Reconciling Inheritance and Subtyping

A Nominal Denotational Model of Object-Oriented Programming

Robert "Corky" Cartwright

Department of Computer Science
Rice University
Houston, Texas 77005
USA

Sektionen för informationsvetenskap, data–och elektroteknik (IDE)
Högskolan i Halmstad
301 18 HALMSTAD
Sweden

cork@rice.edu

14 October, 2013

Joint work with Moez Abdel-Gawad.

What Is Object-Oriented (OO) Programming (OOP)?

An *object* is a record together with a syntactic signature.

- The record binds record label names (symbols) to either:
  - ground values called *fields*, or
  - functions called *methods* that take the record itself (called `this` or `self`) as an implicit first argument.
- Hence, an object is a collection of fields and methods (called *members*) together with a signature.
- In untyped OO languages, the signature is simply a list of member names and their arities.
- In typed OO languages, the signature specifies the types of all members of the record as well as `this` (the record itself).
- Formulating data values as objects (together with inheritance) is the defining feature of OOP.

Note: my terminology is consistent with common Java nomenclature.

In structural models of OOP, signatures are elided.

- Structural models of OOP are based on models for functional programming (FP).
- Objects are simply records of a particular form: the first argument of every method (a record field containing a function) is bound to the record itself (`this`). Hence, all signature information is discarded except for the names of record members.
- In typed structural OO languages, the type of `this` is the record type corresponding to its members.
- This elision, which produces an incorrect and misleading model for mainstream OO languages, has far-reaching implications.

Most OO languages are class-based, nominally typed.

A *class* is a is a template for constructing objects with the same members (modulo different field bindings) and signature.

- Classes have names (name spaces may be segregated).
- In well-designed OO programs, Each class has associated *contracts* describing the behavior of objects in the class. The constracts (which only appear in class documentation), include:
  - An invariant (predicate) for the values of class fields.
  - At least one contract for each method stipulating:
    - what conditions the inputs (including `this`) should satisfy, and
    - what output predicate should hold over the velue returned by the method and what side effects have been performed on `this` and perhaps other objects passed as arguments the method.

    The output predicate may mention the values of arguments and if often called an *input-output* predicate.

In practice, class contracts are expressed in code documentation and may be incomplete. The are typically informal.

Classes in Structural vs Nominal OO languages

Structural OO languages

Early models of OOP pre-date explicitly nominal OO languages.

- Cardelli published his seminal paper on semantics of inheritance in 1984. C++ was in gestation. SmallTalk is largely agnostic.
- In the structural model of OOP, objects are simply records. In particular, class names are not included as part of object denotations.

Classes in Structural vs Nominal OO languages

Structural OO languages

Early models of OOP pre-date explicitly nominal OO languages.

- Cardelli published his seminal paper on semantics of inheritance in 1984. C++ was in gestation. SmallTalk is largely agnostic.
- In the structural model of OOP, objects are simply records. In particular, class names are not included as part of object denotations.

Nominal OO languages

Class names are embedded in objects in mainstream OO languages.

- Some primitive operations depend on the class name (*e.g.*, Java `instanceof`, casting, reflective operations [`getClass`]).
- Class names play a critical role in class signatures which govern inheritance and subtyping.
- Java, C#, and C++ all have nominal type systems.

Framework for Presenting Concrete Examples

Canonical Core Programming Language for OOP

Enhanced Functional Subset of Java with:

- No primitive values ("everything is an object").
- No static members (impure without Class objects).
- No mutation.
- No interfaces ("use abstract classes instead").
- Multiple inheritance. Multiply inherited fields are not duplicated.
- Elided constructors. Outside of model.
- Small set of base classes: `Object`, `Boolean`, `Integer`.
- Optional (first-class) generics. Model easily accommodates generic classes and methods. Orthogonal to sequel.

Sample Classes

Example Classes

- Java Virtual Machines preload thousands of classes. Gross overkill.
- Model makes no commitment to what base classes are available.

Sample Classes

Example Classes

- Java Virtual Machines preload thousands of classes. Gross overkill.
- Model makes no commitment to what base classes are available.

