Inheritance *Is* Subtyping

Robert Cartwright[1,2] and Moez Abdel-Gawad[1]

[1] Rice University, Houston, TX, USA
cork@rice.edu, moez@rice.edu
[2] Halmstad University, Halmstad, Sweden
robert.cartwright@hh.se

Since Luca Cardelli wrote a seminal paper [3] on the semantics of inheritance in 1984, programming language researchers have constructed a variety of structural models of object-oriented programming (OOP) founded on Cardelli's work. Since Cardelli approached OOP from the perspective of functional programming, he identified inheritance with record subtyping—an elegant choice in this context. Although Cardelli did not formally define inheritance, he equated it with record extension and proved that for a small functional language with records, variants, and function types–but no recursive record types–that syntactic and semantic record subtyping were equivalent. William Cook *et al* [4] subsequently added `ThisType` and recursive record types, narrowing the typing of **this** in inherited methods, and reached a profoundly different conclusion: *inheritance is not subtyping.*

Meanwhile, object-oriented (OO) program design emerged as an active area of research within software engineering, spawning class-based OO languages like C++, Java, and C#, which strictly define inheritance in terms of class hierarchies. In these languages, subtyping is identified with inheritance. In contrast to Cardelli's expansive formulation of inheritance based solely on record interfaces (sets of member-name interface pairs)[1], these languages define the type associated with a class C as the set of all instances of C and all instances of explicitly declared subclasses of C. Simply matching the signatures of the members of C–as in record subtyping–is insufficient.

It is easy to show that record subtyping is too weak to capture the notion of inheritance in nominally typed OO languages. Consider classes `Set` and `MultiSet` intended to represent mathematical sets and multi-sets respectively. They can easily have exactly the same visible members with exactly the same types. Assume that `Set` and `MultiSet` are defined independently but have exactly the same visible members. If objects simply denote records, then each class is a subtype of the other, but in a nominal OO language, no such subtyping relationship exists between the two classes. In nominal OO languages, objects are more than mere records.

In Cardelli's semantics and its successors based on functional programming models, the meaning of a class only depends on the members of the class (including *inherited* members), not on the inheritance hierarchy used to define the class. This paper dicusses the implications of a new approach to defining the semantics of OO languages that embeds in each object the signature of the inheritance hierarchy above it. In contrast to record-based semantics, our new approach completely reconciles inheritance and subtyping among classes: a class B is a subtype of a class A iff B inherits from A.

In statically-typed functional languages based on the simply typed lambda-calculus, the issue of subtyping does not arise: every data value belongs to a unique type. Even when such a language is generalized to support parametric polymorphism [9], every value belongs to a unique *monotype* (unquantified type).

Object-oriented languages introduce the idea that composite values (often called records or structures in functional languages) can belong to multiple monotypes. For example, a

[1] Since Cardelli excluded recursive types, every interface in his language can be expressed purely in terms of type constructors applied to primitive types.

`ColorPoint` object with fields `x: Number, y: Number, color: Color` can have type `Point` which omits the `color` field as well as type `ColorPoint`. In *structural* OO languages, object types are based simply on the interfaces of objects: the names and types of their visible record members. Hence, `ColorPoint` is a subtype of `Point` even when it is separately defined without use of inheritance. In *nominal* OO languages, object types are based strictly on the inheritance structure specified in the program: the type associated with class `B` is a subtype of the type associated with class `A` iff the definition of `B` explicitly inherits from the definition of `A`. If object values do not include inheritance information, this restricted definition of subtyping appears capricious. But OO software developers think of an object in the context of its class hierarchy and the contracts associated with its class members, *which are inherited along with the corresponding class members*. For example, in Java, the interface `Comparable<T>`, consisting only of the method `int compareTo(T t)`, has a contract asserting that `compareTo` defines a total ordering on `T`; an arbitrary class with a method `int compareTo(T t)` generally does not obey this contract. When a programmer asserts that a class `C` implements `Comparable<C>`, he is asserting that the `compareTo` defines a total ordering on `T`.

In mainstream OO design, subtyping conforms to the *Contract Presevation* (CP) property: types are characterized by behavioral contracts and every subtype `B` of a type `A` obeys the contracts of the parent type `A`. The earliest formulation of this principle, proposed by Liskov in 1988 [7], was expressed in terms of the substitutability of objects, which was technically problematic, perhaps dissuading researchers in programming theory from giving it much credence. The principle is still called the *Liskov Substitution* principle in most software engineering contexts. A subtype can augment the contracts inherited from its parent type, but the inherited contracts still apply as well. Of course, no decidable type system can fully capture program behavior since any non-trivial aspect of program *behavior* is undecidable by Rice's theorem. In practice, a decidable type system should perform static checks that help programmers confirm that their code obeys CP.

