0
Introduction

This is a book about names and symmetry in the part of computer science that has to do with programming languages. Although symmetry plays an important role in many branches of mathematics and physics, its relevance to computer science may not be so clear to the reader. This introduction explains the computer science motivation for a theory of names based upon symmetry and provides a guide to what follows.

0.1 Atomic names

Names are used in many different ways in computer systems and in the formal languages used to describe and construct them. This book is exclusively concerned with what Needham calls 'pure names':

> A pure name is nothing but a bit-pattern that is an identifier, and is only useful for comparing for identity with other such bit-patterns – which includes looking up in tables to find other information. The intended contrast is with names which yield information by examination of the names themselves, whether by reading the text of the name or otherwise. [. . .] like most good things in computer science, pure names help by putting in an extra stage of indirection; but they are not much good for anything else.
>
> (Needham, 1989, p. 90)

We prefer to use the adjective 'atomic' rather than 'pure', because for this kind of name, internal structure is irrelevant; their only relevant attribute is their identity. Although such names may not be much good for anything other than indirection, that one thing is a hugely important and very characteristic aspect of computer science.

0.2 Support and freshness

The complexity of computer systems has stimulated the development of compositional methods for specifying and constructing them. If one wishes to compose a whole out of parts, then one had better have mechanisms for hiding, or at least controlling access to, the identity of the atomic names upon which each part depends. The prerequisite for devising such mechanisms and understanding their properties is a firm grasp of what it means for a piece of the system to 'depend' upon an atomic name. Although there are syntactic considerations, such as various notions of textual occurrence, this issue really concerns semantics: what does it mean for the behaviour of a software system to depend upon the identity of some atomic names?

A conventional response to this question is simply to parametrize: replace the use of structures of some kind by functions from names to structures. This book develops an alternative approach – a mathematical theory of 'name dependence' based upon the symmetries that a structure exhibits when one permutes names. The fundamental idea is to model systems involving atomic names with mathematical structures for which every permutation of names induces a transformation of the structure. In this case one says that permutations *act* upon the structure and these actions are required to satisfy some simple laws; this is the subject of Chapter 1. A finite collection of atomic names is said to *support* such a structure if any permutation that fixes each name in the collection induces a transformation that leaves the entire structure unchanged with respect to whichever notion of equality of structures is of concern. This notion of support is an old one, mathematically speaking, that we put to new use within computer science.

From this viewpoint, a structure does not depend on a particular atomic name if there is a support set for the structure that does not include that name. This may seem a rather indirect way of getting at the idea of dependence upon names, but it has advantages compared with the more common approach based upon parametrization. In particular the use of functions means that name-dependency is made explicit, whereas often one wants to leave it implicit and work instead with the complementary relation of non-dependence, or *freshness* as we will call it. Chapters 2 and 3 develop the properties of the nominal sets notions of support and freshness. Chapter 7 consider applications of these notions to a fundamental technique in programming language semantics – the use of rule-based inductive and coinductive definitions of subsets of a given set. Chapter 9 uses freshness to model language constructs for hiding the identity of a name outside a given scope.

We turn next to the topic that was the original stimulus for developing the theory of nominal sets.

0.3 Abstract syntax with binders

When defining a programming language it is customary to specify its concrete syntax using context-free grammars that generate the strings of symbols that are legal phrases in the language. Definitions of concrete syntax have to deal with many issues to do with layout, punctuation and comments that are irrelevant to the meaning of programs. If one is primarily concerned with the semantics of programming languages, then what matters is the language's abstract syntax given in terms of parse trees. The use of abstract syntax trees enables two fundamental and inter-linked tools in programming language semantics: the definition of functions on syntax by recursion over the structure of trees; and proofs of properties of syntax by induction on the structure of trees. These techniques have their origin in the classical notions of primitive recursion and induction for the natural numbers, which were extended to abstract syntax trees by Burstall (1969), Martin-Löf (1971) and others.

