JOURNAL OF OBJECT TECHNOLOGY

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

Vol. 7, No. 1, January–February 2008

Nominal and Structural Subtyping in Component-Based Programming

Klaus Ostermann, University of Aarhus, Denmark

In nominal type systems, the subtype relation is between *names* of types, and subtype links are explicitly declared. In structural type systems, names are irrelevant; in determining type compatibility, only the *structure* of types is considered, and a type name is just an abbreviation for the full type. We analyze structural and different flavors of nominal subtyping from the perspective of component-based programming, where issues such as blame assignment and modular extensibility are important. Our analysis puts various existing subtyping mechanisms into a common frame of reference and delineates the frontiers of the subtyping design space. In addition, we propose a new subtyping definition in one particularly interesting corner of the design space which combines the safety of nominal subtyping with the flexibility of structural subtyping.

1 INTRODUCTION

The choice of a nominal or a structural type system is one of the important characteristics of any object-oriented language. In a structural system, an object type t_1 is a subtype of a type t_2 if t_1 is structurally compatible with t_2, meaning that t_1 has at least all methods and fields (*width subtyping*) of t_2, whereby the signatures of the methods and fields in t_1 and t_2 have to be compatible (*depth subtyping*). In the easiest case, *compatible* means *identical*; in type systems supporting some kind of covariance [8] it can be something more sophisticated. Structural type systems are very common in type-theoretic research such as [10, 8, 29] and research languages like Ocaml [22]. In nominal systems, the subtyping relation is a subset of the structural subtyping relation. In addition to structural compatibility, there has to be a declared link between the two types. For example, a type `Point` with methods `int getX()` and `int getY()` is a supertype of `ColorPoint`, which has a method `Color getCol()` in addition to `getX` and `getY`, only if `ColorPoint` explicitly declares (directly or indirectly) that it is a subtype of `Point`. Nominal type systems are used in many main-stream object-oriented languages. In Java [2], for example, the `implements` or `extends` clause is used to declare subtypes; similar constructs exist in many other languages.

Since the nominal subset relation is usually only a subset of the structural subset relation, it is often claimed that structural systems offer more possibilites for software reuse [15, 4, 25, 29]. We will indeed identify several examples where nominal subtyping is an obstacle to reuse. However, we want to shed new light on this debate by analyzing the situation from a component-based perspective; that is, we consider the situation where different parts of the software are made and evolve independently of each other. The main weakness of structural subtyping is that a design decision to make a type a subtype of another one is not documented in the code but is only very implicitly represented by structural compatibility of the types. In particular, there is no place in the code (and hence programmer) which accepts responsibility for the subtype relation - this is the reason why it is hard to assign blame correctly. Our analysis makes this argument very concrete and shows that structural subtyping leads to fragile code: blame cannot be assigned correctly if subtype relations change due to changes in a part of the program.

As a result of the analysis, we develop a concrete set of desiderata for subtyping that make the difference between structural and (different flavors of) nominal subtyping very concrete and can be used to evaluate and compare different subtyping definitions.

These desiderata have been the motivation for the design of a new more flexible model of nominal subtyping which combines its advantages w.r.t. stability and assignment of blame with the flexibility of structural subtyping. In this model, subtype declarations do not need to be part of the declaration of the subtype; they can also be declared in a supertype, or even outside both the subtype and the supertype. Every place in the program which makes use of a subtype relation (i.e., performs an upcast) must refer to a "witness" of the subtype relation: the place in the program where the subtype relation has been declared. We make this model concrete by describing it as an extension to Featherweight Java [17], a small object-oriented language in the Java style, which is well-suited to illustrate the semantics of our proposal.

The primary contributions of this work are:

- A new perspective on subtyping from the point of view of component-based design and blame assignment.
- A concrete set of subtyping desiderata together with an evaluation of different subtyping mechanisms by means of these desiderata. This evaluation gives new insights into the relation between those subtyping mechanisms.
- The design and formal specification of a new powerful model of nominal subtyping which allows to express many subtyping relationships that could previously only be established with structural subtyping without sacrifying the advantages of nominal subtyping. A sketch of the soundness proof is also given.

The remainder of this paper is structured as follows. In Sec. 2 we analyze the interrelation between subtyping and blame assignment. In Sec. 3 we discuss scenarios in which the conventional nominal subtyping model is an obstacle to software reuse. Sec. 4 abstracts from the examples and discusses general desiderata for subtyping. In Sec. 5 we give an informal overview and assessment of our new model of nominal subtyping. In Sec. 6 this model is made precise in the form of an extension to Featherweight Java and shown to be sound. Sec. 7 compares the new model with various other subtyping mechanisms by means of the desiderata from Sec. 4. Sec. 8 discusses other related work. Sec. 9 concludes.

2 SUBTYPING AND BLAME ASSIGNMENT

Consider the following scenario. We have four different components in our program : A component which defines a type Stack with methods like push and pop (the *stack* component), another component which defines a type FastStack which has similar methods as Stack (the *fast stack component*), a *stack client* component which offers a method service(Stack s) requiring a Stack, and finally a *user* component which has access to a FastStack fs and to a StackClient sc and performs an upcast from FastStack to Stack by calling sc.service(fs).

Let us at first assume that the stack component and the fast stack component have been developed independently of each other. In a nominal system, FastStack is then not a declared subtype of Stack, hence the upcast in the user component fails - we cannot combine FastStack with StackClient even if the interfaces of Stack and FastStack are identical. Let us now consider a structural system. If the interfaces are not compatible (e.g., the pop method is called top in FastStack, we are no better off than in the nominal case. The upcast succeeds only if FastStack is a structural subtype of Stack. However, the important point is that this is highly unlikely in this scenario! Names in independently developed interfaces match only by accident. And even if the names would match, it would be highly unlikely that one of the types is a behavioral subtype (Liskov substitution principle [24]) of the other one. Hence we can conclude that structural subtyping offers no advantages over nominal subtyping in this case.

Now assume that FastStack has explicitly been designed to be a subtype of Stack. In this case, we can safely assume that the names match and that FastStack is also behaviorally compatible with Stack. In the nominal system, the design decision to make FastStack a subtype of Stack is also documented in a declared supertype link in FastStack. The combination of FastStack with StackClient in User works with both structural and nominal subtyping.

The situation gets more interesting if we consider changes to Stack or FastStack. If one of these components change, such that they are no longer subtypes of each other, the fast stack component should be blamed, because the designers of FastStack accepted the responsibility for making FastStack a subtype of Stack. The blame is correctly assigned in a nominal system because the compiler will announce that the declared subtype relationship in FastStack does not hold. In a structural system, however, the user component is always (incorrectly) blamed; both the Stack and the FastStack component can still be compiled without an error but the upcast in the user component will lead to an error.

As a final scenario, let us consider the case that FastStack is pre-existing and the programmers of Stack intend to design Stack to be a supertype of FastStack. In a nominal type system such as Java, this cannot be expressed because we would have to change the definition of FastStack in order to make Stack a supertype of FastStack. With a structural type system, the required supertype relation can be established as desired, but the problem with incorrectly assigned blame remains: Although the designers of Stack are responsible for maintaining the supertype relation, the user component will be blamed whenever Stack or FastStack change.

To summarize: Structural subtyping does not have means to express who accepts responsibility for maintaining a subtype relation. Hence blame cannot be assigned correctly. The practical consequence is that the cause of the error and the place where the error occurs are separated in time and space – a highly undesirable situation in a component-based setting. In a nominal system, a type can accept responsibility for maintaining a subtype relation by a corresponding supertype declaration, but the other way around (accepting responsibility for maintaining a supertype relation) is not possible.

```
class Hello {
  String sayHello() { return "Hello"; }
}
class RemoteHello {
  String sayHello() { /* send method call to remote host */ }
}
```

Figure 1: RemoteHello is not a subtype of Hello

```
interface Widget {...}
interface ColoredWidget extends Widget { Color getColor(); }
interface BorderWidget extends Widget {
   void setBorderWidth(int n); }
interface ResizeableWidget extends Widget {
   void setSize(int n); }

class WidgetImp implements Widget {...}

class SomeWidget extends WidgetImp
  implements ColoredWidget, BorderWidget, ResizeableWidget {...}

interface ColoredBorderWidget implements
                      ColoredWidget, BorderWidget {}
interface BorderResizeableWidget implements
                      BorderWidget, ResizeableWidget {}
interface ColoredResizeableWidget implements
                      ColoredWidget, ResizeableWidget {}
```

Figure 2: SomeWidget is not a subtype of the three types at the bottom

3 SUBTYPING AND REUSE

The last scenario in the previous section showed a situation where we cannot express the desired subtype relation with nominal typing. This is a substantial problem, and in this section we will elaborate on it with a set of three examples illustrating the problem in detail. We use Java-like syntax [2] in this paper, but the ideas are easily applicable to other statically typed object-oriented languages.

