

RICE UNIVERSITY

NOOP
A Mathematical Model Of
Object-Oriented Programming

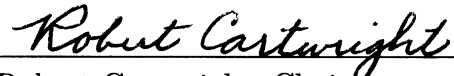
by

Moez A. AbdelGawad

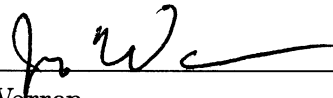
A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

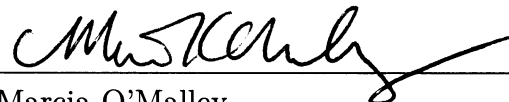
APPROVED, THESIS COMMITTEE:



Robert Cartwright, Chair
Professor of Computer Science



Joe Warren
Professor of Computer Science and
Department Chair



Marcia O'Malley
Associate Professor of Mechanical
Engineering

Houston, Texas

May, 2011

ABSTRACT

NOOP

A Mathematical Model Of Object-Oriented Programming

by

Moez A. AbdelGawad

Computer software is ubiquitous. More than 35×10^{18} computer instructions are executed around the globe each second. As computers dominate more aspects of our lives, there is a growing need to reason more accurately about computer software. Most contemporary computer software is written using object-oriented (OO) programming languages, such as JAVA, C#, and C++. *How should we mathematically characterize object-oriented software?* This is the question this thesis addresses by presenting an accurate domain-theoretic model of mainstream object-oriented programming.

Mainstream object-oriented languages are class-based. In such languages, the name of a class is part of the meaning of an object, a property often called “nominality”. Most mainstream OO languages also conform to a static type discipline. Hence, the focus of this thesis is the construction of an accurate model of nominal, statically-typed OO languages.

In statically-typed nominal OO languages, class names are also part of the meaning of corresponding class types, and class inheritance (subclassing) is explicitly declared; one class is a subclass of another only if it is declared as such. When static

type systems are formulated to describe sets of objects, subtyping is defined so that subclassing is consistent with subtyping. Nevertheless, some programming languages (PL) theoreticians dismiss this identification as a design error because the only published models of OO languages exclude nominal information from objects and define subtyping in a way that ignores nominality.

In nominal OO languages, program behavior depends on the nominal information embedded in objects. This thesis builds a model of OO languages called **NOOP** that includes nominal information and defines static types in accord with mainstream OO language designs. In **NOOP**, the meaning of every object includes its class name. Similarly, types are defined such that objects belong to a particular class type if and only if they are members of classes that inherit from the class corresponding to the class type.

To demonstrate the utility of the model, we show that in **NOOP** inheritance and OO subtyping coincide. This work shows that mainstream OO languages are not technically defective in identifying inheritance and subtyping. In models that include nominal information and define types that respect nominal information, this identification is mathematically correct. The folklore among OO programming language researchers that “inheritance is not subtyping” is incorrect.

Dedication

To my mum, my brother, and my dad ...

Together with Corky, my Ph.D. supervisor, you all suffered, and sacrificed a lot—perhaps much more than you should have—to make it happen ...

And together with all the great respect, and great passion that I, and we all, have *much* of toward my Ph.D., I hope we are all honest and courageous enough to admit that we do *not* believe the heavy total price, paid by us all, was worth it!

Acknowledgments

There are many people that I would like to thank for helping me in my Ph.D., but if there is only one person I am allowed to thank then that person is definitely my corky Ph.D. supervisor, Professor Robert “Corky” Cartwright.

The tremendous amount of guidance and advice—corky and otherwise—Professor Cartwright provided me with, and the method and directions—corky or otherwise—Professor Cartwright employed in supervising me throughout our long journey were simply indispensable, and unparalleled. I am certain the research contained in this thesis is impossible to reproduce without these. There is no way for me, a finite human, to thank Professor Cartwright enough for providing me with his advice and directions.

If allowed to, I also would like to thank Professor Joe Warren much for his strong support of me during my Ph.D. journey, and the great patience he exercised till I was able to produce this thesis.

I would further like to thank Professor Marcia O’Malley for being a member of my thesis exam committee.

Next, without the supportive (and otherwise) environment at the Computer Science Department and at Rice University, I could not have reached this particular moment at which I am writing this acknowledgment. I thank them all, everyone in both, for all they did for (and against) me to make it happen.

Other than Professor Cartwright, Professor Walid Taha has taught me much and motivated me to learn more about PL research. I’d like to thank him much for that.

Throughout my Ph.D. journey, I also benefited much from my ex-colleague Dr. Eric Allen (particularly his opinion about domain theory). I also enjoyed and had

the pleasure of being in the company of Dr. Mathias Ricken, to whom I give special thanks for our friendly academic and non-academic discussions. Throughout the journey, I also had the pleasure to know my Masters colleague Michael Jensen, and every member of the Rice JavaPLT research team.

Finally, just like Professor Cartwright, there is no way I could fully compensate and thank all my friends and family members for all the support, encouragement, motivation, and sacrifices they made to let it happen. Your sacrifices, as well as our non-academic discussions and interactions, provided me with much of the needed fuel to keep feeling alive, and to thus keep me going. Every one of you was special, and each of you deserves a special mention. I console myself by the thought that you know who you are. I can do nothing but simply say to all and each one of you: Thank You.

Contents

Abstract	ii
Acknowledgments	v
1 Introduction	1
1.1 Nominal OOP	2
1.2 Thesis Overview	4
2 Object-Oriented Programming	7
2.1 OOP Notions and Terminology	7
2.1.1 Objects, Fields and Methods	7
2.1.2 Encapsulation	8
2.1.3 Classes, Class Names and Nominality	9
2.1.3.1 Shapes, Object Interfaces and Nominal Typing	11
2.1.4 Object Types, Object Type Expressions and Signatures	15
2.1.5 Inheritance, Subclassing and Nominal Subtyping	17
2.1.6 Signatures, Subsigning and Substitutability	20
2.2 Nominal OOP versus Structural OOP	22
2.2.1 The Meaning of Being an Object	22
2.2.2 The Semantic Value of Nominal Typing	23
2.2.3 Nominal Typing and Subtyping versus Structural Typing and Subtyping	24
2.3 Statically-typed OO Languages versus Dynamically-typed OO Languages	26

3	Motivation and Background	28
3.1	NOOP Motivations	28
3.1.1	Shifting PL Research View of OOP to be Nominal	29
3.1.2	Enabling Progress in Type Systems of Mainstream OO Languages	31
3.2	Related Research	32
3.2.1	Main PL Research Sources	32
3.2.2	PL Research on Functional Programming	34
3.2.2.1	Domain Theory and Denotational Semantics	34
3.2.3	PL Research on Object-Oriented Programming	37
3.2.3.1	Early OO Research	38
3.2.3.2	Recent OO Research	41
4	COOP: A Simple Structural Model of OOP	43
4.1	Records Domain Constructor (\dashv)	46
4.1.1	Record Functions	46
4.1.2	Definition of \dashv	47
4.1.3	Properties of the Records Domain Constructor (\dashv)	49
4.1.3.1	Domain Constructor \dashv is Continuous	49
4.2	COOP Domain Equation	49
4.3	COOP Construction	50
4.3.1	A General COOP Construction Iteration	51
4.3.2	The Solution of the COOP Domain Equation	52
5	Signatures and Nominality	53
5.1	Signature Constructs	53
5.1.1	Class Signatures	55
5.1.2	Signature Environments	58
5.1.2.1	Circularity in Signature Environments	59

5.1.3	Signature Closures	60
5.2	Signature Equality	61
5.2.1	Equality of Class Signatures	61
5.2.2	Equality of Signature Environments and Signature Closures	62
5.3	Extension of Signature Environments	62
5.4	Inheritance and Subsigning	63
6	NOOP: A Domain-Theoretic Model of Nominal OOP	65
6.1	NOOP Domain Equation	67
6.2	Construction of <i>pre</i> NOOP	68
6.2.1	A General <i>pre</i> NOOP Construction Iteration	68
6.2.2	Ranking Finite Domain Elements	69
6.3	Filtering of <i>pre</i> NOOP to NOOP	71
6.3.1	Filtering is a Finitary Projection	74
6.3.2	The NOOP Domain of Objects	75
6.4	Properties of NOOP	75
6.4.1	Semantics of Signatures	76
6.4.2	Signatures Denote Subdomains of \mathcal{O}	76
6.4.2.1	Types as Subdomains	76
6.4.2.2	Exact Object Types	77
6.4.2.3	Nominal Object Types are Subdomains	77
6.5	Reconciling Inheritance with Subtyping	78
7	Discussion and Future Work	81
7.1	Main Research Conclusions and Contributions	81
7.1.1	Main Research Conclusions	82
7.1.1.1	Comparing Nominal and Structural Views of OOP	82
7.2	Incidental Research Contributions	83
7.3	NOOP Limitations	84

7.3.1	NOOP Models Immutable OOP	85
7.3.2	Invariant Subtyping of Method Signatures	85
7.3.3	NOOP is The Universe of a Model	86
7.4	Directions for Future Work	86
	Bibliography	88
	A Domain Theory	96
A.1	Basic Notions	96
A.2	Domains of Functions	102
A.3	Domain Constructors	105
A.3.1	Coalesced Sum (+)	105
A.3.2	Strict Product (\times)	106
A.3.3	Continuous Functions (\rightarrow)	107
A.3.4	Strict Finite Sequences (\mathcal{D}^*)	107
	B Proofs of Important Theorems	109
B.1	The Domain of Record Functions has an Effective Presentation	109
B.2	Domain Constructor \dashv is Continuous	112
B.3	Filtering is a Finitary Projection	114
	C Code Examples	119
C.1	Classes	119
C.2	Shapes	122
C.3	Object Interfaces/Record Types	123
C.4	Structural Subtyping	124
C.5	Signatures and Subsigning	125

Notation

Symbol	Meaning
$\{\dots\}$	Sets
$[\dots]$	Sequences
(\cdot, \cdot)	Pairs
$\{l_i \mapsto d_i\}, 1 \leq i \leq k$	Records
$\cdot _i$	Tuple Projection
$ \mathcal{D} $	Universe of Domain \mathcal{D}
$S_1 + S_2$	Disjoint Union Set Construction
$\mathcal{D}_1 + \mathcal{D}_2$	Coalesced Summation Domain Construction
$S_1 \times S_2$	Cartesian-Product Set Construction
$\mathcal{D}_1 \times \mathcal{D}_2$	Strict Product Domain Construction
$\mathcal{A} \multimap \mathcal{B}$	Strict Continuous Functions Domain Construction
$\mathcal{L} \multimap \mathcal{D}$	Records Domain Construction
$<:$	Subtyping
\triangleleft	Subsigning

Chapter 1

Introduction

I think the fact that [biological] cells are software-driven machines, and that this software is DNA and that truly the secret of life is writing software, is pretty miraculous.

~Dr. Craig Venter, a lead microbiologist in mapping the human genome
(edge.org, November 2010)

Today, around the globe, over thirty-five million trillion (35×10^{18}) computer instructions get executed each second, and this figure has an annual growth rate of more than 55% [35]. Obviously, computer software is becoming ubiquitous.

As software permeates every aspect of our lives, however, software errors and software bugs are becoming ubiquitous too. These could overwhelm us, if we are not careful. Thus, there is an ever-growing need to analyze computer software accurately.

Most contemporary computer software is written using object-oriented programming (OOP) languages. Thus, there is a natural interest in determining how we can properly reason and think about object-oriented software.

How should we mathematically characterize object-oriented software? This is the main question we attempt to answer in this thesis. We do so by presenting a mathematical model of mainstream object-oriented programming. We call this model **NOOP**.

Without a precise mathematical model of OOP, it is hard to reason about and discuss many of the questions related to the properties and behavior of object-oriented software. More importantly, the lack of a precise model of OOP makes it harder to

convincingly answer these questions. An imprecise model of OOP may lead us to make wrong conclusions about OOP and object-oriented (OO) software.

1.1 Nominal OOP

Mainstream OOP is class-based. Classes have names. Class names are associated with behavioral contracts for objects. Because of this, class names play an important role in the way OO developers think about programs and in the structure of type systems of mainstream OO languages. These type systems have the following common characteristics:

1. *Nominal objects*: Objects in mainstream OOP are nominal because the name of the class of an object is carried inside the object as part of the identity (*i.e.*, the meaning) of the object.
2. *Nominal object types* (class types): In statically-typed OO languages, a class name is used as a type name that identifies an object type (the set of all objects constructed using the named class or using explicitly-declared subclasses of the named class).
3. *Nominal subtyping*: Subclassing (*i.e.*, inheritance) in mainstream OOP is explicitly declared between classes, using class names. A class type subtypes another class type if and only if the former is an explicitly declared subclass of the latter.

The type systems of mainstream OO languages all share these characteristics. These type systems are thus often called *nominal OO type systems*.

Earlier models of OOP, such as the well-known model Cardelli developed in [13, 14], do not model the nominal aspects of mainstream OOP. Unlike nominal OOP,

objects and object types in Cardelli’s model only carry information about the structure of objects. Hence, objects in Cardelli’s model are *structural* objects, often called *records*, and object types in Cardelli’s model are structural types. Further, although inheritance is not formalized in Cardelli’s model, his paper informally equates it with subtyping between structural types. This view of inheritance has been rejected by OO software developers and most PL researchers, because structural subtyping does not imply any sharing of type implementations (and thus any sharing of behavior).

OO languages based on structural objects and object types are called *structural OO languages*. Examples of such languages include STRONGTALK [8], MOBY [29], POLYTOIL [10], and OCAML [43]. To make a distinction between different OO languages that have different kinds of type systems, mainstream OO languages are thus sometimes called *nominal OO languages*. Structural OO languages are not common. For the most part, they are only used by OO programming language (PL) researchers. Despite the fact that mainstream OO languages are nominal OO languages, most PL research on OOP and on OO languages, however, assumes and builds on Cardelli’s structural model of OOP.

NOOP, our model of OOP, is based on a nominal view of objects and classes, consistent with the formulation of objects in mainstream OO languages. In contrast to Cardelli’s model (which we call **SOOP** for Structural Object Oriented Programming), our model includes class names in objects and defines class types in a way that respects the declared class hierarchy. As a result, subtyping has a fundamentally different interpretation in our model than it does in Cardelli’s model and in more elaborate models based on Cardelli’s work.

Inheritance is a defining feature of object-oriented programming. As a demonstration of the utility and validity of **NOOP** as a more precise model of mainstream OOP,

we use **NOOP** to prove that nominal subtyping in mainstream OOP *completely* reconciles inheritance and semantic object subtyping. This reconciliation goes against the common folklore among PL researchers, which asserts that “inheritance is not subtyping”. This folklore is based on a mathematical analysis of structural models of OOP (such as **SOOP**). The proof that **NOOP** completely reconciles inheritance and OO subtyping underlines the importance of including the nominal aspects of mainstream OO languages in models of mainstream OOP.

1.2 Thesis Overview

In this thesis, we present the construction of **NOOP** on three steps.

1. First, we construct a simple structural model for OOP, which we call **COOP**.
2. Next, we define *signatures* as pieces of syntax that are similar to object type expressions.
3. Finally, we use signatures to define nominal objects, and we then construct **NOOP** as a model of nominal OOP using essentially the same construction technique that we used to build **COOP**.

After constructing **NOOP** and proving that it is well-defined, we use it to define nominal object types and prove that nominal subtyping completely reconciles inheritance and semantic subtyping.

Thus, the remainder of this thesis is organized as follows:

- Chapter 2, OOP: A Technical Overview, presents a more detailed overview of nominal typing notions in nominal OOP, and contrasts them with their counterparts in structural OOP.

- Chapter 3, Motivation and Background, presents the motivations behind developing our research and an overview of earlier research similar to or related to ours.
- Chapter 4, **COOP**: A Simple Structural Model of OOP, presents **COOP** as a simple structural model of OOP. **COOP** models objects as records. In Chapter 4 we thus also present, in detail, a records domain constructor (\dashv). Using standard domain-theoretic construction methods, we then use the records domain constructor and other standard domain constructors to construct **COOP**.
- Chapter 5, Signatures and Nominality, defines class signatures and all necessary related entities. Consistency conditions and a *subsigning* relation are defined for signature closures, so that signatures have properties that agree with our intuitions about object types in nominal OO languages.
- In Chapter 6, **NOOP**: A Domain-Theoretic Model of Nominal OOP, we construct our domain-theoretic model of mainstream OOP. In **NOOP**, signature closures, of Chapter 5, are paired with records, of Chapter 4, to define nominal objects, as models of objects in mainstream OOP. **NOOP** is constructed using a similar construction method to that used for constructing **COOP**. An additional domain-filtering step is used to guarantee that signature closures are paired with matching records. Finally, in this chapter also we prove that inheritance and subtyping are completely reconciled in mainstream OOP.
- In Chapter 7, Discussion and Future Work, we discuss the work presented in this thesis in less-technical terms, and we make some general conclusions. We also present directions for possible future work.

- Finally, three appendices at the end of this thesis present (1) a brief overview of the main domain theoretic notions used in this thesis (Appendix A), (2) proofs of the important theorems in this thesis (Appendix B), and (3) few code examples that help demonstrate the notions presented in this thesis and the main differences between nominal OO languages and structural OO languages (Appendix C).

Chapter 2

Object-Oriented Programming: A Technical Overview

The beginning of wisdom is to call things by their right names.

~Chinese Proverb

In this introductory chapter, we present an overview of some main notions of mainstream OOP, and we give informal definitions for these technical notions that will help motivate later formal definitions. Later in the chapter, we use these informal technical definitions to make a clear distinction between *structural OOP*, a view of OOP that is commonly held by PL researchers, and *nominal OOP*, the view of OOP commonly held by mainstream OO software developers. In Appendix C, we present a few code examples that demonstrate the concepts and notions discussed in this chapter.

This chapter only assumes some familiarity with mainstream OOP and with basic mathematical notions (like sets and functions), but the chapter does not assume much familiarity with OOP research or PL research in general.

2.1 OOP Notions and Terminology

2.1.1 Objects, Fields and Methods

An informal view of objects in object-oriented programming is “objects as service providers”. In this view, an object is ‘an entity that provides a service’. Objects in

OO software provide their respective services by providing object fields and object methods. A *field* of an object is a binding of a name to an object, while a *method* of an object is a binding of a name to a function that performs a computation (*e.g.*, accessing fields of objects and/or calling their methods) when invoked on zero or more objects, as *method arguments* and returns an object as a result. The object containing a method is always available as an implicit argument to that method under the name **this** or **self**. Collectively, the fields and methods of an object are called the *members* of the object. Fields are sometimes also called *instance variables*.

The set of members of a given object is fixed and finite. Member names are typically plain labels (alphanumeric identifiers). An object responds to a method call by itself calling other methods (of itself or of other objects) then returning a result object. Collectively, the response of an object to field accesses and method calls, and the logical relation between this response and the method arguments, defines the *behavior* of the object. The behavior of an object defines the service the object provides.

2.1.2 Encapsulation

Object-oriented programming is defined by two main features: encapsulation, and inheritance. In this section we discuss encapsulation. We will discuss inheritance in Section 2.1.5.

Because an object is a service provider, the members of an object are not a collection of unrelated, independent members. Rather, members of an object collectively share responsibility for providing the service the object is designed to provide (*i.e.*, they mutually-depend on each other in provide the service) . This mutual dependency particularly pertains to the “active” component of an object, *i.e.*, its methods.

In object-oriented programming, methods of an object often call one another (*i.e.*, an object can recursively call its own methods). The fields of an object record the “state” of the object. Methods of the object can access these fields to obtain this information.

An object is said to *encapsulate* its data because the object pairs its fields with methods which may access and manipulate these fields. Because fields are members embedded inside an object, methods of the object can access the fields of the object (*i.e.*, the data the object encapsulates) in the same way as they can call other methods of the object (via the special variable **this** or **self**).

The encapsulation of data (fields) with functions (methods) that process them was a major motivation behind the development of OOP. SMALLTALK was the language that popularized OOP widely. About SMALLTALK’s existence, and the existence of OOP by implication, Alan Kay wrote [40]:

“Smalltalk’s design, and existence, is due to the insight that everything we can describe can be represented by the recursive composition of a single kind of behavioral building block that hides [*i.e.*, encapsulates] its combination of state and process inside itself and can be dealt with only through the exchange of messages [field accesses and method calls].”

2.1.3 Classes, Class Names and Nominality

Many objects in an OO program respond to the same set of field accesses and method calls in a similar way. These objects, thus, share similar behavior, and they provide the same service, only with some little variations. A *class* is a syntactic programming construct that mainstream OO languages offer for the specification of the common behavior shared by some objects that provide the same service.

Classes are used to create (or produce, or construct) objects with common behavior. A class is a template from which objects that share the same behavior are produced. Objects produced using a certain class are called *instances* of that class.

A class has a name, called a *class name*. A class name is typically associated with a *behavioral contract*, *i.e.*, with a specification (formal or informal) of the service provided by instances of the class.¹ The association of class names with behavioral contracts is commonly used by mainstream OO developers, since it enables developers to design their software based on the behavioral contracts of objects in their software².

A class in mainstream OOP always has a special “meta-method” called an (*object*) *constructor*. The constructor of a class is used to form objects (instances) of the class. Typically, a constructor has code that initializes the fields of an object. As special class methods, constructors usually have the same name as the class name of the objects they construct. In program text, the name of the constructor, which is the same as the class name, *ties the constructed object to its class*.

Thus, in mainstream OOP, instances of a class have the name of that class as part of their identity. Objects with class names embedded inside them are *nominal objects*. Nominal objects are tied to the class that created them, via class names. Having names as part of the meaning of objects (*i.e.*, their identity) is called *nominality*. An OO language with nominal objects is a *nominal* OO language.

The nominality of objects in mainstream OOP implies that two objects produced from classes with different names are not equal objects³. Because class names are

¹Behavioral contracts are usually attached to classes in the form of comments, sometimes even in a standardized comment format like that of JAVADOC comments.

²Via tests like the `instanceof` check in JAVA, and `isMemberOf` in SMALLTALK, etc.

³Mathematical equality of objects is meant here, not programmatic equality. For programmatic equality, where the `equals()` method is program-defined (*e.g.*, in languages like JAVA), it is possible for a program to equate any two objects. This practice is generally regarded as a programming error, but programming equality has no bearing on the mathematical equality under discussion.

associated with behavioral contracts (or, equally, with the service that objects of a class provide), two objects having different class names are considered different in mainstream OOP because the different class names embedded inside them imply that the objects provide different services or, equally, that the objects satisfy, or abide by, different behavioral contracts.

Informally, a class can be considered as a “cookie cutter”, from which cookies (*i.e.*, objects) that behave the same are molded. As being an integral part of the identity of objects, class names are thus “baked into” objects (*i.e.*, into the “cookies”).

As explained above, nominal OOP languages have class names embedded in instances of the classes. An OOP language that does *not* embed class names in objects is called a *structural OOP* language.

Objects in a structural OOP language are simply records containing fields and methods; class names do not appear in structural objects.

Figures C.1, C.2, and C.3, in Appendix C, present JAVA-like code for classes `Object`, `Pair` and few simple classes that we use to demonstrate the concepts we discuss in this chapter.

2.1.3.1 Shapes, Object Interfaces and Nominal Typing

In this thesis we call the set of names of members of an object the *shape* of an object. We believe the notion is important and intuitive enough to have a name of its own. It should be noted that shapes are only sets of labels.

Given that the set of members of an object is fixed, the shape of an object is an invariant of objects of the same class. The shape of an object can, thus, be derived from the class of an object (*i.e.*, the class used to produce the object). Given that the shape of all instances of a class is the same (is invariant), shapes can also be

associated with classes rather than just with objects. The shape of an object or of a class is called the shape *supported by* the object or the class.

A shape that has all member names belonging to another shape, and possibly some more, is called a *supershape* of the other shape. Dually, the other shape (with the smaller set of member names) is called a *subshape* of the first (larger) one. As sets, a subshape is always a subset of its supershapes.

Figure C.4, in Appendix C, presents examples for shapes.

Objects have interfaces. An object interface⁴ specifies how an object is viewed and should be interacted with by other objects, *i.e.*, by “the outside world” (*i.e.*, the interface of an object specifies how objects can access fields and call the methods of the object).⁵

Further, members of an object have interfaces. Member interfaces specify how an object member is viewed and should be interacted with by other objects. Member interfaces and object interfaces mutually depend on each other.

A *field interface* tells the name of the field and the interface of objects that can be bound to the field. A *method interface* tells the name of the method, as well as the interface of objects that can be passed to the method as arguments and also the interface of the result object. An *object interface* includes member (field and method) interfaces of the members of an object.

