UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
A general theory of formality

Permalink
https://escholarship.org/uc/item/5t70z373

Author
Beck, Andrew Edward

Publication Date
2011

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, SAN DIEGO

A General Theory of Formality

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Philosophy

by

Andrew E. Beck

Committee in charge:

Professor Gila Sher, Chair
Professor Samuel Buss
Professor Paul Churchland
Professor Monte Johnson
Professor Sorin Lerner
Professor Clinton Tolley

2011

Copyright
Andrew E. Beck, 2011
All rights reserved.

The dissertation of Andrew E. Beck is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

_____

Chair

University of California, San Diego

2011

DEDICATION

To Jina Park, without your patience, faith, and grace this work could never have been completed.

EPIGRAPH

In the beginning . . . the earth was without form.

—Genesis 1

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Even in a discipline as individualistic as philosophy, nothing is ever accomplished alone. There are many people who deserve my deepest gratitude. I first must thank Gila Sher. Her insight, encouragement, enthusiasm for this project, and most of all her patience, made her an ideal director for this dissertation. It could never have been completed without her. Nor could it have been begun without her, for it was in numerous seminars and independent studies with her that this project took root. I must also thank Clinton Tolley, who read numerous drafts of this work, especially the early, difficult drafts of Chapter 4. This was no easy task, and I owe much to his insightful comments and wisdom. Monte Johnson also read early drafts and provided historical insights which helped shape the way I think about formality. Much of the direction in Chapter 4 resulted from some valuable conversations with Sam Buss, and for this I am grateful. No errors are due them, they are all without question my own.

I also owe a debt of gratitude to the Southern California Working Group in the History of Logic and Mathematics, whose members read and provided insightful comments on a draft of Chapter 3. I gave several presentations at the UCSD Philosophy Graduate Student Colloquia, and there, and on many other occasions, received valuable feedback from fellow graduate students. In particular I must thank Matt Brown, whose astounding willingness to contribute his time and thoughts extends even to this day, and Adam Streed, comrade in logic with whom I honed my thoughts, particularly when this project was in its infancy.

Special thanks are due to Tim Gades, and David Diepenbrock, on whom I inflicted a considerable number of puzzling philosophical and technological questions, and who helped me answer them.

Most of all, I must thank my wife, Jina Park, for her encouragement and patience in the completion of this project, and also, together with our daughter Josephine, for inspiring me in all things.

VITA

2004	B.A. in Philosophy *summa cum laude*, Arizona State University
2009	M.A. in Philosophy, University of California, San Diego
2004-2008	Graduate Teaching Assistant, University of California, San Diego
2011	Ph.D. in Philosophy, University of California, San Diego

ABSTRACT OF THE DISSERTATION

A General Theory of Formality

by

Andrew E. Beck

Doctor of Philosophy in Philosophy

University of California, San Diego, 2011

Professor Gila Sher, Chair

"Form" is one of the first concerns of philosophy, though its centrality to philosophy has somewhat faded. However, in the last few decades it has reemerged as a central idea in a particular area of philosophy — the demarcation of "logicality." I show that it should be of concern more broadly. As invariance, the underlying principle used to characterize logical formality, is not itself essentially logical, important questions arise. Can invariance be used to characterize other formal theories? Can it furthermore form the basis of a contemporary, general theory of formality?

I develop and advocate for just such a general theory of formality (GTOF). This GTOF is based on a notion of invariance which is completely generalized — it consists simply of the stability of features under functional mappings in a given domain. Different domains and functional mappings give rise, then, to different types of formality.

In advocating the view, I show that the GTOF has promise for an acceptable combination of adequacy — it does minimal damage to intuitions about formality — and usefulness. I first show that the GTOF rules geometrical theories to be formal. This is followed by an examination of standard first order logic, which illuminates the possibility that a close correlation between syntactic and semantic formalities may be distinctive of formal languages. This is bolstered by an examination of a number of programming languages and methods. Finally, some deeper consequences of the GTOF are examined and its association to other philosophical theories is explored.

The general theory of formality I develop is a promising candidate to fill a gap in existing theory. Not only does it do justice to some of our deepest intuitions about formality, but it sheds light on important relationships between formal systems that have not been previously recognized.

Chapter 1

Introduction

"Form" is one of the first concerns of philosophy. Understanding forms, both in particular and in general, was a primary goal of the Socratic tradition, which would have us believe that grasping these forms provides insight into the depths of reality. If there was any clear success toward this end, however, no record of it has survived. We have instead a collection of challenging and troubled views complex enough to consume entire philosophical careers. Plato's struggle with "form" first rages in his work and then fades quietly into irresolution. Subsequent thinkers fall just as far from consensus with one another as Plato did within himself. Unlike Plato's vision of an independent world of "ideal forms," Aristotle took them to be part of the corporeal world. He linked "forms" to the very "substance" and " matter" of which particulars consist, and gave them a place in the causal order. Aristotle granted the same license to talk of the "form" of humans as he did to that of geometric figures.

The manifestations of "form" we find in the modern period, though still concerning the ultimately nature of reality, are dramatically different in character. For example, Bacon writes,

> [T]o inquire the Form of a lion, of an oak, of gold; nay, of water, of air, is a vain pursuit: but to inquire the forms of sense, of voluntary motion, of vegetation, of colours, of gravity and levity, of density, of tenuity, of heat, of cold, and all

> other natures and qualities, ... of which the essences, upheld by matter, of all creatures do consist. ... [T]o inquire, I say, the true Forms of these, is that part of metaphysic which we now define of.[1]

In a sense this is considerably less general than the most prominent ancient views, as "form" here only pertains to properties or "natures and qualities" of things, rather than to the objects or things themselves. Yet at the same time Bacon is clearly suggesting that "form" is explanatorily significant.

Form also plays a notable role in the work of Kant, whose views are perhaps the most influential in subsequent thinking about formality. But the "forms" he focuses on most of are certain subjective structures without which, we are told, understanding would not be possible. Space and time, for example, are such structures which render the content of "sensibility" (sensible experience, more-or-less) understandable. Thought, on Kant's view, relies on similar conceptual structures, and it is in this category that logic falls. It is, or is part of, a subjective conceptual structure which is necessary for thought. For Kant the distinction between "form" and "matter" or "content" has markedly epistemological significance, owing to the subjective and *a priori* nature of these "forms." This is the foundation for the infamous analytic-synthetic distinction, and one need not grasp its finer points to see the significant divergence from preceding positions.

"Form," taken as a subject in itself — apart from logic — currently enjoys far less attention. This is not the result of any kind of consensus, but is primarily due to disciplinary specialization and a consequent lack of discourse. Platonic views of form are implicitly endorsed by many mathematicians, and vestiges of Aristotelian "formal causes" have an uncomfortable existence in the biological and medical sciences.[2] There is con-

[1] [Bacon, 2000], II. vii. §5.

[2] This stems from the use of teleological explanation, wherein the end, purpose, or "telos" of a system is provided as the explanation for the system's existence or function. Aristotle referred to such a telos as a *formal* type of cause, and it is consequently closely tied to his account of "form." In most philosophical accounts of scientific explanation, teleological explanation is seen as inferior, at best, and most scientific disciplines have moved away from it. One does not, for example, explain anything about quarks in terms of their "purpose." However, biological evolution and other processes give rise to systems in such a way that

siderable consensus on the idea that logic is formal in some sense,[3] and mathematics as well. Most would likely agree to physics being intuitively "more formal" than chemistry, which itself is possibly "more formal" than biology. "Form" features in literary contexts in contrast to "content," and in many cases the second is considered to be more interesting than the first.[4] Computer scientists talk about "form" in ways that sometimes have much in common with logicians, but also call many things "formal" (and sometimes even "logical") which fall far outside the scope of the logician's use of these terms.

What we may draw from this is that interest in a *general* account of formality, of what it is to be formal or to have one form rather than another, has waned significantly over the last thousand years or so. Fields separated from once all-encompassing Philosophy stand on their own, and, together with newly developed disciplines lacking philosophical ancestry, have found adequate linguistic resources for their own internal purposes. Disciplinary specialization has made it such that, for example, if literary theorists use "form" differently from logicians, little confusion results. There can be no misunderstanding where there is no communication. The result is that there is little internal impetus to develop shared vocabulary or come to terms with differences of use.

But is this a problem? Why not simply embrace pluralism, let a thousand "formals" bloom, and leave it at that? The case for pluralism about "form" and "formality" is strong and easy to make in view of the straightforward and compelling semantic evidence just noted. It is also not something against which we wish to argue. It is not our intention

---

they can be seen as having purposes — they provide or enable a certain competitive edge, successful strategy, etc. — and it is then that teleological explanations can be useful. It is difficult to explain the existence of mammalian hearts, for example, without reference to the need for mammals to circulate blood.

[3] Although there is also considerable debate over how this should be understood (see [Bonnay, 2008], [Etchemendy, 1990], [Feferman, 1999], [MacFarlane, 2000], [McGee, 1996], [Sher, 1991], [Tarski, 1986], and the present work, particularly Chapter 3). For a contrasting view, that logic may not be concerned explicitly with "form," see [Novaes, 2010].

[4] A particular sonnet, for example, is hardly interesting for its rhyme scheme and meter but rather for what it is *saying* — its content. Which is not to say that a poet may not construct something interesting by means of "form," but merely that one does not say *of a particular sonnet* that it is good or interesting on account of those formal features which make it a sonnet. One might make such a judgment about sonnets generally, over and against other poetical forms, but to do so of a particular token seems somehow mistaken.

to dictate to logicians, mathematicians, physicists, engineers, or poets what they should mean when they discuss "form" amongst themselves. There are, however, several reasons why we are not comfortable to merely "leave it at that." To begin with, whether some general account of form can be given is an interesting philosophical question in itself, and it has received relatively little attention in recent decades. Advancements since Plato (philosophical, logical, and otherwise) notwithstanding, the topic is interesting for many of the same reasons it was interesting then. A further reason is that, as we will see, several of the colloquial senses of "formal" and "formality" have a high degree of discipline-specific utility, and there are compelling reasons for thinking that this utility is not really domain-bound. This suggests that a *general* theory of formality, based on similar principles, may well have considerable utility in its own right.

In what follows, we present, expound, and recommend the virtues of what we call the "general theory of formality," — the "GTOF." In Chapter 2 we begin by explaining the fundamental components of the GTOF and how these pieces work together in the context of the seminal work on geometrical invariance by Felix Klein (from whence the GTOF claims conceptual ancestry). We discuss some important and significant aspects of the theory, but reserve more interesting theoretical consequences for Chapter 5. In Chapter 3 we discuss the GTOF in light of the contemporary debate over the demarcation of standard first order logic, first to show that logic is indeed formal according to the GTOF (as it certainly should be), but also to make clear how questions of formality, as we conceive them, are distinct from questions of *logicality*. In Chapter 4 we broaden our view to include several programming languages and technological developments of historical interest. We show not only that the formality of these languages satisfies the GTOF, but also that the GTOF sheds light on the relationships between these languages, their underlying hardware, and the problems they are used to solve. Chapter 5 explores some interesting theoretical consequences of the view and its relationship to other philosophical work on formality and invariance. Some final remarks and speculation on future research are presented in Chapter 6.

The approach and purpose of this inquiry diverge in significant ways from those commonly employed and approved of in contemporary philosophy, and this warrants a few comments to prevent confusion with superficially similar approaches and also to explain its legitimacy.

The first thing to make clear is that this project is not primarily directed toward capturing the meanings of existing uses of "formality" and showing these to be equivalent, or even somehow consistent.[5] This is in contrast to the contemporary "analytic" tradition which focuses on analysis of and intuitions about existing meanings and ideas. Although we will engage in some thought about existing notions of "form" and "formality," this thought does not ground the legitimacy of the proposed theory any more than to show that it has useful and interesting applications (and that it can be legitimately called a "theory of formality" rather than a theory of something else). Hence, if the theory is challenged on grounds that it is not what *anybody* actually means by "formality," and is therefore dubious, we simply respond that our overall purpose is not purely a descriptive conceptual analysis. We are taking for granted, more or less, that none of the existing localized notions of formality are good candidates for a general characterization. [6]

It is on account of this inadequacy, together with the interest in having a general account of formality, that we are proposing this as a *new* theory of formality.[7] And this brings up a question of legitimacy. If the theory is not grounded on what competent speakers of any language mean by "formality," why accept it?

---

[5] Which is as it should be, since the outlook for such a project is unpromising at best (see [Quine, 1960b]).

[6] Arguing for this position would require painfully and tediously proceeding through every identifiable sense of "formality" and showing its inadequacy. Since both pain and tedium are things we hope to avoid insofar as it is possible, we have elected not to pursue this endeavor. In any case, the GTOF seems to our undoubtedly biased opinion to be more adequate and compelling as a general theory of formality than any existing localized notions. The reader is invited to critically consider this position in light of what follows.

[7] To some extent, this approach places us on the "revisionary" side of Strawson's famous distinction between descriptive and revisionary metaphysics in [Strawson, 1959]. We do not purport to describe how anyone currently thinks of "form" or "formality," and we *are* putting forward a new position on the matter. On the other hand, we see the GTOF as attempting to fill a theoretical gap rather than supplanting some existing or intuitive position. Furthermore, we are committed to showing the resonance between the GTOF and existing ways of thinking about "form" and "formality," insofar as this is possible.

It may seem, at first, that an answer to this question could only be forthcoming if compelling evidence is produced for the theory's truth. This is, after all, the reason for theory acceptance most familiar to both philosophers and the general public. This works well enough for theories intended to capture and describe some objects of reality. Physical theories, for example, are usually understood as attempts to accurately describe the reality of the physical world.

Less familiar are those theories which are instead intended to be taken as a way of speaking about or seeing the world which is useful and productive toward some end. These kinds of theories are quite common in mathematics, where they consist of systems of one or more definitions. Philosophers have long accepted the idea that geometry, for example, is not describing the actual world in the same way as physics. It is, however, difficult to deny that geometrical theories have *applications* to the actual world, and it is when they are applied to particular situations in the actual world that we discover whether they are useful in that situation or not.

We take the general theory of formality presented here to be of this second type. The question of its legitimacy should not be grounded on whether it accurately captures any truths, semantic or otherwise. It should instead be grounded on the virtues it has in application.[8] We will not presume to establish this to any great degree by the end of this work. We will primarily focus on the exposition and exploration of the general theory of

[8]The still-disconcerted reader may benefit from considering a generally well-known theory whose pre-eminence is grounded in just this way. Prior to the work of Alan Turing, Alonzo Church, Stephen Kleene, and J.B. Rosser, "computability" was a vague, intuitive notion meaning something like "that which is possible for a computing machine." One of the most significant contributions of these figures, whereby they are deemed the "fathers" of the discipline, was in providing a rigorous characterization of computability. As it happens, there are a number of ways of characterizing computability, some of which may be more appropriate than others for certain types of computation. However, the "Church-Turing Thesis," as it is called, is accepted as the standard definition *not* because it captured what was meant colloquially by "computability" or because it captured the one and only reality of computability, but because it was very useful in a broad range of contexts. The same sort of thing may be said of some of the most famous of Alfred Tarski's projects in the philosophy of logic. In [Tarski, 1983a], contemporary readings notwithstanding, Tarski is not proposing an *analysis* of what anyone at the time meant by "truth," but rather a new way of understanding "truth," which he took to be useful and fruitful when applied to formalized languages. The practice in mathematics is widespread, and many examples can be found there as well.

formality, with the hope of providing enough impetus that others might investigate and prove its usefulness in the future.[9]

If, however, the theory is to be a theory of *formality* and not something else, it should respect the significant features of important and central localized uses. If a consequence of our theory was that logic falls short of being formal, or that a programming language fails to be a formal language, that would challenge its claim to be an adequate theory of *formality* (though it may well be an adequate theory of something else). If, however, our theoretical net turns out not to capture tuxedos and evening gowns, we will not be too worried until a more adequate rival is produced.