The first problem is very simple: In most object-oriented nominal type-systems, classes are also types, but we cannot create a subtype of a class without inheriting from it. This is illustrated in Fig. 1: If we have objects of the class Hello that we want to make available on a remote host, then a proxy (stub) class which is responsible for sending method calls to the remote host, cannot be made a subtype of the Hello, and hence cannot be used transparently everywhere where a Hello object is expected. The only way to make RemoteHello a subtype of Hello is to make it a subclass - but that is obviously completely undesirable in this example.

```
package java.util;
interface Collection {
  boolean contains(Object o);
  boolean isEmpty();
  void clear();
  ...
}
class ArrayList implements Collection {...}

package mypackage;
interface ReadOnlyCollection {
  boolean contains(Object o);
  boolean isEmpty();
}
class ReadOnlyList implements ReadOnlyCollection { ... }
```

Figure 3: ArrayList is not a subtype of ReadOnlyCollection

The second problem of nominal type systems is that it is not possible to specify that an object should have a specific *set* of interfaces. Types are frequently used to model certain features of an object. If more than one feature is required of, say, an object passed as method argument, a new type needs to be defined which stands for the combination of the two features. However, an object which indeed has both types is not a nominal subtype of the combination type except if it is explicitly implemented by the object. The latter strategy requires preplanning and leads to an exponential growth of the number of interfaces. This is illustrated in Fig. 2. The three interfaces at the bottom denote sets of features, but the component SomeWidget which has all the features is not a subtype of these interfaces. SomeWidget could explicitly declare that it is a subtype of these interfaces, but this would require us to preplan and provide an interface for every possible combination of features. Since the number of combinations grows exponentially with the number of features, this approach does not scale.

The third problem is related to the asymmetry of subtype declarations hinted at at the end of Sec. 2. The problem is that in nominal systems one cannot document that only a part of an existing interface is needed for some purpose, and it is not possible to implement new classes having this partial interface. Consider the library types Collection and ArrayList in Fig. 3. Assume we have a component Service which requires only a well-defined subset of the features defined in Collection, such as only those methods that do not change the collection, as indicated in the ReadOnlyCollection interface. With nominal subtyping, we cannot document this property because we cannot make ReadOnlyCollection a supertype of Collection without changing the definition of the latter.

Worse, we might want to create new classes such as ReadOnlyList which has only the features required by Service. It is impossible, however, to use Service with both ReadOnlyList and ArrayList because we cannot declare an appropriate subtype relationship between them without changing the Collection interface.

4 SUBTYPING DESIDERATA

The previous two sections showed different examples that illustrate problems of nominal and structural subtyping in a component-based setting. Now we put these examples into perspective and extract four principles that we believe to be critical for sound and reliable subtyping in large-scale software development. These principles are also concrete criteria which can be used to assess subtyping mechanisms.

Flexible Assignment of Responsibility

Definition

Whether or not an upcast from a type S to T goes wrong depends on three parties: The party performing the upcast, the definition of S, and the definition of T. We say that a subtyping mechanism allows flexible assignment of responsibility if the programmer(s) can control which of these parties will be blamed if the upcast goes wrong.

Rationale

In a component-based setting, proper responsibility assignment (and hence blame assignment) needs to be possible for every important design decision. Sect. 2 showed that all three scenarios (any of the three parties can accept the responsibility) make sense.

Modular extensibility of subtyping relation

Definition

Let P be a program and $<:$ the subtype relation in P. We say that $<:$ is *subtype-open* if it is possible to add type definitions $T$to P such that T is a subtype of an existing type in P without changing P. Similarly, $<:$ is *supertype-open* if it is possible to add type definitions $T$to P such that T is a supertype of an existing type in P without changing P.

Let S and S' be two types in P such that S is a structural subtype of S'. We say that $<:$ is *subtype-supertype-open*, if it is possible to add type definitions $T'$to P such that $S <: T'$ and $T' <: S'$. If we view the subtyping relation of a program as a directed acyclic graph, a subtype-open subtype relation means that the graph can be extended with (sets of connected) nodes that have edges into the existing graph but not vice versa. The same property with the reverse direction of the edges holds for a supertype-open subtype relation. In a subtype-supertype-open subtype relation, we can have edges both in both directions. Note that a subtype-open and supertype-open subtype relation is not necessarily subtype-supertype-open.

Rationale

Modular extensibility (extension without changing code) is a prerequisite for component-based programming. Java-like languages are only subtype-open, but the previous section has shown different examples where this was too limiting.

Unique name introduction

Definition

In object-oriented languages, every method or field in a type has a name. Based on the overloading rules of the language, the signature of the field/method may be part of the name or not. We distinguish between a *name introduction* and a *name usage*. We say that a name m is *introduced* in a type T, if it has not been defined in a supertype S of T, otherwise we way that the m is *used* in T. A subtyping mechanism has the *unique name introduction* property, if, for every type T and every name m, the set of supertypes of T contains at most one name introduction of m.

Rationale

A name introduction can be seen as an implicit definition (in informal comments or only in the mind of the programmer) of the meaning of a name - which is more than a type signature. Since the meaning of a (method) name is more than its signature, it is unlikely that two separate name introductions mean "the same thing", even if their signatures match. A clean distinction between name introductions and name usages is also well-known from λ-calculus, where we are used to the fact that α-conversion (consistent renaming of bound variables) does not change the meaning of a program. Unique name introductions guarantee a similar property: changing a name introductions and all name usages that (uniquely) refer to this name introduction does not change the meaning of the program.

Traceability

Definition

For a type T, let $trace(T)$ be the set of types reachable from T following the declared sub-/supertype links in T. We say that a subtype relation <: is *traceable*, if $S <: T$ depends only on $trace(S) \cup trace(T)$. If there are no super-/suptype declarations, then $trace(T) = \{T\}$.

Rationale

Traceability is important for modular understandability of the code - if a programmer wants to understand why an upcast is correct or not, he can systematically follow the type declarations and does not need a global view on the code. For the same reason, traceability is also important for modular type-checking of the code.

```
class Hello {
  String sayHello() { return "Hello"; }
}
class RemoteHello implements Hello {
  String sayHello() {
     /* method stub generated by RMI compiler */
  }
}
```

Figure 4: RemoteHello is a subtype of Hello

5 OVERVIEW OF OUR PROPOSAL

Based on the desiderata defined in the previous section, we have developed a proposal which addresses these desiderata. Before going into the exact details of the proposal, we will give an overview of its refined subtyping mechanism. Later (in Sec. 7) we will compare the proposed mechanism with respect to the proposed desiderata and related work.

Our proposal differs from the traditional nominal subtyping model such as in Java or C# in three regards:

- The type of a class can be used independently of the class itself. In particular, it is possible to create subtypes of a class type without inheriting the implementation.
- It is possible to define interface sets. Subtyping of interface sets is structural, that is, an interface set is a subtype of another one iff their interface sets are in a subset relation.
- It is possible to declare supertypes of an existing type.

The first two extensions are relatively straightforward and are basically variants or minor improvements over previous proposals such as [30, 6]; the third one is a more fundamental extension of an idea from Sather [35, 33]. The reason why we discuss all three extensions is twofold: First, in order to have a simple and formalized realization of these ideas for the subsequent discussion, detached from the context and languages in which these extensions were proposed. Second, because their *combination* marks an interesting extremal point in the subtyping design space, as we will detail in Sec. 7.

Fig. 4 illustrates the first extension. A class can be freely used as a type in implements clauses, hence the class RemoteHello is a subtype of Hello, but does not inherit its implementation. This would be unsound if public instance variables were allowed because a RemoteHello object does not have the instance variables of Hello. We prevent this problem by disallowing public instance variables; all access to instance variables is allowed via **this** only. This is not a serious restriction because public instance variables can easily be simulated with getter/setter methods.

