$$\frac{\overline{\texttt{x}} : \overline{\texttt{T}}, \texttt{this} : \texttt{C} \vdash \texttt{e}_0 : \texttt{S}_0 \qquad \texttt{S}_0 <: \texttt{T}_0 \qquad \texttt{class C extends D \{...\}} \quad \text{if } \mathit{mtype}(\texttt{m}, \texttt{D}) = \{\overline{\texttt{U}}\to\texttt{U}_0\}, \text{ then } \overline{\texttt{T}} \cong \overline{\texttt{U}} \text{ and } \texttt{T}_0 <: \texttt{U}_0}{\texttt{T}_0\ \texttt{m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\texttt{\{ return e}_0\texttt{; \}} \text{ ok in C}} \qquad \text{(T-Meth)}$$

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

Figure 3: FJ$\vee$ (2)—Subtyping and Typing.



rule T-Case for case analysis is explained as follows: since each branch covers the case where the value of e_0 is an instance of (a subtype of) S_i, the type of the test expression e_0 must be a subtype of the union of S_1 and S_2. Since x_i in each branch is bound to the object after being tested, it can be assumed to have type S_i.

Following the tradition of FJ, the usual subsumption rule, which says, if e is of type T and T is a subtype of T′, then e is of type T′, is merged into other rules, such as, the rules for method invocation and case analysis.

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 subtypes 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.

Operational Semantics

The operational semantics is given by the reduction relation of the form $e \longrightarrow e'$, read "expression e reduces to expression e′ in one step." Here, we write $[\overline{d}/\overline{x}, e/y]$ for capture avoiding substitution of $\overline{d}$ and e for $\overline{x}$ and y, respectively. 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 $C <: T_2$ could be omitted since the type system guarantees that it succeeds; the inclusion of this condition makes the type soundness theorem easier to state. In what follows, we write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

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 [17, 9], which are also proved similarly to FJ. (Recall, in the statement of Theorems 2 and 3, that values are defined by: $v ::= \texttt{new C}(\overline{v})$, where $\overline{v}$ can be empty.) See Appendix A for proofs of the first two theorems, of which Theorem 3 is a simple consequence.

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

Computation:

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

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

$$\frac{\mathtt{C} <: \mathtt{T}_1}{\mathtt{case\ new\ C}(\overline{\mathtt{d}})\ \mathtt{of\ (T}_1\ \mathtt{x}_1\mathtt{)e}_1\ \mathtt{|\ (T}_2\ \mathtt{x}_2\mathtt{)e}_2 \longrightarrow [\mathtt{new\ C}(\overline{\mathtt{d}})/\mathtt{x}_1]\mathtt{e}_1} \quad \text{(R-Case1)}$$

$$\frac{\mathtt{C} \not<: \mathtt{T}_1 \qquad \mathtt{C} <: \mathtt{T}_2}{\mathtt{case\ new\ C}(\overline{\mathtt{d}})\ \mathtt{of\ (T}_1\ \mathtt{x}_1\mathtt{)e}_1\ \mathtt{|\ (T}_2\ \mathtt{x}_2\mathtt{)e}_2 \longrightarrow [\mathtt{new\ C}(\overline{\mathtt{d}})/\mathtt{x}_2]\mathtt{e}_2} \quad \text{(R-Case2)}$$

Congruence:

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

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

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

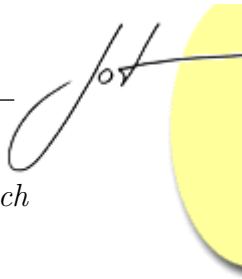
$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i'}{\mathtt{new\ C(}\ldots\mathtt{,e}_i\mathtt{,}\ldots\mathtt{)} \longrightarrow \mathtt{new\ C(}\ldots\mathtt{,e}_i'\mathtt{,}\ldots\mathtt{)}} \quad \text{(RC-New-Arg)}$$