Like the shape of an object, an object interface is an invariant of the objects of a class, and thus it can also be derived from the class of an object.

In nominal OOP, *class signatures* express object interfaces (See Section 2.1.6 for

⁴Despite some similarity, our informal notion of an object interface should not be confused with the more concrete, formal notion of **interfaces** that exists in some languages such as JAVA.

⁵The interface of an object, thus, sort of tells the “set of rules” other objects have to follow to interact with the object.

a discussion of class signatures). A significant difference between structural OOP and nominal OOP is that, in nominal OOP, an object interface (as expressed in *class signatures*) also includes the name of the class of the object, and the names of the superclasses of the class, together with interfaces of members of the class (expressed as *member signatures*) . In nominal OOP, an object interface is thus, roughly, the shape of objects of a class augmented with extra information, *i.e.*, the class name, superclass names, and extra information on member interfaces.

Object interfaces will be the basis for the formal definition of object types and class signatures below. A nominal OO language where nominal objects are also associated with class signatures is a *nominally-typed* OOP language.

Exact Shapes and Exact Object Interfaces When we discuss inheritance, in Section 2.1.5, we will see that an object can be associated with more than one shape, and more than one object interface. For a given object, the object interface of the object derived from the definition of the particular class used to produce the object is called the *exact object interface* of the object. The *exact shape* of the object is the shape derived from the class of the object, which is the same as the shape derived from the exact interface of the object. Other shapes an object can be associated with are subsets (subshapes) of its exact shape.

The exact interface of an object is the object interface that includes member interfaces of *all* members of the object (and nothing more), and member interfaces in the object interface are *exact member interfaces*, *i.e.*, ones where field and method interfaces use exact object interfaces. Relative to a particular object, the exact shape and the exact interface are the most precise and the most specific of the multiple shapes and interfaces that can be associated with the object.

Class Names and Circular Object Interfaces Due to using class names as interface names, an object interface in nominal OOP (when expressed as a class signature) can circularly refer to itself, using its own name. Multiple object interfaces that are mutually-circular also easily refer to each other using their interface names. Circular class definitions are quite common in mainstream OOP⁶. Nominal-typing allows the easy expression of circular object interfaces.

In structural OO languages, due to the lack of class names, self-references inside object interfaces have to be expressed by requiring some *explicit* means for expressing recursive interfaces⁷. Due to requiring explicit recursion, object interfaces that have multiple mutually-recursive interfaces are usually notationally heavier to define, express, and manipulate in structural OO languages than their circular counterparts in nominal OO languages. The ease by which recursive typing notions can be expressed in nominally-typed OO languages is one of the main advantages of nominally-typed OOP. According to Benjamin Pierce [56, p.253], “The fact that recursive types come essentially for free in nominal systems is a decided benefit”.

In structural OO languages objects are associated with structural object interfaces, but not with nominal ones. OO languages where objects interfaces include no class names are *structurally-typed* OO languages.

Figures C.5 and C.6, in Appendix C, present code examples for (structural) object interfaces.

⁶For example, in purely OO languages, the definition of classes `Object` and `Boolean` is usually circular, because class `Object` has a method `equals()` that returns values of class `Boolean`, and class `Boolean` inherits from class `Object` and it also usually has methods which take or return objects of class `Object` (e.g., also the `equals()` method).

⁷Using, e.g., type variables and the μ operator for self-recursive interfaces, and using the `and` operator, together with μ and type variables, for expressing multiple mutually-recursive interfaces (as is done in the functional language OCAML [43], for example).

2.1.4 Object Types, Object Type Expressions and Signatures

In computer programming—whether it is procedural programming, functional programming, or object-oriented programming, or otherwise—every value has a type. The type of a value ensures the proper use of the value⁸. A type of a value may disallow improper use of the value by specifying what operations are allowed for that value (*i.e.*, what computations can be done using the value), and, accordingly, what operations are not allowed for it. In type-checked languages, the language compiler checks type declarations for consistent and proper use of program values.

In OOP, every object is a value, and, accordingly, each object in OOP has at least one object type.⁹ In OO programming language (PL) research, an object type is usually viewed semantically as *a set of objects with similar properties and behavior*. Given that checking the equality of behavior of objects is generally an undecidable problem, OO programming languages use object interfaces and other *syntactic* program features (such as inheritance) to characterize object types and to decide the similarity of the behavior of their objects.

Thus, PL research, a *type* has two meanings: a syntactic meaning, and a semantic one. Given that the semantic meaning of types depends on the syntactic meaning of types (due to practicality considerations, *i.e.*, the decidability of type checking), we first discuss the syntactic meaning of object types in OO PLs, then we discuss their semantic meaning. When discussing types, usually the context is enough to infer which sense of the two is meant. We first discuss the meaning of object types in nominal OOP first, then in structural OOP.

⁸For example, using types, it is not allowed to add `integers` to `booleans`, or to `strings`.

⁹In *pure* OOP, it is also the case that every value is an object. In **NOOP**, we model *pure* nominal OO languages, or “the pure OO subset” of “impure” ones.

The syntactic meaning of ‘object type’ is that an *object type* is a syntactic expression of an object interface. Object types, in their syntactic meaning, are thus sometimes more accurately called *object type expressions*. An object type expression is thus a concrete expression of an object interface.

The semantic meaning of ‘object type’ (which is also called the denotational meaning of an object type) depends on the syntactic meaning. Semantically, an object type is the set of objects (in a domain of objects) denoted by a given object type expression, *i.e.*, the set of objects that have the particular object interface expressed by this object type expression.

In nominal OOP, *class signatures* express object interfaces. In structural OOP, *object type expressions* (which, given the structural view of objects in structural OOP, are the same as *record type expressions*) express structural object interfaces.

Class signatures are nominal constructs (*i.e.*, they have class names as part of their meaning), because they express object interfaces of nominal OOP, which have class names as part of their identity. Names of class signatures are usually also called *type names*. Class signatures can be automatically derived from the source code of classes. Type names, which are the same as interface names and class names, are thus also associated with the same contracts associated with the class names. Because class signatures (*i.e.*, as “nominal object type expressions”) are nominal notions, the set of objects denoted by a class signature is called a *class type* or, synonymously, a *nominal object type*.

In structural OOP, on the other hand, class names are irrelevant to objects and are not included in their object interfaces. Thus, object type expressions in structural OOP are the same as record type expressions. Hence, semantically, *structural object*

types are the same as record types (denotations of record type expressions).¹⁰

Given the likeness between object type expressions and how we concretely expressed object interfaces in earlier code examples, we elide presenting examples for structural object type expressions, referring the reader to Figures C.5 and C.6 instead.

Exact Object Types Syntactically, as a type expression, an *exact object type (expression)* is a concrete expression of an exact object interface (See Section 2.1.3.1). An exact object type is a precise expression of the exact interface of the object. Semantically, as a set of objects, an *exact object type* is the set of objects denoted by an exact object type expression.

2.1.5 Inheritance, Subclassing and Nominal Subtyping

Inheritance is a defining feature of OOP. It is a syntactic notion, defined as a binary relation between classes in an OO program, where a class is said to inherit (or, extend) another class. Inheritance is also called *subclassing*, where the inheriting class is called the *subclass* while the inherited class is called the *superclass*. The main practical motivation behind having inheritance, in OOP, is to simultaneously support software extensibility (*i.e.*, the addition of new members to objects) while supporting software reuse (*i.e.*, the reuse or overriding of existing object members).

In relation to type inheritance, which is our main interest in this thesis¹¹, inher-

¹⁰Record types and record type expressions are familiar notions to functional programmers and to PL researchers. This familiarity caused the confusion of objects with records and is one reason why earlier models of OOP were structural models rather than nominal ones.

¹¹Usually inheritance includes code sharing, where methods code is inherited from a superclass to a subclass, but we ignore this aspect of inheritance in this thesis, because, in **NOOP**, our model of OOP, we more liberally model OO languages that allow each object of a class to have its own implementation code for methods. All method implementations of objects of the same class are required to have the same method signature, however, so as to not affect the common outside view (the object interface expressed in a class signature) of these objects, as well as to stick to the behavioral contract associated with signature names.

itance is a binary relation *between class signatures* of classes. Similar to classes, a class signature is thus said to inherit from another class signature.

As a relation between class signatures, *inheritance* means the class signature of the subclass shares with the class signature of the superclass all field and method signatures of the members in the class signature of the superclass. The shape supported by the subclass signature is thus always a supershape of the shape supported by the superclass signature (The subclass signature may add some new members of its own, together with their member signatures). The subclass signature may also inherit other member signatures from other superclass signatures¹². An object that has some class signature S can always respond to any field access or method call that any of the objects that have any of the supersignatures of S as their class signature can respond to. Under the inheritance relation, class signatures corresponding to classes in an OO program form an inheritance hierarchy.

For a subclass signature (the class signature corresponding to a subclass) and a superclass signature (the class signature corresponding to a superclass), signatures of corresponding members (*i.e.*, members with the same name) in the two signature are required to *match*, where the exact criteria for matching of member signatures may differ from one OO language to another. Typically, however, matching means requiring equality of member signatures (*i.e.*, for the same member name, the member signature in the subclass signature is required to be exactly the same signature of the member with the same name in the superclass signature).

In addition to nominality, and nominal typing, the third significant difference between mainstream OOP and structural OOP lies in how they differently view inheritance.

¹²Note that this definition allows for multiple-inheritance (of class signatures).

In nominal OO languages, inheritance is *explicitly* specified between classes (and thus also between class signatures and object interfaces of these classes), and it is specified *using class names*. Inheritance (subclassing), in nominal OOP, thus is a nominal relation. Because of it being explicitly specified, inheritance in nominal OOP is always an intended relation, and is never accidental. A class explicitly inheriting from another class is declaring, explicitly, that not only does its instances support the shape and the signature of the superclass, with matching member signatures, but that they further *stick to the full behavioral contract* associated with the name of the superclass. Further, because the inheritance relation and the subtyping relation between nominal object types (*i.e.*, the inclusion relation between class types, also informally called the “is-A” relation) are identified in statically-typed nominal OOP (which we prove in Chapter 6), a nominal OO language with nominal subclassing is called a *nominally-subtyped* OO language.

In structural OO languages, though, inheritance is inaccurately interpreted as an (implicitly-specified) relation between structural object interfaces (or, equivalently, as a relation between concrete expressions of structural object interfaces, *i.e.*, record type expressions). An object interface (or, its corresponding record type expression) in structural OOP implicitly inherits from another object interface if and only if each member interface in the subinterface matches with a member interface in the superinterface. As such, inheritance is misinterpreted in structural OO languages as being syntactic structural subtyping between record type expressions. Because it is not specified explicitly, “inheritance” in a structural OO language can be accidental and unintended. Because of the lack of a connection (*e.g.*, via class names) to behavioral contracts, inheritance in structural OOP, thus, may not be a reflection of a true “is-A”

relation¹³.

Even though nominally-subtyped OO languages can sometimes be less flexible than structurally-subtyped OO ones (in the cases where accidental inheritance is useful [47]), having more control over the inheritance relation, so that it reflects true “is-A” relations, is why mainstream OO developers embrace nominally-subtyped OO languages more than structurally-subtyped ones. Biological taxonomy, which has the inheritance relations humans are most familiar with, has nominal inheritance relations (explicitly specified via class names, based on organisms sharing behavioral contracts) rather than derived, implicitly-specified structural inheritance relations.

Figure C.7, in Appendix C, presents examples for record type expressions in the syntactic structural subtyping (“structural inheritance”) relation.

2.1.6 Signatures, Subsigning and Substitutability

In Chapter 5 of this thesis, we formally present the notion of class signatures in detail. We quickly introduce them though in this section.

Class signatures are the formal construct we use in **NOOP** to model the features of class types in nominal OOP. Class signatures are syntactic entities that carry information that is used for typing purposes. A signature closure “closes” a class signature by providing fixed, known class signature bindings for all class names referenced in a class signature.

Similar to a class signature, a signature closure (as a “closed class signature”) thus has a name, contains member signatures (*i.e.*, field and method signatures), and it includes names of its supersignatures. As a nominal object type expression, a

¹³Causing the familiar problem of “spurious subtyping”. See [56].

signature closure, not just a class signature, is the full formal expression of the notion of object interfaces in nominal OO languages (see Section 2.1.3.1).

In the context of signatures, the inheritance (*i.e.*, subclassing) relation between signature closures is called *subsigning*. A signature closure is a subsignature of (sub-signs) another signature closure if and only if corresponding member signatures match *and* the first class signature explicitly declares the second class signature as one its supersignatures (using the name of the second class signature). For the subsigning relation to hold, thus, not only is the structure of objects of a class important in deciding the relation but also the *behavior* of the objects (as expressed in the contract associated with the signature/class name) is equally important.

The inclusion of behavioral contracts in deciding the subsigning relation makes the relation a more accurate reflection of a true ‘is-A’ (substitutability) relationship than the structural syntactic subtyping relation. This makes subtyping in nominal OOP more semantically accurate than structural subtyping. Semantic subtyping is commonly expressed as the ‘Liskov Substitution Principle’ (LSP) familiar to many OO developers. The LSP states that in a computer program, *S* is a subtype of *T*, if and only if objects of type *T* may be replaced with objects of type *S*, without altering any of the main *behavioral* properties of that program. Via including class names associated with behavioral contracts in deciding the subtyping relation, the subtyping relation in nominal OOP is semantically precise (incorporates more behavioral properties) than structural subtyping found in structural OO languages.

Figures C.8 and C.9, in Appendix C, present examples of class signatures, and Figure C.11 presents pairs of signature closures in the subsigning relation.

Class signatures and signature closures play an important role in defining and building our model of mainstream OOP. The embedding of signature closures in the

objects of our model, **NOOP**, makes the objects of **NOOP** model nominal objects of mainstream OOP more precisely. The “baking” of signatures into objects makes **NOOP** be a model of *nominal* OOP.¹⁴

2.2 Nominal OOP versus Structural OOP

Building on the account of OOP presented in Section 2.1, to motivate seeing why a model of OOP that does not include nominality is not a precise model of OOP, this section focuses on discussing the main technical differences between nominal features of nominal OOP and structural features of structural OOP that were discussed sporadically above.

2.2.1 The Meaning of Being an Object

The main semantic difference between nominal OOP and structural OOP is that, given that the type information of an object cannot be inferred based on the structure of the object alone, to support run-time type-dependent operations¹⁵, objects in a nominally-typed OO language must carry their type information at run-time. An object, in a nominal OO language, thus, is not only a record, but is a record paired with a tag that associates the object with a class that expresses the type of the object..

Such a difference in how objects are viewed is the first, and most fundamental difference between nominal OOP and structural OOP. Overlooking this difference causes denotational models of structural OO languages (*e.g.*, Cardelli’s **SOOP** [14])

¹⁴In **NOOP**, full signatures, formalized as *signature closures*, have to be embedded in objects, rather than just signature names, because the domain of objects in **NOOP**, as a model, has to include *all* possible objects in *all* possible OO programs.

¹⁵Such as checking types of field assignments, checking method argument types against method parameter types, **instanceof** checks, and “type casting”.

and denotational models of nominal OO languages (*e.g.*, **NOOP**) to be fundamentally different.

2.2.2 The Semantic Value of Nominal Typing

As explained earlier, in nominal OOP, the class of an object has a name. This name is also used as the name of the exact object type of the object. In structural OOP, this is not the case. The type of an object, in structural OOP, is only a specification of the members (fields and methods) of the object and of the (structural) types of these members. In a structural OO language, using structural object types (record types) disallows accessing a field or calling a method of an object only when that object does not have that field or method as a member¹⁶. In a nominal OO languages, the inclusion of a class name in class types *additionally* disallows accessing the members of an object whose class does not *explicitly* declare that its objects stick to the behavioral contract associated with the class name. This stricter requirement by class types in nominal OO languages is the second fundamental difference between nominally OOP and structural OOP.

Nominal typing is thus useful when the structure of objects (mirrored in the structure of their record types) is *not* enough to express the behavioral constraints satisfied by the objects of the type. For reasons that have to do with the decidability of type checking, type expressions in type systems of practical programming languages can-

¹⁶The notion of structural typing was introduced in the simply typed λ -calculus [20], and subsequently extended to enrichments of the λ -calculus [52, 56]. Hindley-Milner type inference, which is widely used in functional languages to infer the types of programs, critically depends on the property of structural typing. Hindley-Milner type inference depends on the fact that every program value has a unique monotype (program values cannot belong to multiple types). Type-inference becomes muddled in the presence of subtyping, which is why Hindley-Milner type inference does not accommodate explicit subtyping, but only accommodates structural/implicit subtyping (in purely-OO nominal OO languages, every value is an object, making the declared types of functions/methods manifest.)

not express all logical properties of objects in programs that a software developer may wish to express. These properties always hold as true for the objects. These “always true, but inexpressible” properties are considered as extra terms in the behavioral contracts of the objects in the software.¹⁷ Class names, nominality, and nominal typing, provide OO software developers with a simple yet formal means for summarizing, in the class name, the behavioral contract of objects in OOP software.¹⁸ As we noted earlier, nominal typing allows circular object types to be expressed more lightly (See Section 2.1.3.1).

2.2.3 Nominal Typing and Subtyping versus Structural Typing and Subtyping

As discussed in the last section, nominal typing dictates that the name of an object type, in addition to its structure, is taken into consideration when making decisions regarding whether the type is equivalent to another object type, and whether it is a subtype of another object type.

A type-checker, for a statically-typed programming language, must verify that the type of any expression is consistent with the type expected by the context in which that expression appears. For instance, in a method invocation of the form `m(e)`, the inferred type of the expression `e` must be consistent with the declared or inferred type of the formal parameter of method `m`. This notion of consistency, called type-compatibility, is specific to each programming language. Type compatibility involves

¹⁷Documentation and comments accompanying software, *e.g.*, in the style of JAVADOC comments, are usually viewed as an informal specification of the behavioral contracts of the software and its components.

¹⁸A behavioral contract can, for example, express that all members of a subtype of `interface Comparable` (in JAVA) are totally-ordered by the `compareTo()` method they inherit from `interface Comparable`.

checking two types for equivalence, and checking them for subtyping (*i.e.*, one being a subtype of the other).

Based on the discussion in Section 2.2.2, the main difference between the type systems of nominal OO languages (with nominal typing and nominal subtyping) and the type systems of structural OO languages (with structural typing and structural subtyping) lies in how these type systems *differently* answer two main questions, about type equivalence and type subtyping.

The first main question the two different styles of OOP answer differently is: What if two objects have the same members (fields and methods) with the same member signatures, will the two objects then have the same type? Structural typing, in a structural OO type system, says: ‘Yes’, unconditionally. Nominal typing, in a nominal OO type system, says: ‘Only if the structurally-equivalent types also have the same type name’. Thus, nominally-equivalent types of two nominal objects are always structurally-equivalent types, but not necessarily vice versa.

The second main question the two kinds of OO type systems answer differently is related to the first question, but pertains to subtyping. The question is: What if for two objects where the shape of the first object is a supershape of the second object and the first object has the same member signatures as those of the corresponding members in the other object, will the type of the first object be a subtype of the type of the second object? Again, structural subtyping says: ‘Yes’, unconditionally. Nominal subtyping says: ‘Only if the type of the first object explicitly states (using the names of the two types) that the structurally-compatible type of the second object is one of its supertypes (*i.e.*, those of the first object type)’. Thus, nominally-subtyped types of two nominal objects are always structurally-subtyped types, but not necessarily vice versa.

Based on how they differently answer the second question, it is thus said that in structural OOP subtyping happens “by chance” (*i.e.*, is sometimes accidental), while in nominal OOP subtyping takes place “by choice” (*i.e.*, is always intended).

How nominal type systems (of nominal OOP) and structural type systems (of structural OOP) differ in answering these two fundamental questions is behind the fundamental differences between the two kinds of OO type systems. These differences makes the translation of research results reached for one kind of type systems *not* immediately applicable to the other.

2.3 Statically-typed OO Languages versus Dynamically-typed OO Languages

Statically-typed OO languages, where type-checking is done at program compilation time, define the formal notion of object types (See Section 2.1.4) based on the notion of object interfaces (See Section 2.1.3.1). Static type-checking is done based on object types. Examples of mainstream statically-typed OO languages are JAVA [32] and C# [2]. Our focus in this thesis is on modeling statically-typed nominal OO languages.

On the other hand, in dynamically-typed OO languages (where “type-checking” is only done during program run-time) there is no notion of an object type. An example of a dynamically-typed OO language is SMALLTALK [1]. Given that dynamically-typed languages do need to ascertain some form of type-safety (*i.e.*, ascertain the proper use of objects), such languages usually define notions that are similar to types to use and check for consistency at run-time. For example, SMALLTALK [1] requires objects at run-time to conform to ‘protocols’. According to how they are used in SMALLTALK, protocols are actually closer to structural types than they are

to nominal types. In particular, despite them having names, the names of protocols in SMALLTALK, and the nominal subclassing relation between SMALLTALK classes, are *not* relevant at run-time in deciding the conformance (“subtyping”) relation. SMALLTALK is a nominal OO language, but it is dynamically-typed. Because it has no clear notion of types in the first place, SMALLTALK is not a nominally-typed OO language nor is it is not a nominally-subtyped OO language. In this thesis, we are not concerned about modeling non-nominal features of dynamically-typed nominal OO languages.

Chapter 3

Motivation and Background

I see further by standing on the shoulders of giants.

~Isaac Newton

Although a precise mathematical model of nominal OOP has not been developed before, much research on the semantics of programming languages, object-oriented and otherwise, has been done by the programming languages (PL) research community in the last few decades. This thesis depends on this research as a context in which the thesis fits, and as a source of motivation for the particular research presented therein. In this chapter, we present the motivations and the context for developing **NOOP** as a rigorous model of nominal OOP.

3.1 NOOP Motivations

Our development of **NOOP** has two main motivations. These are:

1. Refocusing PL research on models and type systems relevant to mainstream software development.
2. Encouraging the development of more sophisticated type systems for mainstream OOP languages.

We discuss each motivation in the following subsections.

3.1.1 Shifting PL Research View of OOP to be Nominal

As we discussed in Chapter 2, earlier models of OOP (*e.g.*, the model developed by Cardelli in 1988 [14], first presented in 1984 [13]) did not include the name of the class of an object as part of the meaning of the object. Objects in earlier models of OOP are not nominal objects, but are mere records. Cardelli [14], in fact, does not make use of explicit class definitions, but only has a notion of structural object type expressions (*i.e.*, record type expressions). In Cardelli's model of OOP, only the structure of an object (specifying the interface of its members) defines the interface of the object to the outside world. Objects in Cardelli's model only have structural object interfaces. In particular, the contract associated with a class name in OOP is not associated with an object in Cardelli's model. Since Cardelli's framework does not include explicit classes, he does not have a simple way to annotate programs with class contracts. In OO design, the class hierarchy reflects program behavior and method contracts.

All subsequent models of OOP have been built on top of Cardelli's structural model. Thus, nominality has been ignored in subsequent analysis and reasoning about the properties of OOP. Henceforth, we will use the acronym '**SOOP**' to refer to the structural model of OOP [14] Cardelli developed. It should be noted that Cardelli's decision to construct a structural model was well-motivated at the time. OOP was in its infancy at that time. When Cardelli did his seminal work, the dominant OO language (SMALLTALK) was dynamically-typed. The potential importance of nominality in OO design and OO type systems was not yet appreciated.

In the ensuing twenty-five years, PL researchers have generally relied on **SOOP** and its descendants for an intuitive understanding of what OO programs mean. The reliance of subsequent PL research on structural models of OOP has adversely affected

the relevance and impact of that research. To see why, one should consider that three essential features of mainstream OOP make crucial use of class names: namely, (1) circular class dependencies, (2) inheritance, and (3) generics. Even though these three features have counterparts in structural OOP, conclusions about the structural counterparts are not necessarily applicable to nominal OO languages.

We find some indirect support to our opinion, regarding the mismatch between OO PL research on semantics of OOP and mainstream OO software development, among researchers in the PL research community. According to Benjamin Pierce, a leading researcher on type systems ([56, p. 254]):

... [given the practical advantages of nominal typing], it is no surprise to find that nominal type systems are the norm in mainstream programming languages. The research literature on programming languages, on the other hand, is almost completely concerned with structural type systems.