The second extension is illustrated in Fig. 5. An interface stands for a set of other interfaces if it does not define methods headers itself but instead ends its declaration with a semicolon, as illustrated by the three declarations at the bottom of Fig. 5. The class

```
interface Widget {...}
interface ColoredWidget extends Widget {
  Color getColor();
}
interface BorderWidget extends Widget {
  void setBorderWidth(int n);
}
interface ResizeableWidget extends Widget {
  void setSize(int n);
}

class WidgetImp implements Widget {...}

class SomeWidget extends WidgetImp
  implements ColoredWidget, BorderWidget, ResizeableWidget {...}

interface ColoredBorderWidget
     implements ColoredWidget, BorderWidget;
interface BorderResizeableWidget
     implements BorderWidget, ResizeableWidget;
interface ColoredResizeableWidget
     implements ColoredWidget, ResizeableWidget;
```

Figure 5: SomeWidget is a subtype of all three interface sets at the bottom

SomeWidget is a subtype of all these interfaces even though a subtype relation between these types is not explicitly declared. This is because we make a structural comparison: is the set of interfaces a subset of the set of interfaces declared in the interface set specification?

The third extension is the possibility to declare supertypes by means of an **extendedBy** clause, illustrated in Fig. 6. The class ReadOnlyCollection declares that it is a supertype of Collection. This is accepted by the type checker only if the set of methods of ReadOnlyCollection (declared in the interface definition or inherited from superinterfaces) are structurally a supertype of the methods of Collection. If we have some method which accepts objects of type ReadOnlyCollection we can use the method with instances of both ArrayList and ReadOnlyList. The extendedBy declaration means that ReadOnlyCollection accepts the responsibility of maintaining the supertype relation, hence blame can be assigned correctly if Collection or ReadOnlyCollection change.

The extendedBy clause has a number of fundamental implications. Consider the code in Fig. 7, which adds an inherited interface to ReadOnlyCollection. The method m1 of the client code accepts an instance of Collection, which is, through the declaration of ReadOnlyCollection, a subtype of ReadOnlyObject. However, if we would allow a direct upcast to ReadOnlyObject as in the call m2(c), we would need a closed world assumption to type-check this call: There is no direct or indirect link from Collection to ReadOnlyObject. The type ReadOnlyCollection refers to both types, but there is no direct or indirect reference from the two types to ReadOnlyCollection. In the terminology of Sec. 4, the traceability

```
package java.util;
interface Collection {
  boolean contains(Object o);
  boolean isEmpty();
  void clear();
  ...
}
class ArrayList implements Collection {...}

package mypackage;
interface ReadOnlyCollection extendedBy Collection {
  boolean contains(Object o);
  boolean isEmpty();
}
class ReadOnlyList implements ReadOnlyCollection { ... }
```

Figure 6: ArrayList is a subtype of ReadOnlyCollection due to the extendedBy declaration

```
interface ReadOnlyCollection extends ReadOnlyObject
                             extendedBy Collection { ... }

class Client {
  void m1(Collection c) {
    // m2(c); -- static error
    ReadOnlyCollection d = c; //ok
    m2(d); // ok
  }
  void m2(ReadOnlyObject r) { ... }
}
```

Figure 7: The subtype relation is not transitive

of the typing relations would be lost. This is not just a question of whether the type-checker needs a global view of the program, it also means that a programmer who wants to understand the code needs a global view of the system in order to understand why an upcast is correct or not.

The declaration of ReadOnlyCollection is the "witness" of the subtype relation, hence each place in the code which makes use of this subtype relation must refer (via a traceable list of types) to this witness. This can be done by making the upcast in two steps: First, by upcasting the object to a variable of type ReadOnlyCollection. This does not violate traceability because there is a link from ReadOnlyCollection to Collection. Second, by upcasting the temporary object to ReadOnlyObject. Again, this does not violate traceability because there is a declared link from ReadOnlyCollection to ReadOnlyObject. In general, the type checker makes sure that only those upcasts can be performed where the "proofs" that the subtype relation is ok can be reached from the types of the object that are compared by

```
interface Collection { ... }
class ArrayList implements Collection {...}

interface ReadOnlyCollection extends ReadOnlyObject { ... }

class ReadOnlyList implements ReadOnlyCollection { ... }

interface CollectionIsaROC implements ReadOnlyCollection
                           extendedBy Collection;

class Client {
  void m1(Collection c) {
    // m2(c); -- static error
    CollectionIsaROC d = c; //ok
    m2(d); // ok
  }
  void m2(ReadOnlyObject r) { ... }
}
```

Figure 8: CollectionIsaROC is a witness for the subtype relation between Collection and ReadOnlyCollection

following super/subclass declarations. In particular, this means that the subtype relation is not transitive.

Together with the interface set mechanism, we can go even one step further and separate the declaration of the subtype relation completely from the type or its subtype. Fig. 8 shows a variant of ReadOnlyCollection that does not contain an extendedBy link to Collection; neither is Collection declared as a subtype of ReadOnlyCollection. The separate interface declaration CollectionIsaROC links these two types together and declares a subtype relation between them. Since there is no direct link between the two types, an upcast has to be performed in two steps again, as illustrated by the Client code in Fig. 8. If the subtype relation does not hold despite the declaration in CollectionIsaROC, CollectionIsaROC is blamed because it is the declaration where the subtype claim is made.

6 THE FORMAL MODEL

In this section we define the semantics of the new subtyping constructs precisely as an extension to Featherweight Java (FJ) [17]. Our extension is called $\text{FJ}_{<:}$[1].

The formal syntax of $\text{FJ}_{<:}$ is defined in Fig. 9. The formal definitions use a number of syntactic conventions. A bar above a metavariable denotes a list: $\overline{\mathsf{T}}$ stands for $\mathsf{T}_1, ..., \mathsf{T}_k$ for some natural number $k \geq 0$. If $k = 0$ then the list is empty. The length of $\overline{\mathsf{T}}$ is $|\overline{\mathsf{T}}|$. Following common convention [17], $\overline{\mathsf{T}}\ \overline{\mathsf{f}}$ represents a list of pairs $\mathsf{T}_1\ \mathsf{f}_1 \cdots \mathsf{T}_k\ \mathsf{f}_k$ rather than

[1]A pun on the theoretical calculus System $F_{<:}$ [10], which has been used a lot in the study of subtyping.

Grammar

CL	::=	**class** C **extends** C **implements** $\overline{\mathsf{T}}$ { $\overline{\mathsf{T}}\ \overline{\mathsf{f}}$; K $\overline{\mathsf{M}}$ }	*class declaration*
IF	::=	**interface** J **extends** $\overline{\mathsf{T}}$ **extendedBy** $\overline{\mathsf{T}}$ { $\overline{\mathsf{H}}$ }	*interface declaration*
IF	::=	**interface** J **extends** $\overline{\mathsf{T}}$ **extendedBy** $\overline{\mathsf{T}}$;	*interface set declaration*
K	::=	C($\overline{\mathsf{T}}\ \overline{\mathsf{f}}$) { **super**($\overline{\mathsf{f}}$); **this**.$\overline{\mathsf{f}}$ = $\overline{\mathsf{f}}$; }	*constructor*
H	::=	T m($\overline{\mathsf{T}}\ \overline{\mathsf{x}}$)	*method header*
M	::=	H { **return** e; }	*method declaration*
e	::=	x | f | e.m($\overline{\mathsf{e}}$) | **new** C($\overline{\mathsf{e}}$) | (T) e	*expressions*

Identifiers

C, D	*class names*
J	*interface names*
S,T,U,V ::= C | J	*types*
f,g	*field names*
m	*method names*
x	*argument names or* **this**

Figure 9: Syntax of $\mathrm{FJ}_{<:}$

a pair of lists. Sometimes we will abuse syntax and confuse list notation with set notation, e.g., write $\mathsf{T} \in \overline{\mathsf{T}}$, but the meaning will always be clear from the context.

A class declaration consists of the keyword **class**, a class name, a superclass declaration, a list of implemented interfaces, a list of field declarations, a constructor, and a list of methods. An interface declaration specifies a list of extended interfaces (subtype declarations), a list of interfaces that this interface extends (supertype declarations), and a list of method headers that this interface declaration adds. An interface set is declared by omitting the {...} body and using a semicolon ; instead. Constructors, methods, and expressions are almost exactly like in FJ, except that public field access of the form e.f is not allowed but only access to fields of **this** object, hence a field access consists only of a field name. Other available expressions are access to method parameters or **this** via x, method calls e.m($\overline{\mathsf{e}}$), constructor calls **new** C($\overline{\mathsf{e}}$), and type casts (T) e.