Class Object

```
class Object {
  Boolean equals(Object o){ ... }
}
```

Sample Classes

Example Classes

- Java Virtual Machines preload thousands of classes. Gross overkill.
- Model makes no commitment to what base classes are available.

Class Object

```
class Object {
  Boolean equals(Object o){ ... }
}
```

Class Pair

```
class Pair extends Object {
  Object first, second;
  Boolean equals(Object p){ ... }
  Pair swap(){
    return new Pair(second, first);
  }
}
```

Semantic Typing

Semantic Type

A type is a set of object values with similar properties and behavior.

- In statically-typed languages, the compiler checks that program operations respect types.
- Tractable object-oriented type systems rely on characterizing object shapes.

Semantic Typing

Semantic Type

A type is a set of object values with similar properties and behavior.

- In statically-typed languages, the compiler checks that program operations respect types.
- Tractable object-oriented type systems rely on characterizing object shapes.

Liskov Substitution Principle for Semantic Subtyping

If S is a subtype of T, then the contracts for type T hold for type S.

- In nominal OO programming languages, type *names* are associated with (informal) contracts. Hence, in well-written programs, these names have more semantic content that mere shape information.
- Poorly written OO programs do not follow Liskov principle. Most discussions of OO type-checking in PL literature ignore it.

Syntactic Typing

- Types of class members are explicitly declared.
- Syntactic types are expressions describing class shapes, perhaps using class names as tags for contracts restricting the objects belonging to class types.
- Syntactic typing rules are decidable; easily enforced by a compiler.
- Structural versus nominal perspectives:
  - Structural typing ignores class names and inheritance relationships (Liskov principle); suffers from "spurious subtyping".
  - Nominal typing respects inheritance relationships (Liskov substitutability); programs that fail to conform to the (undecidable) Liskov principle may type check.
  - Structural and nominal type checking are generally incompatible; they are based on different semantic models and different definitions of syntactic types.
- Both forms of type-checking are open; each class is type-checked against potential new classes belonging to the types in its declaration.

Spurious Subtyping in Structural OOP

- All statically-typed mainstream OOP languages are nominally typed. Each class C is identified with the semantic type consisting of all instances of the class C and instances of the classes extending C. (subclasses via inheritance).
- Essentially all papers (books, methodologies) on OO design take a nominal perspective. Liskov substitutability is wired into the literature on OOP. From this perspective, structural type checking is not just foreign, it is WRONG.
- In putative structurally-typed OOP languages (in PL papers), a class is identified with the domain of records conforming to its shape *ignoring its name*, Record subytping is allowed (extra members are ignored and method types may be narrowed). The Liskov substitution principle is ignored. In essence, a type is simply a shape; a class name does not serve as a tag for a set of contracts. *Classes with compatible shapes may have different contracts.*

Example of Spurious Subtyping

Class MultiSet

```
class MultiSet extends Object {
Boolean equals(Object ms) { ... }
Void insert(Object o) { ... }
Void remove(Object o) { ... }
Boolean isMember(Object o) { ... }
}
```

Example of Spurious Subtyping

Class MultiSet

```
class MultiSet extends Object {
Boolean equals(Object ms) { ... }
Void insert(Object o) { ... }
Void remove(Object o) { ... }
Boolean isMember(Object o) { ... }
}
```

Class Set

```
Class Set
class Set extends Object {
Boolean equals(Object s){ ... }
Void insert(Object o) { ... }
Void remove(Object o) { ... }
Boolean isMember(Object o) { ... }
}
```

Syntactic Typing

- Concrete text for describing types.
- Type inference systems manipulate syntactic types.
- In nominal (mainstream) OO languages, syntactic types are typically class names with associated superclasses and shapes.
- In putative structurally-typed OO languages, class names are not included in syntactic types. Objects are simply treated as records.
- Avoids undecidability in checking; enforceable by compiler.
- Structural typing versus nominal typing:
  - Structural typing ignores class names and inheritance relationships.
  - Nominal typing focuses on class names as tags (proxies) for sets of contracts. Class extension relationships are paramount because they reveal contract propagation. Subtyping is identified with inheritance following the Liskov Substitution Principle.
  - From the perspective of nominal typing, structural typing is WRONG.