As we discuss in more detail in Section 3.1.2, this non-alignment between how OOP is viewed by OOP researchers, on one hand, and how OOP is viewed by mainstream OO software developers, on the other, could be inhibiting new developments in OO language design, particularly regarding the development of static type systems. Given the ubiquity of computer software, and the non-diminishing popularity of OOP among mainstream software developers, we find this schism between PL researchers and mainstream software developers to be no longer tenable.

To bridge this gap, we developed **NOOP** as a model of nominal object-oriented programming that carefully pays full attention to the nominal structure and nominal aspects of mainstream OOP.

By developing **NOOP**, as a precise mathematical model of *mainstream* OO software that takes nominal features of mainstream OOP in full consideration, we defi-

nately hope to refocus PL research on models and type systems relevant to mainstream programming languages and software development.

3.1.2 Enabling Progress in Type Systems of Mainstream OO Languages

In considering putative language designs similar to JAVA and SCALA, the lack of a clear model for OOP has proven to be an insurmountable obstacle. These putative designs could not be compared in any credible intuitively accessible way.

JAVA wildcards (also called, wildcard types) is a feature of JAVA generics that was added to soften the mismatch between OO subtyping and generics [70].¹ Due to the lack of a clear model for OOP, the mathematical analysis of JAVA wildcards has proven to be unwieldy, thus inhibiting the growth of JAVA, and decreasing the interest of software developers in the language.

As a demonstration of how the inclusion of wildcards in the type system of JAVA has inhibited the growth of JAVA, one of the reasons the addition of JAVA Closures [41] to the JAVA programming language was delayed, was to give time for developing simpler proposals of JAVA Closures. As Joshua Bloch put it, while commenting on proposals for Closures, JAVA has “used up its complexity budget on generics, and in particular, on wildcards.” [41, part 5].

JAVA wildcards also affect other parts of the JAVA type system. The current JAVA type system in fact rests on shaky formal foundations; the JAVA local type inference algorithm is broken and there is no obvious “quick fix” [67]. Newer OO languages like SCALA [54] and X10 [60] have retreated from wildcard types (or, ‘usage-site variance annotations’) in the absence of good models of the current JAVA type system.

¹Having a good understanding JAVA wildcards is important for confidently using some core classes of JAVA, such as the `Hashtable` class.

Having a precise model of nominal OOP, such as **NOOP**, can help prove the type safety of the JAVA type system when it gets extended with other putative features, and thus enable the future growth of the language and its even-wider adoption. Having a precise model of OOP can also help in developing better designs for new OO languages.

3.2 Related Research

Prior research on the semantics of OOP has been strongly influenced by research on the semantics of functional programming. In this section, we identify the primary PL research sources on which this thesis rests, then we review some of the most important research results on the semantics of functional programming (FP) that are relevant to OOP. We conclude this section by reviewing prior research on the semantics of OOP.

3.2.1 Main PL Research Sources

The research in this thesis rests primarily on three sources:

- A monograph on domain theory titled “Domain Theory: An Introduction” by Robert Cartwright and Rebecca Parsons [19];
- Luca Cardelli’s seminal paper on the semantics of OOP titled “The Semantics of Multiple Inheritance” [14]; and
- Igarashi, Pierce, and Wadler’s paper on the operational semantics and type systems of mainstream OO languages titled “Featherweight Java: A Minimal Core Calculus for Java and GJ” [36].

Domain theory is not a readily accessible branch of theoretical computer science. The first source above presents one of the simplest and most accessible introductions to

the field. Domain theory is a branch of computer science that builds on set theory, order theory, and topology. It was developed by Dana Scott, Gordon Plotkin, and others, to provide a framework for defining the mathematical meaning (the denotational semantics) of computer programming languages. The first source [19] is an unpublished revision and simplification of Dana Scott’s tutorial monograph [64] on domain theory. Domain theory can be considered as “the mathematics of computation”. For more details on domain theory see Appendix A. In Section 3.2.2.1 we present a very brief account of the historical development of domain theory.

In the second source, Luca Cardelli presents a domain-theoretic model of structural OOP (which we call **SOOP**) that has served as the basis for nearly all subsequent research on the semantics of OOP. The model of OOP we present in this thesis can be viewed as an updated alternative to **SOOP**. Our model faithfully represents the nominal character of mainstream OO languages, which arose after Cardelli’s work.

The third source [36] presents the most recent major work on the semantics of mainstream OOP. In this paper, the authors present a small language, called Featherweight Java (FJ), that has a number of core features of a mainstream OO PL (*i.e.*, JAVA). Using *operational* semantics, the authors prove the type safety of FJ. The authors, then, extend FJ to FGJ (Featherweight Generic Java), and revise its evaluation rules twice: they revise the rules first to support first-class generics, and they revise them next to support “erased generics”. Again using operational semantics, the authors prove the type safety of the extended language in each case. Igarashi, et al [36], provide strong evidence that the core type system in JAVA with generics (without wildcards) is safe.

Each of these three main sources of PL research rests broadly on core research in functional programming, which we briefly survey in the next section.

Another major source on the semantics of OOP related to this thesis is Kim Bruce’s textbook entitled ‘Foundations of Object-Oriented Languages: Types and Semantics’ [12]. Like **SOOP**, Bruce’s book presumes that statically-typed OO programming languages should have a structural semantics rather than a nominal one. Of course, mainstream OOP has followed a course diametrically opposed to Bruce’s vision. Since this thesis presents a firm theoretical foundation for mainstream, nominal OOP, it can be viewed as the antithesis of Bruce’s work.

3.2.2 PL Research on Functional Programming

Interest in the mathematical meaning of computer software—and, thus, in the mathematical modeling of programming languages—has started since the advent of high-level functional programming languages. Since the production of the pioneering work by McCarthy [48, 49], Strachey [69], and Landin [42] in the 1960s, computer scientists have compiled a deep body of research on program semantics and proof systems rooted in mathematical logic, most notably the λ -calculus, the “mother of all functional languages”.

3.2.2.1 Domain Theory and Denotational Semantics

Since the semantics of programming languages is closely related to the semantics of logical formulas in mathematics, it is not surprising that logician Dana Scott made the seminal breakthrough in program semantics by defining the first algebraic models for the untyped lambda-calculus [61, 62]. Scott’s construction showed how to view function spaces as computational domains, resolving the mismatch in set theory between the cardinalities of $D \rightarrow D$ and D . Scott imposed an ‘approximation’ partial ordering on computational domains and restricted the set-theoretic function space

construction to *continuous* functions according to the approximation orderings on the input and output domains. From Scott’s perspective, infinite computable objects like functions mapping \mathbb{N} into \mathbb{N} and infinite trees over \mathbb{N} are the limits of progressively more defined finite approximations.

Dana Scott [63, 69, 64, 65, 33] and Gordon Plotkin [58, 68, 59, 46, 4] subsequently generalized the constructions used in these models to accommodate arbitrary computational domains. Plotkin played the key role in eliminating the artificial maximum element (\top) present in Scott’s lattice-based models. The work of Scott and Plotkin helped bring into existence the fields of domain theory and denotational semantics.

Domain theory is so-called because it studies the mathematical space (called a domain) that software data, as abstract entities, occupy. In denotational semantics, a program phrase (expressed in abstract syntax [48]) is assigned a denotation as its logical meaning. This denotation is an element of a semantic domain, as formulated by Scott and Plotkin. Abstract syntax, which was invented and named by McCarthy, might more accurately be called ‘algebraic syntax’ since it expresses all program text in terms a set of free algebraic generators called constructors. A denotational model simply interprets these algebraic generators as semantic functions on semantic domains, just as a model of a first order logical language interprets each function symbol in the logical language as a function mapping the input domains (in the model) for the symbol to the output domain (in the model).

An important property of denotational semantics is that it is compositional. Using denotational semantics, the meaning of any term $C(t_1, \dots, t_n)$ in the abstract syntax is simply $\mathcal{M}[C](\mathcal{M}[t_1], \dots, \mathcal{M}[t_n])$ where \mathcal{M} is the meaning function. The value $\mathcal{M}[C]$ is extracted from a table (the meaning of abstract-syntax-tree primitives); the other invocations of \mathcal{M} are recursive.

Using these two fields, Dana Scott and Gordon Plotkin constructed multiple models of the λ -calculus (Scott’s original motivation for working on the mathematical meaning of software though was *not* to find models for λ -calculus. See Stoy’s book [69] for details). Since then, all semantic models of functional programming languages have been rooted in domain theory, and are based on models of the λ -calculus.

Research on domain theory and denotational semantics continued in the 1980’s. PL researchers later built on top of Scott and Plotkin’s work, helping establish and enrich the two fields (See [44, 15, 46, 16, 18, 17, 37]). Jung [28] presents a mathematically-oriented account of the historical development of domain theory and denotational semantics.

Research compiled in the last fifty years on program semantics (and on proof systems, such as LCF [57, 31]) has supported the development of an array of sophisticated statically-typed functional languages, including ML, MIRANDA, HASKELL, OCAML, and F#, and the development of corresponding semantic models rooted in domain theory.

As part of PL research in “the functional world”, a number of proof-assistants (sometimes called “theorem provers”) have also been developed. With the help of the programmer, proof-assistants can ascertain the correctness of (some) functional software (COQ [7], for example, was used to prove security properties of the Java Card system (<http://coq.inria.fr>). Arguably equally-important, proof assistants can also be used to prove pure mathematical theorems). Research supporting recent proof-assistants (*e.g.*, COQ [7], and ISABELLE [55]) was based upon and inspired by research supporting earlier similar tools (*e.g.*, LCF [57, 31], and NUPRL [21]).

Proof assistants such as COQ and ISABELLE depend on the mathematical meaning (the denotations) given to constructs of functional programs to reason about the

correctness of these programs. Having a precise domain-theoretic model of functional programming was, among other factors, an important factor in allowing the development of these tools ([7]), and in allowing better (semi-)automatic reasoning about functional software.

3.2.3 PL Research on Object-Oriented Programming

According to Cardelli [14], “the method of structuring data in OO programming languages can be said to derive from biology and taxonomy”. In FP, the method of structuring data is mathematical. The data domains of FP are inductively defined, using familiar mathematical constructions from set theory (disjoint unions, tuples/cross-products, and functions).

Cardelli [14] states:

Data, in OOP, is organized in a hierarchy of classes and subclasses, and data at any level of the hierarchy *inherits* all the attributes of data higher up in the hierarchy. The top level of this hierarchy is usually called the class of all *objects*; every datum is an object and every datum inherits the basic properties of objects, *e.g.*, the ability to tell whether two objects are the same or not. Functions and procedures are considered as local actions of objects, as opposed to global operations acting over objects.

Explaining the widespread use of OOP among mainstream software developers, and emphasizing the superiority of OOP over other styles of software development, Bruce [12] states that

“there is real substance behind the reasons for the increasing use of object-oriented languages. There seem to be clear advantages for the object-

oriented style in organizing and reusing software components. For example, subtyping and inheritance seem to make it much easier to adapt and reuse existing software components.”

The intuitiveness of OOP data structuring, and the scalability advantages of using OO languages, have helped establish OOP as a mainstream approach/style of computer programming. With roughly about 60% of software developers using OO languages in developing their programs, OOP is currently the dominant style of programming in industrial software development².

3.2.3.1 Early OO Research

Since its inception, OOP has attracted the attention of PL researchers, although not at the same level as functional programming did.

Even though OOP was invented in 1967 (in SIMULA), in the 1980s OOP was still in its infancy. OOP did not impact industrial practice until personal computing with graphical user interfaces become an important mode of computation in the mid-80's. At this point, OOP also attracted the attention of PL researchers.

During this period, most OO programming was conducted in SMALLTALK, a dynamically-typed language. But the dominant mainstream language, C, was statically-typed. Computing researchers and developers anticipated that statically-typed OO languages would emerge, supplanting SMALLTALK, but the precise form of those languages and their type systems was unknown. Initially, PL researchers anticipated building static type systems for OO languages as extensions of the sophisticated type

²See, for example, statistics of the TIOBE index (at <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>), and at <http://www.langpop.com>.

systems [51] they had recently developed and refined for functional programming languages.

Benjamin Pierce [56] presents an account of the development of research on the semantics of OOP, where he states that:

“the first interpretation of objects in a typed λ -calculus was based on recursively-defined records. It was introduced by Cardelli [14] and studied in many variations by Kamin and Reddy [38, 39], Cook and Palsberg [22] and Mitchell [53]. In its typed form, this model was used quite effectively for the denotational semantics of untyped OO languages. In its typed form, it could be used to encode individual OO examples, but it caused difficulties with uniform interpretations of typed OO languages. The most successful effort in this direction was carried out by Cook et al. [23, 24].”

Our assessment of the effectiveness of these models for untyped OO languages (like SMALLTALK) is less sanguine than Pierce’s evaluation, because these models do not accommodate class name information in objects, precluding the definition of operations like SMALLTALK `isMemberOf`, akin to JAVA `instanceof`, and type casting operations. The PL research community did not think such operations were important, but they are critical for confirming inheritance relationships and debugging code that relies on inheritance.³).

Cook’s paper [24], emphatically-titled ‘Inheritance is not subtyping’, is the publication that led PL researchers to the mistaken folklore that inheritance and OO subtyping should not be identified. Cook’s research was predicated on modeling objects as records devoid of class names (following Cardelli’s footsteps [14]), and on

³In fact, it is impossible to write an informative output operation, akin to the JAVA `toString()` method, in the absence of such an operation like `instanceof`, because the program cannot determine the class of an object!

Cook’s modeling of recursive types (*e.g.*, `SelfType`), which Cardelli had intentionally omitted from **SOOP**⁴.

During this period of early OO research, research on functional programming was “the measuring rod” that set the standards for what is right and what is wrong in OOP research. Among PL researchers, research on FP was the ideal to be sought, and on whose footsteps research on OOP should follow. This ideal included the implicit assumption of structural typing, which is a reasonable assumption in functional programming. In short, no one in the PL research community envisioned the critical role that nominality will play in OOP, both in language design and software development.

In the 1990s, research on OOP was focused on even more encoding of OO features using functional ones. In 1994, Bruce presented a paper [11] on the semantics of a functional object-oriented language. PL researchers wanted OOP to be formalized as a dialect of the typed lambda calculus using type constructions familiar to functional programmers. The desire to view OOP as a subset or subdiscipline of FP eliminated nominal information from objects, in conflict with how OO software developers conceptualize OO data.

After writing the above, Pierce [56] further continues that “Meanwhile, frustrated by the difficulties of encoding objects in λ -calculi, Abadi and Cardelli introduced a calculus of primitive objects [3].” Abadi and Cardelli [3] thus did not map objects to corresponding “lower-level” functional constructs. Objects were considered a primitive entity on their own. Abadi and Cardelli gave an operational semantics to their object calculus. The subtyping rules in their work, however, still respected structural subtyping, precluding the inclusion of class names in object denotations. Objects in Abadi and Cardelli’s work [3] did not carry the class name of objects, and inheritance

⁴Cardelli states in [14, page 11] that ‘recursive types are not treated in the formal semantics’

relation was not explicitly specified between object types.

3.2.3.2 Recent OO Research

To provide a comprehensive account of structural OOP, Bruce wrote his book on ‘Foundations of Object-Oriented Languages: Types and Semantics’ [12]. Bruce based his view of OO subtyping on that of Cardelli [12, p.72].

In this book, Bruce further separates inheritance and subtyping by introducing the notion of ‘matching’ as a generalization of structural subtyping. Bruce’s notion of matching did not gain credence among software developers and language designers because it critically relies on the absence of nominal information (class names) in object denotations. Given a structural view of objects, matching makes sense but software developers conceptualize objects as carrying class names and conforming to invariant behavioral contracts associated with their class names.

Another significant foundational work that also adopted the structural view of OOP is that of Anthony Simons, who presented, in the Journal of Object Technology, a series of twenty articles (in 2002-2005) on ‘The Theory of Classification’ [66]. Based on the structural view of objects, Simons also made a clear distinction between subclassing (inheritance) and subtyping in his articles.

The most significant work on the operational semantics of a mainstream OO languages is Igarashi, Pierce and Wadler’s work on FJ/FGJ (as a “featherweight” version of JAVA), presented in [36], [56], and other earlier publications. The development of FJ/FGJ was motivated by the need to prove the type soundness of the “erased generics” semantics used in Generic Java. We briefly discussed the main research results of FJ/FGJ in Section 3.2.1.

Disenchantment with structural OOP Despite this situation where many OO PL researchers have adopted a non-nominal view of objects, the proponents of functional programming appear to have become disenchanted with functionalized, structural formulations of inheritance (OOP). Supporting our belief in this regard is that, in 2002, David MacQueen argued [45] that OO features should *not* be added to STANDARD ML [51], despite them being already included in OCAML [43] (OCAML is another widely-used variant of ML). One of the reasons MacQueen [45] did not mention explicitly for not favoring the mixing ML with OO features is that, if they were added to STANDARD ML (which, like all functional programming languages, is a structurally-typed functional programming language), OO typing features would have to be those of structural OOP, not ones of nominal OOP (so as to make the supposed new OO features of Standard ML mesh well with the already-structural type system of STANDARD ML). But then, MacQueen argued, these (structural) OOP features will not match with the (nominal) typing concepts mainstream OO software developers are familiar with. For this reason, and for other reasons that MacQueen details [45], if ML became object-oriented its OO features are likely to be unused, or will be cumbersome and unnatural to use.

Chapter 4

COOP: A Simple Structural Model of OOP

In this and the following two chapters we present the details of defining and constructing **NOOP**, our model of nominal OOP. First, this chapter presents **COOP** as a simple structural model of OOP. Chapter 5 then presents class signatures and discusses how signatures capture the nominality features of nominal OOP. By enriching **COOP**, Chapter 6 then uses signatures to define *preNOOP*. *preNOOP* is an unfiltered model of nominal OOP because it includes invalid “objects” (whose record components do not match their signatures). In Chapter 6, thus, *preNOOP* is then filtered by a simple projection to produce **NOOP**, as our mathematical model of mainstream OOP.

The reasons for presenting a structural model of OOP (*i.e.*, **COOP**) first, before presenting **NOOP**, are threefold. First, earlier research on structural OOP needs to be put on a more rigorous footing. The literature on models of structural OOP glosses over important technical details like constructing a domain of records, having methods of multiple arity, and being purely OO (*i.e.*, not allowing functions and non-object values have first-class status), which we address. Second, the construction of **COOP** is similar to but simpler than the construction of **NOOP**. Understanding how **COOP** is constructed makes it easier to understand the construction of **NOOP**. Third, and most importantly, the rigorous definition of **COOP** alongside the definition of **NOOP** clarifies the distinction between structural OOP and nominal OOP.

As mathematical models, **COOP** and **NOOP** are collections of domains. In

denotational semantics, domains (partially ordered sets with certain properties as defined by Scott [65] and others [58, 59, 46, 37]) are used to model computational constructs and notions (as explained in more detail in Appendix A). Domains of **COOP** and **NOOP** correspond to the set of all possible object values, and field values, and method values (using JAVA terminology) of structural and nominal OO programs. Similarly, specific subdomains of **COOP** and **NOOP** domains correspond to specific structural and nominal types definable in statically-typed structural and nominal OO languages. **COOP** and **NOOP**, thus, give an abstract mathematical meaning to the most fundamental concepts of structural and nominal OOP.

The domains of **COOP**¹ are the solution of a reflexive domain equation. Appendix A presents a summary of the main definitions and theorems of domain theory presented in Dana Scott’s monograph [64], as updated by Cartwright and Parsons [19]. In Section 4.1 of this chapter, based on the domain theory foundations recounted in Appendix A, we first present a new domain constructor, the records domain constructor \multimap , that formulates records as finite functions.

In Section 4.3 we then show how the **COOP** domains are constructed as the solution of the **COOP** domain equation. The domains of **COOP** are constructed using standard domain theoretic construction methods that make use of standard domain constructors as well as the records domain constructor we described in Section 4.1.

The view of objects in **COOP** is a very simple one. An object in **COOP** is a record of functions mapping sequences of objects to objects. In other words, in **COOP** an object is ‘a finite collection of labeled methods’, where methods are functions from sequences of objects to objects. In **COOP**, we encode fields as zero-ary methods.

¹Similarly, **NOOP** domains are also the solution of a reflexive domain equation. We will, however, drop any further mention of **NOOP** in this chapter unless absolutely necessary.

Given that objects of **COOP**, like those of **SOOP**, miss nominality information, **COOP** is also a structural model of OOP. Given that it is a structural model of OOP, **COOP** closely resembles **SOOP** (*i.e.*, Cardelli’s model of OOP, presented in [13, 14]). The construction of **COOP** shows how to rigorously construct a model like Cardelli’s.

COOP, however, differs from **SOOP** in five respects:

1. Unlike **SOOP**, but similar to many mainstream OO languages, the **COOP** domain equation does not allow functions as first-class values (thus, **COOP** does not support function currying). Only objects are first-class values in **COOP**.
2. Unlike **SOOP**, **COOP** uses the records domain constructor, \multimap , to construct records (rather than the standard continuous functions domain constructor used in **SOOP**). The definition of \multimap is presented in Section 4.1.
3. Unlike **SOOP**, methods in **COOP** objects are multi-ary functions over objects.²
4. For simplicity, **COOP** objects have fields only modeled by (constant) 0-ary functions, not as a separate component in objects. Thus, names of fields and methods in **COOP** objects share the same namespace.
5. Since we do not use **COOP** (nor **NOOP**) to prove type safety results, **COOP** does not need to have a counterpart to the $\mathcal{W} = \{wrong\}$ domain that is used in **SOOP** to detect type errors.

When we present **NOOP**, in Chapter 6, we will show that **COOP**, and thus also **SOOP**, does not accurately capture the notion of inheritance as it has evolved in

²Since **SOOP** defines a domain for a simple functional language with objects based on ML, it is natural to force all functions to be unary (as in ML). In this context, a multi-ary function can be transparently curried.

statically-typed nominal OO languages (like JAVA [32], C# [2], SCALA [54], and X10 [60]).

4.1 Records Domain Constructor ($-\circ$)

As mentioned above, **COOP** models objects as records. In this section, we present the records domain constructor, $-\circ$, which constructs domains of records, and we discuss its mathematical properties. The definition of this constructor makes use of the standard definitions from basic domain theory. We present a summary of these in Appendix A.

4.1.1 Record Functions

A record (or **COOP** object) can be viewed as a finite mapping from a set of labels (member names) to methods. Thus, we model records using record functions, which are explicitly finite. A *record function* is a finite function paired with a tag representing the input domain of the function. The tag of a record function modeling a record thus represents the set of labels of the record. In agreement with our earlier definition of shapes (for objects) in Section 2.1.3.1, we similarly call the set of labels of a record the *shape* of the record. The tag of a record function thus tells the shape of the record.

Due to the finiteness of the shape of an object, and due to the flatness of the shape when it is formulated as an input domain to record functions, modeling objects of **COOP** as records motivates defining a new domain constructor that is similar to but different from conventional functional domain constructors. This domain constructor constructs record functions, which are explicitly finite.³

³In this thesis, we do not follow Cardelli's modeling of records (in [14]) as infinite functions.

4.1.2 Definition of \multimap

Let \mathcal{L} be the flat domain containing all record labels (member names) plus an extra improper bottom label, $\perp_{\mathcal{L}}$, that makes \mathcal{L} be a domain (all domains must have a bottom element). Let \mathcal{D} be an arbitrary domain, with approximation ordering $\sqsubseteq_{\mathcal{D}}$ and bottom element $\perp_{\mathcal{D}}$, that contains the values members of records are mapped to.

We use \subseteq to denote the subdomain relation (See [19, Definition 6.2]). If we use \mathcal{L}_f to range over arbitrary finite subdomains of \mathcal{L} (all domains \mathcal{L}_f contain $\perp_{\mathcal{L}}$), then we define the domain $\mathcal{R} = \mathcal{L} \multimap \mathcal{D}$ as the domain of record functions from \mathcal{L} to \mathcal{D} , where

$$|\mathcal{R}| = \{\perp_{\mathcal{R}}\} \cup \bigcup_{\mathcal{L}_f \in \mathcal{L}} R(\mathcal{L}_f, \mathcal{D}). \quad (4.1)$$

Sets $R(\mathcal{L}_f, \mathcal{D})$ are defined as

$$R(\mathcal{L}_f, \mathcal{D}) = \{tag(|\mathcal{L}_f| \setminus \{\perp_{\mathcal{L}}\})\} \times |\mathcal{L}_f \multimap \mathcal{D}| \quad (4.2)$$

where tag is a function that maps the shape corresponding to a domain \mathcal{L}_f to a unique tag in a countable set of tags (whose format we leave unspecified), and where $\mathcal{L}_f \multimap \mathcal{D}$ is the domain of strict continuous functions from \mathcal{L}_f into \mathcal{D} .⁴

Thus, a record $r = \{l_1 \mapsto d_1, \dots, l_k \mapsto d_k\}$ is modeled by a record function

Cardelli states that he is only interested in the function mapping a cofinite subset of labels to an error value, but this subset of the function space used in Cardelli's domain equation does not form a subdomain of the function defined in Cardelli's domain equation. So the solution to Cardelli's domain equation necessarily contains superfluous elements. In contrast, we use a record function construction (instead of a conventional function space constructor) whose elements are in one-to-one correspondence with Cardelli's set of interest but form a domain. So our model of records is equivalent to Cardelli's in spirit, but our record domain excludes the 'junk' elements that technically exist in Cardelli's domain but Cardelli asks the reader to ignore.