However, abstract syntax trees and their associated structural recursion and induction principles are in one important respect not sufficiently abstract. They do not take into account the fact that some syntax constructors involve binding atomic names to specific scopes. Various schemes have been devised for specifying binding information. One popular option is to use some form of typed λ-calculus (Church, 1940) as a meta-language and express binding forms of the object-language in terms of function abstraction at the meta-level (Pfenning and Elliott, 1988; Harper *et al.*, 1993; Miller, 2000). Some forms of binding do not fit comfortably into this approach; see the discussion in (Sewell *et al.*, 2010, section 3), which describes a flexible mechanism for incorporating binding information in grammars that is part of the Ott tool. Whatever approach is taken, such binding specifications tell us which abstract syntax trees differ only up to consistent renaming of bound names. This is the relation of *α-equivalence*, generalized from its original use in λ-calculus.

To be a 'binder' in the sense that we are using it here, a language construct should have the property that for any reasonable definition of the language's semantics, α-equivalent phrases have equal meanings. Thus in the presence of binders, many syntax-manipulating operations only respect meaning if one operates on syntax at a level of abstraction that respects α-equivalence. So it is natural to regard α-equivalence classes of parse trees, rather than the trees themselves, as the true abstract syntactical structures which are assigned a

meaning in a semantics. The problem is that unlike finite syntax trees, their α-equivalence classes are in general infinite sets and so require indirect methods of construction, computation and proof.

One way round this problem is to devise a scheme for canonical representatives of α-equivalence classes; for example by using indexes instead of names in binders, following de Bruijn (1972). The well-known disadvantage of this device is that it necessitates a calculus of operations on de Bruijn indexes that does not have much to do with our intuitive view of the structure of syntax. As a result there can be a 'coding gap' between statements of results about syntax with bound names and their de Bruijn versions – and hence it is easy to get things wrong. For this reason, most work on programming language semantics that is intended for human rather than machine consumption sticks with ordinary abstract syntax trees involving explicit bound names and uses an informal approach to α-equivalence classes. Yet there is a pressing need for fully formal methods when proving properties of program semantics, caused by the desire for high assurance of correctness in situations where lives or finances are at risk, or by complexities of scale, or both.

The informal approach is usually signalled by a form of words such as 'we identify expressions up to α-equivalence'; see for example Harper (2013, section 1.2) and Remark 10.1 in this book. In this informal mode, one does not make any notational distinction between an α-equivalence class and some chosen representative of it; and if that representative is later used in some context where its particular bound names clash in some way with those in the context, then it is changed to an α-equivalent expression whose bound names are fresh. The theory of nominal sets, with its notion of freshness, is able to fully formalize these common informal practices with bound names via the notion of name abstraction, discussed next.

0.4 Name abstraction

As described above, nominal sets provide a theory of implicit dependence on names (support) and name independence (freshness) based upon the action of name permutations on structures. If one wishes to make explicit how a structure depends upon a name, the traditional approach is to abstract and form a function from names to structures. Quite what a 'function from names to structures' means depends upon the strength of the ambient logical formalism. The notion of function is not as absolute as, say, the notion of 'ordered pair'. This has a complicating effect on logical systems that combine functional representations of binders in abstract syntax with computable functions

operating on those representations (Poswolsky and Schürmann, 2009; Pientka and Dunfield, 2010). The two sorts of function have to be distinguished, leading to meta-meta-distinctions that are perhaps difficult for the average user to appreciate.

Nominal sets contain a form of name abstraction that manages to avoid these problems with function abstraction. Function abstraction models α- and β-conversion (and possibly η-conversion, depending upon how extensional is the notion of function). By contrast, name-abstraction in nominal sets models α-conversion, but only the limited form of β-conversion where one substitutes a fresh name for the bound name. This is just right for representing α-equivalence classes of abstract syntax in a way that captures the informal usage mentioned above. At the same time, nominal name-abstraction has a first-order character that allows one to use formalizations of it in logical systems (Urban, 2008) and in programming languages (Shinwell, 2005b) that mix representation of syntax up to α-equivalence with computation on that syntactical data. Chapter 4 develops the properties of this kind of name abstraction. Chapter 8 uses it to represent abstract syntax modulo α-equivalence as inductive data types with associated principles of 'α-structural' recursion and induction. Chapters 10 and 12 explore the applications of name abstraction within functional programming languages and to computational aspects of logic. At the moment these applications are most accessible to the world via the Isabelle/HOL interactive theorem-proving system (Nipkow *et al.*, 2002). This is because Urban and Berghofer (2006) have implemented a 'nominal datatype' package for it based on the nominal sets notion of name abstraction. This is now part of the official Isabelle software distribution and seems to be a useful tool for formalizing proofs about operational semantics that allows users to retain familiar habits and conventions concerning bound names and their freshness; see for example Bengtson and Parrow (2009).