[9]One important and interesting aspect of the general theory of formality we propose is its relationship to other similar concepts. These include, in particular, the distinctions between types and tokens, concepts and instances, genera and species, and also definitional and demarcational concepts, for example identity conditions and membership conditions. Although far beyond the scope of the present inquiry, we incline toward thinking that each of these involves formality (both intuitively and as we understand it) in some key ways, and the reader is encouraged to consider whether and how this may be as the inquiry proceeds.

Chapter 2

The General Theory of Formality

2.1 Invariance

If presented with a statue of Athena and asked to describe its "form," few of us would be at a loss. Answers may vary to some degree, but it seems likely that all would have something to do with the shape or structure of the statue. This fact is implicit in the verbal construction "That statue was *formed* out of marble by Phidias." If presented with another statue made of wood having exactly the same shape, we would be unlikely to say that it is the same statue as the first. We might, however, be inclined to say that they are both "Phidias' Athena," and in an unreflective moment we could probably be caught saying that the second statue *is the same as* (or even identical to) the first, "but made of wood." This is the case most likely because we recognize that, though made of different materials, the two statues have the same shape or *form*.

Consider also a particular building, say, the Space Needle. Let us also imagine that a tremendous earthquake topples the Space Needle (and also that nobody is hurt in the process). Now, the citizens of Seattle might wish to rebuild their beloved landmark, and let us assume that they carry out construction precisely as the original building was constructed, but without reusing any part or material of the original structure. Is this new

"Space Needle" the *same* building as the first one? Probably not, but we admit to feeling some impulse to say that it is — that there is no need to call it "Space Needle II" rather than just the "Space Needle." We would not, of course, feel this way had the citizens of Seattle dusted off the plans for the Washington Monument to use in reconstruction. The reason is that, in the first case the second building has exactly the same structure or *form* as the first, and this is not so with the Washington Monument. Furthermore, if the rubble from the original Space Needle were reused to construct a *different* structure, perhaps a municipal library, there would be no impulse to say that the library *is* the Space Needle.[1] We are not attempting to make any assertions here about the nature of identity of physical objects. Instead, the examples serve to illustrate what we take as the fundamental characteristic of formality — invariance.

In the case of the statues, a comparison is made between the two. It is recognized that the material used to create the statues is different, but that certain other characteristics remain the same, or invariant, across the two statues. The same is true in the case of the Space Needle and its replacement. What is different in this case, however, isn't even the *type* of material out of which the buildings are constructed, just the particular materials themselves (e.g. they are both made of the same type of concrete, but share no concrete in common).

Now, this does not mean that we fully grasp the natures of the "forms" of statues or buildings. In particular, there is also invariance occurring between the original Space Needle and the library. The materials are invariant (especially if the library used only and all of the original Space Needle material).

Roughly speaking, however, we can say that in the cases in which there is judged

[1] A complication lurking just under the surface is that of the distinction between a *token* and a *type*. One might suggest, for example, that "Space Needle" refers not to the particular building, as such, but to the *type* of building, of which one building currently standing in Seattle is a token. However, even if so, this is not a problem, *per se*. We are not at all concerned with the particular question of identity in this case, but rather with the role played by invariance. Thus, on this alternative account, we would simply ask what it is that enables identification of the new building as being of the same type as the old one. If it is that they have thus-and-such features in common (rather than others), the same points will hold.

to be sameness of form, the "shape" or "structure" of the objects remains invariant. This relationship between certain characteristics and a change or "transformation," as we will come to call them, is the central idea of the GTOF. In some sense the GTOF holds that formality *just is* invariance. Putting it so simply, however, invites misunderstanding and criticism and also belies some of the most interesting aspects of the GTOF. As we considered in the case of the library, we cannot merely point to any invariance whatever in a given context and declare it to be "formal." In the case of physical objects, *certain* transformations are form preserving, while others are not. Saying just which these are hazards a circularity — without more inquiry we can only say that they are those which leave the shape or structure intact.

In elucidating the theory of formality we must first take care to understand what invariance is and the distinctive role played by transformations, as invariance only occurs with respect to transformations. But we must also answer the question in each case of formality: "Why *these* transformations rather than some others?" It is in answering this question that we will find the theory to be illuminating.

In what follows, we begin with Felix Klein's seminal work on invariance in geometry and proceed through several generations of generalizers, namely Alfred Tarski, Gila Sher, and Denis Bonnay. Building upon this foundation, we end the chapter by presenting the GTOF in technical detail.

2.1.1 Invariance in Geometry

Felix Klein was one of the earliest adopters of the notion of invariance. As laid out in [Klein, 1872], Klein uses the idea to understand and characterize the relationships between the newly developing sub-fields of geometry. This endeavor came to be known as the "*Erlangen programme*."

Klein begins with the idea of "space-transformations." These are various ways of mapping a given space (usually the entirety of the space) onto itself. Klein's particular

interest is with space-transformation *groups* — classes of space-transformations for which the composition operation exhibits closure and associativity, and which possess identity and inverse elements under this composition.[2] The first group to be of interest to Klein is the group of space-transformations "by which the geometric properties of configurations in space remain entirely unchanged."[3] This group, called by Klein the "principle group," includes translations, dilations and contractions, and reflections in the plane. Once this group is selected, Klein then defines the "geometrical" properties as those which remain invariant under space-transformations belonging to this principle group.[4]

Klein then proceeds to generalize the idea to include any n-dimensional manifold, such that for any manifold there will be a group of space-transformations which preserve geometrical properties. Importantly, these groups are not identical. When considered in this generalized form, there is no ground for privileging one of these groups over the others (e.g. the principle group). Instead, all groups must be considered as having equal status, and this brings Klein to his primary task: "Given a manifoldness and a group of transformations of the same; to investigate the configurations belonging to the manifoldness with regard to such properties as are not altered by the transformations of the group."[5] Which is geometrical parlance for the problem of demarcating the various sub-fields of geometry. The question posed is, given a certain manifold and group of space-transformations, what configurations (or properties of configurations) remain invariant under that group? We can appeal to these invariants when demarcating and exploring a given geometrical sub-field.

[2]Groups of this kind will be familiar to anyone with knowledge of abstract algebra. Because of some especially useful properties, groups are frequently associated with invariances (or "symmetries" as they are sometimes called), but as we will see in later chapters, not all interesting transformation classes form groups.

[3][Klein, 1872], p. 2.

[4]It is worth noting that an implicit circularity is lurking here. Klein is supposedly demarcating geometry, but he proposes to do so in terms of space-transformations which leave the "geometrical properties" unchanged. How can we know which properties are geometrical without already having some understanding of geometry?

This circularity is analogous to the circularity discussed in the case of the forms of physical objects in the introduction, and it crops up in nearly every case of formality we discuss below. Fortunately the circularity is not vicious, can be dealt with easily, and in some cases proves to be enlightening. We resolve the issue in the particular case of Klein and also discuss it more generally in the concluding section of this chapter.

[5][Klein, 1872], p. 4.

We may say even that they are characteristic of the sub-field. "Particular stress," Klein continues, "is to be laid upon the fact that the choice of the group of [space-]transformations to be adjoined is quite arbitrary." So, other than the elements involved in the application of this approach, we can see two important attributes of the resulting demarcations. First, Klein's formulation is essentially relativistic. Whatever it is that turns out to be invariant, it is invariant *with respect to* a given group of space-transformations of a particular manifold. Second, there is nothing constraining our choice of space-transformation groups (other than that it must be a group).

In addition to being useful for demarcation *simpliciter*, Klein draws attention to the fact that these characterizations in terms of space-transformation groups and invariants can be used to identify the relationships between various sub-fields of geometry. Indeed, this is Klein's primary purpose. We can speak most easily about these relationships in terms of the groups of space-transformations involved (though we could also speak in terms of invariants). If we take for an initial starting point standard Euclidean three dimensional space and the principle group of space-transformations, two possibilities present themselves.[6]

The first is that the group of space-transformations may be *contracted* to a subset of the principle group. Call this operation "transformation-contraction." If we then ask which of the geometric properties remain invariant after a transformation contraction (i.e., invariant under some subset of space-transformations of the principle group), we receive a trivial answer. They all do on pain of contradiction — they cannot be invariant under the principle group and not invariant under a subset of the principle group. What we find, however, is that there are now additional invariants. This is intuitively what we should expect. Fewer space-transformations means fewer respects in which the domain is varying, and this means that more properties will remain static or invariant. An alternative way of implementing a transformation-contraction is to stipulate that a particular configuration in space remain fixed (invariant), *in addition* to the geometrical invariants used to define the

[6]Here and elsewhere the descriptive term "Euclidean" applied to a geometrical system primarily indicates that the system includes the parallel postulate. It does not mean that the methods used,or the resulting theory, closely resemble those of Euclid (although they are similar from an axiomatic perspective).

principle group. In a very simple case, we could stipulate that the identity of a particular point must remain invariant. We would then be considering the properties which remain invariant under the subset of space-transformations of the principle group *which keep this point fixed*. It is easy to find space-transformations from the principle group which leave no point unchanged, and these will be excluded by the transformation-contraction. The resulting class of transformations will therefore be a proper subset of the principle group. What we are left with are the transformations (and invariants) that are constitutive of spherical trigonometry — the properties of polygons on spherical surfaces centered on the fixed point. The invariant properties of spherical trigonometry are, accordingly, a superset of the standard Euclidean invariants (and this is what makes Euclidean geometry more "general" than spherical trigonometry).

The second possibility, "transformation-dilation," involves *expanding* the group of space-transformations to include more than those included in the principle group. What happens then? Well, if we consider the case of the principle group alone, we know that all and only those properties invariant under these space-transformations are invariant (this is trivial and obvious). However, if we add new transformations to the group we are looking at, the invariants will still need to satisfy the condition of being invariant under space-transformations of the principle group, because this group is still "in play." The additional transformations must constitute additional conditions on invariants (otherwise they would belong to the principle group and would not be "new"). As a result, some of the properties invariant under the original group, say the principle group, will fail to be invariant under these new transformations. The resulting class of invariants may be said to contract.[7]

Both of these results are intuitively satisfying. In the first case, we are stipulating additional invariants, hence there are *fewer* ways in which the manifold may "vary," and consequently we have a smaller class of space-transformations. In the second case, we

[7]It is tempting to say that this results in "fewer" invariants, but this language could cause trouble when the classes of invariants have infinite cardinalities. It makes sense, however, to speak of "contraction" even when the resulting sets are equinumerous (e.g. the set of even numbers is a contraction of the set of natural numbers, even though they are equinumerous).

add to the group of space-transformations, hence *more* is "varying" in the system, and we arrive at fewer invariants.

The result of all of this is that, once various geometries are demarcated in terms of groups of space-transformations (or alternatively, invariants), the relationships between them can be understood in terms of the relationships between these groups and invariants. This classificatory structure is at the heart of Klein's Erlangen programme.

2.1.2 Invariance In Logic

In recent decades Klein's basic approach has been taken up by a number of figures in philosophical logic. These philosophers have hoped, in the same vein as Klein, to generalize the theory in a way which demarcates logic and explains some of the special features of logic by establishing clear relations to other theories. To this end we will look at the work of Alfred Tarski, Gila Sher, and Denis Bonnay. Although we are primarily interested in their strategies for generalizing invariance, we will also spend some time considering their various motivations, as these have played a role in the formulation of the GTOF. The views are, we may say, intellectual ancestors of our GTOF, and consequently the GTOF can be better grasped by understanding this history.

Tarski on Invariance

One of the first rigorous attempts at generalizing Klein's work on invariance was Alfred Tarski, in his posthumously published lecture "What are logical notions?"[8] As the title indicates, the primary purpose of the work is a characterization of *logicality*, e.g., a demarcation of logic. Tarski's goal in asking "What are logical notions?" is not, he claims, to capture all of the ways "logical" is actually used. Nor is it to capture any "platonic" idea behind the notion, apart from common usage or normative proposals. Instead, he is trying to make a proposal that nearly enough captures one fairly restricted use of "logical." In

[8] [Tarski, 1986]

the sense that it successfully matches it, it can be taken as descriptive, the way it actually is used in this relatively limited context. Insofar as it deviates, it should be taken as a prescription for how it should be used. He is successful insofar as his definition has enough of a theoretical payoff to justify whatever "damage" is done to the intuitive notion.[9]

Tarski is also not trying to give a complete story about what logic is. Logic, after all, involves other things like truth and consequence, which he has discussed elsewhere. The goal is to provide an account of which "notions" are logical, and what it is that makes them so. By a "notion" Tarski means something like "candidate for semantic denotation."[10] The goal is a *principled* demarcation between those notions which are logical and those which are not. It should be no surprise, given their similarity of purpose, that Tarski should invoke Klein to this end, saying that he will "try to extend [Klein's] method beyond geometry and apply it also to logic."[11]

The most significant aspect of Klein's work for Tarski seems to have been Klein's claim that invariance under a specific class of transformations is the characteristic property of the notions of a specific theory (it determines these notions), and that this class of invariant notions can be expanded or contracted by considering transformation-contraction or transformation-dilation, respectively.

Having acknowledged these points, Tarski directs us to consider what he takes to be the limiting case — the class of *all* transformations.[12] This would presumably result

[9] And for the most part, resistance to Tarski's account, and the demarcation of logic in terms of invariance more generally, has been founded on its perceived mutilation of what it means to be logical. To some extent these objections come from a very different perspective than Tarski's and exhibit conflicting interests. Tarski, as a mathematical logician, has an interest in definitions which are theoretically useful. If, on the other hand, one has an interest in accurately capturing the meanings of "logic" and "logical" for a given community, then it is not surprising that Tarski's conclusion will be unsatisfying. See [Bonnay, 2008], [Etchemendy, 1990], [Feferman, 1999], [Sher, 1991], and [Sher, 2001].

[10] This is essentially what MacFarlane, following Nuel Belnap, terms "presemantics" in [MacFarlane, 2000].

[11] [Tarski, 1986], p. 146. It is worth noting with regard to the present work that Tarski also recognizes the possibility of similar treatments of other sciences.

[12] Although it turns out that Tarski implicitly limits the scope of "all" in this context, we take up the issue of whether anything might be invariant under truly *all* transformations, and correspondingly what happens if we stipulate that *all* notions remain invariant in Chapter 5.

in the fewest and most general notions remaining invariant (not in the sense that all other notions fall under these notions, but in the sense that they are invariant for all objects under all transformations). It is these notions, resulting from the generalization of Kleinian invariance to "all transformations" that Tarski proposes are the logical notions.

According to Tarski, what makes this suggestion reasonable is that it leads to consequences that are in accordance with our sense of logicality. It turns out to be the case, as proved in [Lindenbaum and Tarski, 1983], that all of the notions denoted by the fixed constants of standard first-order logic are identified by this criterion. To see this, we can proceed stepwise through the hierarchy of types and ask which notions satisfy the criterion. At the fundamental level, that of individuals, there are no logical notions as there will always be a transformation from a given individual to some other individual. On the next level, classes of individuals, we have two notions: the universal class and the empty class. Of binary relations, we have self-identity and diversity, as well as the universal relation and empty relation. We may continue up through n-ary relations, for each of which we will find only a handful of invariant notions.