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0'}{\mathtt{case\ e}_0\ \mathtt{of\ (T}_1\ \mathtt{x}_1\mathtt{)e}_1\ \mathtt{|\ (T}_2\ \mathtt{x}_2\mathtt{)e}_2 \longrightarrow \mathtt{case\ e}_0'\ \mathtt{of\ (T}_1\ \mathtt{x}_1\mathtt{)e}_1\ \mathtt{|\ (T}_2\ \mathtt{x}_2\mathtt{)e}_2} \quad \text{(RC-Case)}$$

$$\frac{\mathtt{e}_1 \longrightarrow \mathtt{e}_1'}{\mathtt{case\ e}_0\ \mathtt{of\ (T}_1\ \mathtt{x}_1\mathtt{)e}_1\ \mathtt{|\ (T}_2\ \mathtt{x}_2\mathtt{)e}_2 \longrightarrow \mathtt{case\ e}_0\ \mathtt{of\ (T}_1\ \mathtt{x}_1\mathtt{)e}_1'\ \mathtt{|\ (T}_2\ \mathtt{x}_2\mathtt{)e}_2} \quad \text{(RC-Case1)}$$

$$\frac{\mathtt{e}_2 \longrightarrow \mathtt{e}_2'}{\mathtt{case\ e}_0\ \mathtt{of\ (T}_1\ \mathtt{x}_1\mathtt{)e}_1\ \mathtt{|\ (T}_2\ \mathtt{x}_2\mathtt{)e}_2 \longrightarrow \mathtt{case\ e}_0\ \mathtt{of\ (T}_1\ \mathtt{x}_1\mathtt{)e}_1\ \mathtt{|\ (T}_2\ \mathtt{x}_2\mathtt{)e}_2'} \quad \text{(RC-Case2)}$$

Figure 4: FJ∨ (3)—Operational Semantics.



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

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`.

As a special case of Type Soundness, a "disjunction property," one of important properties of intuitionistic logic with disjunction, holds for FJ$\vee$. Namely, if $\emptyset \vdash$ `e` : `C`$\vee$`D` and `e` reduces to a normal form `e′`, then `e` is of the form `new E(`$\overline{\texttt{v}}$`)` and `E` is a subclass of either `C` or `D`.

4 INTERACTIONS WITH OTHER LANGUAGE FEATURES

We briefly discuss interactions of union types with other common language features, found in Java. Integrating with generics seems fairly straigtforward, while field shadowing and overloading have some subtleties.

Generics and Variant Parametric Types

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, such as `List<Integer∨String>`. Here, we will argue that variant parametric types [10] (a.k.a. wildcards [16] in Java 5.0) give powerful subtyping.

First, let us briefly review the variance problem and 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 generic class. For example, the fact that `String` is a subtype of `Object` does not mean `List<String>` is a subtype of `List<Object>`, as the following code reveals.

```
List<String> list1 = new List<String>("a",null);
List<Object> list2 = list1;     // List<String> <: List<Object> ??
list2.setHead(new Integer(1));  // an Integer is inserted to a string list!
```

Here, `setHead()` is assumed to take an argument of the element type, which is given as a type variable; since `list2` is of type `List<Object>`, this invocation can take any `Object` such as an `Integer`. Notice that this is the same problem as covariant subtyping for array types in Java, in which run-time checks will be performed at every assignment to arrays to ensure run-time safety. Variant parametric types are proposed to ensure static type safety by introducing distinction between invariant and covariant types.[2]

[2]Even contravariant and bivariant types have been introduced [10].

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 `C`: for example, `List<String>` *is* a subtype of `List<+Object>` as in

```
List<String> list1 = new List<String>("a",null);
List<+Object> list2 = list1;        // legal assignment
```

but the type system prohibits the invocation of method `setHead()` on `list2`, since the argument type is a type parameter of `List`. So, `List<+Object>` behaves as if it is a *read-only* list of `Object`s. Giving `list2` the invariant type `List<Object>` would allow the method invocation but assignment of `list1` to `list2` would be prohibited, so type safety is guaranteed statically, without run-time checks.

By using this typing mechanism and union types, 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, both `List<Jpg>` and `List<Gif>` can be regarded as subtypes of `List<+(Jpg∨Gif)>` since `Jpg <: Jpg∨Gif` and `Gif <: Jpg∨Gif`. So, the following method can be applied to a list of `Jpg`s, a list of `Gif`s, or even a heterogeneous list containing `Jpg`s and `Gif`s:

```
void draw_all(List<+(Jpg∨Gif)> l) { for (Jpg∨Gif img : l) l.draw(); }
```

Note that neither `List<Jpg>` nor `List<Gif>` should be a subtype of `List<Jpg∨Gif>` (without `+`), exactly for the same reason above: if it were allowed, a list of `Jpg`s could be given type `List<Jpg∨Gif>`, which allows one to write both `Jpg`s and `Gif`s as elements.