The *subtyping* function defined in Fig. 10 computes the set of known subtype relation as seen from a type T. It does *not* compute the subtypes of T but it represents the set of subtype links visible from T according to the principle of traceability. This set of known subtype relations is computed by following the type declarations to super- or subinterfaces via the helper function *subtypingAux*. The second parameter Z of *subtypingAux* is an accumulator which represents all types visited so far, hence we can stop the search if we encounter a type twice, such that the algorithm does not loop if the subtyping graph contains cycles. Every pair (T, S) in the result stands for a declared sub- or supertype declaration that was encountered when following the sub- or supertype links in the given type. Note that the result of a *subtyping* call, if viewed as a relation, is reflexive, because the pairs (J, J) and (C, C) are added to the result for every visited type. However, the relation is not transitive, in general.

The subtype relation <: is defined in Fig. 11. It is defined in terms of a subtype relation $<:_{\mathsf{S},\mathsf{T}}$ which keeps track of the original types whose subtype relation we want to verify, see rule (S-Intro). The S, T pair in $<:_{\mathsf{S},\mathsf{T}}$ is used to ensure traceability. For a given pair S, T, the relation $<:_{\mathsf{S},\mathsf{T}}$ is transitive by (S-Trans). Note that this does not mean that <:

$$subtyping(\mathsf{T}) = subtypingAux(\mathsf{T}, \emptyset)$$

$$\frac{\begin{array}{c} CT(\mathsf{J}) = \textbf{interface } \mathsf{J} \textbf{ extends } \overline{\mathsf{T}} \textbf{ extendedBy } \overline{\mathsf{S}} \ldots \\ Q_i = \begin{cases} \emptyset \text{ if } \mathsf{T}_i \in Z \\ subtypingAux(\mathsf{T}_i, Z \cup \{\mathsf{J}\}) \text{ else} \end{cases} \\ R_j = \begin{cases} \emptyset \text{ if } \mathsf{S}_j \in Z \\ subtypingAux(\mathsf{S}_j, Z \cup \{\mathsf{J}\}) \text{ else} \end{cases} \end{array}}{\begin{array}{c} subtypingAux(\mathsf{J}, Z) = \\ \{(\mathsf{J},\mathsf{J}), (\mathsf{J},\mathsf{T}_1), \ldots, (\mathsf{J},\mathsf{T}_{|\overline{\mathsf{T}}|}), (\mathsf{S}_1,\mathsf{J}), \ldots, (\mathsf{S}_{|\overline{\mathsf{S}}|},\mathsf{J})\} \\ \cup Q_1 \cup \ldots \cup Q_{|\overline{\mathsf{T}}|} \cup R_1 \cup \ldots \cup R_{|\overline{\mathsf{S}}|} \end{array}}$$

$$\frac{\begin{array}{c} CT(\mathsf{C}) = \textbf{class } \mathsf{C} \textbf{ extends } \mathsf{D} \textbf{ implements } \overline{\mathsf{T}} \ldots \\ \mathsf{T}_0 = \mathsf{D} \\ Q_i = \begin{cases} \emptyset \text{ if } \mathsf{T}_i \in Z \\ subtypingAux(\mathsf{T}_i, Z \cup \{\mathsf{C}\}) \text{ else} \end{cases} \end{array}}{\begin{array}{c} subtypingAux(\mathsf{C}, Z) = \\ \{(\mathsf{C},\mathsf{C}), (\mathsf{C},\mathsf{D}), (\mathsf{C},\mathsf{T}_1), \ldots, (\mathsf{C},\mathsf{T}_{|\overline{\mathsf{T}}|})\} \cup Q_0 \cup \ldots \cup Q_{|\overline{\mathsf{T}}|} \end{array}}$$

Figure 10: Computation of known subtype relations

$$\frac{\mathsf{S} <:_{\mathsf{S},\mathsf{T}} \mathsf{T}}{\mathsf{S} <: \mathsf{T}} \qquad \text{(S-INTRO)}$$

$$\frac{\mathsf{U} <:_{\mathsf{S},\mathsf{T}} \mathsf{V}' \quad \mathsf{V}' <:_{\mathsf{S},\mathsf{T}} \mathsf{V}}{\mathsf{U} <:_{\mathsf{S},\mathsf{T}} \mathsf{V}} \qquad \text{(S-TRANS)}$$

$$\frac{(\mathsf{U},\mathsf{V}) \in subtyping(\mathsf{S}) \cup subtyping(\mathsf{T})}{\mathsf{U} <:_{\mathsf{S},\mathsf{T}} \mathsf{V}} \qquad \text{(S-SUB)}$$

$$\frac{\begin{array}{c} \mathsf{J} <:_{\mathsf{S},\mathsf{T}} \mathsf{J} \\ CT(\mathsf{J}) = \textbf{interface } \mathsf{J} \textbf{ extends } \overline{\mathsf{T}} \textbf{ extendedBy } \overline{\mathsf{S}}; \\ \mathsf{U} <:_{\mathsf{S},\mathsf{T}} \overline{\mathsf{T}} \end{array}}{\mathsf{U} <:_{\mathsf{S},\mathsf{T}} \mathsf{J}} \qquad \text{(S-SET)}$$

Figure 11: Subtyping

itself is transitive. The main subtyping rule is (S-SUB). It specifies that U is a subtype of V if a corresponding super- or subtype declaration is available in the subtyping structure *as seen from* S *or* T. The union of *subtyping*(S) and *subtyping*(T) represents exactly those subtype relations that are traceable starting from T or S.

The rule (S-SET) is responsible for realizing the subtyping of interface sets. A type U is a subtype of an interface set J if U is a subtype of all interfaces $\overline{\mathsf{T}}$ in the set. The condition $\mathsf{J} <:_{\mathsf{S},\mathsf{T}} \mathsf{J}$ makes sure that the subtype relation still remains traceable because $\mathsf{J} <:_{\mathsf{S},\mathsf{T}} \mathsf{J}$ holds only if J is traceable from either S or T. Without this condition, it would

$$\frac{\begin{array}{c}\forall(\mathsf{T}',\mathsf{H}),(\mathsf{T}'',\mathsf{H}')\in\overline{\mathsf{T},\mathsf{H}}:\\ \mathsf{H}=\mathsf{T}\ \mathsf{m}(\overline{\mathsf{T}}),\mathsf{H}'=\mathsf{S}\ \mathsf{m}(\overline{\mathsf{S}})\Rightarrow\mathsf{T}=\mathsf{S},\overline{\mathsf{T}}=\overline{\mathsf{S}}\end{array}}{\mathit{noconflicts}(\overline{\mathsf{T},\mathsf{H}})}$$

$$\frac{\forall(\mathsf{T}',\mathsf{H}')\in\mathit{methods}(\mathsf{T}):\mathsf{H}=\mathsf{H}'\Rightarrow\mathsf{T}=\mathsf{T}'}{\mathit{isIntroduction}(\mathsf{T},\mathsf{H})}$$

$$\frac{\begin{array}{c}\forall(\mathsf{T},\mathsf{H}),(\mathsf{T}',\mathsf{H})\in\overline{\mathsf{T},\mathsf{H}}:\\ \mathit{isIntroduction}(\mathsf{T},\mathsf{H}),\mathit{isIntroduction}(\mathsf{T}',\mathsf{H})\Rightarrow\mathsf{T}=\mathsf{T}'\end{array}}{\mathit{uniqueIntroductions}(\overline{\mathsf{T},\mathsf{H}})}$$

$$\frac{\begin{array}{c}\mathit{noconflicts}(\overline{\mathsf{T},\mathsf{H}})\\ \mathit{uniqueIntroductions}(\overline{\mathsf{T},\mathsf{H}})\end{array}}{\mathit{checkMethods}(\overline{\mathsf{T},\mathsf{H}})}\qquad(\text{CHECK})$$

$$\frac{\textbf{interface}\ \mathsf{J}\ \textbf{extends}\ \overline{\mathsf{T}}\ \textbf{extendedBy}\ \overline{\mathsf{S}}\ \{\}\ \mathsf{OK}}{\textbf{interface}\ \mathsf{J}\ \textbf{extends}\ \overline{\mathsf{T}}\ \textbf{extendedBy}\ \overline{\mathsf{S}};\ \mathsf{OK}}\qquad(\text{T-IFC1})$$