The next level, classes of classes of individuals, may also be thought of as *properties* of classes. The properties of classes which remain invariant are those which concern the number of elements in their constituent classes. That is to say, they concern the numerical properties of extensions (has one element, two, is finite, infinite, etc.). This is where we find the class which contains (or the property of being) the universal class — the logical notion denoted by the universal quantifier — and other quantifiers. Tarski consequently comments that, though it has been claimed that logic is concerned only with extensions, it is concerned with even less — the numerical properties of these extensions. Importantly, there are very many logical notions in this category (and higher orders have even more). Most of these are not denoted by logical constants in standard formulations of first order logic, and this has led to criticisms that the invariance criterion of formality *overgenerates* — it declares more things to be logical than are intuitively thought to be so.[13] It is at the

[13] See [Etchemendy, 1990] and [Feferman, 1999].

level of relations between classes that the notions associated with the logical connectives — inclusion, disjointness, overlap, etc. — are to be found.[14]

Like Klein, Tarski has his mind clearly focused on his native disciplines, logic and mathematics (he suggests as much in his introduction) and he accordingly fails to point out the philosophical virtues of his account. Some important philosophical questions about logic concern the "special properties" that it has. Logic is classically thought to be *a priori*, necessary, and completely general (in the sense that it is always applicable). Does logic really have these properties? If so, why? Whatever the answers, many philosophers have taken logic to be special in important ways. The logicists hoped to reduce mathematics to logic, thinking that mathematics would thereby have a firmer foundation. The logical positivists looked on logic as a model for other sciences, and thought that if a science could be rendered deductive, like logic, then it might acquire some of logic's special properties (like necessity). From almost any perspective, logic seems special. But what makes it so?

Tarski's account of logical notions as invariant under "all transformations" would provide a nice answer to this question. For Klein, considerations of invariants were always and explicitly relativized to a specified class of transformations. But there are different classes to consider, and though some geometrical theories are subordinate to others, there are many possible relationships (two theories may be peers, for example). But Tarski's account demarcates logic as the theory concerned with those notions invariant under *"all"* transformations. The idea is that Tarski has expanded the class of transformations involved

[14]Tarski then proceeds with his account of how mathematical notions (characterized by the membership relation) may be logical notions or not depending on how we think of things. The question hinges on whether the membership relation is logical or not, and Tarski's discussion thus far has been ambiguous with respect to the theory defining this relation. If we define it in terms of a theory of types, in the tradition of *Principia Mathematica*, then it follows that transformations occur essentially only on the fundamental type. These transformations, however, induce transformations among the higher types. As the membership relation only occurs among non-fundamental types (i.e. it is always of type greater than zero), and all of these transformations are induced, it follows that the membership relation is invariant for all types (where it exists), and therefore is logical. If, on the other hand, we consider first order set theory, the membership relation is a primitive relation specified extensionally in the meta-language that holds between individuals in the domain. But we know that there are a limited number of invariant relations between individuals, and the membership relation is not one of them. Consequently, it would not be logical. [Tarski, 1986], pp. 152-153.

to be so large that the resulting invariants are effectively *unrelativized* or absolute.[15] This has the potential to explain just why it is that logic has many of the special features that it has. The necessity of logic, for example, would follow from the extreme generality of the invariants distinctive of logic. If there are no transformations under which the logical notions fail to be invariant, then truths captured exclusively in terms of these invariants could never be false. The ubiquitous applicability of logic also follows straightforwardly, as, again, the logical notions will remain invariant regardless of the transformations which occur within a particular theory. So the logical constants can be legitimately and unproblematically "imported" into any theory whatever. We can also see how something like *a prioricity* follows as well. Depending of course upon what one means by *a prioricity*, on such an account a logical theory would make certain very general facts about objects knowable without encountering any of those objects.

This is Tarski's account of logical notions, filled with theoretical promise. And if this were all there was to say about invariance, all would be well and we could end our exposition immediately. How could one possibly generalize further than "all transformations"? Unfortunately, the situation is not quite so simple.

If we try to take "all transformations" in something like its widest sense, perhaps as denoting the class of all functions *simpliciter*, then at best we are left with things like "is an individual," "is a property of individuals," etc., effectively the properties of being a certain semantic type or other. Crucially, this class does not include any of what are traditionally thought of as logical operators. The permissible transformations must therefore be contracted in some way in order to result in an adequate class of invariants.

It turns out that, although careful in his use of the phrase "all transformations," Tarski implicitly limits his discussion to "all permutations" — all one-to-one transforma-

[15] Strictly speaking, the resulting invariants would still be invariant relative to a certain class of transformations — the universal class, however a consequence of the nature of this class is that they are also invariant over all sub-classes, and hence every transformation class. This makes them appear invariant *simpliciter*, when in fact they are still invariant with respect to certain transformations. It is difficult to imagine something which is *in principle* essentially invariant, come what may.

tions of the world onto itself. This falls somewhat short of the absolute generality brought to mind by "all transformations," but the restriction does permit the notions associated with the standard logical constants to come out as logical.[16]

Restricting the class of transformations to permutations also leaves open the possibility of a host of relationships between logic and other theories. Logic might turn out to be subordinate to yet a more general theory (via transformation-dilation). If this were the case, the "special" properties of logic still might remain intact. Nothing in our contemporary intuitions about logic precludes the possibility, for example, of other *a priori* truths or of an even more general theory than logic. If logic were subordinate to some theory, in the way that Euclidean geometry is subordinate to topology, then the permutation account still might explain these special properties of logic (although the superordinate theory would also certainly share these properties).

On the other hand, if it turns out that logic stands in a peer relationship to other theories in the way that Euclidean geometry stands as a peer to Lobachevskian geometry, not only would these aforementioned explanatory virtues of the Tarskian account come into question, but insofar as the view is considered convincing on its own merits it would actually cast doubt on whether logic even has these purported virtues. The issue is an interesting one for the philosophy of logic, and it will be touched on at points later on.[17]

In either event, claims that logic has pride of place and attempts to explain its special properties will need better arguments than merely that logic is the most general

[16]In fact, there have been criticisms raised that this position (and other similar positions) *overgenerates* — permitting too many notions to be ruled "logical." The main issue is that permutations occur with respect to the same domain, and hence facts about the cardinality of the domain in question will remain invariant over these permutations. This conflicts with the generally held position that logical truths should not rest on facts about particular individuals (or their numbers). We do not delve into this problem here, as it has already received considerable attention in the literature. [McGee, 1996] first raised the issue and is a good starting point.

[17]The crux of the issue is the identification of the class of all transformations with the class of permutations. This class may well be the largest class of one-to-one transformations, but it is far from clear that it is the most general way of thinking about invariance. See [Bonnay, 2008] for a thorough, detailed account of many of these possibilities (only a few of which are discussed below). Bonnay suggests that it is plausible to construe several of these as being in an important sense more general than permutations.

theory in terms of being invariant over the widest possible range of transformations.

Sher on Invariance

We will next consider the related position advocated by Gila Sher which, though developed independently from that of Tarski, makes a further contribution to the generalization of invariance and its use in demarcating logical constants. What Sher's position has in common with the view of [Tarski, 1986] is that they both hold logical constants to denote notions which remain invariant under a certain class of transformations. Unlike Tarski, for whom the inspiration of his position was the geometrical work of Klein, Sher begins with Mostowski's important work on generalized quantifiers.[18] This work is in turn directed toward more specific questions of mathematical and abstract logic. The informal way of understanding the questions Mostowski is addressing is the following. In [Frege, 1967], Frege provided the foundation for a newer, more powerful logical language. Arguably the most important innovation was his recognition that logic can penetrate deeper into language than the propositional level if we treat predication as a function which takes objects as arguments and maps these to truth values. Coming along with this system, however, are some new constants — the logical quantifiers and, on some accounts, identity.[19] But these quantifiers are, more or less, analogues of certain natural language quantifiers, namely "all" or "some" — the ones occurring prominently in syllogistic reasoning. A language with just these two quantifiers is significantly more rich than is required to account for the valid syllogisms, but it is easy to see that there are other quantifiers used in natural language from which related inferences may be drawn. For instance, from "all swans are white" it follows that "there are white swans" (assuming a non-empty domain). This is valid with the standard quantifiers. But the consequence also follows with the same in-

[18] [Mostowski, 1957]

[19] This, along with the fact that it seems clear that the standard logical connectives differ importantly between the propositional and predicational contexts (in the former they are truth-functional, in the later not), is enough to raise Tarski's demarcation questions. Which of these constants are logical? Are there other logical constants? Etc. Sher comes to the same questions from a different direction.

tuitive necessity and *a priori* character from "most swans are white," or for that matter from "a few swans are white." This suggests the possibility of quantifiers other than the standard universal and existential quantifiers.

The questions Mostowski addresses are prior to whether all of these quantifiers could count as logical. Is there a relationship between these quantifiers? If so, what is it? There is *prima facie* some kind of relationship between them, since they all seem to play a similar semantic role. What Mostowski's theory of generalized quantifiers provides is a characterization of what this relationship is. He explains what it is to be a quantifier and provides a general way to define a predicative (numerical) quantifier of our choosing. Mostowski puts forward his account in a model-theoretic language. Quantifiers are associated with functions from cardinalities to truth values. Informally we can understand this to mean that a quantifier is a function which assigns a truth value to a propositional function based on the cardinalities of its extension and its complement. The existential quantifier, for example, is a function which assigns to propositional functions with non-empty extensions the value "true," and to everything else "false." The universal quantifier is a function which assigns propositional functions with empty complements the value "true," and to everything else "false." With this in mind, we can easily construct a function which assigns the value "true" to a propositional function if and only if the cardinality of its extension is greater than the cardinality of its complement. This function is analogous to the monadic natural language quantifier "most."[20]

Sher's primary concern in all of this is the question of which quantifiers are *logical* quantifiers. She cites two conditions Mostowski gives:

Condition LQ1: "Quantifiers enable us to construct propositions from propositional func-

[20]See [Mostowski, 1957], esp. 12-15. There are a number of interesting results concerning generalized quantifiers. For instance, Mostowski shows that any language containing the universal and existential quantifiers together with any quantifier satisfying a certain condition will be incomplete. This condition is that for every denumerable domain, the quantifier assigns both "true" and "false" to denumerably many extension/complement cardinality pairs (where the complement is naturally always denumerably large). It also follows that a language containing the universal and existential quantifiers and the quantifier "uncountably many" is complete, as shown in [Keisler, 1970].

tions."

Condition LQ2: A logical quantifier "does not allow us to distinguish between different elements of the universe."

LQ1 is primarily concerned with language, and it should be nothing new to anyone familiar with a first order language. LQ2, on the other hand, is directly focused on the associated semantics. Sher's glosses LQ2 is as follows: "[L]ogical quantifiers are invariant under permutations of the universe in a given model for the language."[21] The similarity to Tarski's account of logical notions is obvious.[22]

Sher proceeds into an interesting discussion of Mostowskian quantifiers and how the view can be extended. There is no need here to dwell on most of this discussion. The important point Sher makes is that LQ2 can naturally be interpreted as "invariance under permutations" when we are only concerned with predicative quantifiers. When the view is extended to relational quantifiers (which is syntactically straightforward and practically inviting to do), the semantics get messy quickly. The trouble is that predicative quantifiers concern only subsets of the universe, so permutations are sensible, but relational quantifiers concern subsets of Cartesian products. To put the point succinctly, if we are going to countenance relational quantifiers as logical, then they should satisfy LQ2 above. But in interpreting LQ2, what are the permutations of interest? The universe? Subsets of Cartesian products of the universe? Etc. Which brings us back to the question of what counts as logical. Sher wants an account of "logical" that is extensionally adequate, meaning that however we demarcate the logical terms, whether including relational quantifiers or not, we want to be sure to include everything that is logical and nothing that is not.

This, then, is Sher's starting point for answering the question of logicality, and we can already see how it will diverge from Tarski's at least in that it *begins* by recognizing

[21] [Sher, 1991], p. 14.

[22] Mostowski's work on quantification and Tarski's work on logical notions both share the roots of [Mautner, 1946] and [Lindenbaum and Tarski, 1983], so this is not surprising. The inquiries are very similar, with Mostowski's being slightly more limited in scope.

that there may be something inadequate in "invariance under all transformations." Sher's more robust proposal involves five separate conditions which she takes to be necessary and sufficient for logicality.[23] The first three — A, B, and C — though necessary for the view are not strictly characteristic of the position. A and B essentially explain the syntactic role of logical constants and C stipulates that a constant is defined over models. This is done in a straightforward way analogous with definitions of the semantics and truth for logical languages, with the difference being its extreme generality. The characteristic features of Sher's proposal are the final two conditions:

D. A logical constant *C* is defined over *all* models (for the logic).

E. A logical constant *C* is defined by a function f_c [based on the model-theoretic semantics defined in C] which is invariant under isomorphisms.

We are concerned primarily with condition E. This condition is partly due to Per Lindström, who in the course of further generalizing Mostowski's work suggested that logical operators be viewed as invariant under isomorphic transformations rather than permutations.[24] He did not, however, make a connection between this invariance and the nature of logicality, seeming rather to view it as only an interesting and important feature of logical systems. What Sher contributes to condition E is the idea that it characterizes the formality of logic, and as such that it is a necessary condition for logicality.

As is readily apparent, condition E has the same structure as both the geometric principles of Klein and the permutation account of Tarski. The sense, however, in which Sher's position represents a further generalization of invariance is slightly more complicated than a mere transformation-dilation. This is because, by defining invariance in terms of permutations, Tarski selects transformations which map *the* world onto *itself*. Thus, on this view transformations are essentially "world-bound," which means that only the

[23] [Sher, 1991], pp. 54-55.

[24] See [Lindström, 1966a] and [Lindström, 1966b].

"actual" world is considered — i.e., only a single domain is considered. The appropriate transformations are just those bijections defined for the domain.[25]

An important generalizing characteristic of Sher's isomorphism position is that its transformations are not "world-bound" in this way. The associated transformations include transformations that occur between distinct (yet isomorphic) domains. So, in the case of the relatively limited permutations, it is clearest to think of the domain of the particular "world" as both the domain and codomain of the transformations. On the other hand, isomorphisms require us to think in terms of a domain of set-theoretic structures with the transformations defined on this domain.

This transitional generalization, from a particular domain to a domain of model-theoretic structures, is what facilitates the further generalization of Denis Bonnay. But it is Sher's statement that invariance under isomorphisms characterizes the *formality* of logic that gives motivation to Bonnay's task.

Bonnay on Invariance

The notion of invariance is further generalized by Denis Bonnay, whose impetus is a controversy surrounding Sher's claim that invariance under isomorphisms is an apt characterization of what it means to say that logic is "formal." We will focus directly on the question of formality in logic in the following chapter, but for the present purpose we can say that the controversy has resulted in several competing proposals which endorse different types of invariance (i.e. classes of transformations) as proper demarcations of logical constants and characterizations of logical formality.