Early Models of OOP

Cardelli's Model of OOP [1984]

- Cardelli's OO language does not include recursive types.
- Inheritance is simply a syntactic abbreviation. All inheritance relationships in a program can be eliminated by repeatedly expansion of type definitions.
- No recursive types.
- Cardelli's domain (distilled to exclude variants):

$$\mathcal{V} \quad = \quad \mathcal{B} + (\mathcal{V} \to \mathcal{V}) + (\mathcal{L} \to \mathcal{V})$$

  where $\mathcal{B}$ is the domain of base values and $\mathcal{L}$ is the flat domain of labels.
- This model is structural; objects do not contain any nominal information.
- Syntactic inheritance implies subtyping but spurious subtyping can occur.

Subsequent Models of OOP

Elaborations on Cardelli's Model of OOP

- The subsequent quest by Cook for more precise typing (including recursive types) leads to the conclustion: inheritance is not subtyping.
- But Cook's perspective ignores the Liskov principle. In fact, the more precise typings he proposes to support "binary methods" explicitly violate the Liskov principle.
- Common binary method example is a class hierarchy where the `equals` method *only* accepts inputs of the the same type (narrowly defined) as `this`. Such typings for `equals` mean that instances of subclasses will generate run-time type errors when compared with objects belonging to a supertype.
- The OO software engineering community completely ignores PL research on inhertiance and OO typing *for good reason*.
- The denotational models that PL researchers use when thinking about OO programs are WRONG for nominal OO languages.

Overview: Constructing A Nominal Model of OOP

- Each object is a record augmented by a *signature* consisting of a class name and the descriptions of the types of all the class members. Signatures play a critical role in object denotations.
- The signature for an object specifies the signatures of all the class names mentioned in S.
- More precisely, an object denotation is a pair (sig, rec) where sig is *closed* (binds all class names in sig).

Class Signatures

Equations Defining Set of Signatures S

Given a set of class names N and a set of label names L:

$$\begin{aligned} S &= N \times N^* \times S_F^* \times S_M^* \\ S_F &= L \times N \\ S_M &= L \times N^* \times N \end{aligned}$$

where

- S_F is the set of field signatures; and
- S_M is the set of method signatures.

Notes:

- N and L are countable sets of symbols.
- Only the ordering in N^* within S_M matters.
- Every class signature has a name (its first component).
- Class signatures refer to other signatures by name.
- The supersignatures component N^* specifies immediate superclasses.

Signature Environments and Signature Closures

Signature Environment

A signature environment is a finite set of class signatures that:

- Has unique signature names, implying signature environments can be viewed as functions from class names to class signatures.
- Is referentially-closed.
- Has no class signature containing duplicate member names.
- Has no cycles in the supersignatures relation.
- Has member (field and method) signatures of supersignatures included in a class signature (enforcing inheritance constraints).

Signature Environments and Signature Closures

Signature Environment

A signature environment is a finite set of class signatures that:

- Has unique signature names, implying signature environments can be viewed as functions from class names to class signatures.
- Is referentially-closed.
- Has no class signature containing duplicate member names.
- Has no cycles in the supersignatures relation.
- Has member (field and method) signatures of supersignatures included in a class signature (enforcing inheritance constraints).

Signature Closure

A signature closure is a pair (nm, se) consisting of a class name nm and a signature environment se where: nm is defined in se; and every signature in se is reference-accessible from $se(nm)$, implying se is minimal.

Relations on Signatures

Extension of signature environments

- $se1 \blacktriangleleft se2 \quad \Leftrightarrow se1 \supseteq se2.$

Relations on Signatures

Extension of signature environments

- $se1 \blacktriangleleft se2 \quad \Leftrightarrow se1 \supseteq se2.$

Subsigning of signature closures

- Immediate subsigning ($\trianglelefteq_1$):

$$(nm1, se1) \trianglelefteq_1 (nm2, se2) \Leftrightarrow$$

$$(se1 \blacktriangleleft se2) \wedge (nm2 \in supSigs(se1(nm1))).$$

- Subsigning ($\trianglelefteq$):