⁴Since the functions in $\mathcal{L}_f \multimap \mathcal{D}$ are strict, the undefined element $\perp_{\mathcal{L}}$ is always mapped to $\perp_{\mathcal{D}}$. Moreover \mathcal{L}_f is flat, so there is no ordering relationship between label names l_1 and l_2 in $|\mathcal{L}_f| \setminus \{\perp_{\mathcal{L}}\}$. Hence the functions in $\mathcal{L}_f \multimap \mathcal{D}$ precisely correspond to ordinary mathematical functions from $|\mathcal{L}_f| \setminus \{\perp_{\mathcal{L}}\}$ to \mathcal{D} .

$r = (\text{tag}(\{l_1, \dots, l_k\}), \{(\perp_{\mathcal{L}}, \perp_{\mathcal{D}}), (l_1, d_1), \dots, (l_k, d_k)\})$. It should be noted that \dashv does not disallow constructing the (unique) record function $(\text{tag}(\{\}), \{(\perp_{\mathcal{L}}, \perp_{\mathcal{D}})\})$ that models the empty record (one with an empty set of labels, for which $|\mathcal{L}_f| = \{\perp_{\mathcal{L}}\}$).

The approximation ordering, $\sqsubseteq_{\mathcal{R}}$, over elements of \mathcal{R} is defined as follows. $\perp_{\mathcal{R}}$ approximates all elements of \mathcal{R} . The bottom element $\perp_{\mathcal{R}}$ approximates all elements of the domain \mathcal{R} . Elements r and r' in \mathcal{R} with unequal tags are unrelated to one another. On the other hand, elements r and r' with the same tag are ordered by their embedded functions (which must be elements of the same function domain).

Hence, for two non-bottom record functions r, r' in \mathcal{R} that are defined over the same \mathcal{L}_f , where $|\mathcal{L}_f| = \{\perp_{\mathcal{L}}, l_1, \dots, l_k\}$, if

$$r = (\text{tag}(\{l_1, \dots, l_k\}), \{(\perp_{\mathcal{L}}, \perp_{\mathcal{D}}), (l_1, d_1), \dots, (l_k, d_k)\})$$

and

$$r' = (\text{tag}(\{l_1, \dots, l_k\}), \{(\perp_{\mathcal{L}}, \perp_{\mathcal{D}}), (l_1, d'_1), \dots, (l_k, d'_k)\})$$

where d_1, \dots, d_k and d'_1, \dots, d'_k are elements in \mathcal{D} , then we define

$$r \sqsubseteq_{\mathcal{R}} r' \Leftrightarrow \forall_{i \leq k} (d_i \sqsubseteq_{\mathcal{D}} d'_i) \tag{4.3}$$

Theorem 4.1. *Given a flat countable domain of labels \mathcal{L} and an arbitrary domain \mathcal{D} , $\mathcal{L} \dashv \mathcal{D}$ is a domain.*

Proof. Direct consequence of Lemma B.1 in Appendix B. □

4.1.3 Properties of the Records Domain Constructor (\dashv)

Because we will use the records domain constructor \dashv in constructing domains as least fixed points (lfp's) of functions over domains, as subdomains of Scott's universal domain \mathcal{U} , we need to ascertain \dashv has the domain-theoretic properties needed so that it can be used inside these functions. In this section we thus proof that \dashv is a continuous function over its input domain \mathcal{D} (*i.e.*, that, as a function over domains, \dashv is monotonic with respect to the subdomain relation, \subseteq , and that it preserves lubs of domains under the same relation).

4.1.3.1 Domain Constructor \dashv is Continuous

Theorem 4.2. *\dashv is a continuous function on flat domain \mathcal{L} and arbitrary \mathcal{D} .*

Proof. By lemmas B.2 and B.3 in Section B.2 of Appendix B. □

This concludes our proof that \dashv constructs subdomains of Scott's universal domain \mathcal{U} . Armed with \dashv , we now proceed to presenting the construction of **COOP**, as a simple structural model of OOP.

4.2 COOP Domain Equation

The domain equation that defines **COOP** makes use of two simple domains \mathcal{B} and \mathcal{L} . Domain \mathcal{B} is a domain of atomic "base objects". \mathcal{B} could be a domain that contains a single non-bottom value, *e.g.*, *unit* or *null*, or the set of Boolean values $\{true, false\}$, the set of integers, or some more complex set of primitive values that is the union of Boolean values and various forms of numbers (*e.g.*, whole numbers and floats) and other primitive objects, such as characters and strings, etc.

Domain \mathcal{L} is a flat countable non-empty domain of labels. Elements of \mathcal{L} (or, $|\mathcal{L}|$, more accurately) are proper labels used as names of record members (fields and methods), or the improper “bottom label”, $\perp_{\mathcal{L}}$, that is added to proper labels to make \mathcal{L} a flat domain. Elements of \mathcal{L} other than $\perp_{\mathcal{L}}$ (proper labels) will serve as method names in **COOP**.

The domain equation of **COOP** is

$$\mathcal{O} = \mathcal{B} + \mathcal{L} \multimap (\mathcal{O}^* \multimap \mathcal{O}) \quad (4.4)$$

Domain \mathcal{O} is a domain of simple objects, and it is the primary domain of **COOP**. Equation (4.4) states that a **COOP** object (an element of \mathcal{O}) is either (1) a base object (an element of domain \mathcal{B}); or is (2) a record of methods (*i.e.*, a finite mapping from labels, functioning as method names, to functions), where, in turn, methods are functions from sequences of objects to objects.

4.3 COOP Construction

The construction of domain \mathcal{O} , as the solution of domain equation (4.4), is done using standard techniques for solving recursive domain equations (we use the ‘least fixed point (lfp) construction’, which, according to Plotkin [58], is equivalent to the ‘inverse limit’ construction).

Conceptually, the right-hand-side (RHS) of the **COOP** domain equation (Equation (4.4)) is interpreted as a function

$$\lambda \mathcal{O}_i. \mathcal{B} + \mathcal{L} \multimap (\mathcal{O}_i^* \multimap \mathcal{O}_i) \quad (4.5)$$

over domains, from a putative interpretation \mathcal{O}_i for \mathcal{O} to a better approximation \mathcal{O}_{i+1} for \mathcal{O} . Each element in this sequence is a domain. The solution, \mathcal{O} , to the domain equation is the least upper bound (lub) of the sequence $\mathcal{O}_0, \mathcal{O}_1, \dots$.

Thus, the construction of \mathcal{O} proceeds in iterations, numbered $i + 1$ for $i \geq 0$. We use the empty domain as the initial value, \mathcal{O}_0 , for domain \mathcal{O} , and for each iteration $i + 1$ we take the output domain produced by the domain constructions using the domains \mathcal{O}_i , \mathcal{L} and \mathcal{B} (the values for the function given by Formula (4.5)) as the domain \mathcal{O}_{i+1} introduced in iteration $i + 1$.

4.3.1 A General COOP Construction Iteration

For a general iteration $i + 1$ in the construction of **COOP**, the construction method thus proceeds by constructing

$$\mathcal{M}_{i+1} = \mathcal{O}_i^* \dashrightarrow \mathcal{O}_i$$

using the strict continuous functions domain constructor, \dashrightarrow , and the sequences domain constructor, $*$. Then, using the records domain constructor, \dashv , presented in Section 4.1, we construct the domain of records

$$\mathcal{R}_{i+1} = \mathcal{L} \dashv \mathcal{M}_{i+1}$$

and, finally, using the coalesced sum domain constructor, $+$, we construct

$$\mathcal{O}_{i+1} = \mathcal{B} + \mathcal{R}_{i+1}.$$

4.3.2 The Solution of the COOP Domain Equation

Given the continuity of all domain constructors used in the function defined by the lambda expression (4.5), and given that composition of domain constructors preserves continuity, the function defined by the RHS of the **COOP** domain equation is a continuous function [19, Theorem 2.10 and Corollary 2.11]. The least upper bound (lub) of the sequence $\mathcal{O}_0, \mathcal{O}_1, \dots$ of domains constructed in the construction iterations is the least fixed point (lfp) of the function given by Formula (4.5). According to standard theorems of domain theory about lfp of continuous functions, the lub of the domains \mathcal{O}_i (*i.e.*, their “limit”) is simply their union, and this lub is the solution of Equation (4.4).

To complete the construction of **COOP**, we thus construct the solution \mathcal{O} of the **COOP** domain equation by constructing the union of all constructed domains \mathcal{O}_i , *i.e.*, \mathcal{O} will be given by the equation

$$\mathcal{O} = \bigcup_{i \geq 0} \mathcal{O}_i.$$

Chapter 5

Signatures and Nominality

And He taught Adam the names – all of them.

[The Glorious Qur’ān 2:31]

Class signatures are syntactic descriptions of the external interface of classes, and thus of the common interface of the instances of classes. Given the interface information class signatures contain, class signatures can be used as the basis for type systems that confirm that objects are used consistently and properly within a program. Embedding signature closures as tags inside objects of our model of nominal OOP (as we do in Chapter 6) makes objects of our model be nominal objects.

The purpose of having signatures can be summarized by noticing that they will be used as “nominal object type expressions” embedded inside objects as part of the identity of the objects. Class signatures support the nominal typing and nominal subtyping features of nominal object-oriented programs.

5.1 Signature Constructs

For the purposes of building **NOOP** objects we have three signature constructs: class signatures, (class) signature environments, and (class) signature closures. Members (fields and methods) of objects also have field signatures and method signatures.

Signatures are constructed entirely from class names and field/method names, using standard syntactic set constructors. We use signatures to specify the interfaces

of classes, fields, and methods.¹ The signature for a class specifies the class name, the name of the signatures of the superclasses of the class, and the signatures for the class members. A field signature specifies the name of the field and the class name for field values. A method signature specifies the name of the method, a (possibly empty) sequence of class names describing the inputs to the method, and the class name class describing the output of the method. Information in class signatures is thus derived from the text of classes of OO programs. This information includes the names and signatures of the fields and methods of the objects it describes.

The class name inside an class signature is used as a signature name. Signature names are part of the identity of class signatures. Two class signatures with different names are different, even if they carry the same member (field/method) signatures information. Having signature names in class signatures as part of the identity of these class signatures (*i.e.*, part of their meaning) characterizes class signatures as nominal constructs.

A class signature includes names of class signatures corresponding to immediate superclasses of the class. The signatures referred to are called the *supersignatures* of the class signature. Conditions we present in the definition of class signatures ensure that member signatures inside a class signature match those in its supersignatures, thus agreeing with OO inheritance. Explicitly specifying the supersignatures of a class signature identifies the nominal structure of the class hierarchy immediately above the class in question. It also agrees with the inheritance of the behavioral contract associated with class names, which is what is intended to be inherited in nominal OO.

¹As noted earlier, in the context of this thesis, JAVA **interfaces** are treated as (abstract) classes. Our notion of object and class interfaces should not be confused with **interfaces** of JAVA and other OO languages.

Viewed differently, signatures for classes, fields, and methods, describe the intended external interface of these entities, in terms of names. A *class signature* specifies the name of a class and its immediate superclasses, and the signatures of its fields and methods. A *field signature* specifies the name of the class describing the associated values of the field. A *method signature* specifies the names of the classes associated with the result and the input arguments of the methods.

A *signature environment* is a finite set of class signatures such that no class name appears as the name of two signatures. They can be viewed thus as functions from class/signature names to class signatures. A *signature closure* is a pair of a signature name and a minimal signature environment that includes a class signature with that signature name (called a “root” class signature) and that is referentially-closed over all class names in the root class signature.

In the following subsections all syntactic constructs related to class signatures are rigorously defined. Signature environments and signature closures must be internally consistent to model nominal typing and nominal subtyping in OOP. Conditions of our inductive definitions of class signatures, signature environments and signature closures, ensure the defined constructs satisfy the required internal consistency properties.

5.1.1 Class Signatures

Class signatures are the first constructs we define. Class signatures are syntactic elements of a set S . If N is the set of signature names, and L , as before, is the set of member names, then the set S of class signatures is defined by the equation

$$S = N \times N^* \times FS^* \times MS^* \tag{5.1}$$

where \times is the set theoretic cross product set constructor, and $*$ is the finite-sequences constructor, and where

$$FS = L \times N$$

is the set of field signatures, and

$$MS = L \times N^* \times N$$

is the set of method signatures.

The equation for S formally expresses the view presented above that a class signature is composed of four components: (a) a signature name, (b) a sequence of names of supersignatures, (c) a sequence of field signatures, and (d) a sequence of method signatures.

The use of signature names (members of N) in Equation (5.1) characterizes class signatures as nominal constructs. The first component of a class signature (corresponding to the first occurrence of N in Equation (5.1)) denotes the *signature name* of the class signature.

The second component of a signature, the sequence of signature names, N^* , is the *supersignature names* component of the class signature. Note that the supersignatures component of a class signature can be the empty sequence, implying the class signature which this empty sequence is a component of has no supersignatures.

Having the names of supersignatures of a class signature explicitly included as a component of the class signature is an essential and critical feature in the modeling of nominal subtyping in nominal OOP.

The last two components of a class signature are its *field signatures* (a finite sequence of field signatures), and its *method signatures* (a finite sequence of method

signatures). A field signature is a pair of a field name (a member of L) and a class signature name. Similarly, a method signature is a triple of a method name, a sequence of class signature names (for the method parameters), and a signature name for the method result.²

In the sequel we use metavariable nm to range over members of the set N . Metavariables a and b are used to range over members of L , where a stands for field names and b stands for method names. Also, for visual clarity, we use the more suggestive symbol ‘ \rightarrow ’ for syntactic pairing of the second and third components of a method signature, in place of the standard ‘,’ pairing symbol used elsewhere.

Not all members of S are class signatures. To agree with our intuitions about describing the interfaces of classes and their objects, a member s of S is a class signature if its field signatures component (the third component of s) has no duplicate field names and its method signatures component (the fourth component of s) has no duplicate method names (for simplicity, thus, method overloading is not allowed in our modeling of nominal OOP). It should be noted that field names and method names are in separate name spaces and thus a field and a method in s can share the same name..

²An astute reader may note that, in Equation (5.1), the order of elements in sequences of supersignature names, sequences of field signatures, and sequences of method signatures, is actually immaterial. Repetition of elements inside these three kinds of sequences is also not allowed. In Equation (5.1), $*$ can thus be replaced in these three cases by a finite *sets* constructor (where $\wp_f(X)$ or X^* can be used to denote the set of *finite* subsets of X). For the fourth use of $*$, that of signature names of method parameters, order does matter and repetition of signature names inside the sequences is allowed. Instead of using (the more-accurate) finite sets in the three other cases, we keep using (the more-intuitive) finite sequences, but, in Section 5.2, we then define and use an equivalence relation for class signatures that asserts the equivalence of signatures that have equivalent sequences (ignoring the order of their elements and any element repetitions) in these three components. This approach somewhat mimics more closely what language compilers usually do.

5.1.2 Signature Environments

Signature environments are finite sets of class signatures where each signature name is associated with exactly one signature³. If nm is guaranteed to be the signature name of some class signature in a signature environment se , we use the function-application notation $se(nm)$ to refer to this particular class signature.

Similar to class signatures, not all finite sets with unique class names are signature names. In addition to uniqueness of signature names, a finite set of class signatures needs to satisfy the following consistency conditions to function as a signature environments. It first should be noted that a signature environment specifies two relations: an immediate supersignature relation and a direct-reference (adjacency) relation. The first is a subset of the second. These two relations can be represented as directed graphs. The consistency conditions constrain these two relations and their corresponding graphs.

A finite set se of class signatures is a signature environment if and only if

1. A class signature, with the right signature name, belongs to se for each signature reference (by a signature name) in each class signature of se . This is a closure and consistency constraint on signature environments.
2. The graph for the supersignature relation for se must be cycle-free (*i.e.*, an acyclic graph, commonly called a DAG). This well-foundedness constraint forces se to have at least one class signature that has no supersignatures (*i.e.*, the sequence of supersignature names of such a class signature is the empty sequence).

³Defining signature environments as constrained sets, rather than functions from signature names to class signatures, is motivated by redundancy concerns similar to those that could be met in tables of relational databases, given that each “key” (*i.e.*, a signature name) in a signature environment is mapped to a *unique* class signature.

3. The set of member (field and method) signatures of each class signature s in se is a *superset* of the set of member signatures of each supersignature named by the supersignatures component of s (*i.e.*, the set of member signatures of s is a superset of the union of those sets for the supersignatures). This condition makes class signatures in se reflect the explicit inheritance information in nominal OOP, by requiring a class signature to only extend (add to) the set of members supported by an explicitly-specified supersignature..By requiring the members of a class signature to be a superset of the members of all of its supersignatures *exact* matching of member signatures is required. This requirement enforces an *invariant subtyping* rule for field and method signatures.⁴
4. The finite set of class signatures se is a signature environment if and only if it satisfies all constraints above. A finite set of class signatures that does not satisfy one or more of the above constraints above is not a signature environment.

5.1.2.1 Circularity in Signature Environments

Even though consistency conditions for signature environments preclude a class signature from circularly naming itself as one of its own supersignatures (even indirectly via other class signatures), the conditions do not preclude a class signature from referring to itself via its name in the signature of a field, or that of a method parameter or method return value. This kind of reference is called a *circular reference*.

⁴We enforce an invariant subtyping rule of members for the sake of simplicity, and for the sake of mimicking pre-generics JAVA. This constraint can be relaxed to model covariant subtyping of method return value signatures (which is available in mainstream OO languages such as JAVA 5.0+), but we do not do so in this thesis. The constraint can even be further relaxed to model field signature covariant subtyping, as well as to model contravariance of subtyping of method parameter signatures. However, a relaxed constraint that allows covariance of field signatures/types, in particular, is known to be unsound in OO languages with mutation (*e.g.*, in EIFFEL [50, 26]).

The consistency conditions also allow for expressing mutually-dependent class signatures, where a signature refers to itself indirectly via other class signatures. This kind of indirect reference is called a *mutually-circular reference*. Circular references are indeed quite common in mainstream OOP, to allow developers to simulate real-world entities that are circularly-dependent⁵. The nominality of classes in mainstream OOP allows them to readily handle circular references, since they merely involve the use of class names.

5.1.3 Signature Closures

Having defined class signatures and signature environments, a (class) signature closure is a pair of a signature name and a signature environment. The first component of a signature closure, the signature name, tells which class signature in the second component (the signature environment) is the “root” class signature of the signature environment. Thus, not all pairs of signature names and signature environments are signature closures. Such a pair $sc = (nm, se)$ is a signature closure if and only if there is a class signature s inside se with signature name nm (*i.e.*, nm is in the domain of se when se is viewed as a function, and thus $se(nm)$ is always defined) *and* if the direct-reference (adjacency) relation corresponding to se is referentially-closed relative to s . The class signature s then is called the “root” signature of the sc . Relative to the root class signature, a signature environment is thus minimal (contains no unnecessary class signatures). All class signatures in the signature environment of a signature closure are accessible via paths in the adjacency graph of the signature environment starting from the node corresponding to the root signature name.

⁵Check, for example, the circularity in the definitions of the core 2,000, or so, key words used to define the meaning of words in a dictionary.

The set of all signature closures is used to construct a flat domain \mathcal{S} that is used in the construction of **NOOP** in Chapter 6.

5.2 Signature Equality

Checking the equality of signature constructs is needed in multiple places in our modeling of nominal OOP. Equality is defined on signatures as would be expected given their definitions above.

5.2.1 Equality of Class Signatures

For class signatures $S_1 = (nm_1, [nms_1], [fss_1], [mss_1])$ and $S_2 = (nm_2, [nms_2], [fss_2], [mss_2])$, we have

$$S_1 = S_2 \Leftrightarrow (nm_1 = nm_2) \wedge (nms_1 \equiv nms_2) \wedge (fss_1 \equiv fss_2) \wedge (mss_1 \equiv mss_2),$$

where \equiv is an equivalence relation on sequences that ignores the order and repetitions of elements of a sequence, and where for two field signatures (in sequences fss_1 and fss_2), $fs_1 = (a_1, nm_1)$ and $fs_2 = (a_2, nm_2)$, we have

$$fs_1 = fs_2 \Leftrightarrow (a_1 = a_2) \wedge (nm_1 = nm_2),$$

and similarly, for two method signatures (in sequences mss_1 and mss_2), $ms_1 = (b_1, [nms_1] \rightarrow nm_1)$ and $ms_2 = (b_2, [nms_2] \rightarrow nm_2)$, we have

$$ms_1 = ms_2 \Leftrightarrow (b_1 = b_2) \wedge (nms_1 = nms_2) \wedge (nm_1 = nm_2)$$

(note that sequence equality, not sequence equivalence, is used. Order and repetitions do matter for method parameter signatures).

We should particularly note that the names of two class signatures have to be equal for the two signatures to be equal. Two class signatures that share all other components but have different signature names are not equal, since a signature name is part of its identity.

5.2.2 Equality of Signature Environments and Signature Closures

Two signature environments are equal if and only if they are equal as sets.⁶ Two signature closures are equal if and only if they are equal as pairs, *i.e.*, if they have equal components. Equal signature closures have the same signature name and equal signature environments.

5.3 Extension of Signature Environments

A relation between signature environments that we will need when we discuss inheritance in the next section (Section 5.4) is the extension relation. A signature environment se_2 extends a signature environment se_1 (written $se_2 \blacktriangleleft se_1$) if se_2 binds the names defined in se_1 to exactly the same class signatures as se_1 does. Viewed as sets se_2 would be a superset of se_1 . Thus we have

$$se_2 \blacktriangleleft se_1 \Leftrightarrow se_2 \supseteq se_1.$$

It should be noted that se_2 may have extra signature names mapped to extra

⁶It should be noted that because class names (as signature names) are part of the identity of class signatures, the consistent renaming of all class signatures in a signature environment (“alpha-renaming”) does *not* produce an equivalent signature environment. Signature names are thus *not* variable names, because unlike variable names, signature names are bound to behavioral contracts

class signatures not in se_1 . Given it is the same as the subset relation, the extension relation between signature environments is a partial-order relation.

5.4 Inheritance and Subsigning

Inheritance is a defining characteristic of object-oriented programming. Cook [23] defines inheritance as ‘a mechanism for the definition of new program units by modifying existing ones in the presence of self-reference’. Cook has multiple notions of inheritance. For the purposes of this thesis, we are only interested in Cook’s notion of *type inheritance*. For object types, thus, Cook’s defines type inheritance as ‘a mechanism for the definition of new class signatures by modifying existing ones, in the presence of “self-type” ’.

In this thesis, we define inheritance a bit differently. We define inheritance in nominal OOP languages as ‘a mechanism by which a new class signature is defined by adding members to an explicitly-specified set of other class signatures’.

In nominal OOP, inheritance (of class signatures) makes use of class names to explicitly specify the interfaces (implicitly including the informal behavioral contracts) a class and its instances adhere to. In our model of nominal OOP, thus, object interface and contract inheritance is modeled by class signatures explicitly specifying their supersignatures.

The supersignatures component of class signatures defines a syntactic ordering relation between signature closures. We call this relation between signature closures *subsigning*.

A signature closure sc_2 is an *immediate subsignature* (\trianglelefteq_1) of a signature closure sc_1 if the signature environment of sc_2 is an extension (\blacktriangleleft) of the signature environment of sc_1 and the signature name of sc_1 is in the supersignature names component of the

root signature of sc_2 , *i.e.*,

$$(nm_2, se_2) \trianglelefteq_1 (nm_1, se_1) \Leftrightarrow se_2 \blacktriangleleft se_1 \wedge (nm_1 \in \text{super_sigs}(se_2(nm_2))).$$

The subsigning relation \trianglelefteq is the reflexive transitive closure of the immediate subsignature relation \trianglelefteq_1 , where we use the symbol \trianglelefteq to denote the subsigning relation between signature closures.