In nominal OO languages like Java and C#, the static type system identifies subtyping with inheritance: the type corresponding to class `C` consists of all instances of `C` and all instances of subclasses of `C`, which by definition inherit from `C`. Hence, the type for class `B` is a subtype of the type for class `A` iff `B` inherits from `A`. In writing the code for a subclass, the programmer is responsible for confirming that instances of the class conform to the contracts for all superclasses. The preservation of such contracts is a pillar of good OO design. For this reason and to support mutability of object fields, the input types in the signature of an overriding method typically must exactly match those in the overridden method.

According to folklore among programming language researchers, the identification of subtyping and inheritance in mainstream OO languages like Java and C# is misguided, despite the fact that simple versions of these type systems have been proved sound [5, 6] relative to operational semantics for these languages. In earlier work [1], we presented denotational model of OOP, dubbed **NOOP**, akin to Cardelli's record model that justifies the typing conventions in mainstream OO languages and breaks typing rules based on record subtyping. In other words, given what we believe is a proper model of mainstream nominal OOP, *subtyping is inheritance* and the usual structural typing rules are *unsound*.

The key idea in the construction of **NOOP** is to define the meaning of an object in class `C` as a pair consisting of a record (as in structural models) plus the signature closure for class `C`. A *signature* s is a triple consisting of a *key* class name `C`, a (possibly empty) set of superclass names, and a set of syntactic types for the visible members of `C`. A *signature environment* se is a finite set of class signatures where no two key class names are the same. Hence, a signature environment determines a function mapping a finite set of class names to signatures. A signature

environment is *closed* iff every class name that appears anywhere in the environment appears as a key class name. In other words, every class that is referenced in the environment is defined in the finite function corresponding to the environment. A signature environment *se* is a *signature closure for class name* `C` iff (*i*) *se* is closed, (*ii*) *se* defines the name `C`, and (*iii*) *se* only defines the names in the transitive reference closure of `C`.

From the preceding definitions, we deduce that a signature closure for class name `C` is a signature environment consisting of the transitive closure (under class reference) of the singleton set consisting of a signature for class name `C`.

For the details on how to construct **NOOP**, see [1] and [2].

It is straightforward to define a syntactic relation between class signatures called subsigning: the class signature s_1 subsigns the class signature for s_2 iff the information in s_1 includes the information in s_2 and both signatures are well-formed. The details of the construction are given in [1]. We identify subsigning with inheritance. In a nominal OO program `P`, class `B` inherits from class `A` iff the signature closure for `A` subsigns the signature closure for `B`.

For each class `C` in a program `P`, we can define the weak ideal [8] of objects consisting of all objects with signature matching the signature of C in P and all objects in classes with signatures that subsign the signature of `C`. This weak ideal includes the instances of *all possible classes* extending `C`. Moreover, it is not difficult to prove that subsigning implies subtyping. The proof of this property appears in [1]. Hence, in nominal OO languages, inheritance implies subtyping.

References

[1] Moez A. AbdelGawad. A domain-theoretic model of nominally-typed object-oriented programming. *Accepted for publication in the Journal of Electronic Notes in Theoretical Computer Science (ENTCS). Also presented at The 6th International Symposium of Domain Theory and Its Applications (ISDT'13)*, 2013.

[2] Moez A. AbdelGawad. *NOOP: A Nominal Mathematical Model Of Object-Oriented Programming.* Scholar's Press, 2013.

[3] Luca Cardelli. A semantics of multiple inheritance. In *Proc. of the international symposium on Semantics of data types*, pages 51–67, New York, NY, USA, 1984. Springer-Verlag New York, Inc.

[4] William R. Cook, Walter Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '90, pages 125–135, New York, NY, USA, 1990. ACM.

[5] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the java type system sound? *Theor. Pract. Object Syst.*, 5(1):3–24, January 1999.

[6] A. Igarashi, B. Pierce, and P. Wadler. Featherweight java: A minimal core calculus for java and gj. In *OOPSLA*, 1999.

[7] Barbara H. Liskov and Jeanette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16:1811–1841, 1994.

[8] David Macqueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphism. *Information and Control*, pages 95–130, 1986.

[9] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.