Field Shadowing and Overloading

In the last section, the operational semantics for direct member access on union types was actually nothing more than usual member access because the receiver is eventually evaluated to an object and its run-time class determines which method is invoked. Another possible semantics is to consider a direct member access merely an abbreviation of case analysis: for example, `e.m()` where `e` is an expression of type `S∨T` can be expanded into `case e of (S x) x.m() | (T y) y.m()`. Indeed, in FJ∨, there is no difference between the two semantics and the two expressions above will result in the same value. In Java, however, it would not be really the case, since static type information of a receiver matters in the presence of field shadowing and overloading. Here, we discuss subtleties of those features, along with possible semantics of direct member access.

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.

Now, let us consider how the expressions `((Foo)foo).f`, `((Bar∨Foo)foo).f` and `((Foo∨Bar)foo).f` (where `foo` is a variable of `Foo`) should behave. We believe the first expression should always access `f` declared in `Foo` regardless of `foo`'s run-time class so that the original Java semantics is preserved. As for the second and third, we have come up with three possibilities with different rationales:

- Both `Bar∨Foo` and `Foo∨Bar` are compatible with `Foo` and compatible types should behave the same. Hence, both of them are equivalent to `((Foo)foo).f`, which returns `f` in `Foo`.
- They can be expanded to case expressions by a simple transformation that puts case branches in the same order that summands occur in a type expression. Thus, they are equivalent to `case foo of (Bar x) x.f | (Foo y) y.f` and `case foo of (Foo x) x.f | (Bar y) y.f`, respectively, and so the second can return a different `f`—depending on `foo`'s run-time class—but the third always returns `f` in `Foo` (remember the first-matching semantics of `case`).
- It looks easy to arrange these case branches in an "appropriate" order by taking possible values of `foo` into account and both of them should be equivalent to `case foo of (Bar x) x.f | (Foo y) y.f` so that they return a different `f`.

The first alternative is not very easy to understand because one always has to "canonicalize" types into a simpler form, while the other alternatives mean that compatible types are not really compatible. The second one looks most weird due to the loss of commutativity of ∨, although this very simple expansion scheme is easy to explain. The third one looks most natural at this moment, but, before drawing any conclusion, let us consider method overloading.

In Java, overloading allows one class to have methods of the same name but different signatures. It takes two steps to determine which method to invoke. For example, consider the following code:

```
class Foo { ...
  void m(Number x) {... /* 1 */ }
  void m(Integer x){... /* 2 */ }
}
class Bar extends Foo { ...
  void m(Number x) {... /* 3 */ } // overriding 1
  void m(Integer x){... /* 4 */ } // overriding 2
}
Foo foo = ...;  foo.m(new Integer(10));
```

At compile-time, the signature of the invoked method is determined from static type information; in this case, `void m(Integer x)` will be chosen since it is more "specific" than the other, which takes `Number`, a supertype of `Integer`. So, we can know at compile-time that neither method body marked `/* 1 */` nor `/* 3 */` will be executed. At run-time, the run-time class of the receiver will determine which method body to execute: if `foo` is an instance of `Bar` at run-time, then the method body marked `/* 4 */` will be executed. Note that the compiler signals error if the most specific method signature does not exist: for example,

```
class Baz {
  void m(Integer x, Number y) {...}
  void m(Number x, Integer y) {...}
}
Baz baz = ...;  baz.m(new Integer(1), new Integer(2));
```

will be an error, since both method signatures match the actual argument but neither is more specific than the other. See Ancona, Zucca, and Drossopoulou [1] for more detailed accounts on interactions between overloading and overriding.

Although method invocation like `((Bar∨Foo)foo).m(new Integer(10))` could be discussed exactly the same as field shadowing, there is an even more subtle case when interfaces are taken into account. Consider two interfaces

```
interface I { void m(Integer x); }
interface J { void m(Number x); }
```

and reexamine the three possibilities discussed above for `((I∨J)x).m(new Integer(1))` and `((J∨I)x).m(new Integer(1))`. The first one is not relevant here since `I` and `J` are not comparable. As for the third one, one might imagine an expansion like `case x of (I x) x.m(new Integer(1)) | (J y) y.m(new Integer(1))` and expect it does not matter which case branch comes first. However, it *does* matter, in fact—`x` may be an instance of a class that implements *both* `I` and `J` and, in this case, the first branch will always be taken and different behavior will be exposed, depending on which branch to put first. So, it seems that it is required to investigate even overloaded method signatures in finding an "appropriate" order of branches, which will be a rather complex process.