Chapter 6

NOOP: A Domain-Theoretic Model of Nominal Object-Oriented Programming

A model focuses on parts rather than the whole. It is a caricature which overemphasizes some features at the expense of others. A model is a fetish in which the importance of one key part of the object of interest is obsessively exaggerated until it comes to represent the object's quintessence.

~Physicist Prof. Emanuel Derman

This chapter presents the construction of **NOOP**: a mathematical, domain-theoretic model of nominal OOP. The construction of **NOOP** uses the same construction method we employed for building **COOP** (our simple structural model of OOP, presented in Chapter 4). The construction of **NOOP** also uses the same domain constructors we used for constructing **COOP**. The algebraic structure of **NOOP**, however, is different from that of **COOP**.

To model nominality, **NOOP** pairs appropriate signature closures with records of fields and methods to model objects in nominal OOP. A **NOOP** object contains two records—one for field bindings and one for method bindings—to accommodate separate namespaces for fields and methods.¹

The formulation of objects in **NOOP** is rich enough that no base domain (like domain \mathcal{B} in the definition of **COOP**) is necessary.

The construction of **NOOP** proceeds in two steps. First, we use a simple recursive domain equation to define a domain of objects containing signature information as

¹As is the case in mainstream OO languages such as JAVA and C#.

well as member bindings in separate field and method records. Second, we “filter out” invalid objects where the signature information is inconsistent with the member bindings. A simple recursive definition of objects with signature information does not force the signature information embedded in objects to conform with their member bindings. It is easy to define a projection on this raw domain that eliminates invalid objects.

We define **NOOP** as being the model with only the domain of valid objects in the constructed solution of the **NOOP** domain equation. Hence, the solution of the **NOOP** domain equation does not define the actual **NOOP** domain of objects but constructs a larger domain of a model that we call *preNOOP*. We then define the domain of **NOOP** object as the image (range) of a filtering function on *preNOOP* that only retains objects with consistent signature information. A filtering function on domain is well-defined if its image is a subdomain of the input domain (See Definition A.20 in Appendix A).

As constructed, **NOOP** presents our answer for how we should mathematically think of object-oriented software. We conclude this chapter by discussing and proving some of the fundamental properties of **NOOP**, then we use these results to show how the inclusion of nominal information completely reconciles inheritance and subtyping in nominal OOP.

NOOP is a nominal model of OOP because objects include signature information for their visible interfaces. This information provides a framework of naturally partitioning **NOOP** into nominal types. The “exact” nominal type corresponding to a class **C** is the set of all objects tagged with the signature closure for **C**.²

²In JAVA, for example, the objects in the exact type for a class **C** are precisely those for which the `getClass()` method returns the class object for **C**.

A cardinal principle of OOP is that objects from subclasses of C conform to the visible interface of C and can be used in place of objects from class C . Hence, the natural type associated with C consists of the objects in class C plus the objects in *all* subclasses of C . In typed nominal OO languages, the type designated by C is not the “exact” nominal type for C but the union of all the exact types for the classes D that are subtypes of C (including C itself).

In **NOOP**, a class B inherits from class A if the signature closure for B subsigns the signature closure for A .³ In contrast to other models of OOP, we prove that inheritance in **NOOP** is completely consistent with subtyping: a class B inherits from class A if and only if class B is a subtype of A . In other words, inheritance *is* subtyping, overturning one of the mantras of OOP research.

6.1 NOOP Domain Equation

The domain equation that defines *preNOOP* makes use of two simple domains \mathcal{L} and \mathcal{S} , where domain \mathcal{L} is the same flat domain of labels as in Chapter 4, and \mathcal{S} is the flat domain of signature closures (see Section 5.1.3).

The recursive definition for *preNOOP* is fundamentally different from the definitions of other OO domains like Cardelli’s **SOOP** [14] because every object in *preNOOP* contains a signature closure specifying its external interface. From an intuitive perspective, the signatures embedded in objects are certificates authenticating their interfaces.

³Note that subsigning is defined so that the JAVA subclassing relation (prior to the addition of generics in JAVA 5) exactly matches the subsigning relation on the signatures of those classes. The same observation applies to C# (without generics) and C++ (without templates).

The **NOOP** domain equation, which defines and describes *preNOOP*, is

$$\hat{\mathcal{O}} = \mathcal{S} \times (\mathcal{L} \multimap \hat{\mathcal{O}}) \times (\mathcal{L} \multimap (\hat{\mathcal{O}}^* \multimap \hat{\mathcal{O}})) \quad (6.1)$$

where $\hat{\mathcal{O}}$ is the domain of (valid and invalid) objects of *preNOOP*, \times is the strict product domain constructor, and domains \mathcal{L} and \mathcal{S} are as described above. Equation (6.1) states that every object in *preNOOP* (and **NOOP**) is a triple composed of: (1) a signature closure (a member of the domain \mathcal{S}), (2) a fields record (a member of $\mathcal{L} \multimap \mathcal{O}$), and (3) a methods record (a member of $\mathcal{L} \multimap (\mathcal{O}^* \multimap \mathcal{O})$).

In the next section, we present the construction of *preNOOP* as the solution of the **NOOP** domain equation, and we define the filtering function that maps *preNOOP* onto **NOOP**.

6.2 Construction of *preNOOP*

Similar to **COOP**, the construction of *preNOOP* proceeds in iterations, driven by the structure of the RHS of the **NOOP** domain equation, viewed as a continuous function from domains to domains.

6.2.1 A General *preNOOP* Construction Iteration

Similar to the construction of **COOP**, a general iteration $i + 1$ in the construction of *preNOOP* proceeds by forming

$$\mathcal{M}_{i+1} = \hat{\mathcal{O}}_i^* \multimap \hat{\mathcal{O}}_i$$

using the strict continuous functions domain constructor $\dashv\rightarrow$, and the sequences domain constructor $*$. Then, using the records domain constructor $\dashv\circ$, we construct the domain of method records

$$\mathcal{MR}_{i+1} = \mathcal{L} \dashv\circ \mathcal{M}_{i+1}$$

and the domain of field records

$$\mathcal{FR}_{i+1} = \mathcal{L} \dashv\circ \hat{\mathcal{O}}_i$$

and, finally, using the strict product domain constructor, \times , we construct the domain of objects

$$\hat{\mathcal{O}}_{i+1} = \mathcal{S} \times \mathcal{FR}_{i+1} \times \mathcal{MR}_{i+1}.$$

It should be noted that a non-bottom object $o_1 = (osc_1, fr_1, mr_1)$ in $\hat{\mathcal{O}}_{i+1}$ approximates a non-bottom object $o_2 = (osc_2, fr_2, mr_2)$ if and only if osc_1 approximates osc_2 , fr_1 approximates fr_2 , and mr_1 approximates mr_2 . Given the flatness of \mathcal{S} , o_1 approximates o_2 only if they have the *same* signature closure. Using domains $\hat{\mathcal{O}}_i$ we, then, finally define the domain $\hat{\mathcal{O}}$ of *preNOOP* objects as the directed closure (also called the *ideal completion*) of the infinite union of the different $\hat{\mathcal{O}}_i$'s:

$$\hat{\mathcal{O}} = IdealCompletion\left(\bigcup_{i \geq 0} \hat{\mathcal{O}}_i\right).$$

6.2.2 Ranking Finite Domain Elements

For purpose of proving properties of *preNOOP*, it is useful to define a ranking notion on the finite elements of the *preNOOP* domain of objects. The solution to domain

equation (6.1) builds a sequence $\hat{\mathcal{O}}_0, \hat{\mathcal{O}}_1, \hat{\mathcal{O}}_2, \dots$, of domains where $\hat{\mathcal{O}}_i \subseteq \hat{\mathcal{O}}_{i+1}$. In this solution, we can assign finite objects of $\hat{\mathcal{O}}$ a rank, when we note that every finite object in $\hat{\mathcal{O}}$ is introduced in some finite iteration $\hat{\mathcal{O}}_i$. Every finite element o of $\hat{\mathcal{O}}$ is constructed in some approximating domain $\hat{\mathcal{O}}_i$. The *rank* of a finite object o of domain $\hat{\mathcal{O}}$ is the integer index of the iteration in which the element is first constructed. Thus, in each $\hat{\mathcal{O}}_i$, the new finite elements of $\hat{\mathcal{O}}_i$ have rank i .⁴ The only object of rank 0 is $\perp_{\mathcal{O}}$ (the only element of the empty domain $\hat{\mathcal{O}}_0$). All finite non-bottom objects of domain $\hat{\mathcal{O}}$ constructed after *one* application of the domain construction function (corresponding to the RHS of Equation (6.1)) to domain $\hat{\mathcal{O}}_0$ have rank 1. The finite objects of rank 1 are objects having the form $(\text{os}, \{\dots, l_i \mapsto \perp_{\mathcal{O}}, \dots\}, \{\dots, l_j \mapsto \perp_{\mathcal{M}}, \dots\})$, in which all fields are bound to the bottom object $\perp_{\mathcal{O}}$, and all methods are bound to the bottom element of the methods domain $\perp_{\mathcal{M}}$ (*i.e.*, the only element in the domain $\hat{\mathcal{O}}_0^* \multimap \hat{\mathcal{O}}_0$).

A finite object constructed in some iteration of the construction method always has a higher rank than that of the finite objects used to construct this element (these objects were constructed in earlier iterations of the construction method). Thus, when an object o is represented by a derivation tree, whose root is the object, and the derivation trees representing the objects used to construct o as its subtrees, the depth of this tree is the rank of o . The rank of an object can thus be viewed as the “construction depth” of the object, *i.e.*, the length of the longest path from the top of the tree representing the object to a leaf of the tree (Given the **NOOP** domain equation has no base objects, all leaves in trees representing elements of the *preNOOP* domain of objects are nodes that only represent $\perp_{\mathcal{O}}$).

⁴All of the elements in every domain $\hat{\mathcal{O}}_i$ are finite. Hence, the elements of $\bigcup \hat{\mathcal{O}}_i$ form a finitary basis for $\hat{\mathcal{O}}$. Therefore any element not in $\bigcup \hat{\mathcal{O}}_i$ is not finite. In a Scott domain constructed from a finitary basis, an element is finite if and only if it is a member of the finitary basis.

6.3 Filtering of *pre*NOOP to NOOP

We define a filtering function called **filter** to map domain $\hat{\mathcal{O}}$, the *pre*NOOP domain of objects, onto the NOOP domain of valid objects. We use the symbol \mathcal{O} to denote the domain of valid objects produced by **filter**.

Definition 6.1. A finite object o in $\hat{\mathcal{O}}$ is *valid* if it is the bottom object $\perp_{\mathcal{O}}$, or if it is a non-bottom object (sc, fr, mr) such that

- the sets of names of fields and methods (the field shape and the method shape) of the root class signature of sc are exactly the same as the set of names of fields in (*i.e.*, the shape of) the fields record fr and the set of names of methods (*i.e.*, the shape of) the methods record mr , respectively,
- non-bottom objects bound to field names in fr have signature closures that subsign the signature closures for the corresponding fields in the root class of sc , and
- non-bottom functions bound to method names in mr conform to the corresponding method signatures in the root class of sc . By conformance, the functions are required take in sequences of valid objects that subsign (component-wise) the corresponding sequences of method parameter signatures in the root class of sc prepended with sc itself, and to return valid objects with signatures that subsign the corresponding return value signatures specified in the method signatures in the root class of sc . (Note that sc must be prepended to sequences of method parameter signatures when checking for validity of method arguments, because in an application of a method in any nominal OO language, the first argument is bound to a receiver object **this/self** whose signature closure must subsign sc).

Definition 6.2. The function `filter` mapping $\hat{\mathcal{O}}$ into $\hat{\mathcal{O}}$ is defined by the recursive definitions in Figure 6.1.

```

fun filter(o: $\hat{\mathcal{O}}$ ): $\mathcal{O}$ 
  match o with ((nm,se), fr, mr)
    if (sf-shp(se(nm)) != rec-shp(fr))  $\vee$ 
      (sm-shp(se(nm)) != rec-shp(mr))
      return  $\perp_{\mathcal{O}}$  // because of non-matching shapes
    else // lazily construct closest valid object to o
      match se(nm), fr, mr with
        (_, _, [(ai, snmi) | i=1, ..., m ],
              [(bj, mi_snmj, mo_snmj) | j=1, ..., n]),
        (fr-tag, {ai  $\mapsto$  oi | i=1, ..., m}),
        (mr-tag, {bj  $\mapsto$  mj | j=1, ..., n})
        let si = se_clos(se, snmi)
        let misj = map(se_clos(se), [nm::mi_snmj])
            // nm is prepended to mi_snmj to handle 'this'
        let mosj = se_clos(se, mo_snmj)
        return ((nm,se),
              (fr-tag, {ai  $\mapsto$  filter-obj-sig(si,oi) | i=1, ..., m}),
              (mr-tag, {bj  $\mapsto$  filter-meth-sig(misj, mosj, mj)
                        | j=1, ..., n}))

fun filter-obj-sig(ss: $\mathcal{S}$ , o: $\hat{\mathcal{O}}$ ): $\mathcal{O}$ 
  match o with (s, _, _)
    if (s  $\trianglelefteq$  ss)
      return filter(o) // closest valid object to o
    else
      return  $\perp_{\mathcal{O}}$  // because of no subsigning

fun filter-meth-sig(in_s: $\mathcal{S}^+$ , out_s: $\mathcal{S}$ , m: $\hat{\mathcal{M}}$ ): $\mathcal{M}$ 
  return ( $\lambda o^*$ . let vo* = map2(filter-obj-sig, in_s, o*)
          in filter-obj-sig(out_s, m(vo*)) )

```

Figure 6.1 : Filtering *pre*NOOP to NOOP

In Figure 6.1, to compute shapes of signatures we have

```
sf-shp((_, _, [(ai, _) | i=1,...,m], _) = {a1,...,am}
sm-shp((_, _, _, [(bj, _, _) | j=1,...,n]) = {b1,...,bn}
```

and to compute shapes of records we have

```
rec-shp((_, { li ↦ _ | i=1,...,k })) = {l1, ...,lk}
```

The function `se_clos(se, nm)` in Figure 6.1 computes a signature closure (a pair of a signature name and a signature environment) corresponding to signature name `nm` whose first component is `nm` and whose second component is the minimal subset of signature environment `se` that makes `se_clos(se, nm)` a signature closure. Note that by the referential-closure of `se`, `nm` is guaranteed to be in the domain of `se`, and that the class signature referenced by `nm` in `se` is the root signature of `se_clos(se, nm)`. Note that to handle `this/self` a “curried” version of `se_clos` is passed to the `map` function.

Also, in Figure 6.1 the domain \mathcal{S}^+ is the domain of non-empty sequences of signature closures (signature closure sequences passed to `filter-meth-sig` are always non-empty because object methods are always passed in at least one object argument, corresponding to the value `this/self`). The function `map2` is the two-dimensional version of `map`, which takes a binary function and two input lists as its arguments.

Figure 6.2 : Filtering Auxiliary Definitions

It should be noted that all functions in Figures 6.1 and 6.2 are not eager (“call-by-value”) functions but *lazy* (“call-by-name”) functions.

The definition of the filtering function `filter` in Figures 6.1 and 6.2 thus states that it takes an object o of $\hat{\mathcal{O}}$ and returns a corresponding valid object of \mathcal{O} . If the object is invalid because of unequal shapes in the signature of o and its member records, `filter` returns the bottom object $\perp_{\mathcal{O}}$ ($\perp_{\mathcal{O}}$ is the closest valid object to an invalid object with non-equal shapes in its signature and records). Otherwise, o has equal signature and record shapes but may have objects bounds to its fields, or taken in or returned by its methods, whose signature closure does not subsign the corresponding signatures in the signature closure of o . In this case, `filter`

then lazily constructs and returns the closest approximating valid object to o , where all non-bottom fields and non-bottom methods of o are guaranteed (via functions `filter-obj-sig` and `filter-meth-sig`, respectively) to have signatures that subsign the corresponding signatures in the signature closure of o .

Function `filter-obj-sig` checks if its input object o has a signature closure s that subsigns a required declared signature closure ss (also input to `filter-obj-sig`). If s is not a subsignature of ss , `filter-obj-sig` returns $\perp_{\mathcal{O}}$. If it is, the function calls `filter` on o , thereby returning the closest valid object to o .

For methods, when `filter-meth-sig` is applied to a method m it returns a valid method that when applied to the same input o^* as m (a member of $\hat{\mathcal{O}}^*$) returns the closest valid object to the output object of m that subsigns the declared out signature closure `out_s` corresponding to the sequence of valid objects closest (component-wise) to o^* that (component-wise) subsigns the declared sequence of input signature closures `in_s` prepended with the signature closure of the object enclosing m (to properly filter the first argument object in o^* , which is passed in as a value for `this/self`).

6.3.1 Filtering is a Finitary Projection

For domain \mathcal{O} to be well-defined as a subdomain of $\hat{\mathcal{O}}$, the filtering function `filter` needs to be a finitary projection (See Section 8 in [19]).

Theorem 6.1. *filter is a finitary projection.*

Proof. See Theorem B.1 in Section B.3. □

By proving that `filter` is a finitary projection, Theorem 6.1 proves thus that domain \mathcal{O} is well-defined.

6.3.2 The NOOP Domain of Objects

Given that the filtering function `filter` is computable its behavior on infinite objects is completely determined by its behavior on finite objects.

The behavior of `filter` is thus completely defined by its behavior on elements of domains $\hat{\mathcal{O}}_i$. The `filter` function in fact can be considered as a function that maps domains $\hat{\mathcal{O}}_i$ to domains \mathcal{O}_i . Given that `filter` is a continuous function, the lub of its output domains \mathcal{O}_i is the result of applying `filter` to the domain $\hat{\mathcal{O}}$ (*i.e.*, our sought-after domain \mathcal{O}) because, given continuity, we have

$$\mathcal{O} = \text{filter}(\hat{\mathcal{O}}) = \text{filter}(\sqcup \hat{\mathcal{O}}_i) = \sqcup \text{filter}(\hat{\mathcal{O}}_i) = \sqcup \mathcal{O}_i.$$

Thus, \mathcal{O} is defined as the union (*i.e.*, the lub) of domains $\mathcal{O}_i = \text{filter}(\hat{\mathcal{O}}_i)$ rather than as `filter`($\hat{\mathcal{O}}$).

6.4 Properties of NOOP

Using the **NOOP** domain of objects, \mathcal{O} , in Section 6.4.1 we associate nominal object types with signature closures (where nominal object types, as discussed in Section 2.1.4, are certain subsets of \mathcal{O} having similar objects). Signatures, thus, are associated with certain subsets of the domain \mathcal{O} of **NOOP**. These sets of objects associated with signature closures are called nominal object types. In Section 6.4.2, we then show that class types (nominal object types) are subdomains of \mathcal{O} .

Finally, in Section 6.5 we show how our association of class types with signature closures (which embody inheritance in nominal OOP) enables **NOOP** to completely reconcile inheritance with subtyping. That is, that in our model **NOOP** of nominal OO programming, the type associated with a class **B** is a subtype of the type associated

with class **A** if and only if the signature of class **B** inherits from the signature of class **A**. In other words, we show that inheritance and subtyping in mainstream OOP *exactly* coincide, and that the shibboleth “inheritance is not subtyping” among PL researchers is simply *wrong*.

6.4.1 Semantics of Signatures

Using notions of nominal OOP that we introduced less formally in Section 2.1.4, we now provide a formal definition for nominal object types (also called class types).

Given that objects in **NOOP** have signatures embedded inside them, the nominal object type in **NOOP** that interprets a signature closure sc (which we denote by $\mathbb{S}[sc]$) is simply defined by the equation

$$\mathbb{S}[sc] = \{(scs, fr, mr) \in \mathcal{O} \mid scs \trianglelefteq sc\} \cup \{\perp_{\mathcal{O}}\} \quad (6.2)$$

In other words, the interpretation of a signature closure sc is the set of all objects (scs, fr, mr) in domain \mathcal{O} with a signature closure scs that subsigns sc , or the bottom object $\perp_{\mathcal{O}}$. The approximation ordering of elements of $\mathbb{S}[sc]$ is the one inherited from \mathcal{O} .

6.4.2 Signatures Denote Subdomains of \mathcal{O}

6.4.2.1 Types as Subdomains

To prove that class signatures denote subdomains of \mathcal{O} (Check Definition A.20 in Appendix A), we start by defining the notion of an ‘exact object type’ corresponding to a signature closure.

6.4.2.2 Exact Object Types

The *exact object type* corresponding to a signature closure sc is the set of objects in domain \mathcal{O} that have a signature closure sc , or $\perp_{\mathcal{O}}$

$$\mathbb{SE}[sc] = \{(sc, fr, mr) \in \mathcal{O}\} \cup \{\perp_{\mathcal{O}}\} \quad (6.3)$$

with the approximation ordering inherited from \mathcal{O} .

Two posets are *disjoint* if their intersection is the empty poset (which only contains the bottom element). Given the definition of exact object types in Equation (6.3), the sets $\mathbb{SE}[sc]$ are disjoint posets for different sc (because of the flatness of \mathcal{S}). Thus, given the definition of $\mathbb{S}[sc]$ in Equation (6.2), with little effort we can immediately see that

$$\mathbb{S}[sc] = \bigcup_{scs \trianglelefteq sc} \mathbb{SE}[scs] \quad (6.4)$$

since

$$\begin{aligned} \bigcup_{scs \trianglelefteq sc} \mathbb{SE}[scs] &= \bigcup_{scs \trianglelefteq sc} (\{(scs, r) \in \mathcal{O}\} \cup \{\perp_{\mathcal{O}}\}) = \bigcup_{scs \trianglelefteq sc} (\{(scs, r) \in \mathcal{O}\}) \cup \bigcup_{scs \trianglelefteq sc} (\{\perp_{\mathcal{O}}\}) \\ &= \bigcup_{scs \trianglelefteq sc} \{(scs, r) \in \mathcal{O}\} \cup \{\perp_{\mathcal{O}}\} = \{(scs, r) \in \mathcal{O} \mid scs \trianglelefteq sc\} \cup \{\perp_{\mathcal{O}}\} = \mathbb{S}[sc]. \end{aligned}$$

6.4.2.3 Nominal Object Types are Subdomains

We now establish that a nominal object type is a subdomain of \mathcal{O} in two steps, where we, first, prove that each exact object type is a subdomain of \mathcal{O} , then, in a second step based on the first one, we show that a nominal object type, as a union of exact object types, is a subdomain of \mathcal{O} .

Lemma 6.1. *Exact object types are subdomains of \mathcal{O} .*

Proof. Based on the definition of exact object types, properties (1), (2) and (3) of Definition A.20 in Appendix A are immediate. For property (4), notice that exact object types are closed under lubs, because all non-bottom objects in a (finite or infinite) chain in an exact object type have the same signature. Hence, the lub in \mathcal{O} must have the same signature, and thus, by the definition of exact object types, the lub belongs to the exact object type. \square

Lemma 6.2. *The union of a disjoint collection of subdomains of \mathcal{O} is a subdomain of \mathcal{O} .*

Proof. Properties (1) and (2) of Definition A.20 in Appendix A are trivial to prove. For properties (3) and (4), notice that due to the disjointness of the members of the collection, any (finite or infinite) chain in the union must lie entirely (excepting bottom) in one exact member of the collection. Thus, the approximation and lub relations in the union of the collection are the same as in \mathcal{O} . \square

Theorem 6.2. *Nominal object types are subdomains of \mathcal{O} .*

Proof. Immediate, from Lemma 6.2, Lemma 6.1, the definition of nominal object types as given by Equation 6.4, and noting that exact object types are disjoint posets. \square

6.5 Reconciling Inheritance with Subtyping

Now we can easily see what it means for nominal OO type systems to completely reconcile inheritance and subtyping: Two signature closures are in the subsigning relation if and only if the nominal object types (class types) denoted by the two signature closures are in the subset relation (*i.e.*, are in the nominal subtyping relation).

Theorem 6.3 (Subsigning \Leftrightarrow Nominal Subtyping). *For two signature closures sc_1 and sc_2 denoting non-empty class types $\mathbb{S}[sc_1]$ and $\mathbb{S}[sc_2]$, we have*

$$sc_1 \trianglelefteq sc_2 \Leftrightarrow \mathbb{S}[sc_1] \subseteq \mathbb{S}[sc_2] \quad (6.5)$$

Proof. Based on Equation (6.2), the proof of this theorem is simple.