$$\frac{\begin{array}{c}\mathit{checkMethods}(\mathit{methods}(\mathsf{J}))\\ \mathit{signatures}(\mathit{methods}(\overline{\mathsf{S}}))\supseteq\mathit{signatures}(\mathit{methods}(\mathsf{J}))\end{array}}{\textbf{interface}\ \mathsf{J}\ \textbf{extends}\ \overline{\mathsf{T}}\ \textbf{extendedBy}\ \overline{\mathsf{S}}\ \{\overline{\mathsf{H}}\}\ \mathsf{OK}}\qquad(\text{T-IFC2})$$

$$\frac{\begin{array}{c}\mathit{checkMethods}(\mathit{methods}(\mathsf{C}))\\ \mathit{signatures}(\mathit{methods}(\mathsf{C}))\supseteq\mathit{signatures}(\mathit{methods}(\overline{\mathsf{T}}))\\ \mathsf{K}=\mathsf{C}(\overline{\mathsf{S}}\ \overline{\mathsf{g}},\overline{\mathsf{U}}\ \overline{\mathsf{f}})\ \{\textbf{super}(\overline{\mathsf{g}});\textbf{this}.\overline{\mathsf{f}}=\overline{\mathsf{f}};\}\\ \mathit{fields}(\mathsf{D})=\overline{\mathsf{S}}\ \overline{\mathsf{g}}\qquad\overline{\mathsf{M}}\ \mathsf{OK}\ \ \mathsf{IN}\ \ \mathsf{C}\end{array}}{\textbf{class}\ \mathsf{C}\ \textbf{extends}\ \mathsf{D}\ \textbf{implements}\ \overline{\mathsf{T}}\ \{\overline{\mathsf{U}}\ \overline{\mathsf{f}};\mathsf{K}\ \overline{\mathsf{M}}\}\ \mathsf{OK}}\qquad(\text{T-CLASS})$$

$$\frac{\overline{\mathsf{x}}:\overline{\mathsf{T}},\textbf{this}:\mathsf{C}\vdash\mathsf{e}\in\mathsf{S}\qquad\mathsf{S}<:\mathsf{T}}{\mathsf{T}\ \mathsf{m}(\overline{\mathsf{T}}\ \overline{\mathsf{x}})\{\ \textbf{return}\ \mathsf{e};\}\ \mathsf{OK}\ \ \mathsf{IN}\ \ \mathsf{C}}\qquad(\text{T-METHOD})$$

Figure 12: Method, Class, and Interface Typing

be possible to “guess” such an interface set type as V' in (S-TRANS) which is outside the set of traceable type declarations.

The other interesting part of the language semantics are the constraints on method, class, and interface declarations to be type-safe. These are the places where blame is assigned if a declared subtype relation does not hold. The formal definitions are in Fig. 12. The predicate *noconflicts* checks whether a set of method headers has conflicts, i.e., whether there are two methods with the same name but different signatures. The *uniqueIntroductions* check makes sure that the unique name introduction property defined in Sec. 4 holds.

The check for interface sets (T-IFC1) is the same as the check for interfaces (T-IFC2). An interface specification is ok (T-IFC2) if there are no conflicts among the inherited and defined methods and if all types after the **extendedBy** keyword have at least all methods that this type defines. The auxiliary function *methods*, which is defined in the appendix (Fig. 13), collects all method headers and the place where they are declared in a type and all its supertypes. The function *signatures* extracts the headers. The notation $\mathit{signatures}(\mathit{methods}(\overline{\mathsf{S}})) \supseteq \mathit{signatures}(\mathit{methods}(\mathsf{J}))$ is a shorthand for $\mathit{signatures}(\mathit{methods}(\mathsf{S}_1)) \supseteq \mathit{signatures}(\mathit{methods}(\mathsf{J})), ..., \mathit{signatures}(\mathit{methods}(\mathsf{S}_{|\overline{\mathsf{S}}|})) \supseteq \mathit{signatures}(\mathit{methods}(\mathsf{J}))$ (similarly in other places).

For a class declaration to be ok (T-Class), it must not have method conflicts and it has to implement (directly or in a superclass) all methods required by all implemented interfaces $\overline{\mathsf{T}}$. In addition, the constructor parameters must match with the fields defined in C and its superclass D. The helper function *fields* collects all fields of a class and its superclasses and is again formally defined in the appendix (Fig. 13). Finally, all methods of the class must be ok.

A method is ok (T-Method), if the method body can be type-checked in the appropriate environment and if the type of the body is a subtype of the annotated return type.

Type checking of expressions and the operational semantics of $\mathrm{FJ}_{<:}$ are not very interesting; they are a straightforward translation of the corresponding FJ definitions. For this reason, these definitions are given in the appendix (App. A Fig. 13 and 14) and not discussed here. Note, however, that these definitions differ from the FJ definitions in that they use the refined subtype relation from Fig. 11.

Soundness of the type system is established via the standard progress and preservation theorem[2]. A sketch of the soundness proof as a delta to the Featherweight Java soundness proof is available in App. B.

7 DISCUSSION

We will now compare $\mathrm{FJ}_{<:}$ and different other subtyping definitions with respect to the desideratas defined in Sec. 4. The results are summarized in Table 1.

Structural subtyping such as in OCaml [22] or [25] is on one end of the spectrum. Structural subtyping is traceable because only the structure of two types needs to be compared; no lookup of other types is necessary. Name introductions are not unique - with structural subtyping it is not possible to differentiate name introductions from name usages. Since subtyping is not declared, the blame for every failing upcast will be assigned to the party performing the upcast, hence it is not possible to assign blame flexibly. The big advantage of structural subtyping is its modular extensibility. One just needs to define a type and it is automatically a subtype or supertype of all other types with a corresponding structure.

Nominal subtyping such as in most mainstream languages is on the other end of the

[2]The preservation theorem requires that the type of a rewritten program is a subtype of the previous type. This cannot hold if the subtyping relation that is used for rewritten programs is not transitive. For this reason, the subtype definition discussed here is only used for type-checking the original program, whereas for rewritten programs a wider subtype relation that is defined in the appendix is used. This is basically the same trick as the “stupid cast” rule in FJ.

	Traceability	Unique Name Introduction	Flexible Assignment of Responsibility	Modular Extensibility
Structural Subtyping	Yes	No	No, the party performing the upcast is always blamed	Since subtype relations are not declared, all kinds of subtype openness are automatically given
Nominal Subtyping in Java [2]	Yes	No	Not possible to assign responsibility to the supertype	only subtype-open for interface types; intervened with inheritance for class types
Sather [35, 33]	Yes	No	responsibility can be assigned to both sub- and supertype	subtype-open and supertype-open
Compound Types [6]	Yes	Yes*	Not possible to assign responsibility to a supertype	subtype-open and subtype-supertype-open w.r.t. interface sets
$FJ_{<:}$	Yes	Yes/No*	responsibility can be assigned to both sub- and supertype	subtype-supertype-open
AspectJ Inter-type declarations [19]	No	No	No, the party performing the upcast is always blamed	subtype-supertype-open
“Duck Typing”	No	No	The first party which calls a method on an object that does not exist is blamed at runtime	subtype-supertype-open

Table 1: Comparison of different subtyping mechanisms

spectrum. We will consider subtyping as defined in Java [2] in more detail. In Java, the subtype relation is traceable; in order to determine whether T is a subtype of T' it is sufficient to follow the tree of superclasses and implemented interfaces of T and search for T'. Although, in principle, it would be possible to distinguish name introductions from name usages, Java does *not* have unique name introductions: As long as the signatures are compatible, the same method can be inherited from two distinct, unrelated interfaces. As the examples in Sec. 2 showed, the subtype declarations in Java always assign responsibility to the subtype; the other way around is not possible. The major disadvantage of nominal subtyping is its limited extensibility: It is only possible to add new subtypes of interface types; for class types, subtyping is intervened with inheritance because it is not possible to create a subtype of a class type without inheriting the implementation of the class. More importantly, it is not possible to add new supertypes in a modular way.

Subtyping in $FJ_{<:}$ combines and generalizes the subtyping mechanisms in Sather [35, 33] and compound types [6]. Sather is a language with nominal subtyping where type declarations have a supertype clause similar to our **extendedBy** clause, in addition to the usual subtype clause. Hence Sather allows flexible assignment of responsibility. The subtype and supertype clauses render Sather subtype-open and supertype-open. The main difference to $\mathrm{FJ}_{<:}$ is that Sather is *not* subtype-supertype-open because, as the authors say [33], this would prevent modular type checking (i.e., traceability). The advantage of $\mathrm{FJ}_{<:}$ over Sather is that the non-transitive subtype relation enables to *combine* traceability with subtype-supertype-openness.