Bonnay takes as his starting point a particular consequence of the Sherian position on logicality. The consequence, welcomed by some and deplored by others, is that

[25]It is worth noting that, although this is the predominant interpretation of Tarski in the rest of the literature, the limitation of the position to a single domain has been challenged by Sher as uncharitable towards Tarski. However, even if the view is expanded to include permutations on multiple domains, the nature of permutations means the transformations will still be world-bound in being restricted to each particular domain, and a move to isomorphisms will therefore still represent a significant generalization (although in this case, the move involves a straightforward transformation-dilation).

invariance under isomorphisms "overgenerates" logical notions, effectively rendering the foundational mathematical notions of set-theory "logical."[26] This effectively "collapses" logic into set-theory, since logic turns out to be based on a fragment of set-theory. Bonnay prefaces his generalization of formality with respect to this consequence saying, "Formality is a property of logic that is shared by set-theory and other branches of mathematics: it is not a surprise that taking formality as the starting point of an analysis of logicality yields a collapse of logic into mathematics. A proper analysis of the distinctive feature of logic should take into account the fact that it is even more "content-free" than set-theory."[27] What this makes clear is that Bonnay is examining the role of invariance with the goal of finding a new formulation which takes into account the peculiar characteristics of standard first order logic. "Generalized invariance," as he puts it, is the first step towards accomplishing this goal.

When all is said and done, there are actually two senses in which generalization occurs in Bonnay's work. The less important of these is transformation-dilation, with which we are already somewhat familiar. It is in this sense that Euclidean geometry is more general than spherical trigonometry, and also the sense in which logic is more general than geometry under both Tarski and Sher's accounts. Bonnay presents several criteria for transformation classes that are more general even than isomorphisms in the sense that they are produced by transformation-dilation.

More importantly, Bonnay generalizes the formulation of what we call invariance principles — the criteria by which transformation classes are defined (e.g., principle group, permutations, isomorphisms, homomorphisms, etc.). He both presents a generalized schema for generating invariance principles (on set-theoretic structures) and broadens the class of relations used to construct these principles.

[26] Although it must be stressed that for Sher, the standard arithmetical operators do *not* come out as logical.

[27] [Bonnay, 2008], pp. 10–11. It is worth noting that Bonnay takes up Sher's claim that isomorphic invariance captures the formal characteristic of logic unquestioningly. This is what makes the collapse of logic into set-theory "not a surprise." Other positions on the formality of logic will not necessarily generate this same result. This may well weigh in favor of Sher's position, but Bonnay does not himself explain his acceptance of the position.

Bonnay's concern with the demarcation of logic directs his attention primarily to invariant operators. The details of these operators, though of importance to questions of logicality, make no difference in understanding how the generalization works. What does matter is that Bonnay takes these operators to define classes of structures. "For any operator Q and for any structure M," writes Bonnay, " we will use the notation $Q(M)$ to indicate that M belongs to Q."[28] It is then these structures which are related to one another by various similarity relations. Bonnay continues, " A *similarity relation* S is a relation between structures respecting signatures (*i.e.*, S is a family of relations S_σ between σ-structures for all signatures σ). The notation is MSM'."

An operator Q, then, is invariant under a similarity relation S, or S-invariant, if and only if for any structures M and M', where MSM', $Q(M)$ if and only if $Q(M')$. In language we are more familiar with, the relation S is an invariance principle which designates a class of transformations. These transformations are defined on the class of structures (e.g., they are transformations between structures). An operator, Q, is invariant under the transformations defined by S if and only if every transformation from a structure containing Q terminates in a structure also containing Q and every transformation from a structure not containing Q terminates in a structure not containing Q — that is, the transformations leave the class of structures containing Q unchanged. It is then Bonnay's task to find a relation S such that "an operator Q is logical iff it is S-invariant."[29]

The class of similarity relations Bonnay is considering includes all of the transformation classes so far considered — permutation, isomorphism, and groups of space transformations — and many others, including several proposals that have been put forward elsewhere in the literature.[30]

[28] [Bonnay, 2008], p. 11.

[29] [Bonnay, 2008], p. 12.

[30] One of the more important such proposals is that of Solomon Feferman, which makes use of homomorphisms. The class of invariants resulting from the use of homomorphisms is contracted from that resulting from the use of isomorphisms. This is as we should expect, since the class of isomorphisms is a proper subclass of the homomorphisms. Importantly, the identity operator, "=," along with all numerically related operators (which are invariant under isomorphisms) are *not* invariant under homomorphisms. Hence, the debate over whether to use isomorphisms or homomorphisms is in part a question about whether or not

Other examples presented by Bonnay include $Univ$, a very weak relation which holds between structures just in case they have the same signatures; $Bool$, which holds just in case two structures have identical Boolean elements; and App, which holds just in case two structures "satisfy exactly the same atomic formulas of a language whose signature is the signature of the structures, and also exactly the same Δ_0 formulas of that language."[31] App is, Bonnay tells us, "the smallest similarity relation respecting functional application." Of these, $Univ$ is by far the most general, so much so that its class of invariants is very small and of little interest for logical demarcation. After proving that these relations form a partial ordering and identifying $Univ$ as the lowest item in this ordering, Bonnay proceeds to search for a relation less general than $Univ$ (or even App) but more general than isomorphism, and which is justified as being characteristic of logicality.

Bonnay makes a few remarks which are worth considering from the perspective of generalizing invariance. "There are two main differences," we are told, "between generalized invariance and Klein's idea of invariance by a group of transformations. The first one is that we lose the group idea. ... The second difference is that elements of a group of transformations were [technical] transformations, namely one-one function[s] from a set onto itself. ... [T]his means that similarity is always induced by a bijection."[32] This is recognition, first, that the relations used need not have a specific structure. They need not define groups or be equivalence relations. The second is the recognition that the schema for creating invariance principles permits relations which are not bijective. These points together amount to the fact that "any kind of relation between structures ... can be used as a similarity relation."[33] This is, for our purposes, the crucial generalization. For, while providing and discussing relations like $Univ$ and App is certainly useful and interesting from a purely logical perspective, it is the suggestion that *any* relation can be used as an

these operators are logical. For the original proposal, see[Feferman, 1999]. For an interesting and nuanced response, see [Bonnay, 2008], pp. 13-17.

[31][Bonnay, 2008], p. 18.

[32][Bonnay, 2008], p. 13.

[33][Bonnay, 2008], ibid.

invariance principle that provides the foundation for a truly generalized account of invariance. Although Bonnay limits his consideration explicitly to relations between structures — no doubt a result of his focus on logical demarcation — there is no principled reason for this restriction. In the following section, we develop a truly generalized account of invariance based on the idea that *any* relation whatsoever may serve as an invariance principle to define a class of transformations. It is this account of generalized invariance that lies at the heart of our GTOF.

2.1.3 Generalized Invariance

We said at the outset that the GTOF essentially holds that formality consists of invariance. Now that we have seen how invariance works and has been generalized in some contexts, it is time to make clear precisely what we mean by this. In doing so, we will introduce some new vocabulary. In some cases this will seem unnecessary given existing technical vocabulary. For example, we use the term "feature" when in many cases "property" might work equally well. We do not do this to be revisionary or in any way disparage existing technical vocabulary. The GTOF is formulated in such a way as to be extremely broad and general, and we have found some existing vocabulary to be implicitly narrow or biased toward a specific domain or way of thinking (e.g., "property" carries significantly more metaphysical baggage than "feature"). In particular, we will be demonstrating applications of the GTOF both at the level of language and of semantics (i.e., notions, presemantics, etc.) and wish to avoid any confusion which would result from using language specific to one domain in applications to the other.

We will take as our starting point the generalized invariance proposed by Bonnay. On this view, there are several elements involved in each instance of "S-invariance." First, of course, there is the similarity relation itself. There is also the class of operators, which remain invariant with respect to the similarity relation. Finally, there is the class of structures which participate in the similarity relations. We can step back from the particularities

of the logico-mathematical focus of Bonnay and think of these elements in terms of the roles that they play in the resulting invariance structure. Perhaps the easiest element to do this with is the class of *S*-invariant operators. From the perspective of understanding invariance (as opposed to the demarcation of logical operators), it does not matter that these are operators. They could be anything. What matters is that they remain invariant. Hence, we can just refer to these as a "class of invariants" or merely "invariants."

We can make a similar move with respect to the similarity relation itself. In the logical context, it is the particular nature of these similarity relations that result in different classes of invariants, but the role in every case remains the same. The similarity relation essentially defines a mapping between structures. From the perspective of invariance, this mapping transforms, metaphorically, one structure into another. Those features which remain intact under these transformations are the invariants. Hence, the role played by the similarity relations is what we will familiarly call a "class of transformations" or just "transformations."[34]

Finally, we come to the role which structures play, which is that they are relata for the similarity relations. Or, in the new parlance we have established, they sit on either end of the transformations. From this abstract perspective, then, the structures really play the role of particulars from which the similarity relations are constructed, and together they form a domain (or a domain and codomain) upon which the transformations are defined and act. So, in the case of Bonnay, the "domain" is the class of set-theoretic structures, as he defines them, and the "particulars" are the individual structures which participate in various similarity relations.

And so we can say that there are three general roles at play in Bonnay's account of generalized invariance: domains and particulars, transformations, and invariants. It is easy to see that each of the other accounts of invariance above also satisfy this general schema,

[34]It should be clarified here that we are *not* using "transformation" in its technical, mathematical sense, as Klein does. We still use the term, however, because it is evocative of the underlying character of invariance — that it involves constancy over change. And, of course, everything which is a transformation in the technical sense is also a transformation in our sense.

and it should not be surprising that it is from this that we will develop our generalized account of invariance which is central to the GTOF. In the state that it is in, however, the schema merely states roles which must be satisfied in an instance of invariance. It does not say anything about what these roles are like or what types of things can satisfy them. This is the work immediately before us.

Domains and Particulars

A domain is a class, and any class can play the role of a domain. This includes the classes that are familiar domains in many theories — sets and proper classes, spaces, integers, complex numbers, physical objects, etc. — as well as less familiar classes. These include, in particular, classes whose members are not conventionally thought of as objects *per se*. One could have a domain consisting of properties (e.g. colors), which would otherwise be considered *properties* of objects, not objects themselves. Other domains might consist of sentences, expressions, thoughts, propositions, or emotions. We could even consider domains of mythical beasts, fictional detectives, or other non-existent "objects." Gerrymandered domains, like that consisting of a unicorn, the color white, anxiety, and this very sentence, might even be considered, though they may fail to be useful for anything. In short, all classes are permissible domains within the bounds of our theory of formality. The reason the criterion for domain-hood is so low is that the role is not particularly demanding. All that is required are non-empty classes of items such that mappings can occur between them.[35] Although many of the domains we will consider are actually sets, the role domains play within the GTOF does not distinguish between sets and proper classes — that is, the difference is of little consequence. For this reason, and also because there are a number of interesting domains which are not classes, we do not discriminate between domains which are sets and those which are not.

[35]The non-emptiness requirement is implicit in the idea of a transformation, which *must* obtain between two (possibly identical) particulars. The empty set cannot, therefore, play the role of a domain of invariance since *no* transformations can be defined on it. The empty set may, of course, itself belong to such a domain and participate in transformations.

Although it is conventional to refer to the members of a domain as "objects," since many domains we wish to consider do not consist of what would conventionally thought of as objects we will often use the terms "particular" or "element" instead. We will reserve the term "object" only for conventional, physical objects or elements of a model-theoretic domain.

Transformations

The fundamental notion of transformations we will use is that of a mapping between particulars. At a most basic level, this is a further generalization of Bonnay's result — namely that *any* binary relation (not just between structures) is a transformation. On this view, a class of transformations would then be a class of such relations. In principle, this could be as simple as a single relation holding only between a single member of a domain and itself, or a huge number of relations holding between every pair of individuals in the domain. However, for the sake of simplicity we will impose some artificial constraints on our thinking and consider only a very well-understood type of mapping — functional mappings. Since all functions can be recharacterized as binary relations, every such function, including non-mathematical functions and those of mixed type (i.e. from physical objects to numbers), is a transformation. It just happens that every class of transformations we will be discussing either is a function or can be constructed by taking an appropriate set or class of functions.[36] The domains and codomains of the functions which define these classes of transformation are domains of the type defined above. A "class of transformations" is just a class containing one or more of these functions.

[36]This is possible in nearly every case, since considerations of invariance are themselves insensitive to the particular character of a transformation. What matters is whether or not a transformation occurs between a given ordered pair, not how that pair was defined.

Invariants

The central focus of our theory of formality is on invariants and their relationships to classes of transformations. Just what can count as an invariant is quite broad and there are no familiar ideas we can adopt from other theories to help us understand it.[37] Generally speaking, however, most invariants are easily thought of as properties or in a way similar to properties (e.g., operators having an extension in a structure).[38] For example, invariants are generally the sort of thing attributable to a particular from the domain, or perhaps even to the domain itself. There is, furthermore, also no limit placed on the order of an invariant. A property of a relation between properties of particulars could be an invariant, for example. In principle there is no way to specify what can and cannot satisfy the invariant role. The role itself places no constraints on what can count as an invariant other than that it is invariant over transformations.

The category of potential invariants or "features" is a large category indeed, and the question may arise of whether just anything, quite broadly, might be an invariant. While this is a topic discussed in further detail in Chapter 5, we can say here that the answer is affirmative, with the qualification that the choice of domain places considerable constraint on which features turn out to be salient. If, for example, we are considering numerical transformations, "corporeality" will likely not be salient. It may be so if we consider transformations from physical objects to numbers (though as such it is unlikely to be invariant). These constraints will, in most cases, prevent us from drowning in oceans of features.

The constraining effect of domain choice stems from a kind of ontological priority

[37] Although there is the closely related notion of "symmetries" in physics, this idea is far from "familiar" to most and is nearly always understood in a mathematically ladened way that it is not useful in this context. This does not mean that such symmetries are not invariants, merely that they are not helpful in explaining invariance in a generalized way. For more on symmetries from a philosophical perspective, see [Brading and Castellini, 2003].

[38] That is, as long as the property is not at the same time playing the role of a particular. In such a case, the property could not be an invariant, though a property or feature of it may well *have* invariants under some transformations. For example, the color orange may have the feature of being more similar to red than to purple.

the domain and its particulars have over features, in that the domain must be determined prior to any consideration of which features are salient (let alone invariant). On the other hand, features may be thought to have the ontological upper hand in a way that will prove significant later on. There is, as we have seen, a certain flexibility in what can count as a particular, and considerably more flexibility in what counts as a domain. This flexibility consists in our ability to construct (or select) domains and stipulate what the particulars will be. At this point, once the domain has been stipulated together with its particulars, there is no corresponding freedom with respect to features. Our only flexibility is in judging which features attributable to the particulars or the domains are interesting for our purposes. The difference between the two can be seen in the truth of certain subjunctive conditionals. With respect to the domain and particulars, these could have been otherwise had we so chosen. Once the domain and particulars are stipulated, however, the features, whatever they are, could not have been otherwise.[39]

2.2 The Theory of Formality

We are now in a position to understand invariance, and consequently the GTOF, more rigorously. For a given case of invariance, we begin with two domains, D_1 and D_2. These domains need not be distinct, and in many cases they will not be. The domains consist of particulars p_{i_j} such that $\{p_{1_1}, p_{1_2}, \ldots, p_{1_n}\} \in D_1$ and $\{p_{2_1}, p_{2_2}, \ldots, p_{2_n}\} \in D_2$. We can then define a class of transformations, T, consisting of functions $f_1, f_2, \ldots, f_n$ from p_{1_j}s in D_1 to p_{2_j}s in D_2. A feature h is an invariant with respect to $D_1$and D_2 under T if and only if the functions in T have no effect on h. Just how this is understood depends, in part, on the types of features being considered. For example, let the domains consist of physical objects, and let a feature h be a property which some of those objects have

[39]This appears to be a kind of necessity, but we are reluctant to invoke this term as it might lead to confusion elsewhere. In particular, that a certain feature is ascribed to a certain particular may well not hold in all possible worlds, but it still has the degree of inflexibility described above. Perhaps we can call this a kind of *factual* necessity, since it is not in any way sensitive to choice.

and some do not. Feature h is invariant with respect to $D_1$and D_2 under T if and only if, for every $f_k \in T$, every $p_{2_j} = f_k(p_{1_i})$ has feature h if and only if p_{1_i} also does. That is, every function in T only ever maps objects having h to objects with h, and objects without h to objects without h. The example can be easily expanded to include all other types of particulars and their features. It can also be straight forwardly extended to relations, features of features, and so on.