Case: The \Rightarrow (only if) direction:

If $sc_1 \trianglelefteq sc_2$, then by the definition of $\mathbb{S}[sc_2]$ an element of $\mathbb{S}[sc_1]$ belongs to $\mathbb{S}[sc_2]$ (the variable scs in Equation (6.2) is instantiated to sc_1).

While the \Rightarrow direction in Theorem 6.3 is important, and is trivial to prove, the \Leftarrow direction is more significant.

Case: The \Leftarrow (if) direction:

By Equation (6.2), a non-bottom object o of $\mathbb{S}[sc_2]$ with signature closure osc has $sc \trianglelefteq sc_2$, and, similarly, a non-bottom object o' of $\mathbb{S}[sc_1]$ has a signature sc' such that $sc' \trianglelefteq sc_1$. Given $\mathbb{S}[sc_1] \subseteq \mathbb{S}[sc_2]$, by inclusion, we also have $sc' \trianglelefteq sc_2$ for all sc' in objects of $\mathbb{S}[sc_1]$. Thus, for each signature closure sc' that is a subsign of sc_1 , we have that sc' is a subsign of sc_2 . That is, we have

$$\forall sc'. \forall o = (sc', _, _). (sc' \trianglelefteq sc_1 \Rightarrow o \in \mathbb{S}[sc_1] \Rightarrow o \in \mathbb{S}[sc_2] \Rightarrow sc' \trianglelefteq sc_2).$$

By reflexivity, we have $sc_1 \trianglelefteq sc_1$. Hence, $sc_1 \trianglelefteq sc_2$. □

We should notice that nominality, where signatures are embedded into objects, is what makes $\mathbb{S}[sc_2]$ being a superset of $\mathbb{S}[sc_1]$ imply that sc_1 has sc_2 as one of its ancestor supersignatures. The definition of \mathbb{S} guarantees that the set $\mathbb{S}[sc_2]$ does not have elements o of \mathcal{O} that have a signature closure sc unless $sc \trianglelefteq sc_2$ or $o = \perp_{\mathcal{O}}$. By

set inclusion, the set $\mathbb{S}[sc_1]$ does not have objects that are not in $\mathbb{S}[sc_2]$ (even though, generally, the opposite is not true, unless $sc_1 = sc_2$).

Chapter 7

Discussion and Future Work

In this chapter, we discuss the implications of the nominal model of OOP presented in this thesis. We also discuss some of the limitations of the model, and we explore some of the possible paths for building more elaborate nominal models.

7.1 Main Research Conclusions and Contributions

In the preceding chapters, we have shown that mainstream OO programming languages have straightforward denotational models that have quite a different internal structure than models for structural OO programming languages. Unfortunately, the research literature on OO programming languages is rife with misstatements and faulty intuitions such as “inheritance is not subtyping”, based on incorrectly assuming that structural OO models are representative of mainstream OO programming languages.

Our model demonstrates that in fact inheritance is completely consistent with subtyping in nominal OO languages. This is not a dispute about taste or subjective preferences but about the fundamental mathematical and technical properties of OO languages. The structural models of OO programming (like **SOOP**) are *wrong* when they are applied to nominal OO languages, because they are *faulty* and *incomplete*. They are faulty because the meaning of class types includes object values that *do not belong to the corresponding class types* in nominal languages. They are incomplete

because they do not include nominal information in the object denotations so that fundamental operations, like type casting, cannot be defined, because they critically depend on missing information.

7.1.1 Main Research Conclusions

The main conclusions we reached based on research presented in this thesis are:

1. Extant models of OOP do not precisely model mainstream OOP, because the extant models are founded on a structural view of the meaning of objects while mainstream OO languages are based on a nominal view of the meaning of objects. Extant models thus can lead to making wrong conclusions about OOP. Incorporating nominal information is a must for any precise model of OOP.
2. Nominal models of OOP that embed class names and signatures in objects readily support program operations that require nominal information, since they accommodate the accurate definition of class types. In such models, inheritance and subtyping are perfectly consistent.
3. Although nominal models incorporate specific class names in objects, it is still possible to define comprehensive nominal models of OOP that incorporate all possible object values from *all* possible programs.
4. Nominal models of OOP unfeignedly reflect the properties of programs (that conform to the limitations of the model) written in mainstream OO languages.

7.1.1.1 Comparing Nominal and Structural Views of OOP

When **NOOP** is put in comparison with earlier similar work, it should be noted that Theorem 6.3, about the complete reconciliation of inheritance and subtyping in

NOOP, is similar to Cardelli’s ‘Semantic Subtyping’ theorem in Section 11 of [14]. Because Cardelli does not model recursive types, Cardelli was able to make the identification of an inaccurate notion of inheritance (as syntactic structural subtyping) and semantic subtyping, despite his model being a structural one.

Cook’s model of OOP in [23] and [24], which does model OO recursive types like **SelfType**, does not allow making the identification of inheritance and subtyping, because recursive type variable names get rebound when they get inherited. The reason circular signatures (as the nominal counterpart of structural recursive types) allow making the identification of OO inheritance and OO subtyping is that names of signatures have a *fixed* binding, even when they get inherited.

From a practical point of view, it is worthy to note that the identification of OO inheritance with OO subtyping makes mainstream OOP conceptually simple, due to the parsimony of concepts an OO developer has to deal with. In structural OO languages, not identifying inheritance and subtyping creates significant problems from the perspective of OO program design.

7.2 Incidental Research Contributions

In the course of addressing the issue of how to accurately define the meaning of mainstream OO programs, we generated some new technical machinery in the realm of domain theory and programming language semantics that warrants discussion. The two most interesting pieces of new machinery are

1. A rigorous definition of the finite records domain constructor, \rightarrow , including proofs that the constructor is continuous and computable.
2. An inductively-defined set of class signatures, which can be used to precisely to

tag objects with nominal information of mainstream OOP.

7.3 NOOP Limitations

In the remainder of this chapter, we discuss the limitations of the model presented in this thesis, and how they may be rectified in future work. Simple, clear models necessarily simplify the artifact or phenomenon being modeling. The art in developing a good model is wisely selecting which critical features to capture in the model. Structural models of OO programming languages ignore the nominal information in OO programs which plays a critical role in reasoning about program behavior (because class names are associated with contracts and behavior) and in defining the semantics of type casting, class inheritance, and reflection. Without nominality, models of OO programming produce wrong answers to basic questions about program structure (*e.g.*, subtyping) and program behavior (*e.g.*, casts).

To make our model accessible and amenable to intuitive reasoning, we made it as simple as possible. For example, we forced objects to be immutable (a choice that is almost universally followed in the research literature on modeling the semantics of OO programs) and ignored the complications of generic type systems. We also forced some common restrictions on the typing of inherited class members, which can be relaxed in some situations. We discuss these restrictions and some others in more detail below. We believe that all of these restrictions can be addressed at the cost of complicating the model. We are confident that enhanced models will have the same fundamental structure as the simple model presented in this thesis.

7.3.1 NOOP Models Immutable OOP

While **NOOP** provides a more precise model of mainstream OOP, it is important to observe that **NOOP**, like most models of OOP, does not model mutation. **NOOP** is a set of possible object values; modeling mutation introduces a distinction between objects and the values they can assume via mutation. Ignoring mutation simplifies the task of modeling the data domains of programming languages.

Except for few important places where they do interact, it is well-known among PL researchers that modeling mutation is largely orthogonal to modeling the typing properties of programming languages. For example, **SOOP** [13, 14] does not model mutation. Featherweight Java (FJ/FGJ) [36] does not model mutation either. The same no-mutation limitation applies to most models of OOP and FP. By focusing on the immutable subset of programs of nominal OO languages, we are thus following the footsteps of a long tradition in the semantics of mostly-functional (*i.e.*, with little mutation) languages (*e.g.*, SCHEME and ML) and OO languages. Further, in our presentation of **NOOP**, the few places where mutation and typing interact were clearly noted.

7.3.2 Invariant Subtyping of Method Signatures

In the definition of subsigning, we require signatures of fields and methods in a super-signature to be exactly the same as corresponding ones inherited by a subsignature. As a result, **NOOP** forces inherited members of subclasses to have exactly the same types as the corresponding member in the superclass. We adopted this requirement for simplicity, and because exact matching of method and field types is required (with minor exceptions) in mainstream OO languages like JAVA.

At the expense of complicating the model and the proof of its properties, this rule

can be relaxed to allow full co-variant subtyping of inherited members as in Cardelli's model **SOOP**. This generalization breaks except for the narrowing of method output types (as in JAVA 5/6/7) in the presence of mutation.¹

7.3.3 **NOOP** is The Universe of a Model

Strictly-speaking, **NOOP** is the universe of a model of nominal OOP, not a full model of it. A full model of for a mainstream OOP language requires a meaning function that maps program expressions to values in a semantic domain like **NOOP**. Such a meaning function depends on the details of the particular language being modeled. **NOOP**, which is largely independent of language, provides a universe for defining the meaning of a variety of nominal OO languages.

7.4 Directions for Future Work

This thesis lays the foundations for building more complex models of mainstream OOP that address complications such as data mutation, generic types, and more flexible typing of inherited members. A possible future work that can be built on top of **NOOP** is to define a minimal nominal OO language, in the spirit of FJ [36], then give the denotational semantics of program constructs of this language in **NOOP**. The type safety of this language can then be proven using this denotational semantics. Another possibility is developing a model of nominal OOP that incorporates mutable fields.

Another possible future work that can be based on **NOOP** is to develop a model of

¹It is well-known that a covariant subtyping rule for field types is unsound if fields are mutable. EIFFEL [26] is a language whose development has been affected by this unsoundness. Given that “write-only” fields are of no practical use, a contravariant subtyping rule for field types is of no practical use either.

generic OOP, and prove that OO inheritance and OO subtyping are also completely reconciled in generic OOP. Such a model would be a simple extension of **NOOP**, where class signatures and method signatures would be “generified” to generic class signatures and generic method signatures that are constructed using signature constructors. Such a model of generic OOP may be then used to reason about generic OOP features such as JAVA wildcards and polymorphic methods.

Bibliography

- [1] *ANSI Smalltalk Standard*. 1998.
- [2] C# language specification, version 3.0. <http://msdn.microsoft.com/vcsharp>, 2007.
- [3] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
- [4] Martin Abadi, Benjamin C. Pierce, and Gordon D. Plotkin. Faithful ideal models for recursive polymorphic types. In *IEEE Symposium on Logic in Computer Science*, 1989.
- [5] Samson Abramsky and Achim Jung. Domain theory. In Dov M. Gabbay S. Abramsky and T. S. E. Maibaum, editors, *Handbook for Logic in Computer Science*, volume 3. Clarendon Press, 1994.
- [6] Lloyd Allison. *A Practical Introduction to Denotational Semantics*. Cambridge University Press, 1986.
- [7] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.
- [8] G. Bracha and D. Griswold. Strongtalk: typechecking smalltalk in a production environment. In *OOPSLA '93*, pages 215–230, 1993.
- [9] Joseph Breuer. *Introduction to the Theory of Sets*. Dover Publications, 2006 (first published 1958).

- [10] K. Bruce, A. Schuett, R. van Gent, and A. Fiech. PolyTOIL: A type-safe polymorphic object-oriented language. *ACM Transactions on Programming Languages and Systems*, 25(2):225–290, 2003.
- [11] Kim B. Bruce. A paradigmatic object-oriented programming languages: Design, static typing and semantics. *Journal of Functional Programming*, (4(2)), April 1994.
- [12] Kim B. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, 2002.
- [13] Luca Cardelli. A semantics of multiple inheritance. In *Proc. of the international symposium on Semantics of data types*, volume 173, pages 51–67. Springer-Verlag, 1984.
- [14] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.
- [15] Robert Cartwright. Types as intervals. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '85, pages 22–36. ACM, 1985.
- [16] Robert Cartwright and Alan Demers. The topology of program termination. pages 296–308, 1988.
- [17] Robert Cartwright and Jim Donahue. Lazy data domains, 1992. Earlier version under title ‘The Semantics of Lazy Evaluation’ appeared in Proceedings of the 1982 ACM Conference on LISP and Functional Programming.

- [18] Robert Cartwright and Matthias Felleisen. Observable sequentiality and full abstraction. In *Conference Record 19th ACM Symposium on Principles of Programming Languages*, pages 328–342. ACM, 1992.
- [19] Robert Cartwright and Rebecca Parsons. *Domain theory: An introduction*, 1988. Monograph.
- [20] Alonzo Church. A formulation of the simple theory of types. *JSL*, 5, 1940.
- [21] Robert L. et al Constable. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice Hall, Englewood Cliffs, NJ, 1986.
- [22] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *ACM Symposium on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 433–444, 1989.
- [23] William R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.
- [24] William R. Cook, Walter L. Hill, and Peter S. Cannin. Inheritance is not subtyping. In *POPL'90 Proceedings*, 1990.
- [25] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, first edition, 1990.
- [26] ECMA-367. Eiffel: Analysis, design and programming language. Standard ECMA-367, June 2006.
- [27] Herbert B. Enderton. *Elements of Set Theory*. Academic Press, New York, 1977.

- [28] Marcelo Fiore, Achim Jung, Eugenio Moggi, Peter O’Hearn, Jon Riecke, Giuseppe Rosolini, and Ian Stark. Domains and denotational semantics: History, accomplishments and open problems, Jan 1996.
- [29] K. Fisher and J. Reppy. The design of a class mechanism for Moby. In *PLDI*, 1999.
- [30] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. W. Mislove, and D. S. Scott. *Continuous Lattices and Domains*, volume 93 of *ENCYCLOPEDIA OF MATHEMATICS AND ITS APPLICATIONS*. Cambridge University Press, 2003.
- [31] Michael J. Gordon and Christopher P. Milner, Arthur J. annd Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*. Springer-Verlag, 1978.
- [32] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2005.
- [33] C. A. Gunter and Dana S. Scott. *Handbook of Theoretical Computer Science*, chapter Semantic Domains. 1990.
- [34] Paul R. Halmos. *Naive Set Theory*. D. Van Nostrand Company, Inc., 1960.
- [35] Martin Hilbert and Priscila Lopez. The world’s technological capacity to store, communicate, and compute information. *ScienceXpress*, February 2011.
- [36] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.
- [37] Gilles Kahn and Gordon D. Plotkin. Concrete domains, May 1993.

- [38] Samuel N. Kamin. Inheritance in smalltalk-80: A denotational definition. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 80–87, 1988.
- [39] Samuel N. Kamin and Uday S. Reddy. Two semantic models of object-oriented languages. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics and Language Design*, pages 464–495. MIT Press, 1994.
- [40] Alan C. Kay. The early history of smalltalk. *ACM SIGPLAN Notices*, 28, March 1993.
- [41] Klaus Kreft and Angelika Langer. Understanding the closures debate: Does Java need closures? three proposals compared. *JavaWorld*, June 2008.
- [42] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9:157–166, 1966.
- [43] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system. Available at <http://caml.inria.fr/>.
- [44] D. B. MacQueen and Ravi Sethi. A semantic model of types for applicative languages. In *Proceedings of the 1982 ACM symposium on LISP and functional programming*, LFP '82, pages 243–252. ACM, 1982.
- [45] David B. MacQueen. Should ML be object-oriented? *Formal Aspects of Computing*, 13:214–232, 2002.
- [46] David B. MacQueen, Gordon D. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

- [47] Donna Malayeri and Jonathan Aldrich. Is structural subtyping useful? an empirical study. In *ESOP*, 2009.
- [48] John McCarthy. A basis for a mathematical theory of computation. *Computer Programming and Formal Systems*, pages 33–70, 1963.
- [49] John McCarthy. Towards a mathematical science of computation. *Science*, 1996.
- [50] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1995.
- [51] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
- [52] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
- [53] John C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *ACM Symposium on Principles of Programming Languages (POPL)*, San Francisco, California, pages 109–124, 1990.
- [54] Martin Odersky. The scala language specification, version 2.7. <http://www.scala-lang.org>, 2009.
- [55] Lawrence C Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.
- [56] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- [57] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

- [58] Gordon D. Plotkin. \mathbb{T}^ω as a universal domain. *Journal of Computer and System Sciences*, 17:209–236, 1978.
- [59] Gordon D. Plotkin. Domains. Lecture notes in advanced domain theory, 1983.
- [60] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. *X10 Language Specification: Version 2.2*, May 2011.
- [61] Dana S. Scott. Outline of a mathematical theory of computation. In *4th Annual Princeton Conference on Information Sciences and Systems*, 1970.
- [62] Dana S. Scott. The lattice of flow diagrams. In E. Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 311–366. Springer-Verlag, 1971.
- [63] Dana S. Scott. Data types as lattices. *SIAM Journal of Computing*, 5(3):522–587, 1976.
- [64] Dana S. Scott. Lectures on a mathematical theory of computation. Technical Monograph PRG-19, Oxford University Computing Laboratory, May 1981.
- [65] Dana S. Scott. Domains for denotational semantics. Technical report, Computer Science Department, Carnegie Mellon University, 1983.
- [66] Anthony J. H. Simons. The theory of classification, part 1: Perspectives on type compatibility. *Journal of Object Technology*, 1(1):55–61, May-June 2002.
- [67] Dan Smith and Robert Cartwright. Java type inference is broken: Can we fix it? *OOPSLA*, pages 505–524, 2008.
- [68] M. B. Smyth and Gordon D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11:761–783, 1982.

- [69] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.
- [70] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. In *SAC*, 2004.
- [71] Franklyn Turbak, David Gifford, and with Mark A. Sheldon. *Design Concepts in Programming Languages*. MIT Press, 2008.

Appendix A

Domain Theory

According to Abramsky and Jung [5], domain theory is ‘a mathematical theory that serves as a foundation for the semantics of programming languages’. Domains form the basis of a theory of partial information, which extends the familiar notion of partial function to encompass a whole spectrum of “degrees of definedness”, so as to model *incremental higher-order computation* (*i.e.*, computing with infinite data values). General considerations from recursion theory dictate that partial functions are unavoidable in any discussion of computability. Domain theory provides an appropriately abstract setting in which the notion of a partial function can be lifted and used to give meaning to higher types, recursive types, etc.

In this Appendix, we present the definitions of basic domain theoretic notions and domain constructors we use in this thesis.

A.1 Basic Notions

Domain theory builds on set theory, order theory (the theory of partially-ordered sets, *i.e.*, posets), and topology (*i.e.*, the theory of topological spaces). It is relatively easy to digest the basic definitions of domain theory once the computational motivations behind these definitions are understood.

Standard references on set theory include [9, 27, 34]. A standard reference on

order theory is [25]. Chapter 5 in [71] presents a simple introduction to fixed points particularly suited for mathematically-inclined programmers.

In [30], Gierz, et al, present a detailed encyclopaedic account of domain theory that connects it to order theory and topology. Otherwise, literature on domain theory is somewhat more fractured, and its terminology is somewhat less standard than that of set theory and order theory. Accordingly, there is no standard formulation of domain theory. References to domain theory include [63, 69, 64, 65, 59, 6, 19, 33, 37, 5]. Stoy's book [69] is a particularly detailed account of the motivations behind domain theoretic definitions (even though for domains Stoy, following Scott's original formulation [63], uses complete lattices rather than cpos).

In this section we present definitions of domain theory notions used in constructing **NOOP**. In the next section we focus on the definitions of the domain constructors used in the construction.

Definition A.1 (Partial Order). A *partial order* (also called a partially ordered set, a poset) is a pair $(\mathcal{X}, \sqsubseteq)$ consisting of a set \mathcal{X} (called the *universe* of the ordering), and a binary relation \sqsubseteq on the set \mathcal{X} , such that

- $\forall x \in \mathcal{X}, x \sqsubseteq x$ (\sqsubseteq is reflexive)
- $\forall x, y \in \mathcal{X}, x \sqsubseteq y \wedge y \sqsubseteq x \implies x = y$ (\sqsubseteq is antisymmetric)
- $\forall x, y, z \in \mathcal{X}, x \sqsubseteq y \wedge y \sqsubseteq z \implies x \sqsubseteq z$ (\sqsubseteq is transitive)

\sqsubseteq is usually called 'less than or equals' in when discussing general posets, and 'approximates' in domain theory. A poset $(\mathcal{X}, \sqsubseteq)$ is usually referred to using just the symbol for its universe, \mathcal{X} . We do so below.

Remark A.1. The ordering upon which domain theory is based is called ‘the approximation ordering’. The approximation ordering is defined on computational data values. The approximation ordering has intuitive connections to information theory. A computational value that approximates a second computational value is considered no more informative than the second value. The approximation ordering is a qualitative expression of the relative informational content of computational values (elements of the universe of the ordering). Computational values higher in the approximation ordering are more informative than ones lower in the ordering.

Remark A.2. The least computational value is divergence (as in an “infinite loop”). It gives no information, and thus is the least informative computational value. Given that divergence gives no information, the abstract mathematical value denoting divergence is called ‘bottom’, and is usually denoted by the symbol \perp .

Definition A.2 (Upper bound). Given a subset \mathcal{S} of a poset \mathcal{X} , an *upper bound* of \mathcal{S} (in \mathcal{X}) is an element $x \in \mathcal{X}$ such that $\forall s \in \mathcal{S}, s \sqsubseteq x$.

Definition A.3 (Bounded). A subset \mathcal{S} of a poset \mathcal{X} is *bounded in \mathcal{X}* iff \mathcal{S} has an upper bound in \mathcal{X} .

Definition A.4 (Least Upper Bound). An upper bound of a subset \mathcal{S} in a poset \mathcal{X} is a *least upper bound* (also called a *lub*, or LUB) of \mathcal{S} iff it approximates all upper bounds of \mathcal{S} in \mathcal{X} . If it exists¹, the lub of \mathcal{S} is denoted $\sqcup \mathcal{S}$.

Definition A.5 (Downward-Closed). A subset \mathcal{S} of a poset \mathcal{X} is a *downward-closed* set iff all elements x of \mathcal{X} that approximate some element in \mathcal{S} belong to \mathcal{S} . Thus, \mathcal{S} is downward-closed iff $\forall x \in \mathcal{X}. ((\exists s \in \mathcal{S}, x \sqsubseteq s) \implies x \in \mathcal{S})$.

¹A lub of a subset \mathcal{S} may not exist, either because \mathcal{S} has multiple upper bounds that have no least element (minimum) or because \mathcal{S} has no upper bounds.

Definition A.6 (Chain). A countable subset \mathcal{S} of elements $s_i \in X$ is a chain if $\forall i, j \in \mathbb{N}. i \leq j \rightarrow s_i \leq s_j$.

Remark A.3. Every finite chain includes its lub (maximum element of the chain). Infinite chains (like set \mathbb{N} under the standard ordering) do not necessarily have maximal elements.

Definition A.7 (Anti-chain). A countable subset \mathcal{S} of elements $s_i \in X$ is an anti-chain if $\forall i, j \in \mathbb{N}. i \neq j \rightarrow s_i \not\leq s_j$.

Remark A.4. A flat poset \mathcal{R} is an anti-chain \mathcal{S} with elements s_i and an additional bottom element $\perp_{\mathcal{R}} \neq s_i$, such that, in \mathcal{R} , $\perp_{\mathcal{R}} \leq s_i$. A flat poset, thus, always *lifts* an anti-chain.

Definition A.8 (Directed). A subset \mathcal{S} of a poset \mathcal{X} is *directed* iff every finite subset of \mathcal{S} is bounded in \mathcal{S} .

Remark A.5. Every chain is a directed set, but not necessarily vice versa.

Definition A.9 (Consistent). A subset \mathcal{S} of a poset \mathcal{X} is *consistent in \mathcal{X}* iff every finite subset of \mathcal{S} is bounded in \mathcal{X} .

Remark A.6. In general posets, every bounded set is consistent, but not necessarily vice versa. Consistency requires bounds for finite subsets only, and thus is a weaker condition than boundedness (*all* subsets of a bounded set are bounded).

Remark A.7. Because \mathcal{S} is a subset of \mathcal{X} , boundedness in \mathcal{S} implies boundedness in \mathcal{X} , and thus every directed set \mathcal{S} is a consistent set, but not necessarily vice versa. Directedness is a stronger condition than consistency.

Definition A.10 (Ideal). A subset \mathcal{S} of a poset \mathcal{X} is an *ideal* iff it is downward-closed and directed.

Definition A.11 (Lower set). A subset \mathcal{S}_x of a poset \mathcal{X} is a *lower set* of an element $x \in \mathcal{X}$ iff it contains all elements of \mathcal{X} that are less than or equal to x (and nothing else). Thus, for $x \in \mathcal{X}$, \mathcal{S}_x is the lower set of x iff $\mathcal{S}_x = \{s \in \mathcal{X} \mid s \sqsubseteq x\}$.