Compound types [6] are a proposal to integrate intersection types into nominal object-oriented languages, similar to our interface sets. The new extensibility dimension of compound types is hence that they are subtype-supertype-open, but only with respect to interface sets. This is because subtyping of interface sets is defined as a subset relation between the respective interface sets, i.e., structural. Does this extension mean that one inherits the disadvantages of structural subtyping through the backdoor? This is true to some degree: In our example in Fig. 5, we cannot document that SomeWidget is designed to be a subtype of ColoredBorderWidget. For example, removal of the ColoredWidget interface from SomeWidget silently removes this subtype relation, and a client upcasting SomeWidget to ColoredBorderWidget would be blamed for this error. The important difference to general structural subtyping, however, is that such intersection types are compatible with unique method name introductions. The unique name introduction property may not hold because of other properties of the underlying type system, but compound types themselves are compatible with it because an interface set does not introduce new method names. One could argue that an interface set introduces a new name, namely the name of the interface set, and that this name represents more than just the set of interfaces it represents. However, this name is never matched against any other name, hence it does not influence unique name introductions.

Compound types / interface sets are a very interesting corner point in the subtyping design space which is illustrated by the comparison with Sather and $\mathrm{FJ}_{<:}$. $\mathrm{FJ}_{<:}$, as defined in the previous section, guarantees unique name introductions and is also subtype-supertype-open, whereas compound types are subtype-supertype-open only with respect to interface sets. However, the unique name introduction property in $\mathrm{FJ}_{<:}$ *enforces* that only those subtype-supertype extensions are accepted that could also be represented by compound types! Consider the `CollectionIsaROC` declaration in Fig. 8. Every method that is introduced in `ReadOnlyCollection` which does not stem from an interface that is also implemented by `Collection` leads to a non-unique name introduction in `CollectionIsaROC` because the same method has to be available in `Collection`. Hence this situation would be prevented by the *uniqueIntroductions* check in (Check) (Fig. 12).

On the other hand, if all method names in `ReadOnlyCollection` stem from interfaces that are also implemented by Collection, then the **extendedBy** Collection clause is not necessary in the declaration of `CollectionIsaROC` - that is, the same type could also be declared as an interface set / compound type only. Interface sets hence seem to represent the maximum in subtype-supertype-openness that can be reconciled with unique name introductions.

Since the uniqueness check *uniqueIntroductions* in (Check) (Fig. 12) is not necessary for type soundness, we can also consider it optional: With this check enabled, subtype-supertype-extensibility boils down to interface sets; without the check, name introductions are no longer unique, but other extensibility scenarious not expressible with interface sets are also possible. This would be useful if the link between two name introductions is outside the source code itself. For example, two teams may have negotiated on the phone that a method name x represents a particular concept. In this case, it would be ok to merge two name introductions of x, because they are name introductions only syntactically and in reality both stand for the negotiated meaning of x. The fact that most subtype definitions used in practice do not have unique name introductions suggests that this may indeed be a common situation and hence the option to drop the uniqueness check may be useful.

Our subtyping desiderata are also insightful in the comparison with other subtyping mechanisms not discussed in this work so far. With AspectJ [19], the subtype structure can be extended by means of *inter-type declarations*. For example, the declaration `declare parents : X implements Y` in an aspect makes the class `X` a subtype of the interface `Y`, provided that `X` implements all interfaces required by `Y` (whereby these methods could also be added by the aspect, but this mechanism is not in the scope of this paper). With this mechanism, all extensibility scenarios we considered can be encoded. However, this mechanism is *not* traceable: A client upcasting `X` to `Y` does not have to refer to the aspect in any way, hence a global view is required in order to understand the upcast. Blame is assigned incorrectly: Since the (client) upcast site does not refer to the witness of the subtype relation (the intertype declaration), the client is always blamed.

The term “duck typing” is frequently used to describe typing in dynamically-typed languages such as Ruby [37] (“If it walks like a duck and quacks like a duck, it must be a duck”). In these languages, there is no well-defined subtype relation, hence our criteria for comparison do not fit 100%. Whether an object is ok (does not lead to a dynamic type error) as an argument to a method is not traceable. In fact, even with a global view on the system this is (in general) not decidable. Most dynamically typed languages do not differentiate name introductions from name usages. Blame is assigned to those places in the code which (at runtime) call a method on an object that does not understand the method. This makes it very hard to track the error down to the place responsible for the error. Although our terminology does not quite fit to dynamic typing, duck typing can be seen as the most flexible mechanism with respect to modular extensibility - even more flexible than structural subtyping, since the set of methods required from an object are only those that are actually called in a particular control flow.

8 RELATED WORK

Languages which extend nominal subtyping: The relation to Sather [35, 33], compound types [6], safe structural conformance [25] and AspectJ [19] has already been discussed in detail in the previous section. The language gbeta [14] has a nominal core but uses structural subtyping on sets of mixins, which are closely related to interfaces. A similar approach is choosen in McJava [18]. The latter work also describes an implementation strategy for subtyping of mixin sets on the Java Virtual Machine [23]. *Induction* [34] is a technique with which the commonalities between two independent types can be computed and made available in the form of a a new common supertype. Some languages allow the user to choose between nominal and structural subtyping with mechanisms such as *branding* in Modula [9]. However, with such a mechanism one can only choose between these two mechanisms but not combine the advantages of both mechanisms.

Conflict between inheritance and subtyping: This conflict is well-known and has been discussed in many works [26, 20, 36, 11, 3, 31, 27]. Our proposal to create subtypes of a class without inheriting its implementation is a step towards a better separation of these two concepts. Similar ideas have been proposed in Emerald [3, 31] and some other works, but the idea to simply use the class name as a type is, to the best knowledge of the author, new.

Subtyping recursive types: In structural type systems, it is possible to establish subtype relations between recursive types [1], e.g., `interface C { C clone(); }` can be a subtype of `interface D { D clone(); }`. Such subtype relations cannot be established in any of the nominal subtype relations we have considered; making nominal systems powerful enough to express such subtype relations is hence part of our future work.

Discussions of nominal versus structural type systems: There are only few works with an explicit comparison of these approaches. Hölzle [16] argues that structural subtyping may lead to unintended subtype relations (Cowboy.draw versus Display.draw) and a *loss of abstraction* because the type system cannot differentiate between, say, a number representing a length and a number representing a weight. He argues in favor of structural subtyping that a new common supertype can easily be retrofitted into an existing type hierarchy simply by defining the type. A similar effect can be achieved with our **extendedBy** clause, but without sacrifying the model of explicitly declared subtype relations. Day et al note that nominal systems make renaming of features in subtypes possible [13]. Leavens and Weihl argue that subtypes should be designed "with subtyping in mind" and that nominal subtyping makes modular program verification easier [21]. Findler et al [15] state that nominal typesystems are mainly choosen because they are easier to implement and understand and argue that nominal type systems prevent component reuse, hence they propose the possibility to perform structural casts in an otherwise nominal system. We think this argument in favor of structural subtyping is relativized by the weakness of structural subtyping w.r.t. blame assignment.

Findler et al also propose to add structural type casts to a nominal language [15]. Their work also tries to achieve a better assignment of blame if subtyping goes wrong. They consider interface contracts that cannot be verified statically in that the party which performs the cast specifies (in a string) which party is to blame if a pre- or postcondition, respectively, fail. The work does not consider the problem of assigning blame if the statically verifiable structure in the upcast does not match. Our work does not consider dynamically checked contracts. We think, however, that the need for structural casts is decreased by the mechanisms proposed here.

Generalization: There are a few works on *generalization* or *exheritance*, a concept dual to inheritance [12, 28, 32]. These works focus mainly on the definition and meaning of generalization, e.g., by removing methods from the subclass. Neither of these works considers the implications of generalization for static nominal subtyping.

Matching: Matching and the notion of *MyType* has been proposed as an alternative to subtyping [5]. Matching has advantages over subtyping with respect to the treatment of the type of **this**, as required, e.g., for binary methods [5]. This problem is orthogonal to our work. It would be interesting, though, to investigate whether the idea of the **extendedBy** clause would also work for matching.