These are the most common and important cases, and it is best to leave characterizations of other kinds of features until they are needed, but it should be clear more or less how this can be accomplished. For the sake of brevity, when $D_1 = D_2$, we shall simply call the domain "D," and shall also say that h is invariant with respect to D under T. In fact this situation is so common and important that in the abstract we will proceed to speak only of D when this will not cause problems. Finally, the class of features invariant with respect to D under T we will generally designate "I."

This characterization of invariance contains an important dynamic that may not be *prima facie* evident. As written, it implies that when D and T are given, the class of invariant features I is determined. This is an effect stemming from the factual determinacy of features discussed above, namely that the facts about features are determined simply by the specification of D (having the particular members that it does). Specifying T does not *change* anything about these features, but merely functions to select which features belong to I. Different choices for T will select different invariant features, and each T will always select exactly the same I, for a given D, on account of this fixity of features.

Because of this fixity, the same dynamic works in the opposite direction. An alternative to specifying a given T is to instead *stipulate* some features as invariant, as belonging to I (this happened earlier with the move from Euclidean geometry to spherical trigonometry). We then define the corresponding class of transformations T in terms of these invariants, namely that T consists of all transformations under which the selected features remain invariant. This may appear trivial, for if formality consists in invariance under a class of transformations and the class of transformations itself is defined by stip-

ulating invariants, then we should not be surprised when the selected features turn out invariant over that class of transformations (which they were used to defined). If so, this would reduce formality to stipulation, and thereby convention - hardly a novel proposal.[40] Fortunately, invariances are rarely if ever isolated, and in most cases our stipulations will only characterize a proper subset of I. The invariances used to define T will belong to I, that much *is* trivial, but I will often include additional features which are co-invariant with the original stipulated invariants. Owing to the fixity of features, this co-invariance is a matter of *fact*, not stipulation, and hence the result is not trivial at all. The most convincing case for triviality would be if the stipulated invariances happen to be the *only* features to remain invariant under T. But the fact that the stipulated invariants exhaust I is usually something we did not know beforehand. It is trivial only in a limited sense, and it will rarely, if ever, be obvious and uninteresting.

There are many interesting questions to be asked about the relationship between T, I, and in some cases even D. Might one of T or I be empty? Could we hold *every* feature invariant, and if so, what class of transformations would be so defined? What are the implications of the role played by stipulation (either of T or I)? Some light will be shed on these issues throughout, as we engage in applications of the GTOF, and we will take them up more directly in Chapter 5.

Given this account of invariance, it remains to be described what the connection is to formality. One possibility would be to say, in the spirit of Tarski, that formality, or what is genuinely formal, is that which is invariant over all classes of transformations for all domains. This is similar to the sweeping claims made on behalf of logic over the last few centuries concerning its necessity and preeminence over all scientific theory. Although we have good reasons for thinking that first order logic does not hold this status, perhaps the resulting "most formal" theory would be considered even more "fundamental" than logical theory. While we would certainly be interested to learn of a theory with such special

[40]Early in his career Rudolf Carnap championed such a view, which was echoed by many other logical positivists. See [Carnap, 1937].

invariance characteristics, not only does it seem difficult to prove such a case for *every* class of transformations on *every* domain, but it is also unnecessarily restrictive. Just as we wish to propose a theory of formality which is independent of *logicality*, why should we think that formality must be identified with any other particular class of invariants against which everything else is defined as "non-formal."

Instead we take the following perspective. Invariance, as we have defined it, is relative to a domain and a class of transformations on that domain. It makes sense to us to say that formality comes in "types," and that a given type of formality is determined by the choice of domain and class of transformations with which it is associated. In the case of many theories generally considered "formal," this choice of domain and class of transformations is definitive of a field of knowledge or discipline of research, as is the case with logic and geometry. In such cases we will call the invariances associated with that field, for example logic, constitutive of "logical formality." According to our GTOF it is a type of formality in virtue of consisting of invariants of the character described above, and it is logical in having as its subject the domain and kinds of transformations of interest to logicians. Not all types of formality can be so easily named, but the underlying idea remains the same.

A consequence of this perspective is that the GTOF might just as easily be called a "general theory of *types* of formality," since the theory is a proposal for a core characteristic of various types of formality. But this does not make our GTOF any less a theory of formality, since what we are proposing is that this core characteristic is *what it is to be formal*. This is the level at which we intend to explore the theory of formality, showing that many things conventionally considered "formal" are actually built upon invariants of the described sort, and hence upon distinguishable types of formality. It will be useful to have in hand a common framework by which to express these types of formality, and one is directly suggested by our generalized characterization of invariance. We should be able to formulate an "invariance principle" for each type of formality which satisfies the following "invariance schema":

Invariance Schema: A type of formality $\mathfrak{F}$ consists of the class of features I which remain invariant with respect to (possibly identical) domains $D_1$and D_2 under the class of transformations T (which may itself be defined with respect to some fixed features).

Our first major task will be to apply the GTOF to standard first order logic to verify that the GTOF holds logic to be formal (as it ought) and to make clear that the GTOF makes a distinction between logicality and formality. We take this up in the following chapter, but it seems useful to show how in general we can go about this by first applying it in the case of geometry.

2.3 The Formality of Euclidean Geometry

Showing that geometry or any theory either characterizes or is grounded on a type of formality according to the GTOF requires first identifying which elements play the role of domain, particulars, features, and class of transformations, and secondly explaining why these identifications are appropriate (e.g., that they do not violate the corresponding definitions). We then need to formulate the appropriate invariance principle. In some cases, we will be able to proceed further to indicate how the particular type of formality in question is related to other types of formality in terms of transformation-contraction or transformation-dilation operations.

In Klein's account of Euclidean geometry, the obvious candidate to play the role of domain is the space (or the manifolds thereof). If we take as particulars the points which make up the space, then the space is clearly a set of such points, and thus the space and its various manifolds satisfy the definition of a domain. As Klein describes transformations as mapping the space onto itself — by this meaning that the transformations map every point onto another point of the same space — points are clearly the appropriate particulars (as opposed to lines, planes, etc.). From this we know that the domain and codomain are the same for all of the functions. Since these "space-transformations" are functional mappings

between particulars in the domain, they are transformations in the sense we have defined above.

But Klein is not interested merely in transformations of the space onto itself. He is instead interested in the effect these transformations have on something he calls "configurations in space." These are just loci of points, or regions of space which satisfy certain properties. These configurations are therefore composed of points, and as a result cannot be features of points, *per se*. Spatial configurations are instead best thought of as consisting of sets of relations between points. The spatial configuration of a line, for example, can be characterized in terms of a three-place collinearity relation. Now each of the pointillistic transformations with which Klein is concerned will also induce a transformation on the relations in which those points participate. For example, a relation R between points p_1 and p_2 is preserved under functional transformation t if and only if $R(t(p_1),t(p_2))$. A region consisting of certain points P_1 will be effectively transformed into the region consisting of the points P_2 to which the original P_1s are mapped if and only if the points are mapped in such a way as to preserve the relations between points which constitute the region.

The focus, further, is really on which features of these configurations are left untouched by the underlying transformations of points. In principle, some configurations might remain invariant. This would happen if the class of transformations contained only the identity function (in which case all configurations would be invariant). In most cases, the invariant features will be features of the configurations, and hence features of sets of relations between particulars.[41] In the case of Euclidean geometry, Klein is concerned with the principle group, which is clearly an appropriate class of transformations. This puts us in a position to produce the requisite invariance principle:

Euclidean (geometrical) formality consists of those configurations in space

[41] Above we did not present a general picture of how relations, features of relations, features of features, etc. could remain invariant. Generally speaking, transformations of the particulars in the domain induce transformations in relations, features, features of relations, etc. If one of these more complex features is invariant with respect to all functions induced by a given class of transformations, then the complex feature is also invariant under that class of transformations. Examples of how this works in the case of a simplified domain can be found in [Sher, 1991], pp. 72-73.

> which remain invariant under transformations from the principle group.

We arrived at this principle by a simple substitution of the invariance schema. The domain, D, consists of the space. The transformations T consist of the principle group. The resulting invariances constitute the type of formality, $\mathfrak{F}$, which is Euclidean. From this point it should be apparent how additional invariance principles could be produced for various other subdisciplines of geometry, substituting the appropriate transformation classes.

So, it looks as though geometry is a paradigmatic case of formality according to the GTOF, which should not be surprising given its intellectual ancestry. There is still, however, the matter of the apparent circularity noted above. We know that the type of the formality, which in this case is euclidean, is determined by the specified class of transformations (once the domain is fixed). If we want to know more about geometrical formality, we should ask how the class of changes was defined. In the case of the principle group it was defined as those transformations which leave the geometrical properties unchanged. For example, they leave the ratios between the sides of polygons invariant, though the actual length of these sides may change. The measure of angles is also always invariant under these transformations. A quick substitution in the invariance principle above gives us a new principle which makes the apparent circularity explicit.

> Geometrical formality consists of those configurations in space which remain invariant under transformations which leave geometrical properties invariant.

This certainly looks problematic, but we should be wary of labeling every circularity as "vicious." In this case it is reasonable to think that a class of transformations and the resulting invariants are received as a "package deal." This is a consequence of the determinacy of features discussed above. It is also why transformations or invariants (or both) may be taken as the characterization of a type of formality. Provided associated classes of transformations and invariants, we could *in every case* define each in terms of the other. This will always allow for a substitution resulting in a circularity — a class of invariants defined by a class of transformations which was defined by the class of invariants. There

is no inconsistency or regression here. Neither is there a serious problem about how the circularity "gets off the ground." We can simply start by selecting (arbitrarily or otherwise) either a class of transformations or invariants. The true source of discomfort comes from the freedom involved. Why select one class of transformations (or class of invariants) rather than another? Why the principle group rather than some other? Why call anything the "principle group"?

This deeper difficulty affords a nice opportunity to explore the idea of a "type" of formality in a way which is illuminating. This freedom is implicit in the GTOF — for any given type of formality, many different classes of transformations could have been selected. But we do not really believe that Klein selected the principle group in a random manner. Although anything may be permissible, not everything is beneficial. Klein had a particular theory in mind (or really a class of theories) with which he was antecedently familiar. Even if Klein's antecedent notion of "geometrical" involved the principle group or the geometrical invariants explicitly, at some point in history geometers came upon one or the other of these classes on account of some interest apart from studying the classes themselves.[42] We select classes of transformations out of a tremendous number of possibilities, usually on account of particular interests and purposes preceding, and often instigating, the inquiry itself.

This perspective allows us to engage in some speculation on how a Kleinian story might go.[43] Consider the historical beginnings of geometry (actual or imaginary). It is plausible to think that such inquiries began with basic interest in *shapes* and their interrelations (in particular we might think that different shapes can be composed out of the same basic shapes, etc.). Now we notice, so immediately that we may take it for granted, that objects of the same shape can have different colors, or that changing the colors of objects

[42]Of course the reality is likely much more complicated and almost certainly involves a gradual development of the class of geometrical properties. The point is that this development did not originate from an understanding of the principle group. This story gains plausibility from the historical fact that many geometrical properties were known long before group theory came into being.

[43]Tarski offers a similar story in [Tarski, 1986]. Though ours was developed independently, we are happy to credit Tarski with the original idea.

does not change their shape (e.g., by painting or dyeing). It is immediately apparent that we have invariance entering into the picture. This is not problematic precisely because we have defined neither color nor shape in terms of invariance with respect to the other. It would surely be a problem if we did. Instead, we developed a non-explicit understanding that the class of properties called "shape" (whatever it may be) is insensitive to changes in the class of properties we call "color." It is also not *prima facie* obvious that shape and color should be insensitive to one another. This is at least not obvious for (macroscopic) physical objects. It might have turned out to be true that changes in shape corresponded sometimes to changes in color. Something like this is true if we count micro-structure as "shape," since changes at this level can result in changes of perceived color.

At any rate, we can see how simple experience with a subject matter might lead us to develop intuitive notions about what is distinctive of a discipline. This story, or a similar one, could easily be applied to the Kleinian situation to resolve any discomfort the circularity may cause. There is an intuitive notion of what is euclidean which says that it is those things which are insensitive to changes in location, size, and handedness. These intuitions might be wrong (as has turned out in some cases), but they give us something to work with.

We may then take these intuitions about the interplay between certain features and transformations and explore them. They face the tribunal of experience, so to speak, and thereby become more refined and specific. These "working" intuitions merely *hypothesize* that some features are insensitive to certain changes. They are doxastic, and as such lend themselves not only to questions of empirical adequacy but also to being incomplete. There is a high probability that other properties not originally included in our intuitive notion of geometry will end up being geometrical, and some that we thought were might not be.[44]

We see this tension between the freedom we have in making determinations of for-

[44]Which is, of course, the same thing as saying that the considered class of transformations might be different.

mality and the perceived "truth" or effectiveness of things which are formal (in particular formal theories) under the guise of asking "Which logic, geometry, etc., is the 'true' or 'right' one?" At least on the present view, this comes down to a question about why we should choose one class of transformations rather than another, and the only answer available appeals to our purposes, interests in, and experience of, a subject matter. We will see this issue surface regularly in the work that follows.

Chapter 3

Formality and First Order Logic

"Form" and "formality" have traditionally been discussed in conjunction with two mostly distinct subjects — shape (geometry) and language. The historical connection between these ways of thinking is loose, at best, and it is not immediately obvious that the "formality" of a language should have much at all to do with the formality of objects or shapes.[1]

In Chapter 2 we considered several views which take invariance in geometry as a starting point and generalize it ultimately to set-theoretic structures. These are prominent examples of what we call "objectualist" views of logical formality. A central claim of such views is that the semantics of standard first order logic is built upon set-theoretic invariance (of one or another kind), and that the particular character of this invariance (isomorphisms, homomorphisms, etc.) can serve as an adequate demarcation of logical *notions*, and hence of the discipline of logic itself.[2] While the proliferation of objectualist views raises impor-

[1]For an account of the transition from a metaphysical to logical focus on formality, and for some arguments that the relation between the two is weak, see [Novaes, 2010].

[2]Within the scope of the present chapter, we always mean by "first order logic" *standard* first order logic. By this we mean the system that is the general focus of mathematical logic, having a language which consists of variable and function terms (with 0-ary functions denoting individuals), predicate symbols, one or both of the standard quantifier symbols ($\forall$, $\exists$), and the standard connectives (or an appropriate minimal set of these — often $\neg$ and $\vee$). In general, we also mean to include identity ($=$), and we will be explicit any time this is brought into question. All well-formed formulae are finite in length, and we assume a standard

tant questions about the nature of logic and prods the logical community to seek consensus on just which operators they are willing to countenance as "logical," objectualist views are interesting as a whole in that they all give semantics a very prominent role in determining the overall character of the language. Among the views we have considered this is particularly true for Sher and Bonnay, who explicitly characterize "logical formality" in these objectual terms, and in so doing make a connection between logical formality and geometrical formality.

Furthermore, if objectualist semantics can genuinely serve as an adequate account of logical formality, there is an added advantage for our task in that it should be easy to show that this "logical formality" is a type of formality according to our GTOF. We undertake this task in the following section, but since the GTOF is effectively a generalization of objectualist views, the conclusion follows intuitively. Hence every objectualist account of first order logic embodies an instance of *some* kind of formality.