0.5 Orbit-finiteness

Nominal sets provide a theory for mathematical structures involving atomic names based upon the symmetries exhibited by a structure when names are permuted. Finiteness obviously plays a fundamental role in the study of data structures and algorithms on them. Taking symmetry into account allows one to extend the reach of that study to encompass structures that are infinite, but only have finitely many different forms modulo symmetry. Name-abstractions provide an extreme example; they are singletons modulo symmetry and this partly explains why it is possible to use them to develop such a well-behaved

theory of representation and computation for syntax with binders. In general, the property of having only finitely many orbits for the action of name permutations seems to be a very useful relaxation of the notion of finiteness. It is studied in Chapter 5. One application of orbit-finiteness is to the denotational semantics of programs based upon the use of partial orders, where potentially infinite behaviour is modelled as a limit of finite approximations; this is explored in Chapter 11.

0.6 Alternative formulations

One approach to formalizing the notion of freshness for names is to make use of the technique of 'possible worlds' stemming from Kripke semantics for intuitionistic and modal logics (Kripke, 1965). Structures are indexed by worlds that contain (at least) the names that are known at that stage. One then has to give morphisms between structures induced by moving from one world to another, for example by adding a name not already in the current world. The mathematics of this approach is best treated as part of the category theory of *presheaves* (Borceux, 2008, vol. 3, chapter 2). Fiore *et al.* (1999) use presheaves in their algebraic treatment of abstract syntax with binders. This is closely related to the technique of using 'well-scoped' de Bruijn indexes within functional programming and interactive theorem proving based upon constructive type theory; see for example Pouillard (2012), who compares this technique with nominal ones. Techniques based upon possible worlds bring with them a certain amount of book-keeping to do with change of world, for example when a world is weakened by adding a new name. By contrast, within the theory of nominal sets dependence on names is implicit – it is a property of an object (its support), rather than extra structure that has to be explicitly specified and manipulated.

As mentioned in the Preface, nominal sets arose from presheaves, sheaves and topos theory (Johnstone, 2002). The category of nominal sets was designed to be a conveniently concrete presentation of an existing topos of sheaves, known as the *Schanuel topos*. That uses a category of worlds consisting of injective functions between finite ordinals, thought of as the number of different well-scoped indexes currently in use. The passage to nominal sets involves first replacing finite ordinals by finite subsets of some fixed, infinite collection of atomic names; and then replacing injections between finite subsets by permutations of the whole collection of names. Neither of these steps change things up to category-theoretic equivalence; but the final result, the category of nominal sets that is studied in this book, is a much simpler setting in which

to carry out the constructions and calculations relevant to the topics discussed in this introduction. This is particularly noticeable when it comes to higher-order functions; exponentials of nominal sets are appreciably easier to work with than exponentials of the **Set**-valued functors that are the objects of the Schanuel topos.

Chapter 6 describes the equivalence between the Schanuel topos and nominal sets. It also discusses some other equivalent formulations, notably the concept of *named set* that arose in the work of Montanari and Pistore (2000) on automated verification for mobile processes (Milner *et al.*, 1992).

0.7 Prerequisites

A prerequisite for understanding Part One of the book, on the theory of nominal sets, is some familiarity with naive set theory and higher-order logic; see Andrews (2002), for example.