9 CONCLUSIONS AND FUTURE WORK

We have presented an analysis of subtyping from a component-based perspective. Nominal subtyping can give more safety guarantees like unique name introductions and correct blame assignment, whereas structural subtyping excels at modular extensibility of the subtyping relation. We have developed concrete terminology to compare subtyping mechanisms and proposed a definition of subtyping that makes nominal subtyping almost as flexible as structural subtyping without sacrifying clear assignment of responsibility and blame for subtype relations.

Our future work will concentrate on analyzing other kinds of structural subtypes, such as those required for F-bounded parametric polymorphism [7] or the structural subtyping mechanism on interface sets proposed in this paper and check whether they can also be replaced by advanced nominal subtyping mechanisms. The long-term future work is to find a general mechanism with which a program part can accept responsibility for any kind of design decision, not just subtyping.

REFERENCES

[1] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

[2] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language (4th edition)*. Addison-Wesley, 2005.

[3] A. P. Black, N. C. Hutchinson, E. Jul, and H. M. Levy. Object structure in the Emerald system. In *OOPSLA'86: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 78–86, 1986.

[4] K. B. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, 2002.

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

[6] M. Büchi and W. Weck. Compound types for Java. In *OOPSLA'98: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 362–373, October 1998.

[7] P. S. Canning, W. Cook, W. L. Hill, J. C. Mitchell, and W. G. Olthoff. F-bounded polymorphism for object-oriented programming. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture (FCPA) '89*, pages 273–280, 1989.

[8] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, 1988.

[9] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 language definition. *SIGPLAN Not.*, 27(8):15–42, 1992.

[10] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[11] W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. In *POPL'90: ACM Symposium on Principles of Programming Languages*, pages 125–135. ACM, 1990.

[12] P. Crescenzo and P. Lahire. Using both specialisation and generalisation in a programming language: Why and how? In *OOIS '02: Proceedings of the Workshops on Advances in Object-Oriented Information Systems*, pages 64–73, London, UK, 2002. Springer-Verlag.

[13] M. Day, R. Gruber, B. Liskov, and A. C. Meyers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 156–168, 1995.

[14] E. Ernst. Propagating class and method combination. In *Proceedings ECOOP'99*, LNCS 1628, pages 67–91. Springer-Verlag, 1999.

[15] R. B. Findler, M. Flatt, and M. Felleisen. Semantic casts: Contracts and structural subtyping in a nominal world. In *ECOOP '2004: Proceedings of the European Conference on Object-Oriented Programming*, pages 364–388. Springer-Verlag, 2004.

[16] U. Hölzle. Integrating independently-developed components in object-oriented languages. In *Proceedings ECOOP '93*, LNCS 707, pages 36–56. Springer, 1993.

[17] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 1999.

[18] T. Kamina and T. Tamai. McJava - a design and implementation of Java with mixin-types. In W.-N. Chin, editor, *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004. Proceedings*, volume 3302 of *Lecture Notes in Computer Science*, pages 398–414. Springer, 2004.

[19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP '01*, 2001.

[20] W. LaLonde, D. Thomas, and J. Pugh. Subclassing $\neq$ subtyping $\neq$ is-a. *Journal of Object-Oriented Programming*, 3(5), 1991.

[21] G. T. Leavens and W. E. Weihl. Reasoning about object-oriented programs that use subtypes. In *Proceedings of OOPSLA/ECOOP '90*, pages 212–223, New York, NY, USA, 1990. ACM Press.

[22] X. Leroy. The objective caml system, release 3.09, 2005. http://ocaml.org.

[23] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[24] B. H. Liskov and J. M. Wing.. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[25] K. Läufer, G. Baumgartner, and V. F. Russo. Safe structural conformance for Java. *Computer Journal*, 43(6):469–481, 2001.

[26] O. L. Madsen, B. Magnusson, and B. Møller-Pedersen. Strong typing of object-oriented languages revisited. In *Proceedings of OOPSLA/ECOOP '90*, pages 140–150, New York, NY, USA, 1990. ACM Press.

[27] K. Ostermann and M. Mezini. Object-oriented composition untangled. In *Proceedings OOP-SLA '01*, ACM SIGPLAN Notices 36(11), pages 283–299, 2001.

[28] C. H. Pedersen. Extending ordinary inheritance schemes to include generalization. In *OOP-SLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 407–417, New York, NY, USA, 1989. ACM Press.

[29] B. C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002.

[30] R. K. Raj, E. Tempero, and H. K. Levy. Emerald: A general-purpose programming language. *Software – Practice and Experience*, 21(1):91–118, 1991.

[31] R. K. Raj, E. D. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul. Emerald: A general-purpose programming language. *Softw., Pract. Exper.*, 21(1):91–118, 1991.

[32] M. Sakkinen. Exheritance - class generalisation revived. In *The Inheritance Workshop at ECOOP'02*, 2002.

[33] D. Stoutamire and S. Omohundro. The Sather 1.1 specification. Technical Report TR-96-012, International Computer Science Institute, 1996.

[34] B. Swen. Object-oriented programming with induction. *SIGPLAN Not.*, 35(2):61–67, 2000.

[35] C. Szyperski, S. Omohundro, and S. Murer. Engineering a programming language: The type and class system of Sather. In J. Gutknecht, editor, *Programming Languages and System Architectures*, pages 208–227. LNCS 782, Springer, 1993.

[36] A. Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):439–479, 1996.

[37] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby*. Pragmatic Bookshelf, 2004.

ABOUT THE AUTHORS

Klaus Ostermann is associate professor for computer science at the University of Aarhus, Denmark. His main research interests are in programming languages and software engineering for modular software development.

A AUXILIARY FUNCTIONS, EXPRESSION TYPING, AND COMPUTATION OF FJ$_{<:}$

The formal definitions are given in Fig. 13 and 14.

B SOUNDNESS

In this section we sketch how the soundness proof in [17] needs to be adapted in order to prove soundness of FJ$_{<:}$.

In a small-step semantics, it is not possible to establish soundness with a non-transitive subtype relation. Hence we explictly distinguish two different type systems: A strict type system which is used for type-checking the original program, and a type system with a wider, transitive subtype relation that is used for intermediate programs. Every program that is accepted by the strict type system is also accepted by the relaxed type system. The traceability property (which is only relevant for the original program) is only guaranteed by the strict type system, however.

The rules for runtime subtyping are in Fig. 14. The existential quantification in (SR-Intro) means that it is sufficient to find a matching subtype declaration *anywhere* in the program. Similarly to the stupid cast rule in FJ, the runtime subtyping rules may only be used for rewritten programs.

With this extension, the FJ proof in [17] goes through virtually unchanged. The only non-trivial change to the proof is Lemma A.1.1 of [17], hence we will re-state and prove it here:

Auxiliary functions:

$$methods(\mathsf{Object}) = \emptyset$$

$$\frac{\begin{array}{c} CT(\mathsf{C}) = \textbf{class}\ \mathsf{C}\ \textbf{extends}\ \mathsf{D}\ \textbf{implements}\ \overline{\mathsf{T}} \\ \{\overline{\mathsf{S}}\ \overline{\mathsf{f}}; \mathsf{K}\ \mathsf{H}_1\ \{\textbf{return}\ \mathsf{e}_1\}...\mathsf{H}_n\ \{\textbf{return}\ \mathsf{e}_n\}\} \end{array}}{methods(\mathsf{C}) = \{(\mathsf{C},\mathsf{H}_1),...,(\mathsf{C},\mathsf{H}_n)\} \cup methods(\mathsf{D})}$$

$$\frac{\overline{\mathsf{T},\mathsf{H}} = (\mathsf{T}_1,\mathsf{H}_1),...,(\mathsf{T}_n,\mathsf{H}_n)}{signatures(\overline{\mathsf{T},\mathsf{H}}) = \{\mathsf{H}_1,...,\mathsf{H}_n\}}$$

$$\frac{CT(\mathsf{J}) = \textbf{interface}\ \mathsf{J}\ \textbf{extends}\ \overline{\mathsf{T}}\ \textbf{extendedBy}\ \overline{\mathsf{S}}\ \{\overline{\mathsf{H}}\}}{methods(\mathsf{J}) = \{(\mathsf{J},\mathsf{H}_1),...,(\mathsf{J},\mathsf{H}_{|\overline{\mathsf{H}}|}) \cup \bigcup_{\mathsf{T}\in\overline{\mathsf{T}}} methods(\mathsf{T})}$$