Definition A.12 (Principal Ideal). A subset \mathcal{S}_x of a poset \mathcal{X} is a *principal ideal* (determined by x) iff it is the lower set of x .

Theorem A.1. (*Principal Ideals are Ideals*) A subset \mathcal{S} of a poset \mathcal{X} is an ideal if it is a principal ideal.

Proof. Note that, by definition and transitivity of \sqsubseteq , a lower set of an element $x \in \mathcal{X}$ is downward-closed. The lower set of x is also directed because it contains x and x is a bound for all its (finite) subsets. \square

Definition A.13 (Weak Ideal). A non-empty subset \mathcal{S} of a poset \mathcal{X} is a *weak ideal* iff it is downward-closed and is closed under lubs of its chains.

Remark A.8. Every flat poset is a weak ideal. Chains in flat posets have two elements, the lower of which is always \perp .

Definition A.14 (Finitary Basis). A poset \mathcal{X} is a *finitary basis* iff its universe, $|\mathcal{X}|$, is countable and every finite bounded subset \mathcal{S} of \mathcal{X} has a lub in \mathcal{X} .

Remark A.9. For a finitary basis \mathcal{X} , the fact that a finite subset \mathcal{S} of \mathcal{X} is bounded is equivalent to \mathcal{S} having a lub. Generally, this statement is true only in one direction for an arbitrary poset (*i.e.*, the trivial \Leftarrow direction, which asserts the boundedness of a set if it has a lub. In a finitary basis, the opposite direction is true as well).

Definition A.15 (Complete Partial Order). A poset \mathcal{X} is a *complete partial order* (*cpo*, or, sometimes, *dcpo*) iff every directed subset \mathcal{S} of \mathcal{X} has a lub in \mathcal{X} .

Theorem A.2 (Ideals over a FB form a cpo). *Given a finitary basis \mathcal{X} , the set $\mathcal{I}_{\mathcal{X}}$ of ideals of \mathcal{X} is a cpo under the subset ordering.*

Definition A.16 (Constructed Domain). Given a finitary basis \mathcal{X} , the set $\mathcal{I}_{\mathcal{X}}$, of ideals of \mathcal{X} , is a poset $(\mathcal{I}_{\mathcal{X}}, \subseteq)$ that is called the *domain determined by the \mathcal{X}* or the *ideal completion* of \mathcal{X} . $\mathcal{I}_{\mathcal{X}}$ is, thus, called a *constructed domain* (*i.e.*, one that is determined by the finitary basis \mathcal{X}).

Remark A.10. By Theorem A.2, the ideal completion of a finitary basis is a cpo.

Definition A.17 (Finite Element of a CPO). An element d of a cpo \mathcal{D} is a *finite element* (or *isolated* or *compact*) iff d belongs to each directed subset \mathcal{S} that d is a lub of. The set of finite elements of a cpo \mathcal{D} is denoted by \mathcal{D}^0 .²

Definition A.18 (Isomorphic Partial Orders). Two posets are *isomorphic* iff there is an order-preserving one-to-one onto function between them.

Definition A.19 (Domain). A cpo \mathcal{D} is a *domain* iff its finite elements \mathcal{D}^0 form a finitary basis and \mathcal{D} is isomorphic to the domain determined by \mathcal{D}^0 .

Definition A.20 (Subdomain). A domain \mathcal{D} is a *subdomain* of a domain \mathcal{E} iff (1) $|\mathcal{D}| \subseteq |\mathcal{E}|$, (2) $\perp_{\mathcal{D}} = \perp_{\mathcal{E}}$, (3) $\forall d_1, d_2 \in \mathcal{D}, d_1 \sqsubseteq_{\mathcal{D}} d_2 \Leftrightarrow d_1 \sqsubseteq_{\mathcal{E}} d_2$ (*i.e.*, approximation ordering for \mathcal{D} is the approximation ordering of \mathcal{E} restricted to elements of \mathcal{D}), and (4) $\forall d_1, d_2, d_3 \in \mathcal{D}, (d_1 \sqcup_{\mathcal{D}} d_2 = d_3) \Leftrightarrow (d_1 \sqcup_{\mathcal{E}} d_2 = d_3)$ (*i.e.*, lub relation for \mathcal{D} is the lub relation of \mathcal{E} restricted to elements of \mathcal{D}).

Remark A.11. The domain determined by \mathcal{D}^0 is isomorphic to the domain determined by $\mathcal{E}^0 \cap \mathcal{D}$, which must be a finitary basis.

²This definition is weaker than the usual definition for cpos. In the context of domains, the two definitions are equivalent.

Remark A.12. We use Scott’s definition of subdomains because we define **NOOP** domains as subdomains of Scott’s universal domain \mathcal{U} . Scott [19] shows that every domain is isomorphic to a subdomain of \mathcal{U} . The subdomains of \mathcal{U} form a domain. All domains given in a domain equation and all recursively defined domains in the equation are elements of this space of domains (which consists of all of the subdomains of \mathcal{U}). Thus, fixed-points for domains (as elements of the domain of subdomains of \mathcal{U}) are defined in the same way as fixed-points of recursive definitions over the elements of any other domain.

A.2 Domains of Functions

To model computable functions, domain theory provides functional domains, whose elements are particular mathematical functions mapping elements from one computational domain to another. To define functional domains, we will introduce the domain theoretic notions of ‘approximable mappings’ (AMs), ‘finite-step mapping’, and ‘continuous functions’.

Given the importance of these three notions, we only introduce them in this section, then we discuss them in more detail in Section A.3 when we present domain constructors that make use of these three notions. These domain constructors are used to construct the functional domains of **NOOP**.

Definition A.21 (Approximable Mapping). Given two finitary basis A and B , with ordering relations \sqsubseteq_A and \sqsubseteq_B , respectively, a relation $f_{am} \subseteq |A| \times |B|$ is an *approximable mapping* (AM) iff

1. Condition 1: $(\perp_A, \perp_B) \in f_{am}$
2. Condition 2: $\forall a \in A. \forall b_1, b_2 \in B. ((a, b_2) \in f_{am} \wedge b_1 \sqsubseteq_B b_2 \rightarrow (a, b_1) \in f_{am})$

3. Condition 3: $\forall a \in A. \forall b_1, b_2 \in B. ((a, b_1) \in f_{am} \wedge (a, b_2) \in f_{am} \rightarrow (a, b_1 \sqcup_B b_2) \in f_{am})$
4. Condition 4: $\forall a_1, a_2 \in A. \forall b \in B. ((a_1, b) \in f_{am} \wedge a_1 \sqsubseteq_A a_2 \rightarrow (a_2, b) \in f_{am})$

Definition A.22 (Set Image under a Relation). Given sets A , B and a relation $r \subseteq A \times B$, the *set image* of a subset \mathcal{S} of A under r , denoted by $r(\mathcal{S})$, is the set of all $b \in B$ related in r to some element in \mathcal{S} . r is thus viewed as function over subsets of A . Thus, for a subset \mathcal{S} of A , we have $r(\mathcal{S}) = \{b \in B \mid \exists a \in \mathcal{S}. (a, b) \in r\}$. The set image of a relation r allows viewing r as a function $r : A \rightarrow \wp(B)$, where $r(a) = r(\{a\})$ for $a \in A$. For an element $a \in A$, function r , thus, returns the set of all $b \in B$ related to a in r .

Theorem A.3 (AMs map ideals to ideals). *Given finitary basis A and B , if f_{am} is an approximable mapping from A to B , and if I is an ideal in A , then $f_{am}(I)$, the set image of I under f_{am} , is an ideal in B .*

Proof. From the definition of an ideal, and using AM Condition 2 (which guarantees the downward-closure of the set image), and AM Condition 3 (which guarantees that the set image is directed). □

Theorem A.4 (AMs are monotonic). *Given finitary basis A and B , if f_{am} is an approximable mapping from A to B , and if I_1 and I_2 are ideals in A such that $I_1 \subseteq I_2$, then $f_{am}(I_1) \subseteq f_{am}(I_2)$ in B .*

Proof. By AM Condition 4. □

Definition A.23 (Finite-Step Mapping). Given finitary basis A and B , an approximable mapping f_{am} is a *finite-step mapping* iff it is the smallest approximable mapping containing some finite subset of $|A| \times |B|$.

Definition A.24 (Continuous Function). Given domains \mathcal{A} and \mathcal{B} , a function $f : \mathcal{A} \rightarrow \mathcal{B}$ is a *continuous function* iff the value of f at the lub of a directed set of a 's in \mathcal{A} is the lub, in \mathcal{B} , of the directed set of function values $f(a)$.

Remark A.13. Continuity of a function requires the value of the function at a (non-finite) limit point a to equal the limit of the function's values on the finite approximations to a . Continuous functions are thus said to “have no surprises at the limit”.

Remark A.14. Because of the four AM conditions, every approximable mapping in $|A| \times |B|$ determines a continuous function in $\mathcal{A} \rightarrow \mathcal{B}$, and vice versa. Check Cartwright and Parsons' “Domain Theory: An Introduction” [19] and other domain theory literature for proof and more details.

Remark A.15. To motivate the preceding definitions, it should be noted that continuous functions capture the fact that computation is of a “finitely-based” nature. Only finite data values can have canonical representations inside a computing device. From a domain-theoretic perspective, a function can be computable only if its value “at infinity” (*i.e.*, at an infinite input data value) is the one we expect by only seeing (and extrapolating from) the values of the function at all finite inputs that approximate the infinite input data value (finite inputs are all we can represent inside computers, and thus they are all we can compute with). See Stoy's book [69] for more details on motivation and intuitions behind domain theoretic definitions.³

Remark A.16. Approximable mappings offer the means to accurately characterize and define continuous functions (which capture the finitely-based nature of computation). Finite-step mappings, as the “finite/representable parts” of AMs, offer the means by which continuous functions can be constructed from more elementary parts that can be represented in a computing device.

³Via Roger's work, Dana Scott managed to connect the notion of continuous functions to the notion of computable functions in computability theory. See Stoy's book [69] for more details.

A.3 Domain Constructors

In this section we present the domain constructors used to define **COOP** and **NOOP**. Since there is a one-to-one correspondence between domains and their finitary bases, and given that the latter are simpler, and more intuitive, we will actually be describing in this section how each of these “domain” constructors construct and define new finitary basis using other finitary basis defined earlier.

A.3.1 Coalesced Sum (+)

The first domain constructor we present is the *coalesced sum* domain constructor, $+$. We use the expression $\mathcal{A} + \mathcal{B}$ to denote the coalesced sum of two domains \mathcal{A} and \mathcal{B} , with approximation ordering relations $\sqsubseteq_{\mathcal{A}}$ and $\sqsubseteq_{\mathcal{B}}$, respectively. A coalesced sum is an domain-theoretic counterpart of the standard set-theoretic disjoint union operation.

If $\mathcal{C} = \mathcal{A} + \mathcal{B}$ then

$$|\mathcal{C}| = \{\perp_{\mathcal{C}}\} \cup \{(0, a) | a \in (|\mathcal{A}| \setminus \{\perp_{\mathcal{A}}\})\} \cup \{(1, b) | b \in (|\mathcal{B}| \setminus \{\perp_{\mathcal{B}}\})\}$$

where 0 and 1 are used in \mathcal{C} to tag non-bottom elements from \mathcal{A} and \mathcal{B} , respectively.

The ordering relation $\sqsubseteq_{\mathcal{C}}$, on elements of \mathcal{C} , is defined, for all $c_1, c_2 \in \mathcal{C}$, by the predicate

$$\begin{aligned} c_1 \sqsubseteq_{\mathcal{C}} c_2 \Leftrightarrow & (c_1 = \perp_{\mathcal{C}}) \vee (c_1 = (0, a_1) \wedge c_2 = (0, a_2) \wedge a_1 \sqsubseteq_{\mathcal{A}} a_2) \\ & \vee (c_1 = (1, b_1) \wedge c_2 = (1, b_2) \wedge b_1 \sqsubseteq_{\mathcal{B}} b_2) \end{aligned} \quad (\text{A.1})$$

A.3.2 Strict Product (\times)

Next, we present the *strict product* domain constructor, \times .⁴ We use $\mathcal{A} \times \mathcal{B}$ to denote the strict product of two domains, \mathcal{A} and \mathcal{B} , with approximation ordering relations $\sqsubseteq_{\mathcal{A}}$ and $\sqsubseteq_{\mathcal{B}}$, respectively. A strict product is an order-theoretic counterpart of the standard set-theoretic cross-product operation.

If $\mathcal{C} = \mathcal{A} \times \mathcal{B}$ then

$$|\mathcal{C}| = (|\mathcal{A}| \setminus \{\perp_{\mathcal{A}}\}) \times (|\mathcal{B}| \setminus \{\perp_{\mathcal{B}}\}) \cup \{\perp_{\mathcal{C}}\} \quad (\text{A.2})$$

Strictness of \times means that in \mathcal{C} , $\perp_{\mathcal{C}}$ replaces all pairs $(a, b) \in \mathcal{A} \times \mathcal{B}$ where $a = \perp_{\mathcal{A}}$ or $b = \perp_{\mathcal{B}}$. Similar to the definition of the coalesced sum constructor, this strictness is achieved in the definition above by excluding $\perp_{\mathcal{A}}$ and $\perp_{\mathcal{B}}$ from the input sets of the set-theoretic cross product. Sometimes the strict product $\mathcal{A} \times \mathcal{B}$ is called their ‘smash product’.

The ordering relation $\sqsubseteq_{\mathcal{C}}$, on elements of \mathcal{C} , is defined as follows. $\forall c_1, c_2 \in \mathcal{C}, \forall a_1, a_2 \in \mathcal{A} \setminus \{\perp_{\mathcal{A}}\}, \forall b_1, b_2 \in \mathcal{B} \setminus \{\perp_{\mathcal{B}}\}$ where $c_1 = (a_1, b_1)$ or $c_1 = \perp_{\mathcal{C}}$, and $c_2 = (a_2, b_2)$ or $c_2 = \perp_{\mathcal{C}}$

$$c_1 \sqsubseteq_{\mathcal{C}} c_2 \Leftrightarrow (c_1 = \perp_{\mathcal{C}} \vee (a_1 \sqsubseteq_{\mathcal{A}} a_2 \wedge b_1 \sqsubseteq_{\mathcal{B}} b_2)). \quad (\text{A.3})$$

⁴In agreement with the convention in domain theoretic literature, the symbol \times is overloaded in this thesis: it is used to denote the strict product of ordered sets, and it is also used to denote the standard set-theoretic cross product (which ignores any ordering on its input sets). It should always be clear from context which meaning is attributed to \times .

A.3.3 Continuous Functions (\rightarrow)

Functional domains of **NOOP** are the domain \mathcal{M} , whose members are strict continuous functions modeling object methods, and the domain \mathcal{R} , whose members are ‘record functions’ modeling the record component of objects. A record function is a function defined over a finite set of labels. Functional domains and functional domain constructors allow **NOOP** to model nominal OOP more accurately.

Making use of the definitions of domain theoretic notions presented in Section A.2, for the details of the definitions of the functional domain constructors we now refer the reader to Chapter 3 of the Cartwright and Parsons’ update [19] of Scott’s [64]. \rightarrow is the standard continuous functions domain constructor.

In this thesis, we use the symbol \multimap to denote the strict continuous functions domain constructor, which simply constructs a space as the continuous function space from domain \mathcal{A} to domain \mathcal{B} but where all one-step functions of the form $\perp_{\mathcal{A}} \mapsto b$ (for $b \in \mathcal{B} \setminus \{\perp_{\mathcal{B}}\}$) are eliminated (are mapped to the one-step function $\perp_{\mathcal{A}} \mapsto \perp_{\mathcal{B}}$ which is the bottom element of the constructed function space). Strict continuous functions thus map $\perp_{\mathcal{A}}$ only to $\perp_{\mathcal{B}}$, thus modeling strict computable functions (*i.e.*, ones which have “call-by-value” semantics).

A notable property of functional domain constructors is that the set of continuous functions is itself a domain. This property has been behind much of the development of domain theory.

A.3.4 Strict Finite Sequences (\mathcal{D}^*)

For purposes of constructing the methods of **NOOP**, one more simple domain constructor is needed: the constructor of the domain of strict finite sequences. This constructor is used to construct the finite sequences of objects passed as arguments

to object methods. Sometimes the domain \mathcal{D}^* of finite sequences of elements of domain \mathcal{D} is called the Kleene closure of domain \mathcal{D} .

The Kleene closure, \mathcal{D}^* , constructs a domain of finite sequences of elements of its input domain, \mathcal{D} , including the empty sequence. It excludes sequences of \mathcal{D} where a member of the sequence is $\perp_{\mathcal{D}}$. Thus, $*$ constructs *strict* finite sequences.

The Kleene closure is defined as a set of all n -tuples of elements of \mathcal{D} (where n is a natural number). Thus

$$|\mathcal{D}^*| = \{\perp_{\mathcal{D}^*}\} \cup \bigcup_{n \in \mathbb{N}} \{ \langle d_0, \dots, d_i, \dots, d_{n-1} \rangle \mid d_i \in (|\mathcal{D}| \setminus \{\perp_{\mathcal{D}}\}) \}$$

An element a of \mathcal{D}^* approximates another element b iff $a = \perp_{\mathcal{D}^*}$ or the lengths of a and b are equal to a natural number k , and $a_i \sqsubseteq_{\mathcal{D}} b_i$ for all $0 \leq i < k$.

Appendix B

Proofs of Important Theorems

B.1 The Domain of Record Functions has an Effective Presentation

It is straightforward to confirm that \dashv constructs a domain. To prove that \dashv constructs domains given an arbitrary domain \mathcal{D} and a domain \mathcal{L} (with a fixed interpretation as a flat domain of labels), we build an effective presentation of the finite elements of $\mathcal{L} \dashv \mathcal{D}$, assuming an effective presentation of the finite elements of \mathcal{D} and \mathcal{L} . Since \mathcal{L} has a fixed interpretation, \dashv , as a domain constructor, can be considered as being parametrized only by domain \mathcal{D} . We prove that these finite elements form a finitary basis of the records domain.

Given an effective presentation of \mathcal{L} , $L = [\perp_{\mathcal{L}}, l_1, l_2, \dots]$ define, for all $n \in \mathbb{N}$, the finite sequences

$$L_n = [l_{j_1}, \dots, l_{j_k}]$$

where $0 < j_1 < \dots < j_k$, and

$$2n = \sum_{0 < i \leq k} 2^{j_i}. \tag{B.1}$$

The size k , of L_n , is the number of ones in the binary expansion of n , and thus $k \leq \log_2(n + 1)$ with equality only when n is one less than a power of 2. $k = 0$ only when $n = 0$, and in this case $L_0 = []$ (the empty sequence)¹. It is easy to confirm

¹The definition of L_n is patterned after a similar construction presented in Dana Scott's "Data

that there is a one-to-one correspondence between the set of natural numbers \mathbb{N} and the set of distinct finite label sequences L_n .

Given an effective presentation of the finite elements of \mathcal{D} , $D = [\perp_{\mathcal{D}}, d_1, d_2, \dots]$, an effective presentation of the finite elements of \mathcal{D}^k , the domain of (non-strict) sequences of length k ($k \geq 0$) of elements of \mathcal{D} , is

$$(\mathcal{D}^k)_{\pi^k(n_1, n_2, \dots, n_k)} = [d_{n_1}, \dots, d_{n_k}]$$

where, for $k > 2$,

$$\pi^k(n_1, n_2, \dots, n_k) = \pi(\pi^{k-1}(n_1, \dots, n_{k-1}), n_k)$$

$\pi^k(\cdot)$ is the one-to-one k -tupling function (also called the Cantor tupling function), and

$$\pi(p, q) = \frac{1}{2}(p+q)(p+q+1) + q = \pi^2(p, q)$$

is the one-to-one Cantor pairing function.

Now, let

$$f(n, m) = \{(\perp_{\mathcal{L}}, \perp_{\mathcal{D}})\} \cup zip(L_n, (\mathcal{D}^k)_m)$$

where, again, k is the number of ones in the binary expansion of n , and

$$zip([l_{j_1}, \dots, l_{j_k}], [d_{n_1}, \dots, d_{n_k}]) = \{(l_{j_1}, d_{n_1}), \dots, (l_{j_k}, d_{n_k})\}.$$

The sequence $\mathbf{R} = [r_0, r_1, \dots]$ of the finite elements of \mathcal{R} can then be presented as

Types as Lattices" [63]. Unlike the case in Scott's construction, n is doubled in the LHS of Equation (B.1)—*i.e.*, the binary expansion of n is "shifted left" by one position—to guarantee $j_i > 0$, and thus guarantee that $l_0 = \perp_{\mathcal{L}}$ is never an element of L_n .

$r_0 = \perp_{\mathcal{R}}$, and for $n, m \geq 0$,

$$r_{\pi(n,m)+1} = (\text{tag}(L_n), f(n, m)).$$

Given the decidability of the consistency ($\cdot \uparrow_{\mathcal{D}} \cdot$) and lub ($\cdot \sqcup_{\mathcal{D}} \cdot = \cdot$) relations for finite elements of \mathcal{D} , the presentation \mathbf{R} of the finite elements of \mathcal{R} is effective, since, for record functions r and r' as defined in Section 4.1.2, under the approximation ordering defined by Equation 4.3, the consistency relation

$$r \uparrow_{\mathcal{R}} r' \Leftrightarrow \forall_{i \leq k} (d_i \uparrow_{\mathcal{D}} d'_i) \quad (\text{B.2})$$

is decidable (given the finiteness of records), and the lub relation

$$r \sqcup_{\mathcal{R}} r' = (\text{tag}(\{l_1, \dots, l_k\}), \{(\perp_{\mathcal{L}}, \perp_{\mathcal{D}}), (l_1, d_1 \sqcup_{\mathcal{D}} d'_1), \dots, (l_k, d_k \sqcup_{\mathcal{D}} d'_k)\}) \quad (\text{B.3})$$

is recursive (handling $r = \perp_{\mathcal{R}}$ or $r' = \perp_{\mathcal{R}}$ in the definitions of $\uparrow_{\mathcal{R}}$ and $\sqcup_{\mathcal{R}}$ is obvious. All record functions are consistent with $\perp_{\mathcal{R}}$, and the lub of a record function r and $\perp_{\mathcal{R}}$ is r).

Lemma B.1 (\dashv constructs domains). *Under $\sqsubseteq_{\mathcal{R}}$, elements of \mathbf{R} form a finitary basis of \mathcal{R} .*

Proof. Given the countability of \mathcal{L} and of the finite elements of \mathcal{D} , element of \mathbf{R} are countable. A consistent pair of elements $r, r' \in \mathbf{R}$, according to Equation (B.2), has a lub $r \sqcup_{\mathcal{R}} r'$ defined by Equation (B.3). Given that \mathcal{D} is a domain, the lub $d \sqcup_{\mathcal{D}} d'$ of all consistent pairs of finite elements d, d' in \mathcal{D} exists, thus the lub $r \sqcup_{\mathcal{R}} r'$ also exists.

Lemma B.1 actually proves that \dashv is a computable function mapping flat domains \times domains to the corresponding record domains. The presumption is that no effective

presentation is necessary for the flat domain because distinct indices for elements of \mathcal{L} will simply mean distinct labels l_i . If \mathcal{L} is a flat countably infinite domain (which implies it has an effective presentation) and \mathcal{D} is an arbitrary domain, then $\mathcal{L} \multimap \mathcal{D}$ is a domain with an effective presentation that is constructible from the effective presentations for \mathcal{L} and \mathcal{D} . \square

B.2 Domain Constructor \multimap is Continuous

Lemma B.2 (\multimap is monotonic). *For domains \mathcal{D} and \mathcal{D}' , and a flat domain of labels \mathcal{L} , $\mathcal{D} \subseteq \mathcal{D}' \Rightarrow (\mathcal{L} \multimap \mathcal{D}) \subseteq (\mathcal{L} \multimap \mathcal{D}')$*

Proof. First, we prove that \multimap is monotonic with respect to the subset relation on the universe of its input, *i.e.*, that $|\mathcal{D}| \subseteq |\mathcal{D}'| \Rightarrow |\mathcal{L} \multimap \mathcal{D}| \subseteq |\mathcal{L} \multimap \mathcal{D}'|$. Then, given that the approximation ordering on \mathcal{D} (as a subdomain of \mathcal{D}') is the restriction of the approximation ordering on \mathcal{D}' , we prove that the elements of $\mathcal{L} \multimap \mathcal{D}$ (as members of $\mathcal{L} \multimap \mathcal{D}'$) form a domain under the approximation ordering of $\mathcal{L} \multimap \mathcal{D}'$, and thus that $\mathcal{L} \multimap \mathcal{D}$ is a subdomain of $\mathcal{L} \multimap \mathcal{D}'$.