Having discovered those subtleties, we are inclined to advocate the second alternative (to expand direct member access to case expressions in the order of the occurrences of summands) as the semantics of direct member access in the presence of shadowing and overloading in favor of accessibility, even though it loses *behavioral* compatibility of compatible types and commutativity of the union type constructor.

As a final remark, let us mention that problems in overloading seem to go away when direct method invocation requires all possibly invoked methods to have the compatible argument types (as formalized in the conference version of the paper [8]): under this rule, the above code fragments will be illegal, as the most specific version of `m` in `Foo` (or `I`) and that of `m` in `Bar` (or `J`, respectively) have different argument

types. So, although allowing different signatures for direct method invocation is type safe and would make more flexible programming, it may be worth restricting it (if we have to live with overloading).

5 RELATED WORK

We already compared our union types with similar traditional language features in Section 2. In this section, we briefly discuss other related work in the literature.

Kyas discussed the introduction of union types (as well as their dual notion of intersection types) to the type system for the Object Constraint Language (OCL), which is a formal specification language [12]. However, no formal semantics of the extended OCL is given and so type soundness issues are not discussed.

More recently, untagged union types have been studied in the context of programming languages for semi-structured data such as XML [4, 7]. Subtyping supports distributivity of unions over record field types, exemplified as

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

(`{a:S,b:T}` is a type for records that have a field `a` of type `S` and `b` of `T`). This has inspired us in the development of our direct member access mechanism. The type system of Xtatic [5], an extension of C# with mechanisms for native XML processing, is equipped with regular expression types [7, 6], 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.

A mechanism called intertype declarations [15] to add supertypes to existing classes can be found in the AspectJ language [11], an aspect-oriented extension of Java. Our direct access mechanism, which allows access to 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 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. As far as we have investigated, union types can smoothly integrate with generics and variance (in the style of variant parametric types), but field shadowing and method overloading—already notorious for its complex semantics [1]—have been discovered to expose subtle interactions with 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 [14] may be applied.

We do not discuss implementation issues in this article and leave them for future work. Straightforward implementation would be by *erasure* [3, 9]: a union type `C∨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 article. 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 and Grant-in-Aid for Young Scientists (B) No. 18700026 from JSPS.

REFERENCES

[1] Davide Ancona, Elena Zucca, and Sophia Drossopoulou. Overloading and inheritance. In *Proceedings of the International Workshop on Foundations of Object-Oriented Languages (FOOL8)*, London, England, January 2001.

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

[3] 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 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pages 183–200, October 1998.

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

[5] 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.

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

[7] 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.

[8] Atsushi Igarashi and Hideshi Nagira. Union types for object-oriented programming. In *Proceedings of the 21st Annual ACM Symposium on Applied Computing (SAC2006)*, pages 1435–1441, Dijon, France, April 2006.

[9] 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.

[10] Atsushi Igarashi and Mirko Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM Transactions on Programming Languages and Systems*, 28(5):795–847, September 2006.

[11] 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 *LNCS*, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[12] Marcel Kyas. An extended type system for OCL supporting templates and transformations. In *Proceedings of the 7th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS2005)*, volume 3535 of *LNCS*, pages 83–98. Springer-Verlag, 2005.

[13] 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.

[14] 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 *LNCS*, pages 248–174, Darmstadt, Germany, July 2003. Springer-Verlag.

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

[16] 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 Techonolgy*, 3(11), December 2004. Special issue: OOPS track at SAC 2004, pp. 97–116.

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

A PROOF OF SUBJECT REDUCTION THEOREM

In this appendix, we detail proofs of Subject Reduction Theorem (Theorem 1) and Progress Theorem (Theorem 2). Before giving the proofs, we develop a number of required lemmas. The first three lemmas below can be proved by straightforward induction.

Lemma 1 (Weakening) *If* $\Gamma \vdash \mathtt{e} : \mathtt{T}$, *then* $\Gamma, \mathtt{x} : \mathtt{S} \vdash \mathtt{e} : \mathtt{T}$.

Lemma 2 *If* $\mathit{ftype}(\mathtt{f}, \mathtt{T}) = \mathtt{U}$ *then,* $\mathit{ftype}(\mathtt{f}, \mathtt{S}) <: \mathtt{U}$ *for any* $\mathtt{S} <: \mathtt{T}$.