$$fields(\mathsf{Object}) = \emptyset$$

$$\frac{\begin{array}{c} CT(\mathsf{C}) = \textbf{class}\ \mathsf{C}\ \textbf{extends}\ \mathsf{D}\ \textbf{implements}\ \overline{\mathsf{T}}\ \{\overline{\mathsf{S}}\ \overline{\mathsf{f}};...\} \\ fields(\mathsf{D}) = \overline{\mathsf{T}}\ \overline{\mathsf{g}} \end{array}}{fields(\mathsf{C}) = \overline{\mathsf{T}}\ \overline{\mathsf{g}}, \overline{\mathsf{S}}\ \overline{\mathsf{f}}}$$

$$\frac{(\mathsf{T}', \mathsf{U}\ \mathsf{m}(\overline{\mathsf{S}}\ \overline{\mathsf{x}})) \in methods(\mathsf{T})}{mtype(\mathsf{m},\mathsf{T}) = \overline{\mathsf{S}} \to \mathsf{U}}$$

$$\frac{\begin{array}{c} CT(\mathsf{C}) = \textbf{class}\ \mathsf{C}\ \textbf{extends}\ \mathsf{D}\ \textbf{implements}\ \overline{\mathsf{T}}\ \{\overline{\mathsf{S}}\ \overline{\mathsf{f}}; \mathsf{K}\ \overline{\mathsf{M}}\} \\ \mathsf{T}\ \mathsf{m}(\overline{\mathsf{U}}\ \overline{\mathsf{x}})\ \{\textbf{return}\ \mathsf{e};\} \in \overline{\mathsf{M}} \end{array}}{mbody(\mathsf{m},\mathsf{C}) = (\overline{\mathsf{x}},\mathsf{e})}$$

$$\frac{\begin{array}{c} CT(\mathsf{C}) = \textbf{class}\ \mathsf{C}\ \textbf{extends}\ \mathsf{D}\ \textbf{implements}\ \overline{\mathsf{T}}\ \{\overline{\mathsf{S}}\ \overline{\mathsf{f}}; \mathsf{K}\ \overline{\mathsf{M}}\} \\ \mathsf{m}\ \text{is not defined in}\ \overline{\mathsf{M}} \end{array}}{mbody(\mathsf{m},\mathsf{C}) = mbody(\mathsf{m},\mathsf{D})}$$

Congruence:

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

$$\frac{\mathsf{e}_i \longrightarrow \mathsf{e}_i'}{\begin{array}{c}\mathsf{e}_0.\mathsf{m}(...,\mathsf{e}_i,...) \\ \longrightarrow \mathsf{e}_0.\mathsf{m}(...,\mathsf{e}_i',...)\end{array}} \qquad \text{(RC-Invk-Arg)}$$

$$\frac{\mathsf{e}_i \longrightarrow \mathsf{e}_i'}{\begin{array}{c}\textbf{new}\ \mathsf{C}(...,\mathsf{e}_i,...) \\ \longrightarrow \textbf{new}\ \mathsf{C}(...,\mathsf{e}_i',...)\end{array}} \qquad \text{(RC-New-Arg)}$$

$$\frac{\mathsf{e}_0 \longrightarrow \mathsf{e}_0'}{(\mathsf{T})\mathsf{e}_0 \longrightarrow (\mathsf{T})\mathsf{e}_0'} \qquad \text{(RC-Cast)}$$

Figure 13: Operational semantics and auxiliary functions

Computation:

$$\frac{\begin{array}{c} mbody(\mathsf{m}, \mathsf{C}) = (\bar{\mathsf{x}}, \mathsf{e}_0) \\ fields(\mathsf{C}) = \bar{\mathsf{T}}\ \bar{\mathsf{f}} \end{array}}{\begin{array}{c} (\mathbf{new}\ \mathsf{C}(\bar{\mathsf{e}})).\mathsf{m}(\bar{\mathsf{d}}) \\ \longrightarrow [\bar{\mathsf{d}}/\bar{\mathsf{x}}, \mathbf{new}\ \mathsf{C}(\bar{\mathsf{e}})/\mathbf{this}, \bar{\mathsf{e}}/\bar{\mathsf{f}}]\mathsf{e}_0 \end{array}} \qquad \text{(R-Invk)}$$

$$\frac{\mathsf{C} <: \mathsf{T}}{(\mathsf{T})\ (\mathbf{new}\ \mathsf{C}(\bar{\mathsf{e}})) \longrightarrow \mathbf{new}\ \mathsf{C}(\bar{\mathsf{e}})} \qquad \text{(R-Cast)}$$

Expression Typing:

$$\Gamma \vdash \mathsf{x} \in \Gamma(\mathsf{x}) \qquad \text{(T-Var)}$$

$$\frac{\begin{array}{c} \Gamma \vdash \mathbf{this} \in \mathsf{C}_0 \\ fields(\mathsf{C}_0) = \bar{\mathsf{T}}\ \bar{\mathsf{f}} \end{array}}{\Gamma \vdash \mathsf{f}_i \in \mathsf{T}_i} \qquad \text{(T-Field)}$$

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

$$\frac{\begin{array}{c} fields(\mathsf{C}) = \bar{\mathsf{T}}\ \bar{\mathsf{f}} \\ \Gamma \vdash \bar{\mathsf{e}} \in \bar{\mathsf{U}} \qquad \bar{\mathsf{U}} <: \bar{\mathsf{T}} \end{array}}{\Gamma \vdash \mathbf{new}\ \mathsf{C}(\bar{\mathsf{e}}) \in \mathsf{C}} \qquad \text{(T-New)}$$

$$\frac{\Gamma \vdash \mathsf{e}_0 \in \mathsf{S} \qquad \mathsf{S} <: \mathsf{T}}{\Gamma \vdash (\mathsf{T})\mathsf{e}_0 \in \mathsf{T}} \qquad \text{(T-UCast)}$$

$$\frac{\begin{array}{c} \Gamma \vdash \mathsf{e}_0 \in \mathsf{S} \\ \mathsf{T} <: \mathsf{S} \qquad \mathsf{T} \neq \mathsf{S} \end{array}}{\Gamma \vdash (\mathsf{T})\mathsf{e}_0 \in \mathsf{T}} \qquad \text{(T-DCast)}$$

$$\frac{\begin{array}{c} \Gamma \vdash \mathsf{e}_0 \in \mathsf{S} \\ \mathsf{T} \not<: \mathsf{S} \qquad \mathsf{S} \not<: \mathsf{T} \end{array}}{\Gamma \vdash (\mathsf{T})\mathsf{e}_0 \in \mathsf{T}} \qquad \text{(T-SCast)}$$

Runtime subtyping rules:

$$\frac{\exists \mathsf{S}, \mathsf{T} : \mathsf{S}' <:_{\mathsf{S},\mathsf{T}} \mathsf{T}'}{\mathsf{S}' <: \mathsf{T}'} \qquad \text{(SR-Intro)}$$

$$\frac{\mathsf{U} <: \mathsf{V}' \quad \mathsf{V}' <: \mathsf{V}}{\mathsf{U} <: \mathsf{V}} \qquad \text{(SR-Trans)}$$

$$\mathsf{U} <: \mathsf{U} \qquad \text{(SR-Ref)}$$

Figure 14: Typing rules

Lemma A.1.1 : If $\mathit{mtype}(\mathsf{m}, \mathsf{S}) = \overline{\mathsf{T}} \to \mathsf{T}_0$ then $\mathit{mtype}(\mathsf{m}, \mathsf{T}) = \overline{\mathsf{T}} \to \mathsf{T}_0$ for all $\mathsf{T} <: \mathsf{S}$.

Proof: Easy induction in the derivation of $\mathsf{T} <: \mathsf{S}$ using the *noconflicts* predicate and the method subset relations defined in (T-IFC2) and (T-CLASS).

Since $\mathrm{FJ}_{<:}$ does not need an evaluation rule for field access (field names are substituted in (R-INVK)), the corresponding case from the proof of the progress theorem goes away. Instead, the (R-INVK) case of the proof needs to take the additional substitutions into account, but this is easy due to the term substitution lemma A.1.2.

The *uniqueIntroductions* check in (CHECK) (Fig. 12) is not needed for the soundness proof; all that matters for type soundness is the compatibility of method signatures, which is already guaranteed by *noconflicts*. Hence it is ok to make this check optional.

The rest of the proof works exactly as the original FJ proof, except that meta-variables for classes have to be replaced by meta-variables for types in several places.