The question before us might then seem to be, "Which objectualist view characterizes *logical* formality?" And from the perspective of objectualists arguing with one another this is effectively what is at stake. Unfortunately, the situation is not quite so clear as objectualist accounts tend to make little fuss over characteristics of first order logic related to the fact that it is a language. To be fair, of course all of those who are objectualists about the semantics of the language recognize that it *is* a language. However, the *language* of first order logic does not consist of set-theoretic structures, operations on structures, or invariants under operations on structures. It consists of symbols, syntactic rules, and derivational rules, together with a semantics. When the objectualists characterize logicality or logical formality in terms of denotational semantics, they often either spend little time discussing the syntactic characteristics that have historically been cited as hallmarks of logicality, or they take for granted that the syntactic features arise from (or

interpretation of the language in terms of satisfaction over standard models with a non-empty universe. What we have to say about proof theory applies equally to standard axiomatic and natural deduction approaches. Although much of what we say is also applicable to a broader range of first order systems, we will hold these considerations to Chapter 5.

supervene in some way on) the underlying objectual formality of the semantics. But before objectualism was conceived, philosophers and logicians were not at a loss as to why the language was "formal." Formality was even seen as something essential to the language, such that first order logic would be formal even if uninterpreted, *without any semantics at all*.

Our contention here is that those who provide objectual accounts of first order logic *do* provide one way in which the language might be deemed "formal." However, it also turns out that logic may be considered formal on purely syntactic grounds, and that a straightforward reduction of this "syntactic formality" to "objectual formality" is not possible. The relationship between the two is somewhat weaker, in that while objectual formality does not *determine* the syntactic characteristics of first order logic, it imposes significant *constraints* on these characteristics. Much stronger constraints are imposed by the inclusion of derivational rules, which as we will see straddle the syntactic-semantic divide.

To answer the question of what constitutes logical formality, we take a closer look at the relationship between syntactic and objectual formality. The result is a fair degree of confidence in the claim that first order logic comes out as formal according to our theory of formality, and that objectual formality is in some sense primary. In addition, an interesting conclusion is raised regarding the nature of formal languages more generally, and this sets the stage for Chapter 4.

First, however, we should take a brief moment to establish the fact that objectual accounts of semantics are grounded on a type of formality according to our GTOF. Given that the GTOF was developed in Chapter 2 as a further generalization of objectualist views, doing so is relatively straightforward and involves no innovation.

3.1 Objectual Formality and First Order Logic

The clearest way to proceed will be to do for Sher's view what we did in the example of geometry in Chapter 1 — produce the relevant invariance schema for this type of formality.

The most obvious element is the class of transformations. This will consist of isomorphisms.[3] These isomorphisms are understood to be functions between set theoretic structures, and so the underlying domain will consist of these structures. It is important to note a point of possible confusion about this. Generally speaking, every set theoretic structure itself has a domain, and the transformations in question can be seen, in part, as functions between these domains. However, none of these domains is *the* domain from the perspective of invariance, as no single one (or two) of these can be used to define the class of transformations (isomorphisms). These transformations are defined on the class of set theoretic structures, and hence the class of the domains of these set theoretic structures. In this case the domain, from the perspective of invariance, effectively consists of set-theoretic domains. In what follows, the unqualified expression "domain" should always be taken as the domain of invariance. When speaking of a domain of objects belonging to a structure, or in any other sense, we will be explicit.[4] From this, we can produce the following invariance principle:

> Objectual formality consists of those features of model theoretic structures which remain invariant under isomorphisms between those structures.

Given the nature of model-theoretic structures, these invariant features will all be functions

[3]Of course, other objectualist views will involve a class of transformations based on whatever is considered by that view to demarcate logic. For Tarski, the class of transformations consists of permutations or automorphisms. For Fefermann it is "strong homeomorphisms." As we have said before, just which is chosen has much more to do with questions of logicality than of formality, although in some cases intuitive reasoning about degrees of formality is brought into play. Reviewing these considerations is beyond the scope of the present work, but the idea of degrees of formality and comparisons of types of formality, more generally, will be revisited in Chapter 5

[4]Nothing of course prevents the use of a set-theoretic domain as an invariance domain in defining some other type of formality, in a different context.

which belong to the particular structures (but are, of course, exemplified in multiple such structures).

So, it is reasonably easy to see that any account of "logical formality" based on transformations (or as Bonnay calls them, "similarity relations") between set-theoretic structures will constitute a type of formality according to our general theory of formality. These resulting types of formality are what we have called "objectual formality."

3.2 "Logical Formality"

In this section we turn our focus to the question of logical formality and the justification for thinking it is syntactic or semantic (objectual) in nature. First, of course, we must have an understanding of what it means to hold that logical formality is syntactic in nature. For this we will turn to some of Tarski's earlier work. It will then be relatively easy to see why one would think logic is formal in virtue of syntax. This sets the stage for considering the reasons Sher presents for thinking that logic is formal in virtue of its semantics, which is representative of most objectualist positions. We conclude this section by considering whether the syntax of first order logic reduces to or is entirely determined by its semantics.

3.2.1 Syntax

It is exceedingly rare for a philosopher or logician to explicitly address the notion of "syntactic formality" independently of other considerations. The topic nearly always only comes up during characterizations and discussions of formal languages, and for this reason we will here be considering some notable passages from the seminal works of [Tarski, 1983a] and [Tarski, 1983c], in which Tarski discusses his understanding of formalized languages and formal consequence.

In [Tarski, 1983a] Tarski tells us that:

> [Formalized languages] can be roughly characterized as artificially constructed languages in which the sense of every expression is unambiguously determined by its form. Without attempting a completely exhaustive and precise description, which is a matter of considerable difficulty, I shall draw attention here to some essential properties which all the formalized languages possess: (a) for each of these languages a list or description is given in structural terms of all the *signs with which the expressions of the language are formed*; (b) among all possible expressions which can be formed with these signs those called *sentences* are distinguished by means of purely structural properties.[5]

We are therefore presented with the following two key characteristics of formalized languages:

- Formalized languages are artificially constructed.
- Formalized languages are such that the sense of every expression is unambiguously determined by its form.

What does it mean for a language to be "artificially constructed"? As philosophers and logicians, we are likely in the habit of construing this to mean something like the properties Tarski labels "(a)" and "(b)." This is what is usually meant when we call something an "artificial language." From a broader context, however, it can quickly become unclear what artificiality (however it is taken) has to do with formalized languages. The usual distinction is between "artificial" and "natural" languages, and it is often easier to have a clearer idea of the latter of these. Natural languages, generally speaking, are those languages which have evolved on their own without intentional human intervention (from a linguistic perspective). That is, neither the basic lexicon nor the rules governing composition have been entirely prescribed. We can easily point to a great many such languages. If we understand "artificial languages" as those which are not "natural" in this sense, it becomes difficult to see how the distinction is directly relevant to formalized languages. Although these will certainly turn out to be artificial, so will a great many non-formalized languages.

[5] [Tarski, 1983a], p. 166.

Esperanto is a clear case of an "artificial" language, and many languages we may want to call "natural" have undergone considerable intentional human intervention.[6] Any novice philosopher could think of many possible languages which are artificial but not intuitively "formalized."

In what sense, then, are formalized languages artificial? What Tarski is more likely suggesting is that artificiality is something distinctive of formalized languages in more than just a historical or causal sense. Instead, an "artificially constructed" language is one which has artificiality at its core in such a way as to be a different kind of language from a natural language. Conditions (a) and (b) seem to characterize such a language. We are told by (a) that the language must be presented as a list or description of the signs of the language. (b) implies the presence of rules of formation such that for every expression these rules are either satisfied, and therefore the expression is deemed a sentence, or not, and therefore it is deemed a "non-sentence."

Characteristic (b), however, goes further than the idea of artificiality. Since Tarski has made clear that he is only concerned with "interpreted languages" — languages which have been assigned a semantics — (b) takes on greater significance because the notion of "sentence" takes on greater significance. In an interpreted language, sentences are *meaningful* expressions of the language. They are the sorts of things that, in a logical language, can be true or false. If, according to (b), the sentences can be distinguished from non-sentences by purely structural means, it follows directly that the sense of every expression is determined by its "form" or structure.[7]

Though, as Tarski himself acknowledges, this characterization is neither exhaustive nor precise, we can infer that Tarski means to explain the notion of form by reference to

[6]Examples of this include modern Hebrew, French, and the dialect of Korean spoken in the Democratic People's Republic of Korea. Each of these languages has an associated governmental body intentionally affecting the languages development. The revival of Modern Hebrew, in particular, involved considerable intentional change to what had come to be an exclusively religious language.

[7]In the case of first order logic, the notion of "sentence" should probably be expanded to include all well-formed formulae since, even though not all well-formed formulae evaluate to truth values, they can all be assigned meaningful semantic values. Well-formed formulae which are not sentences can be evaluated to sets or extensions.

the "structure" of sentences. Otherwise, his "essential properties," both of which refer only to such structure and not explicitly to "form," would be essential to a formalized language but have nothing to do with form, and would consequently not qualify even as a minimal account of logical formality.

This is all Tarski says about form in [Tarski, 1983a], but [Tarski, 1983c] says more. Here, Tarski discusses "formalized deductive theories." In these theories "proof" results from the application of minimal rules of inference. "These rules," writes Tarski, "tell us what transformations of a purely structural kind (i.e. transformations in which only the external structure of sentences is involved) are to be performed upon the axioms or theorems already proved in the theory ...."[8] This suggests, again, that the notion of "form" involved pertains to the *external structure* of sentences of the language. And though Tarski rejects the view that logical consequence can be exhaustively explained in terms of deduction or deducibility, he glosses logical consequence by saying, "[W]e are concerned here with the concept of logical, i.e. *formal*, consequence, and thus with a relation which is to be uniquely determined by the form of the sentences between which it holds ...."[9] What is important for our present purpose is that Tarski takes logical consequence to be consequence based on form. Given what Tarski has said earlier in this text and in [Tarski, 1983a], this suggests that logic is formal in the sense that it is concerned with the external structure of sentences. Though it is not entirely clear what Tarski means in calling the structure "external," some help is to be found in [Tarski, 1983b]:

> The description of a language is exact and clear only if it is purely structural, that is to say, if we employ in it only those concepts which relate to the form and arrangement of the signs and compound expressions of the language. Not every language can be described in this purely structural manner.

[8] [Tarski, 1983c], pp. 409-410.

[9] [Tarski, 1983c], p. 414. The excerpt is from a longer passage which plays a role in Sher's reasoning regarding logical formality. Given this, there is certainly room for debate over what Tarski means by "formal" in this context (as opposed to elsewhere). However, we are only here interested in the relation between sentences, which is determined by their form. Since objectual formality does not contain resources for describing the forms of *sentences*, we believe we are justified in our reading of Tarski's use of "form" in the passage. We return to the interpretation of "formal" below.

> The languages for which such a description can be given are called *formalized languages*.[10]

The use of "purely" for "external" notwithstanding, it seems reasonable to think that by all of this Tarski takes the relevant structure to involve the *arrangement* of terms in the sentences in exclusion of other relations. This point is again reinforced in [Tarski, 1941] when, after Tarski has presented the result that "All theorems proved on the basis of a given axiom system remain valid for any interpretation of the system," he adds:

> The common source of the methodological phenomena discussed here is the fact [that] ... we disregard the meaning of the axioms and take into account only their form. It is for this reason that people ... speak about the purely FORMAL CHARACTER of deductive sciences and of all reasonings within these sciences.[11]

This amounts to saying that *form* is what remains when we disregard questions of meaning. This will obviously exclude any kind of relation that is not based exclusively on the arrangement of the symbols of the language.[12]

Most languages have grammatical rules which determine that some combinations of symbols of the language are not sentences. In such cases it is necessary that an expression — a string of primitive symbols (or words) — be grammatically permissible to be meaningful.[13] However, as is well known, grammatical permissibility does not always guarantee meaningfulness — certainly this is so in English. The difference between a

[10] [Tarski, 1983b], p. 403.

[11] [Tarski, 1941], p. 128.

[12] There is a sense, however, in which the invocation of a rule of inference makes a deduction meaning-bound, at least insofar as the constants characteristic of the rule of inference are concerned. We address the important role of inferential systems below. The idea here is that *once the rules of inference (or axioms) have been formulated*, we may disregard all meaning in using said rules. It is true that one can know whether a set of sentences implies a certain sentence without knowing anything about meanings, *even* the meanings of the logical constants. Whether anything is thereby gained is another question entirely.

[13] We readily grant that the situation for natural language is far murkier than we make it appear, and that grammaticality may itself be an amorphous thing. It is nevertheless the case that strings in a natural language can fail to be meaningful on grammatical grounds. One does not explain the failure of "Green green, green green green," to be meaningful in English in terms of vocabulary choice alone.

formalized language and one which is not is what we said above, that for the formalized language satisfaction of the syntactic rules is also sufficient for an expression to be meaningful. While at the outset we phrased all of this in terms of something called "structure," we can now be more nuanced. What the preceding suggests is that in a formalized language meaningful sentential expressions are distinguishable from non-sentences entirely in terms of *the order of symbols in the strings*. This is a result of the fact that the meanings of expressions of such a language are entirely determined by the arrangements of their symbols. And of course, it is the syntactic rules of a language which separate the meaningful sentences from the meaningless strings.

In the case of first order logic, and most other formal languages of note, the syntactic rules make explicit use of particular constants of the language. In first order logic, these are the quantifiers, connectives, and the identity symbol — the logical constants. Such symbols are *characteristic* of the language because the syntactic rules are effectively defined in terms of them. That is, creating a well-formed expression of the language becomes in large part a matter of the location of these characteristic constants within expressions of the language. And since this ordering, as we have just arrived at, is the structure of the expression, it is only a short step toward saying that these constants *constitute* the structure of expressions of the language. This is convenient, if strictly speaking false, since non-characteristic (in the present case non-logical) constants play such a secondary role.

All of this should be familiar to anyone who has studied the development of logic in the 19th and 20th Centuries. It is an important component in the views of a number of important figures of the era, perhaps most notably Hilbert and Carnap. A major difference, though, is that while Hilbert, Carnap, and Tarski all held the view that formalized languages are distinguished primarily in terms of their syntaxes, Hilbert and Carnap (at least earlier in his career) both held hyperbolic forms of the position. It is an important part of their views that logic is *essentially* syntactic. This was particularly true of Carnap. In their explorations of logic they consequently focused almost exclusively on syntax. In contrast, Tarski held a more moderate position in which formalized languages are *at least*

syntactic in nature. In addition to his views that logic also involves semantic notions like truth and consequence, there are compelling suggestions in [Tarski, 1983c] that he also endorsed objectual formality as somehow associated with logical formality.[14]

Finally, it stands to reason that any claim that "logical formality" is (or is constituted of) syntactic formality would appeal to the significant role syntactic rules play in defining the language (especially sentence-hood) and on the role that syntactic form plays in identifying which sentences imply which others, according to the rules of inference.

3.2.2 Semantics

Although we are already familiar with objectual formality, we have yet to see what reasons lie behind the claim that it is (or is constitutive of) logical formality. This is our present task.

Recall that Sher's account of logical formality hinges on what she labels conditions D and E:

D. A logical constant *C* is defined over *all* models (for the logic).

E. A logical constant *C* is defined by a function f_c [based on standard model-theoretic semantics] which is invariant under isomorphisms.