Lemma 3 *If* $\mathit{mtype}(\mathtt{m}, \mathtt{T}) = \mathcal{S}$, *then, for any* $\mathtt{S} <: \mathtt{T}$ *and* $\overline{\mathtt{U}}' \to \mathtt{U}'_0 \in \mathit{mtype}(\mathtt{m}, \mathtt{S})$, *there exist* $\overline{\mathtt{U}} \to \mathtt{U}_0 \in \mathcal{S}$ *such that* $\overline{\mathtt{U}} \cong \overline{\mathtt{U}}'$ *and* $\mathtt{U}_0' <: \mathtt{U}_0$.

Lemma 4 (Term Substitution Preserves Typing) *If* $\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{e} : \mathtt{U}$, *and* $\Gamma \vdash \overline{\mathtt{d}} : \overline{\mathtt{S}}$ *with* $\overline{\mathtt{S}} <: \overline{\mathtt{T}}$, *then* $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e} : \mathtt{U}'$ *for some* $\mathtt{U}' <: \mathtt{U}$.

Proof: By induction on the derivation of $\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{e} : \mathtt{U}$.

Case T-VAR: $\mathtt{e} = \mathtt{x} \qquad \mathtt{U} = \Gamma(\mathtt{x})$

If $\mathtt{x} \notin \overline{\mathtt{x}}$, then the conclusion is immediate, since $[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{x} = \mathtt{x}$. On the other hand, if $\mathtt{x} = \mathtt{x}_i$ and $\mathtt{U} = \mathtt{T}_i$, then, letting $\mathtt{U}' = \mathtt{S}_i$ finishes the case, because $[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{x} = [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{x}_i = \mathtt{d}_i$ and $\Gamma \vdash \mathtt{d}_i : \mathtt{S}_i$ by assumption.

Case T-FIELD: $\mathtt{e} = \mathtt{e}_0.\mathtt{f}_i \qquad \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{e}_0 : \mathtt{U}_0 \qquad \mathit{ftype}(\mathtt{f}, \mathtt{U}_0) = \mathtt{U}$

By the induction hypothesis, there is some $\mathtt{U}_0'$ such that $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0 : \mathtt{U}_0'$ and $\mathtt{U}_0' <: \mathtt{U}_0$. By Lemma 2, $\mathit{ftype}(\mathtt{f}, \mathtt{U}_0') <: \mathit{ftype}(\mathtt{f}, \mathtt{U}_0)$. Therefore, letting $\mathtt{U}' = \mathit{ftype}(\mathtt{f}, \mathtt{U}_0')$ finishes the case because $\Gamma \vdash ([\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0).\mathtt{f}_i : \mathtt{U}'$ by the rule T-FIELD.

Case T-INVK: $\mathtt{e} = \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) \qquad \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{e}_0 : \mathtt{U}_0 \qquad \mathit{mtype}(\mathtt{m}, \mathtt{U}_0) = \mathcal{S}$
$\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \overline{\mathtt{e}} : \overline{\mathtt{W}} \qquad \forall(\overline{\mathtt{V}} \to \mathtt{U} \in \mathcal{S}).\ \overline{\mathtt{W}} <: \overline{\mathtt{V}} \qquad \mathtt{T} = \bigvee_{\overline{\mathtt{V}} \to \mathtt{U} \in \mathcal{S}} \mathtt{U}$

By the induction hypothesis, there are some $\mathtt{U}_0'$ and $\overline{\mathtt{W}}'$ such that $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0 : \mathtt{U}_0'$ and $\mathtt{U}_0' <: \mathtt{U}_0$ and $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}} : \overline{\mathtt{W}}'$ and $\overline{\mathtt{W}}' <: \overline{\mathtt{W}}$. Let $\mathcal{S}'$ be $\mathit{mtype}(\mathtt{m}, \mathtt{U}_0')$. By Lemma 3, $\overline{\mathtt{W}}' <: \overline{\mathtt{W}} <: \overline{\mathtt{V}}'$ for any $\overline{\mathtt{V}}' \to \mathtt{V}' \in \mathcal{S}'$. Therefore, by the rule T-INVK, $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0.\mathtt{m}([\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}}) : \bigvee_{\overline{\mathtt{V}}' \to \mathtt{V}' \in \mathcal{S}'} \mathtt{V}'$. Finally, by Lemma 3, $\bigvee_{\overline{\mathtt{V}}' \to \mathtt{V}' \in \mathcal{S}'} \mathtt{V}' <: \bigvee_{\overline{\mathtt{V}} \to \mathtt{U} \in \mathcal{S}} \mathtt{U}$.