$$\trianglelefteq \quad = \quad ReflexiveTransitiveClosure(\trianglelefteq_1).$$

Signature Examples

Class Signatures

$$Obj = (\texttt{Object}, [], [], [(\texttt{equals}, [\texttt{Object}], \texttt{Boolean})])$$

$$P = (\texttt{Pair}, [\texttt{Object}], [(\texttt{first}, \texttt{Object}), (\texttt{second}, \texttt{Object})],$$
$$[(\texttt{equals}, [\texttt{Object}], \texttt{Boolean}), (\texttt{swap}, [], \texttt{Pair})])$$

$$Bool = ..$$

Signature Examples

Class Signatures

$$Obj = (\texttt{Object}, [], [], [(\texttt{equals}, [\texttt{Object}], \texttt{Boolean})])$$

$$P = (\texttt{Pair}, [\texttt{Object}], [(\texttt{first}, \texttt{Object}), (\texttt{second}, \texttt{Object})],$$
$$[(\texttt{equals}, [\texttt{Object}], \texttt{Boolean}), (\texttt{swap}, [], \texttt{Pair})])$$

$$Bool = ..$$

Signature Environments

$$ObjSE = Obj, Bool$$

$$PairSE = Obj, Bool, P$$

Signature Examples

Class Signatures

$$Obj = (\texttt{Object}, [], [], [(\texttt{equals}, [\texttt{Object}], \texttt{Boolean})])$$

$$P = (\texttt{Pair}, [\texttt{Object}], [(\texttt{first}, \texttt{Object}), (\texttt{second}, \texttt{Object})], [(\texttt{equals}, [\texttt{Object}], \texttt{Boolean}), (\texttt{swap}, [], \texttt{Pair})])$$

$$Bool = ..$$

Signature Environments

$$ObjSE = Obj, Bool$$

$$PairSE = Obj, Bool, P$$

Signature Closures

$$ObjSC = (Object, ObjSE)$$

$$PairSC = (Pair, PairSE)$$

Signature Examples

Class Signatures

$$Obj = (\texttt{Object}, [], [], [(\texttt{equals}, [\texttt{Object}], \texttt{Boolean})])$$

$$P = (\texttt{Pair}, [\texttt{Object}], [(\texttt{first}, \texttt{Object}), (\texttt{second}, \texttt{Object})], [(\texttt{equals}, [\texttt{Object}], \texttt{Boolean}), (\texttt{swap}, [], \texttt{Pair})])$$

$$Bool = ..$$

Signature Environments

$$ObjSE = Obj, Bool$$

$$PairSE = Obj, Bool, P$$

Signature Closures

$$ObjSC = (Object, ObjSE)$$

$$PairSC = (Pair, PairSE)$$

Environment Extension

$$PairSE \blacktriangleleft ObjSE$$

Signature Examples

Class Signatures

$$Obj = (\texttt{Object}, [], [], [(\texttt{equals}, [\texttt{Object}], \texttt{Boolean})])$$

$$P = (\texttt{Pair}, [\texttt{Object}], [(\texttt{first}, \texttt{Object}), (\texttt{second}, \texttt{Object})], [(\texttt{equals}, [\texttt{Object}], \texttt{Boolean}), (\texttt{swap}, [], \texttt{Pair})])$$

$$Bool = ..$$

Signature Environments

$$ObjSE = Obj, Bool$$

$$PairSE = Obj, Bool, P$$

Signature Closures

$$ObjSC = (Object, ObjSE)$$

$$PairSC = (Pair, PairSE)$$

Environment Extension

$$PairSE \blacktriangleleft ObjSE$$

Subsigning

$$PairSC \trianglelefteq_1 ObjSC$$

$$PairSC \trianglelefteq ObjSC$$

Constructing A Domain $\mathcal{O}$ of Nominal Objects

Domain Equation for $\hat{\mathcal{O}}$

$$\hat{\mathcal{O}} = \mathcal{S} \times (\mathcal{L} \multimap \hat{\mathcal{O}}) \times (\mathcal{L} \multimap (\hat{\mathcal{O}}^{+} \multimap\!\!\rightarrow \hat{\mathcal{O}}))$$

where $\mathcal{S}$ is the flat domain of signature closures, $\mathcal{L}$ is the flat domain of label names, $\multimap$ constructs the domain of finite records over the specified flat label domain and value domain, and $\multimap\!\!\rightarrow$ is the strict function space constructor.