Conditions D and E, claims Sher, satisfy the two desiderata Tarski put forward when defining logical consequence. The first is that logical consequences are necessary, and condition D is supposed to capture this. The second, which Sher labels (C2), is that logical consequence is *formal*. The relevant portion of the Tarski text is the very same included above:

> Moreover, ... we are concerned here with the concept of logical, i.e. *formal*, consequence, and thus with a relation which is to be uniquely determined

[14] It is worth noting that Tarski's influential publications on semantics were strongly encouraged by Carnap, who at that time was considerably later on in his career. An interesting discussion of this is included in [Carnap, 1963], pp. 60-67. Of particular interest are the reasons for and degree of resistance among philosophers and physicists of the time. Much of Carnap's later work on semantics builds on this work of Tarski.

> by the form of the sentences between which it holds, this relation cannot be influenced in any way by empirical knowledge, and in particular by knowledge of the objects to which the sentence X or the sentences of the class K refer. The consequence relation cannot be affected by replacing the designations of the objects referred to in these sentences by the designations of any other objects.[15]

On Sher's account, this becomes:

> CONDITION C2: Not all necessary consequences fall under the concept of logical consequence; only those in which the consequence relation between a set of sentences K and a sentence X is based on *formal* relationships between the sentences K and X do.[16]

On the face of it, this is entirely consistent with a syntactic view about logical formality. It is in spelling out the connection between E above and condition C2 that we see a significant different. Sher explains C2 in the following way:

> The condition of formality, (C2), has several aspects. First, logical consequences, according to Tarski, are based on the logical form of the sentences involved. The logical form of sentences is in turn determined by their logical terms (see Tarski's notion of a well-formed formula in "The Concept of Truth in Formalized Languages").[17] Therefore, logical consequences are based on the logical terms of the language. Second, logical consequences are not empirical. This means that logical terms, which determine logical consequences, are not empirical either. Finally, logical consequences 'cannot be affected by replacing the designations of the objects . . . by other objects.'[18]

Here we see that Sher has included what Tarski says in the second half of the passage above, regarding non-empiricality and insensitivity to the identities of objects, as part of what logical formality means. Whether or not Tarski himself intended this is difficult to discern from the passage. Since, as we will see, there *is* a significant relationship between the syntactic characteristics of logical consequences and the underlying semantics,

[15][Tarski, 1983c], p. 414.
[16][Sher, 1991], p. 40.
[17][Tarski, 1983a]
[18][Sher, 1991], p. 40.

he might just be stating that non-empiricality and insensitivity to the identities of objects are properties that consequences have *in addition* to syntactic (logical) formality. On the other hand, given the context it seems quite likely that Tarski takes these as intuitive consequences of the formality involved — that is, the consequence relation has those qualities in virtue of being formal.

Although Sher's reading of Tarski follows this second line of thinking, the exegetical question is one of mostly historical interest.[19] Even if Tarski did not intend these properties to be part of "logical formality," it *might* still be true that these properties follow intuitively from our antecedent understanding of logical formality. Furthermore, it might be the case that these properties bear some significant relation to the syntactic forms of sentences such that they give rise to, account for, or explain them.

The final step is to explain how this results in condition E, which Sher gives as follows:

> Why don't we ... follow Mostowski's construal of 'not distinguishing the identity of objects' as invariance under permutations ... Generalizing Mostowski, we arrive at the notion of a logical term as formal in the following sense: being formal is, semantically, being invariant under isomorphic structures.[20]

And now we see that objectual formality enters the picture as a more detailed account of what it means for logical consequence to be insensitive to the identities of objects. If we think that such insensitivity is constitutive of logical formality then we can arrive at the conclusion that objectual formality — invariance over transformations between set theoretic structures — just is logical formality.

This line of reasoning finally results in the claim,"This criterion [E] is almost universally accepted as capturing the intuitive (semantic) idea of formality."[21] And so we can see that the claim that objectual formality characterizes logical formality ultimately stems

[19] In [Sher, 1991] Sher cites [Tarski, 1983c] as a source for the position, and reiterates this connection in [Sher, 1996]. She has since claimed we should understand this as an independent proposal.

[20] [Sher, 1991], p. 53.

[21] [Sher, 1991], p. 56.

from the intuition that logical consequences are formal in being insensitive to the identities of particular objects.

3.2.3 The Irreducibility of Syntax to Semantics

We are now faced with two candidates for the title of "logical formality." One of these we have already shown to be a type of formality according to our GTOF, but the other we have not. However, before we engage in a separate inquiry into formality and syntax, we should take a moment to eliminate the possibility that the syntactic characteristics of first order logic, though distinct from its semantics, are reducible to or otherwise determined by it.

What would need to be accomplished to show that the syntactic characteristics of a formalized language can be accounted for in terms of objectual formality? One significant feature of semantic formality is that, while it provides the notions which are denoted by the primitive logical constants of the language (and in addition those notions which are denoted by complex expressions formed exclusively out of primitive logical constants), it does not on its own tell us how these primitive symbols should be combined into expressions of the language. This means that any reasoning showing the syntactic features of logic arise from the objectual formality of the notions denoted by its symbols must appeal primarily to syntactic features of these symbols, not of complex expressions.

Fortunately, we can avail ourselves of some of our earlier considerations to make some headway, as follows:

1. Logical consequences are formal consequences, and as such are determined by the form of the sentences involved. (Characteristic feature of logic.)

2. The form of a sentence is determined by the order of primitive symbols in the sentence, and in particular the locations of characteristic constants within the sentence. (Definition of "form" or "structure" of a sentence.)

3. Logical consequences are determined by the order of symbols — especially the location of characteristic constants — of the sentences involved. (Follows from 1 and 2.)

The justifications provided are straightforward and should be uncontroversial. What the reasoning serves to do is to resolve logical consequence, usually characterized at the sentential level (e.g., between sentences), into a statement about the symbols of the language.

What we need is to find some additional reasoning to bridge the gap between (3) and the conclusion that the syntactic relationship called "logical consequence" is determined by the denotations of the logical constants involved.

Sher has herself suggested a helpful interpretation of Tarski's familiar claim that logical consequences "cannot be affected by replacing the designations of the objects . . . by other objects." Her idea is that the fact that logical consequence is insensitive to the identities of individuals (members of the model-theoretic domain or universe) is *evidence* that is in need of explanation. Furthermore, according to Sher, the objectual formality of the denotations of logical constants is what explains this evidence.[22] Logical consequences are insensitive to the identities of individuals because the logical constants denote notions which are objectually formal (i.e., they are insensitive to these same individuals). If we take the characteristic constants, the logical constants, of the sentences involved in consequences to denote the kinds of invariants which Sher suggests, then we could consider these consequences to characterize laws which hold for these denotations.[23] It would then be in virtue of the fact that these laws are insensitive to the identities of individuals that the consequences are also insensitive.

The virtue of this way of thinking is that it allows us to say the following:

1. Logical consequences are formal consequences, and as such are determined by the form of the sentences involved. (Syntactic feature of logic.)

[22] Gila Sher, 2/6/2008, in conversation.

[23] In [Sher, 1996], [Sher, 1999], and [Sher, 2001], Sher calls these "formal laws," a phrase I avoid only to prevent confusion.

2. The form of a sentence is determined by the order of primitive symbols in the sentence, and in particular the locations of characteristic constants within the sentence. (Definition of "form" or "structure" of a sentence.)

3. Logical consequences are determined by the order of symbols — the location of characteristic constants — of the sentences involved. (Follows from 1 and 2.)

4. Logical consequences characterize laws which hold for the denotations of the characteristic constants of the language.

$\therefore$ Logical consequences are determined by the notions denoted by the characteristic constants occurring in the associated sentences.

$\therefore$ The syntactic character of a logical consequence is a direct result of the objectual formality of the notions denoted by the characteristic constants of the language.

Statement (4) is designed to capture our preceding reasoning about explanation. The first conclusion is what should interest us, as the second just makes its significance explicit. It is also fairly evident that the premises one would use to arrive at this conclusion, if at all, would be (3) and (4). Is this possible?

No. Although there is clearly a significant relationship between a law which holds for the domain of semantic invariance and the corresponding logical consequence, this relationship does not hold *in virtue* of the syntax of the involved sentences over and above the fact that the senses of the sentences manage to capture or express the law. This means that, while the sentences will involve characteristic constants with the appropriate denotations, the actual ordering of symbols in the sentences and the locations of the characteristic constants is under-determined. The correlation is not between the syntactic relation and the law, but between the correlated semantics of the sentences involved and the law.

To make the point a bit clearer, the objectual formality of the notions denoted by the characteristic constants of logic clearly places certain constraints on the syntax of the language. It must, we can say, be capable of expressing the formal laws which

hold between these notions. But this stops short of *determining* the syntax which must be used. Many different formalized languages, with distinct syntaxes, can be used to express the same laws. Notational variation alone can permit two languages to express exactly the same consequence (formal law) but with different syntactic form (although always with the same underlying characteristic notions). What is more, these laws can even be expressed, and indeed proofs carried out, in English or many other non-formalized languages.

So, it is difficult to see how the syntax of a language can be fully determined by its semantics. For all but the simplest, most impoverished languages, the constraints placed by the semantics upon the syntax simply are not strong enough. And so, we will need to pursue an analysis of the syntactic features of standard first order logic independently from the denotations of its characteristic constants. On the other hand, the preceding reasoning may suggest that while the objectual formality of the logical constants does not wholly determine or explain the syntax of sentences, it certainly places considerable constraints on the syntax and explains the presence of characteristic constants within the particular expressions involved. This is significant and interesting, and it is something we explore in the final section of this chapter.

3.3 Formality in Language

We can now turn to an examination of the syntactic features of first order logic and answer the question of whether or not these are a type of formality according to our GTOF. Most of what follows is applicable well beyond the case of first order logic, and this opens up some interesting questions to be explored in later chapters.

Generally speaking the syntactic features of all languages, formal and non-formal, can be divided into two categories. The first of these is a grammar, or what we might call the "syntax of the language." This is what is most commonly at issue when philosophers speak of "syntax," and it essentially consists of two parts: primitive symbols, and the rules of composition for forming complex expressions out of these. In some languages, like

first order logic, certain primitive symbols play an important role in the rules of composition. These constants are then distinctive of the language, and they are what we have been calling "characteristic constants." The second category we will call the "syntax of expressions." This is what we characterized above in terms of the non-meaning-based relations (usually spatial or ordering) between symbols or words within a sentence or well-formed formula of the language.

In the case of first order logic, the syntax of the language consists of the logical (characteristic) constants (connectives, quantifiers, and usually identity), non-logical constants (predicates and functions, including 0-ary constant functions), variable symbols, and auxiliary symbols,[24] together with the general rules about how these may be combined to produce well-formed formulae of the language. Every such well-formed formula has a syntax of expression, as well, which consists of the ordering of its actual component symbols. For example, the syntax of the expression "$\forall xFx$" is that it begins with a universal quantifier symbol, followed by the variable symbol x, followed by the constant predicate F, followed by another instance the variable symbol x.[25]

In addition to these two formal characteristics, some formal languages have a third component related to syntax — a set of derivational rules for manipulating expressions in the language. In the case of standard first order logic (and many other languages) this takes the form of a proof theory, with rules of inference and rules of transformation. Essentially what these rules do is establish a relation on the basis of the expressional syntax of formulae, almost always given in terms of characteristic constants. Thus, in first order logic the formulas "$\forall x(Fx) \rightarrow \forall x(Gx)$" and "$\forall x(Fx)$" bear an implication relation to "$\forall x(Gx)$" in virtue of the positions of symbols within the expressions.[26] At a more general

[24]So called because their only contribution to the language is to resolve ambiguity (e.g., "$A \wedge B \vee C$" becomes "$(A \wedge B) \vee C$" or "$A \wedge (B \vee C)$").

[25]This likely seems more restrictive than what is usually presented. It is more usual to describe the syntax of a particular formula in terms of the syntax of the language. Such hybrid descriptions are certainly more useful, however they make characterization in terms of invariance needlessly complicated. It is best for our purposes to deal with these independently, and later address their combined use (as we do in section 3.3.2).

[26]The implication relation here should not be confused with a *consequence* relation. Implication is a syntactic relation, whereas consequence concerns semantics. However, the completeness of first order logic

level, this third component can vary somewhat from a familiar proof or inferential system, and it is not a component of all formal languages (as is often the case for diagrammatic languages).

We will consider these three syntactic characteristics in turn as they pertain to first order logic.

3.3.1 Grammatical Formality

As we know, the syntax of a language, or the grammar, consists of a description or list of the basic symbols of the language and a set of rules of composition for constructing complex expressions from these symbols. How this works in the case of standard first order logic is well known, so there is no need for a review here, and we can get directly to our analysis of whether the grammar of a language is grounded on a type of formality according to our GTOF. This involves identifying the appropriate elements — domain, particulars, transformations, invariants — and constructing the associated invariance schema.

Given the role of grammar in the composition of complex expressions, two domains present themselves as plausible candidates: the domain of all strings of symbols of the language, and the domain of all well-formed formulae of the language. The second of these is, of course, a subset of the first.

If we take the class of all strings of symbols of the language as the domain, and then choose as the transformation class all functions between individuals in this domain, we certainly will not end up with anything characterizing the composition rules for well-formed formulae. If we think about what might remain invariant under transformations from meaningless strings to well-formed formulae, we can see that we will not arrive at anything like a grammar. The invariants will be limited to features of strings of symbols generally, and as this set presumably includes every possible ordering of primitive symbols, nothing structural will turn out to be invariant except the feature of being a string,

guarantees a correspondence between the two such that conflation is easy and often not problematic. The present context make distinguishing them imperative.

which is neither notable nor useful. This puts us off the mark. We will need to look elsewhere, and the intuitive thing to do is to limit our consideration only to well-formed formulae of the language, since this is what the grammar characterizes. This will induce a restriction, so to speak, on the class of all functions between individuals in the domain, such that the class of transformations only contains functions between well-formed formulae of the language.[27]

Having identified a plausible domain, we can turn to the evaluation of classes of transformations. We have mentioned the class of transformations between all well-formed formulae, but there are other possibilities. One of these is to consider those transformations which preserve "logical form." Since we know that the grammar of first order logic distinguishes well-formed formulae based on their logical form, this class contains essentially only the identity function on sentences.[28] The class of invariants generated by this class of transformations includes, among other things, every well-formed formula of the language. We know that the syntactic formality of the language does not consist in the fact that particular formulae can be constructed in the language, at least not directly. So, this meager class of transformations will not do.[29]

A more likely proposal is to stick with the class of transformations between all particulars of the domain. That is, the class of transformations which maps every well-

[27]There is an additional component worth noting. The inclusion of function symbols in the language means that there is also such a thing as a "well-formed term" of the language which is not itself a formula. If we wish to be very inclusive, we might also add such strings to the domain we are considering Doing so will have little effect, however, as the class of *all* well-formed formulae includes formulae containing all possible well-formed terms.

[28]There is, of course, a small issue concerning the choice of names for variables. We follow [Quine, 1960a] and [Quine, 1976b] in thinking that, whatever the difference may be between $\forall xFx$ and $\forall yFy$, it is not a grammatical difference. From a grammatical perspective, these sentences have the same form.