Case T-NEW: $\mathtt{e} = \mathtt{new}\ \mathtt{D}(\overline{\mathtt{e}}) \qquad \mathit{fields}(\mathtt{D}) = \overline{\mathtt{U}}\ \overline{\mathtt{f}}$
$\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \overline{\mathtt{e}} : \overline{\mathtt{V}} \qquad \overline{\mathtt{V}} <: \overline{\mathtt{U}} \qquad \mathtt{U} = \mathtt{D}$

By the induction hypothesis, there are $\overline{\mathtt{V}}'$ such that $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}} : \overline{\mathtt{V}}'$ and $\overline{\mathtt{V}}' <: \overline{\mathtt{V}}$. Then, $\overline{\mathtt{V}}' <: \overline{\mathtt{U}}$, by transitivity of $<:$. Therefore, by the rule T-NEW, $\Gamma \vdash \mathtt{new}\ \mathtt{D}([\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}}) : \mathtt{D}$.

Case T-CASE: $\mathtt{e} = \mathtt{case}\ \mathtt{e}_0\ \mathtt{of}\ (\mathtt{V}_1\ \mathtt{y}_1)\mathtt{e}_1\ |\ (\mathtt{V}_2\ \mathtt{y}_2)\mathtt{e}_2$ $\quad \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{e}_0 : \mathtt{U}_0$
$\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{y}_i : \mathtt{V}_i \vdash \mathtt{e}_i : \mathtt{U}_i \quad (\text{for } i = 1, 2)$ $\quad \mathtt{U}_0 <: \mathtt{V}_1 \vee \mathtt{V}_2$
$\mathtt{U} = \mathtt{U}_1 \vee \mathtt{U}_2$

By the induction hypothesis, there is $\mathtt{U}_0'$ such that $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0 : \mathtt{U}_0'$ and $\mathtt{U}_0' <: \mathtt{U}_0$. Similarly, there are $\mathtt{U}_1'$ and $\mathtt{U}_2'$ such that $\Gamma, \mathtt{y}_i : \mathtt{V}_i \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_i : \mathtt{U}_i'$ and $\mathtt{U}_i' <: \mathtt{U}_i$ for $i = 1, 2$. By transitivity, $\mathtt{U}_0' <: \mathtt{V}_1 \vee \mathtt{V}_2$. Then, by rule T-CASE, $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e} : \mathtt{U}_1' \vee \mathtt{U}_2'$. Letting $\mathtt{U}' = \mathtt{U}_1' \vee \mathtt{U}_2'$ finishes the case since $\mathtt{U}_1' \vee \mathtt{U}_2' <: \mathtt{U}_1 \vee \mathtt{U}_2$. ∎

Lemma 5 *If* $mtype(\mathtt{m}, \mathtt{C}_0) = \{\overline{\mathtt{T}} \to \mathtt{T}_0\}$, *then, there exist* $\overline{\mathtt{x}}$, $\mathtt{e}$, $\mathtt{D}_0$ *and* $\mathtt{T}_0'$ *such that* $mbody(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{x}}.\mathtt{e}$ *and* $\mathtt{C}_0 <: \mathtt{D}_0$ *and* $\overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{D}_0 \vdash \mathtt{e} : \mathtt{T}_0'$ *and* $\mathtt{T}_0' <: \mathtt{T}_0$.

Proof: By induction on the derivation of $mtype(\mathtt{m}, \mathtt{C}_0)$. The base case (where $\mathtt{m}$ is defined in $\mathtt{C}_0$) is easy, since $\mathtt{m}$ is defined in $CT(\mathtt{C}_0)$ and $\overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C}_0 \vdash \mathtt{e} : \mathtt{T}_0'$ by T-METHOD. The induction step is also straightforward. ∎

Proof of Subject Reduction Theorem: By induction on a derivation of $\mathtt{e} \longrightarrow \mathtt{e}'$, with a case analysis on the reduction rule used.