$\hat{\mathcal{O}}$ is the least domain satisfying this equation.

Constructing A Domain $\mathcal{O}$ of Nominal Objects

Domain Equation for $\hat{\mathcal{O}}$

$$\hat{\mathcal{O}} = \mathcal{S} \times (\mathcal{L} \multimap \hat{\mathcal{O}}) \times (\mathcal{L} \multimap (\hat{\mathcal{O}}^{+} \multimap\!\!\rightarrow \hat{\mathcal{O}}))$$

where $\mathcal{S}$ is the flat domain of signature closures, $\mathcal{L}$ is the flat domain of label names, $\multimap$ constructs the domain of finite records over the specified flat label domain and value domain, and $\multimap\!\!\rightarrow$ is the strict function space constructor.
$\hat{\mathcal{O}}$ is the least domain satisfying this equation.

Filtering $\hat{\mathcal{O}}$ to form $\mathcal{O}$

The domain $\hat{\mathcal{O}}$ is a simple construction that includes many ill-formed objects with member names or values that are inconsisent with the specified signature closure. Every inconsistent finite object in the basis for $\hat{\mathcal{O}}$ can be filtered out to form a clean basis for objects with proper signatures.

Inheritance = Subtyping

Semantic Type of Signature Closure

The class type corresponding to a signature closure sc is the subset of $\mathcal{O}$ containing the bottom object and all objects that have a signature closure that subsigns sc.

$$\mathcal{O}[sc] = \{(scs, fr, mr) \in \mathcal{O} \mid scs \blacktriangleleft sc\} \cup \{\bot_{\mathcal{O}}\}$$

Inheritance = Subtyping

Semantic Type of Signature Closure

The class type corresponding to a signature closure sc is the subset of $\mathcal{O}$ containing the bottom object and all objects that have a signature closure that subsigns sc.

$$\mathcal{O}[sc] = \{(scs, fr, mr) \in \mathcal{O} \mid scs \blacktriangleleft sc\} \cup \{\bot_{\mathcal{O}}\}$$

Key Theorem

Subsigning $\Leftrightarrow$ Subtyping: For two signature closures $sc1$ and $sc2$,

$$sc1 \blacktriangleleft sc2 \Leftrightarrow \mathcal{O}[sc1] \subseteq \mathcal{O}[sc2]$$

In other words, inheritance between two classes, formalized as subsigning, directly corresponds to semantic subtyping between between these classes.

Conclusions

Extended Technical Exercise in Denotational Semantics

The construction of the model proceeded much as I expected with two surprises:

- All possible nominal objects neatly fit into a single domain. OO programs introduce so many new forms of data that I originally expected build domains on a per program basis.
- The fact that the nominal OO domain is in some sense *universal* casts some light on why compilers for nominal OO languages can accurately type check open programs that are subsequently extended in arbitrary ways. All of the possible new object classes and instances (assuming the existing code base does not change) are already in the domain and included in the types specified in the existing program signatures.

Future of Nominal OO Languages

Better OO Type Systems?

- Scala and F# are far ahead of major commercial platforms.
- Perhaps Java 8+K will support a richer type system, but backward compatibility is an impediment.
- Better type systems for nominal OO languages is still a fertile research area. Nominal OO type systems is an under-researched area. Perhaps a more appropriate semantic model can help.

Future of Nominal OO Languages

Better OO Type Systems?

- Scala and F# are far ahead of major commercial platforms.
- Perhaps Java 8+K will support a richer type system, but backward compatibility is an impediment.
- Better type systems for nominal OO languages is still a fertile research area. Nominal OO type systems is an under-researched area. Perhaps a more appropriate semantic model can help.

Better Compilers?

- Object-inlining is the key to major increases in performance on a fixed architecture. CLR based languages have better prospects than JVM based languages in this regard. Some unfortunate design decisions in the evolution of Java/JVM (which are difficult to reverse given Java's commitment to backward compatibility) are likely to impede this form of optimization. The decision to use the existing wrapper classes in `java.lang` for autoboxing in Java 5 particularly stands out.