Inheritance *Is* Subtyping

Robert Cartwright

[1] Rice University
Houston, Texas U.S.A.
cork@rice.edu
[2] Halmstad University
Halmstad, Sweden
robert.cartwright@hh.se

Extended Abstract

Since Luca Cardelli wrote a seminal paper [1] 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 subtyping 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* [2] subsequently added recursive record types, including a more accurate typing for **this** in 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 is insufficient.

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 presents 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.

Since antiquity, mathematicians have implicity used types in describing mathematical constructions. In describing functions and other constructions, mathematicians typically designate

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

the sets (types) to which variables are presumed to belong. In addition, they usually indicate the set to which the output of a function or construction belongs. For example, the formula for the volume of a sphere (a function of the radius), the radius is presumed to be a real number and the volume output is also a real number. Of course, some functions like the identity function and the equality function are applicable to arbitrary mathematical objects. More recently, logicians and computer scientists have been concerned with defining families of similar objects (types) and precisely characterizing the input and output types of functions. For example, a higher-order function *twice* that that takes a unary function f as an argument and composes f with itself accepts all functions of type `T -> T` for some `T` and returns a function of the same type

In simple, 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 [5], 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 `ColorPoint` object with fields `x: Number`, `y: Number`, and `color: Color` can have type 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 fields. 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, `DefaultPlainDocument` and `DefaultStyledDocument` are two different specializations of the abstract class `AbstractDocument` but they share the common behavior (contracts) associated with `AbstractDocument`.

In mainstream OO design, subtyping conforms to the *Liskov substitution principle* [4]: types are characterized by behavioral contracts and every subtype `B` of a type `A` obeys the contracts of the parent type `A`. 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. In practice, a decidable type system should perform static checks that help the programmers confirm that their code obeys the Liskov substitution principle.

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 subclasses of `C`. Hence, a 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, the signature of an overriding method typically must exactly match the signature of 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 [3] relative to operational semantics for these languages. This paper presents a denotational model of OOP 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*.

References

[1] 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.

[2] 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.

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

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

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