Case R-FIELD: $\mathtt{e} = \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{e}}).\mathtt{f}_i \quad \mathtt{e}' = \mathtt{e}_i \quad fields(\mathtt{C}_0) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$

By rule T-FIELD and the definition of *ftype*, we have $\Gamma \vdash \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{e}}) : \mathtt{D}_0$ and $\mathtt{T} = \mathtt{T}_i$ for some $\mathtt{D}_0$. Again, by the rule T-NEW, $\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}}$ and $\overline{\mathtt{S}} <: \overline{\mathtt{T}}$ and $\mathtt{D}_0 = \mathtt{C}_0$. In particular, $\Gamma \vdash \mathtt{e}_i : \mathtt{S}_i$, finishing the case, since $\mathtt{S}_i <: \mathtt{T}_i = \mathtt{T}$.

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

By the rules T-INVK and T-NEW, we have $\Gamma \vdash \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{e}}) : \mathtt{C}_0$ and $mtype(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{T}} \to \mathtt{T}$ and $\Gamma \vdash \overline{\mathtt{d}} : \overline{\mathtt{S}}$ and $\overline{\mathtt{S}} <: \overline{\mathtt{T}}$ for some $\overline{\mathtt{S}}$ and $\overline{\mathtt{T}}$. By Lemma 5, $\overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{D}_0 \vdash \mathtt{e}_0 : \mathtt{T}_0$ for some $\mathtt{D}_0$ and $\mathtt{T}_0$ where $\mathtt{C}_0 <: \mathtt{D}_0$ and $\mathtt{T}_0 <: \mathtt{T}$. By Lemma 1, $\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{D}_0 \vdash \mathtt{e}_0 : \mathtt{T}_0$. Then, by Lemma 4, $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{e}})/\mathtt{this}]\mathtt{e}_0 : \mathtt{T}_0'$ for some $\mathtt{T}_0' <: \mathtt{T}_0$. Then $\mathtt{T}_0' <: \mathtt{T}$ by transitivity of $<:$. Finally, letting $\mathtt{T}' = \mathtt{T}_0'$ finishes this case.

Case R-CASE1: $\mathtt{e} = \mathtt{case}\ \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{d}})\ \mathtt{of}\ (\mathtt{T}_1\ \mathtt{x}_1)\mathtt{e}_1\ |\ (\mathtt{T}_2\ \mathtt{x}_2)\mathtt{e}_2 \quad \mathtt{C}_0 <: \mathtt{T}_1$
$\mathtt{e}' = [\mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{d}})/\mathtt{x}_1]\mathtt{e}_1$

By the rules T-CASE and T-NEW, we have $\Gamma \vdash \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{d}}) : \mathtt{C}_0$ and $\Gamma, \mathtt{x}_i : \mathtt{T}_i \vdash \mathtt{e}_i : \mathtt{S}_i$ for $i = 1, 2$ and $\mathtt{T} = \mathtt{S}_1 \vee \mathtt{S}_2$. By Lemma 4, there exists $\mathtt{S}_1'$ such that $\Gamma \vdash [\mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{d}})/\mathtt{x}_1]\mathtt{e}_1 : \mathtt{S}_1'$ and $\mathtt{S}_1' <: \mathtt{S}_1$. Letting $\mathtt{T}' = \mathtt{S}_1'$ finishes the case.

The case for R-CASE2 is similar. The cases for congruence rules are easy. ∎

Proof of Progress Theorem: Since $\mathtt{e}$ is a closed non-value term, we have the following cases, each of which are easy to prove:

(i) The case where $\mathtt{e}$ contains a subterm of the form $\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}).\mathtt{f}$: Then, by T-FIELD, there exist $\overline{\mathtt{T}}$ and $\overline{\mathtt{f}}$ such that $fields(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ and $\mathtt{f} \in \overline{\mathtt{f}}$ and $|\overline{\mathtt{f}}| = |\overline{\mathtt{e}}|$.

(ii) The case where e contains a subterm of the form new C($\overline{e}$).m($\overline{d}$): Then, by Lemma 5, there exist $\overline{x}$ and e_0 such that $mbody(m, C) = \overline{x}.e_0$ and $|\overline{x}| = |\overline{d}|$.

(iii) The case where e contains a subterm of the form case new C($\overline{e}$) of (T_1 x_1)d_1 | (T_2 x_2)d_2: Then, by T-Case, either C<:T_1 or C<:T_2 holds. ∎

ABOUT THE AUTHORS

Atsushi Igarashi is an associate professor at Kyoto University. His home page is at http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/.

Hideshi Nagira is a Ph.D student at Graduate School of Informatics, Kyoto University.