We also assume a knowledge of the basics of category theory. The first work on nominal sets (Gabbay and Pitts, 1999, 2002; Gabbay, 2000) took a set-theoretic approach. It used *FM-sets*,[1] the cumulative hierarchy of hereditarily finitely supported sets devised by Fraenkel in the 1920s and used by him and Mostowski to get independence results for Zermelo–Fraenkel set theory with atoms; see Gabbay (2011) for a discussion of these historical sources. Nominal sets are essentially the FM-sets that depend upon no particular names (that is, whose support is empty). On the other hand the universe of FM-sets can be given a category-theoretic construction; and an 'algebraic' set-theory (Joyal and Moerdijk, 1995) can be developed for it within a category of (large) nominal sets. Should one develop the theory of nominal sets using set theory or category theory; in other words, should one be 'element-oriented' or 'morphism-oriented'? Here both approaches are used, as appropriate. Nevertheless, emphasis is placed upon category-theoretic concepts, particularly the use of various universal properties that characterize constructions uniquely up to isomorphism. So familiarity with the basic concepts of category theory – category, functor, natural transformation, adjunction and equivalence – is assumed. There are many suitable introductions, some aimed specifically at computer science applications, such as those by Pierce (1991) and Crole (1993). The classic text by MacLane (1971) still holds its own; and the three volumes by Borceux (2008) are usefully comprehensive. Some familiarity is needed with

[1] The term 'nominal set' was first used by Pitts (2003).

the connections between category theory, typed λ-calculus and higher-order logic, as described by Lambek and Scott (1986), for example.

The distinctive feature of nominal sets is their reliance upon some simple mathematics to do with symmetric groups and their actions on sets. This is not material that is likely to be familiar to the intended reader. So the book contains a self-contained account of the small amount of this well-developed topic that is needed.

For Part Two of the book, on computer science applications of nominal sets, the reader needs to be familiar with the basic techniques of programming language semantics; see for example Winskel (1993).

0.8 Notation

Being about a new subject, the literature on nominal sets does not always agree on matters of notation. The subject-specific notations used in this book are collected in an *Index of notation*. Apart from that we use more-or-less standard notations and conventions from logic, naive set theory and category theory, some of which are listed below.

Logic We write $\varphi \wedge \psi$ for conjunction, $\varphi \vee \psi$ for disjunction, $\varphi \Rightarrow \psi$ for implication, $\varphi \Leftrightarrow \psi$ for bi-implication, and $\neg\varphi$ for negation. Quantification is written $(\mathsf{Q}\, x)\ \varphi$ with the convention that the scope of the quantifier Q extends to the right as far as possible. So, for example, $(\forall x)\ \varphi \wedge \psi$ means $(\forall x)\ (\varphi \wedge \psi)$ rather than $((\forall x)\ \varphi) \wedge \psi$.

Sets We denote by $X - Y$ the set subtraction, $\{x \in X \mid x \notin Y\}$.

Functions Function application is written without punctuation: $F\ x$ means the result of applying function F to argument x. Multiple applications associate to the left. So, for example, $G\, F\, x$ means $(G\, F)\, x$, rather than $G\, (F\, x)$. If an expression $e(x)$ denotes an element of a set Y as x ranges over the elements of a set X, then the function from X to Y that it determines will be denoted by either of the notations

$$x \in X \mapsto e(x) \in Y \quad \text{or} \quad \lambda x \in X \to e(x)\,.$$

When the set X has some structure, we use notations for patterns; for example if $X = X_1 \times X_2$ is a cartesian product, we write $\lambda(x_1, x_2) \in X_1 \times X_2 \to e(x_1, x_2)$.

Categories If $\mathbf{C}$ is a category, its collection of objects will also be denoted $\mathbf{C}$. If X and Y are objects of $\mathbf{C}$, then we write $f \in \mathbf{C}(X, Y)$, or just $f : X \to Y$, to indicate that f is a morphism in $\mathbf{C}$ whose domain is X and whose codomain is Y. The identity morphism for X is written as id_X, or just id. Composition is written in application order: $g \circ f \in \mathbf{C}(X, Z)$ denotes the composition of the morphism $f \in \mathbf{C}(X, Y)$ followed by the morphism $g \in \mathbf{C}(Y, Z)$.