Since $|\mathcal{D}| \subseteq |\mathcal{D}'|$, then $\{d_1, \dots, d_k\} \subseteq |\mathcal{D}| \Rightarrow \{d_1, \dots, d_k\} \subseteq |\mathcal{D}'|$. For arbitrary \mathcal{L}_f where $|\mathcal{L}_f| = \{\perp_{\mathcal{L}}, l_1, \dots, l_k\}$, we thus have

$$f = \{(\perp_{\mathcal{L}}, \perp_{\mathcal{D}}), (l_1, d_1), \dots, (l_k, d_k)\} \in |\mathcal{L}_f \multimap \mathcal{D}| \Rightarrow f \in |\mathcal{L}_f \multimap \mathcal{D}'|.$$

Thus, $|\mathcal{L}_f \multimap \mathcal{D}| \subseteq |\mathcal{L}_f \multimap \mathcal{D}'|$. Accordingly, for sets $R(\mathcal{L}_f, \mathcal{D})$ (the elements of $\mathcal{L} \multimap \mathcal{D}$ with tag $\text{tag}(\mathcal{L}_f \setminus \{\perp_{\mathcal{L}}\})$) and $R(\mathcal{L}_f, \mathcal{D}')$ (the elements of $\mathcal{L} \multimap \mathcal{D}'$ with tag $\text{tag}(\mathcal{L}_f \setminus \{\perp_{\mathcal{L}}\})$), as defined in Equation 4.2 of Section 4.1.2, we have $R(\mathcal{L}_f, \mathcal{D}) \subseteq$

$R(\mathcal{L}_f, \mathcal{D}')$. Thus,

$$\left(\{\perp_{\mathcal{R}}\} \cup \bigcup_{\mathcal{L}_f \in \mathcal{L}} R(\mathcal{L}_f, \mathcal{D}) \right) \subseteq \left(\{\perp_{\mathcal{R}}\} \cup \bigcup_{\mathcal{L}_f \in \mathcal{L}} R(\mathcal{L}_f, \mathcal{D}') \right).$$

Thus,

$$|\mathcal{L} \multimap \mathcal{D}| \subseteq |\mathcal{L} \multimap \mathcal{D}'|. \quad (\text{B.4})$$

Next, since \mathcal{D} is a subdomain of \mathcal{D}' when restricted to elements of \mathcal{D} , we know:

(i) the approximation relation on \mathcal{D} is the approximation relation on \mathcal{D}' restricted to \mathcal{D} ; (ii) consistent pairs of \mathcal{D} are consistent pairs in \mathcal{D}' ; and (iii) lubs, in \mathcal{D} , of consistent pairs of elements of \mathcal{D} are also their lubs in \mathcal{D}' . Thus, for $d_i, d_j \in \mathcal{D}$, $d_i \sqsubseteq_{\mathcal{D}} d_j \Leftrightarrow d_i \sqsubseteq_{\mathcal{D}'} d_j$, $d_i \uparrow_{\mathcal{D}} d_j \Leftrightarrow d_i \uparrow_{\mathcal{D}'} d_j$ and $d_i \sqcup_{\mathcal{D}} d_j = d_i \sqcup_{\mathcal{D}'} d_j$.

Hence, according to the definition of the approximation, consistency and lub relations for \multimap (Equations (4.3), (B.2) and (B.3)), the lub, in $\mathcal{L} \multimap \mathcal{D}$, of a consistent pair of records is also their lub in $\mathcal{L} \multimap \mathcal{D}'$. That is, respectively, for $r, r' \in |\mathcal{L} \multimap \mathcal{D}|$, we have

$$r \sqsubseteq_{(\mathcal{L} \multimap \mathcal{D})} r' \Leftrightarrow r \sqsubseteq_{(\mathcal{L} \multimap \mathcal{D}')} r', \quad (\text{B.5})$$

$$r \uparrow_{(\mathcal{L} \multimap \mathcal{D})} r' \Leftrightarrow r \uparrow_{(\mathcal{L} \multimap \mathcal{D}')} r' \quad (\text{B.6})$$

and

$$r \sqcup_{(\mathcal{L} \multimap \mathcal{D})} r' = r \sqcup_{(\mathcal{L} \multimap \mathcal{D}')} r'. \quad (\text{B.7})$$

From equations (B.4), (B.5), (B.6), (B.7), and the fact that $\perp_{\mathcal{R}}$ is the bottom element of both $\mathcal{L} \multimap \mathcal{D}$ and $\mathcal{L} \multimap \mathcal{D}'$, we can conclude using Definition 6.2 in [19] that

$$\mathcal{L} \multimap \mathcal{D} \in \mathcal{L} \multimap \mathcal{D}'.$$

□

In addition to being monotonic, continuity of a domain constructor asserts that the lub of domains it constructs using a chain of input domains is the domain it constructs using the lub of the chain of input domains (*i.e.*, that, for \multimap , the lub \mathcal{D} of a chain of input domains \mathcal{D}_i gets mapped by \multimap to the lub, say domain \mathcal{R} , of the chain of output domains $\mathcal{R}_i = \mathcal{L} \multimap \mathcal{D}_i$).

Lemma B.3 (\multimap preserves lubs.). *For a chain of domains \mathcal{D}_i , if $\mathcal{D} = \sqcup \mathcal{D}_i$, $\mathcal{R}_i = \mathcal{L} \multimap \mathcal{D}_i$, and $\mathcal{R} = \mathcal{L} \multimap \mathcal{D}$, then $\mathcal{R} = \sqcup \mathcal{R}_i$.*

Proof. Let \mathcal{Q} be the lub of the chain of domains $\mathcal{R}_i = \mathcal{L} \multimap \mathcal{D}_i$ (\mathcal{R}_i 's form a chain by the monotonicity of \multimap). Domain \mathcal{Q} is thus the union of domains \mathcal{R}_i , *i.e.*, $\mathcal{Q} = \sqcup \mathcal{R}_i = \cup_i (\mathcal{L} \multimap \mathcal{D}_i)$.

Domain \mathcal{Q} is equal to $\mathcal{R} = \mathcal{L} \multimap \mathcal{D} = \mathcal{L} \multimap \cup_i \mathcal{D}_i$ because each element q in \mathcal{Q} (q is a record function) is an element of a domain $\mathcal{L} \multimap \mathcal{D}_i$ for some i . Given \mathcal{D}_i is a subset of $\mathcal{D} = \cup_i \mathcal{D}_i$, q will also appear in \mathcal{R} .

Similarly, a record function r in \mathcal{R} is an element of a domain $\mathcal{L} \multimap \mathcal{D}_i$ for some i , because every *finite* subset of $\cup_i \mathcal{D}_i$ has to appear in *one* \mathcal{D}_i (given that \mathcal{D}_i is a chain of domains). Thus, by the definition of \mathcal{Q} , r is also a member of \mathcal{Q} .

This proves that $\mathcal{Q} = \mathcal{R}$. □

Lemmas B.2 and B.3 prove that \multimap is computable given effective presentations for \mathcal{L} and \mathcal{D} (or, equivalently, an effective presentation for \mathcal{D}).

B.3 Filtering is a Finitary Projection

In this section we prove that function `filter`, as defined in Section 6.3.1, is indeed a finitary projection, and thus that the domain \mathcal{O} of valid objects (see Definition 6.1

in Section 6.3) defined by the filtering function is indeed a domain (*i.e.*, a subdomain of Scott’s universal domain \mathcal{U}).

To do so, we first prove a number of auxiliary properties.

Proposition B.1. *In domain $\hat{\mathcal{O}}$, higher-ranked objects do not approximate lower-ranked ones, i.e., $\text{rank}(o_1) < \text{rank}(o_2) \implies o_2 \not\sqsubseteq o_1$*

Proof. By strong induction on rank of objects. □

Proposition B.2. *In domain $\hat{\mathcal{O}}$, all approximations of a valid object are valid objects.*

Proof. We outline a proof that uses induction on the rank of valid objects. Objects of rank 1 with non-empty member records have $\perp_{\mathcal{O}}$ as the object bound to their fields. Thus invalid rank 1 objects are invalid *only* because of have non-matching signature shapes and record shapes.

None of the objects of rank 1 approximate each other (a rank 1 object is approximated only by $\perp_{\mathcal{O}}$ and itself). The mismatch between signatures and shapes of invalid objects in objects of rank 1 that causes them to be invalid cannot be made valid (get “fixed”) in objects (of higher rank) that are approximated by the invalid rank 1 objects (since approximation dictates that those approximated objects have the *same* signatures and shapes as the invalid ones approximating them).

For higher ranks, objects of a higher rank that valid rank 1 objects (that have matching signature and record shapes) approximate but that have embedded in their records objects with signatures that do not subsign the signature in the corresponding field signature of (which is the same in all these objects of ranks starting 1 and higher) make the approximated objects be invalid. Given that signatures form a flat domain, the signature of objects embedded inside objects that approximate each other is the same in all objects that approximate each other, so objects that are approximated

by such higher-ranked invalid objects are also invalid ones. An object in $\hat{\mathcal{O}}$ that is approximated by an invalid object is, thus, guaranteed to be an invalid object. \square

Proposition B.3. *In domain $\hat{\mathcal{O}}$, no two distinct valid objects that do not approximate each other may approximate the same invalid object.*

Proof. By strong induction on rank of objects, making use of Proposition B.1. \square

In the sequel, we use the inductively defined predicate **valid** (as defined by Definition 6.1 in Section 6.3) that applies to objects of $\hat{\mathcal{O}}$. Note that in addition to $\perp_{\mathcal{O}}$, objects with empty field and method records also provide base cases for the definition of **valid**.

Lemma B.4 (**filter** returns valid object closest to input object). *For an object o of $\hat{\mathcal{O}}$, $\mathbf{filter}(o) \sqsubseteq o \wedge \mathbf{valid}(\mathbf{filter}(o)) \wedge \forall o' (o' \sqsubseteq o \wedge \mathbf{valid}(o') \implies o' \sqsubseteq \mathbf{filter}(o))$*

Proof. By strong induction on rank of objects, noting that, for the base case, **filter**(o) diverges (“returns” $\perp_{\mathcal{O}}$) for the rank 0 object $\perp_{\mathcal{O}}$, and if an object o of rank 1 is invalid then **filter**(o) also returns $\perp_{\mathcal{O}}$ (No distinct objects of rank 1 approximate each other). Proposition B.1 is used for the inductive case. \square

Theorem B.1. ***filter** is a finitary projection.*

Proof. We prove that **filter** is a finitary projection, on four steps. \square

1. **filter** is a retraction: $\mathbf{filter}(\mathbf{filter}(o)) = \mathbf{filter}(o)$

Proof. Obvious from definition of **filter**, and that, by Lemma B.4, function **filter** returns a valid object (*i.e.*, $\mathbf{valid}(\mathbf{filter}(o))$). \square

2. **filter** approximates identity: $\mathbf{filter}(o) \sqsubseteq o$

Proof. Obvious from definition of **filter**, and that, by Lemma B.4, $\mathbf{filter}(o)$ returns an object that approximates o (*i.e.*, $\mathbf{filter}(o) \sqsubseteq o$). \square

3. **filter** is a continuous function (*i.e.*, is monotonic, and preserves lubs of chains²)

Proof. By cases, where we assume o_1 and o_2 are two finite elements in a chain in \hat{O} where $o_1 \sqsubseteq o_2$, and each case show that $\mathbf{filter}(o_1) \sqsubseteq \mathbf{filter}(o_2)$ (monotonicity), and that $\mathbf{filter}(o_2) = \sqcup\{\mathbf{filter}(o_1), \mathbf{filter}(o_2)\}$ (lubs preserved).

Case 1. If o_1, o_2 are both valid

- (a) Monotonicity: Immediate.
- (b) LUB: o_2 .

Case 2. If o_1, o_2 are both invalid

- (a) Monotonicity: Simple (By cases, and function **filter** returning closest valid object).
- (b) LUB: Let $v_1 = \mathbf{filter}(o_1)$, $v_2 = \mathbf{filter}(o_2)$. Then, by Proposition B.3 we have $v_2 \sqsubseteq o_1$ (since $o_1 \sqsubseteq v_2$ is impossible, by Proposition B.2). Thus, we have $v_1 = v_2$, and thus $\sqcup\{\mathbf{filter}(o_2), \mathbf{filter}(o_1)\} = \mathbf{filter}(o_2)$.

Case 3. If o_1 is valid, and o_2 is invalid

- (a) Monotonicity: Simple (by function **filter** returning closest valid object).

²Preserving lubs of directed-sets could be used, but, in the context of finitarily-based domains (which have a countable basis), the two definitions of continuity are equivalent.

(b) LUB: We have $o_1 = \mathbf{filter}(o_1) \wedge o_1 \sqsubseteq \mathbf{filter}(o_2)$. Thus, $\sqcup\{\mathbf{filter}(o_2), \mathbf{filter}(o_1)\} = \sqcup\{\mathbf{filter}(o_2), o_1\} = \mathbf{filter}(o_2)$

Case 4. If o_1 is invalid, and o_2 is valid

Proof. Impossible, by Proposition B.2. □

□

4. **filter** is finitary

Proof. The condition in point 2 of Theorem 8.5 in [19]

$$a(x) = \{y \in \mathcal{O} \mid \exists x' \in x. x' a x' \wedge y \sqsubseteq x'\}$$

can be rewritten, for the filtering function **filter**, as

$$\mathbf{filter}(o) = \{p \in \mathcal{O} \mid \exists o' \in \mathcal{O}. o' \sqsubseteq o \wedge o' = \mathbf{filter}(o') \wedge p \sqsubseteq o'\}. \quad (\text{B.8})$$

Objects of domain $\hat{\mathcal{O}}$ are in 1-1 correspondence with principal ideals over their finitary basis. The filtering function **filter** returns, as its output, the closest valid object to its input object (the object returned is a well-defined object, and it is a fixed point of the filtering function). Thus, given that objects are strong ideals in the finitary basis of $\hat{\mathcal{O}}$, they are downward-closed sets. Thus, the condition (B.8) is true for all objects in $\hat{\mathcal{O}}$.

Thus, the projection defined by **filter** is a finitary projection. □

Appendix C

Code Examples

C.1 Classes

In this appendix, we present code examples that concretely demonstrate the concepts and notions we discuss in this thesis. Unless otherwise noted, code examples in this thesis use the syntax of JAVA-like OO languages. In the code examples it is *not* assumed that all classes inherit from a single superclass (like `Object`).

First, we assume a declaration of class `Object` as in Figure C.1.

```
class Object {
    // Classes with no explicit constructors are always
    // assumed to have a default constructor that
    // initializes the fields of the object, if any.

    Boolean equals(Object o){
        return (o is Object);
        // Equivalent to: 'o.getClass() == Object.class'
    }
}
```

Figure C.1 : Class `Object`

As demonstrated in class `Object`, we make use of the standard class `Boolean`, which we assume has boolean values `true` and `false` (or equivalents) as its objects, and that it supports standard boolean operations on boolean values.

In the following code examples, we will make use of the declarations of classes `A`,

B, C, D and E presented in Figure C.2. These simple classes serve no purpose but to demonstrate the concepts and notions we present.

```
class A { // no superclasses
}

class B extends A {
    // add no members
}

class C extends B {
    D foo(D d) { return d; }
}

class D { // no superclasses
    A bar() { return new A(); }
}

class E extends D {
    A meth() { return new A(); }
}
```

Figure C.2 : Classes A, B, C, D and E

To demonstrate a more complex example, we also assume the declaration of class `Pair` presented in Figure C.3.

```
class Pair extends Object {
    Object first;
    Object second;

    Boolean fstEqSnd(){
        return first.equals(second);
    }
    Boolean equalTo(Pair p){
        return first.equals(p.first) &&
            second.equals(p.second);
    }
    Boolean equals(Object p){
        if(p instanceof Pair)
            return equalTo((Pair)p);
        return false;
    }
    Pair setFirst(Object nf){
        return new Pair(nf, second);
    }
    Pair setSecond(Object ns){
        return new Pair(first, ns);
    }
    Pair swap(){
        return new Pair(second, first);
    }
}
```

Figure C.3 : Class Pair

C.2 Shapes

The shape of instances of class `Object` (in Figure C.1) is the set

```
{equals}
```

of member (field and method) names in class `Object`.

The shape of instances of classes `A`, `B`, `C`, `D` and `E` (in Figure C.2), respectively, are the sets

```
{}, {}, {foo}, {bar}, {bar, meth}
```

(note that the shapes of instances of `A` and `B` are the same). The shape supported by class `E` is a supershape of the shape of class `D`, which in turn, is a supershape of the shapes of classes `A` and `B`.

The shape of instances of class `Pair` (in Figure C.3) is the set

```
{equals, first, second, fstEqSnd, equalTo, setFirst,  
setSecond, swap}
```

of member names in class `Pair`. The shape of (instances of) class `Pair` is a supershape of the shape of (instances of) class `Object`^a.

^aNote that because class `Object` has no fields, all its instances are mathematically-equivalent. Mathematically-speaking, thus, class `Object` has only one instance.

Figure C.4 : Shape Examples

C.3 Object Interfaces/Record Types

The structural object interface (of instances) of class `Object` is

```
OSOI  $\triangleq$  object_interface  $\mu$ O. {
  B equals(O)
} and  $\mu$ B. {...interface of class Boolean...}
```

The object interface (of instances) of classes A, B, C, D and E, respectively, are

```
ASOI  $\triangleq$  object_interface {}
BSOI  $\triangleq$  object_interface {}
// note that ASOI and BSOI are the same.
DSOI  $\triangleq$  object_interface {
  BSOI bar()
  // Note need to include full interface. BSOI is a "macro".
  // BSOI is used, rather than ASOI, to make a point: Interface
  // names here are just "macro names". The names can be changed
  // without changing the meaning of the defined interfaces.
}
CSOI  $\triangleq$  object_interface {
  DSOI foo(DSOI)
}
ESOI  $\triangleq$  object_interface {
  ASOI bar(), // BSOI, or 'object_interface {}'
              // could be used in place of ASOI
  ASOI meth()
}
```

Figure C.5 : Object Interface/Record Type Examples

While the object interface (of instances) of class `Pair` is

```
PSOI  $\triangleq$  object_interface  $\mu P$ . {
  B equals(O),
  O first, O second,
  B fstEqSnd(),
  B equalTo(P),
  P setFirst(O),
  P setSecond(O),
  P swap()
} and  $\mu O$ . {...interface of class Object...}
  and  $\mu B$ . {...interface of class Boolean...}
```

Figure C.6 : Object Interface/Record Type Examples

C.4 Structural Subtyping

Figure C.7 presents examples for structural object types in the structural subtyping relation.

The following structural object types/interfaces, from Figures C.5 and C.6, are in the structural subtyping relation, $<:$.

```
BSOI <: ASOI (and ASOI <: BSOI, because ASOI = BSOI)
CSOI <: BSOI (a genuine "is-A")
DSOI <: BSOI (unwarranted "is-A")
ESOI <: DSOI (a genuine "is-A")
OSOI <: BSOI (unwarranted "is-A")
PSOI <: OSOI (a genuine "is-A")
```

Figure C.7 : Structural Subtyping Examples

Note that pairs in structural subtyping relations could express genuine “is-A” relations or unwarranted accidental ones. For example,

```
object_interface {} <: object_interface {}
```

which we expressed above as $\text{BSOI} <: \text{ASOI}$ (and $\text{ASOI} <: \text{BSOI}$), intuitively holds true when in reference to objects of class **B** being (or “is-A”, or are substitutable for) objects of class **A**. This is something the developer (of class **B**) intended. It is thus a genuine is-A relation. The same relation does not hold true, however, when in it refers to objects of class **A** being objects of class **B**. Viewing objects of **A** as objects of **B** may not have been intended by the developer of class **A**. It is an accidental (“spurious”) is-A relation. It is only a result of the fact that structural subtyping does not capture the full intention of class developers.

The subtyping pairs $\text{CSOI} <: \text{BSOI}$, $\text{ESOI} <: \text{DSOI}$ and $\text{PSOI} <: \text{OSOI}$ express genuine is-A relations when referencing objects of classes **C** being **B**'s, **E**'s being **D**'s, and **Pair**'s being **Object**'s, respectively. The pairs $\text{DSOI} <: \text{BSOI}$ and $\text{OSOI} <: \text{BSOI}$ express an unwarranted is-A relation when referencing objects of **D** being **B**'s, and **Object**'s being **B**'s.

C.5 Signatures and Subsigning

Figures C.8 and C.9 present examples of class signatures, and Figure C.11 presents pairs of signature closures in the subsigning relation.

The signature (of instances) of class `Object` is

```
Obj  $\triangleq$  sig Object {
  equals: Object→Boolean
}
```

The signature of (instances of) classes `A`, `B`, `C`, `D` and `E` are

```
A  $\triangleq$  sig A {}
B  $\triangleq$  sig B ext A {}
C  $\triangleq$  sig C ext B {
  foo: D→D
}
D  $\triangleq$  sig D {
  bar: ()→A
}
E  $\triangleq$  sig E ext D {
  bar: ()→A,
  meth: ()→A
}
```

Figure C.8 : Signature Examples

While the signature (of instances) of class `Pair` is

```
Pair  $\triangleq$  sig Pair ext Object {
  equals: Object→Boolean,
  first: Object,
  second: Object,
  fstEqSnd: ()→Boolean,
  equalTo: Pair→Boolean,
  setFirst: Object→Pair,
  setSecond: Object→Pair,
  swap: ()→Pair
}
```

Figure C.9 : Signature Examples

It should be noted that the syntax used to present examples of class signatures

in Figures C.8 and C.9 is different from the syntax generated by the mathematically-oriented abstract syntax rules presented in Chapter 5. Even though equally informative, the syntax of signatures we used here is closer to the syntax of classes, which most mainstream OO developers are thus familiar with.

The signature environments and signature closures in Figure C.10 use class signatures presented in Figures C.8 and C.9.

```

ObjectSE = { Obj, Bool }
  // where Bool is the class signature of class Boolean
Ase = { A }
Bse = { A, B }
Cse = { A, B, C }
Dse = { A, D }
Ese = { A, D, E }
PairSE = { Obj, Bool, Pair }

ObjSC = (Object, ObjSE)
Asc = (A, Ase), Bsc = (B, Bse), Csc = (C, Cse)
Dsc = (D, Dse), Esc = (E, Ese)
PairSC = (Pair, PairSE)

```

Figure C.10 : Signature Environment and Signature Closure Examples

Figure C.11 presents signature environments, and signature closures (from Figure C.10) that are in the extension relation \blacktriangleleft , and the subsigning (inheritance) relation \trianglelefteq , respectively.

$Bse \triangleleft Ase$ (but $Ase \not\triangleleft Bse$)
 $Cse \triangleleft Bse$
 $Ese \triangleleft Dse$
 $PairSE \triangleleft ObjSE$

and by rules of subsigning (See Section 5.4)

$Bsc \trianglelefteq Asc$ (but $Asc \not\trianglelefteq Bsc$)
 $Csc \trianglelefteq Bsc$
 $Esc \trianglelefteq Dsc$
 $PairSC \trianglelefteq ObjSC$

Note that pairs in subsigning relation only express genuine “is-A” relations. In particular, unlike we had for structural subtyping (in Figure C.7), for subsigning we have

- $Dsc \not\trianglelefteq Bsc$

($Dsc \trianglelefteq Bsc$ is unwarranted by rules of subsigning,
since $Dse \not\triangleleft Bse$)

- $Dsc \not\trianglelefteq Asc$

($Dsc \trianglelefteq Asc$ is unwarranted by rules of subsigning, since,
even though $Dse \triangleleft Ase$, but $A \notin super_sigs(D) = \phi$)

- $ObjSC \not\trianglelefteq Bsc$

($ObjSC \trianglelefteq Bsc$ is unwarranted by rules of subsigning,
since $ObjSE \not\triangleleft Bse$)

Figure C.11 : Extension and Subsinging Examples

Using the class declarations presented in Section C.1, the reader is invited to construct more examples of signature closures in and outside the subsigning relation. Unlike structural subtyping ($<:$), the examples for subsigning demonstrate that subsigning (=nominal subtyping) *fully captures the intention of class developers*.