[29]Under sufficiently idealized conditions, there is an analogous idea for natural languages which is a bit more interesting. If we take the class of transformations that maps every sentence to every sentence with exactly the same characteristic form (e.g., grammatical form in that language), the invariants would turn out to be the possible *grammatical structures* of expressions of the language. That is, the invariants would amount to the possible descriptions of sentences in terms of the grammar, which for natural languages is non-unique (many sentences may share the same description).

formed formula to every other.[30] Which are the features invariant over this class? Certainly the property of being a string of primitive symbols of the language will belong to this class, as this is true of all well-formed formulae. But this time there is more. A very important property is preserved by these transformations, that of being a well-formed formula. At first, this looks trivial; of course they are well-formed formulae! That is how the class of transformations was defined. But from our earlier discussions of syntax, we know that a string of symbols is a well-formed formula if and only if it satisfies the syntax of the language. So, the class of invariants, in addition to containing features relating to string-hood, also contains features relating to the rules of composition, including the satisfaction of the rules of composition of the language. In fact, the class of invariants contains little of substance other than these rules of composition. In addition to string-hood, which is syntactic in nature, it will also contain the fact that each formula consists only of the symbols listed in the definition of the language, but none of these symbols itself is invariant, since the rules of composition of standard first order logic do not require any given constant be included within a particular formula.[31] A less interesting invariant will be that all formulae contain at least one symbol. But this, again, is a consequence of the rules of composition. Many more invariants can be found in this class, but none which are not trivial or uninteresting. What is interesting, however, is that the class seems to include anything syntactic that pertains to the language as a whole, and this result generalizes. If, for example, we consider a language using some form of representation other than strings

[30] As this is the first class of transformations we will work with which is not easily defined by a function type (e.g. isomorphisms), a brief description of how we can define this class seems appropriate. The simplest way to do this is to take the class of constant functions which take any well-formed formula as input and map it to a constant well-formed formula, with one such function mapping to each formula. This will in fact result in a set, since the number of well-formed formulae is denumerable. However, for our purposes the details do not matter, provided that the class of transformations is such that for every pair of formulae, there is some function which maps the first to the second.

[31] Incidentally, the same cannot be said of all formal languages. Many programming languages, in particular, require that a valid statement begin or, more usually, terminate with a specific constant (often ";"). For such languages, constants like these will turn out to be invariant features under all transformations between valid statements. But, of course, this will also be captured by the rules of formation of the language, which are themselves invariant under these same transformations.

of symbols, the rules of composition would still be invariant. Likewise, it seems clear that since any rule of composition of the language is, by fiat, satisfied by every formula of the language, these rules *must* be among the invariants.

This puts us in a position to formulate the following provisional invariance principle for what we will call "grammatical formality":

> Grammatical formality consists of those features of well-formed formulae which remain invariant under transformations between all well-formed formulae of the language.

Given the inter-dependency between invariants and transformations, we can revise this to capture the more interesting point we have just presented as follows:

> Grammatical formality consists of those features of well-formed formulae which remain invariant under transformations which preserve well-formed-formula-hood.

An astute reader will have noticed even before these principles were given that we face a similar circularity to that which we encountered in our discussion of geometry in Chapter 2. We want to characterize grammatical formality in terms of invariance under a class of transformations, but because of the interdependence between invariants and transformations, we appear to need a characterization of grammatical formality in order to identify this class of transformations. That is, it *looks* as though we need the grammar in order to construct the set of well-formed formulae. However, just as before, no contradiction or regress results. Neither is there a theoretical problem of being unable to grasp either the grammatical rules or the formulae first. We could dream up the syntactic rules, for example, and from these generate the well-formed formulae of the language. Likewise, we could write down some strings and, if we are lucky and they are formulae, the grammatical rules would be invariant under transformations between them.

However, in the case of first order logic we are at much less of a loss for how the system can be set up, since for an interpreted formal language, like first order logic, there is an independent, non-grammatical criterion of well-formed-formula-hood, and especially of sentence-hood — meaningfulness. That is, if we allow ourselves access to the semantics of the language, such that we can distinguish meaningful formulae from nonsense, we can see how the invariance relationship might have been constructed. This suggests a conclusion we arrived at earlier from another direction — that the semantics of first order logic play an important role in the particular character of the grammatical formality that constitutes the syntax of the language. We discuss this conclusion directly in section 3.4, but not before we encounter it twice more.

What we should take as the conclusion here is that what we have called the "syntax" of the language, or "grammar," is in fact a kind of formality according to our GTOF. In fact we have carried on in such a general manner that we can say something more general than this. For any system whereby complex objects are constructed out of primitive objects according to a fixed set of rules, those rules are formal with respect to the objects so created. This holds true for all formal languages and also many fragments of natural languages.[32] That is to say, these languages exhibit grammatical formality, and their rules of composition are "grammatically formal."

3.3.2 Sentential Formality

We will now turn our attention to the syntax of expressions. The syntax of an expression, we already know, consists of the ordering of symbols in the expression. What we also know, from our consideration of Tarski above, is that this is what we are left with

[32]The trouble with applying the GTOF to the syntax of natural languages is that natural languages involve innumerable exceptions, deviations, and are mutable by nature. All of these cause problems for the GTOF as presented in Chapter 2 because it assumes an essentially static, exceptionless domain (which works perfectly well for formalized languages). We expect that this way of thinking could be broadened to provide a characterization of "approximate" formality or something of that sort. The issue arises again in section 5.2.3.

when we "disregard" the meaning of an expression. This means that, for any given sentence, the syntactic structure of the expression should be the same regardless of meaning assignments to the constituent parts.

This makes our task reasonably easy in showing that the syntax of expressions is indeed a type of formality. A consistent theme so far, and one that will continue to develop, is that focusing on invariance is essentially a way of systematically disregarding or "washing out" particular aspects of whatever is in focus. Alternatively, it is a way of systematically selecting and isolating certain details (a structure) which we wish to attend to or focus on. In the case of grammatical formality above, what is disregarded are the particular structural characteristics of expressions (by choosing transformations between all sentences) and what is attended to is what we might call the *shared* structural characteristics. From this perspective it seems reasonable to think that the transformations we wish to consider in this context involve the meanings or interpretations of expressions, since this is what is getting systematically disregarded.

There are a number of perspectives from which this might be pursued. We might, for example, take each sentence of the language individually and consider reinterpretations of the symbols of that sentence. This would result in an invariance schema for each sentence of the language, and hence a "unique" type of formality for each sentence. This is a bit more complicated than necessary, however, since these can quite naturally be grouped as a family, and it is likely that we might be considering the same meaning transformations in each case.

Instead we will take a more holistic approach and think in terms of interpretations of the language. For the present purpose, we intend the notion of "interpretation" to be quite a bit broader than how it is standardly understood in logic. We need not, for example, require even that the "interpretation" results in all sentences of the language being meaningful (since meaning is what we are washing out). And importantly, none of the denotations of any of the symbols remains fixed (not even the logical constants, which are usually kept fixed). So, by an "interpretation" we mean the following. An interpretation of

a formal language, in this case standard first order logic, consists of the list of symbols of the language and rules of composition, together with a denotation function which assigns a denotation to each of these symbols.

Our domain consists, then, of ordered pairs consisting of the set of sentences of the language, as defined by the grammar, together with an interpretation. We must also stipulate that the interpretations included among these pairs must be of a sufficiently great variety that every symbol is assigned at least two distinct denotations, but ideally several. The class of transformations, then, will be taken such that every pair is mapped onto every other pair by some function in the class of transformations.

It is intuitively clear that the class of invariants will not contain any semantic feature, other than the relatively mundane fact that the symbols of the language all have *some* denotation under an interpretation (unless we include a "null" denotation function as an interpretation). What remains invariant with respect to the set of sentences of the language? Only that each sentence consists of the symbols that it does in the order in which they occur. This is all. But this is precisely what we mean by the "external" or "pure" syntax of an expression.

With this in hand, we can now define what we will call "sentential formality" in terms of the following invariance principle:

> Sentential formality consists of those features of sentences which remain invariant under all transformations between interpretations.

And so, this is legitimately a type of formality according to our GTOF. There are several issues worth considering at this point.

First, on its own this is a relatively "weak" kind of formality from a practical perspective. It really does not amount to much. In fact, if in constructing our pairs above we had taken arbitrary strings of the language rather than sentences, these would exhibit "sentential" formality as well, since they also consist of sequences of symbols of the lan-

guage. Furthermore, providing a description of an expression in terms of a sequence of symbols, whether the description is meaningful or not, is not a particularly useful ability. This is something of a surprise, seeing as the view that logical formality is syntactic in nature implies that this is what is interesting about expressions. It still follows, however, that consequence relations are determined by this ordering of symbols. The same cannot be said of non-sentence strings, even though they do exhibit structure. The major difference, of course, is that sentences satisfy the rules of composition and are appropriately related by means of a rule of inference. That is, what matters for formal consequence is not merely that a sentence has certain symbols in a certain order, but that having those symbols in that particular order satisfies the rules of composition of the language in such a way that an implication relationship results between them.[33] It is no surprise, then, that the complexity of the functional terms does not involve logical constants.

Fortunately, it is also the case that the grammatical rules of the language also belong to the class of invariants under interpretation transformations. That is to say, the grammatical rules of the language do not change under changes in interpretation, and neither do the facts about which rules are used to construct each sentence. They are, we may say, also "sententially formal." The virtue of this is that it allows us to employ the grammatical rules in providing descriptions of sentential syntax. These descriptions, it should be noted, are something of a hybrid between the purely grammatically formal and purely sententially formal descriptions. Grammatical formality only affords description in terms of which rules were used to construct an expression, but it does not allow reference to any particular symbols from within grammatical classes (other than those explicitly involved in the rules). Sentential formality, on the other hand, permits reference to all symbols in an

[33] Another way to see this more clearly is in contrast with well-formed complex functional terms (of arity greater than zero). Such terms have a structure of their own, but this structure plays no role in establishing consequence relations, for which purpose they are effectively treated as non-composite. Complex well-formed formulae, on the other hand, are generated by logical constants, and it is this resulting structure that serves as a basis for logical consequence. That the internal composition of complex functional terms is also formal (though not logical) is not something we will establish here, but which can be inferred from the discussion in Chapter 4, in particular section 4.2.3.

expression but does not recognize grammatical classes. Hybrid descriptions suffer none of these drawbacks, which we can see by considering the simple sentence "*Pa*", where *P* is a one-place predicate and *a* is an individual constant (zero-place function).

Grammatical: A predicate constant followed by (or applied to) an individual constant.

Sentential: The symbol "*P*" followed by the symbol "*a*".

Hybrid: The predicate constant "*P*" followed by (or applied to) the individual constant "*a*".

It is easy to see that neither descriptions of the first or second sort make it easy to see the implication relations between two sentences. In the case of grammatical descriptions, we have no idea which non-logical symbols are shared between sentences, which is usually necessary to determine a consequence relation. In the case of the sentential description, it is unknown how symbols relate to one another other than the order in which they appear. In the case of first order logic, we cannot even presume that each occurrence of a symbol has the same denotation as every other, since the logical connectives behave differently depending on whether their arguments are sentences or open formulae.[34] It is only by application of the grammatical rules to the ordering of symbols that we have a complete picture of what is going on syntactically within a sentence and, crucially for the case of first order logic, *between* sentences.

It additionally might be wondered why we defined grammatical formality as a distinct type of formality when it is also invariant under interpretation transformations. Should it not be combined with sentential formality? It is first important to recognize that it is not a problem for features to be formal in more than one respect. Why this is so, and how it works, is something we discuss in greater detail in Chapter 5.[35] Whether the

[34]Though of course, if the language were altered to remove this ambiguity, perhaps by providing additional symbols, the condition might be satisfied.

[35]In fact, as we make clear there, most things (and possibly all) are formal in more than one respect.

multiple formal aspects are related, and how, is often an interesting question and one we take up below with respect to first order logic.

The second thing to recognize is that the fact that something exhibits a certain type of formality does not mean that it can be easily grasped or constructed by considerations limited only to that type of formality. This is true in the present case. Assume for a moment that we are unaware of the rules of composition of the language and are merely presented with the set of strings of the language (which of course, contains the set of sentences as a subset). Now, the rules of composition of the language are going to be invariant under transformations between all particulars in the domain of sentences in this case as well. But since we do not know these rules of composition, we must somehow recover them. And we cannot do this without the set of all sentences of the language — we will not know which strings need to satisfy the rules and which do not.[36] We also cannot rely on the semantic criterion of sentence-hood — meaningfulness — because of course we have washed away meaning by means of invariance. If we had a distinction between sentences and mere strings, we could most likely reconstruct the grammar from looking at what is invariant over the sentences. But where could such a distinction come from if not from the meaningfulness or grammaticality criteria? It could not. Even if the distinction was made for us, by some outside source, we would still be relying on that source's semantic or grammatical knowledge, and so would not be in any better position.

This allows us to make a distinction between a feature being "essentially" formal or "incidentally" formal in a given respect. A feature or theory is *essentially* formal if invariance under a certain class of transformations is constitutive of what it is to be that feature or is somehow significant in characterizing it. If this is not the case, it is incidentally formal. In the present example, the rules of composition of first order logic are

[36] A notable variation on this is combinatorial logic, discussed at the end of Chapter 4, in which the only rule of construction is juxtaposition of primitive symbols, and sentences can be of any length. For such a language, every string is a sentence. However, from our ignorant, "original position," so to speak, we would not know this to be the case, since that is grammatical knowledge. We simply do not know whether all, some, or none of the strings are in fact sentences.

essentially grammatically formal but only incidentally sententially formal.[37]

3.3.3 Derivational Formality

The final syntactic component of standard first order logic that we must say something about is the proof theory.[38] In doing this we must be careful to avoid implicit inclusion of semantic considerations, since as we will see, a proof theory actually imposes considerable semantic constraints on a language. Perhaps because of this philosophers have a tendency to speak loosely about proof theory as involving "logical consequence." In one of the passages above, from [Tarski, 1983c], Tarski himself characterizes "logical consequence" as "a relation which is to be uniquely determined by the form of the sentences between which it holds." Now, this can only be taken as a fact about the relationship, not as constitutive of it, since later in that same article Tarski gives his famous semantically laden definition of logical consequence. A few pages before this passage, however, Tarski tells us that the rules of inference and transformation "tell us what transformations of a purely structural kind (i.e. transformations in which only the external structure of sentences is involved) are to be performed upon the axioms or theorems already proved in the theory ..."[39] This, we think, is the proper way to think of proof theory and derivation from the perspective of syntax alone.

Although we wish to exclude semantic considerations, we cannot do so by looking

[37]This point also has bearing on a controversial claim the GTOF commits us to, which we discuss in greater detail in Chapter 5. The claim is that everything can turn out to be formal in some respect (e.g., according to some type of formality). One concern this raises is the possibility that the GTOF cannot fail to proclaim logic to be formal since it proclaims *everything* to be formal. This is certainly not an intuitive result, and our response is to say that our project here is not to show that logic is incidentally formal in some way, but rather to show that it is *essentially* formal in at least one respect.

[38]Treating the derivational theory of a formal language as distinct from its grammar raises an interesting question. Although it is a standard among logicians to define the rules of formation of a language apart from its proof theory, if the primary purpose of the language is the construction of derivations rather than solitary expressions, might not the proof theory merely constitute grammatical rules that hold at the the higher level of statements rather than symbols? Whether this is so may well depend on the purpose and applications of the language in question, and like many things, this is beyond our scope of inquiry. It does seem that, at the very least, there is a convenient practical distinction between the two.

[39][Tarski, 1983c], pp. 409-410.

at invariance under interpretation in the same way we did for first order logic. This is because derivational rules are *not* invariant with respect to changes in the denotation of logical connectives. All that is required to see this is to include among our interpretations one that exchanges the denotations of conjunction and disjunction. Clearly, this will not affect sentence-hood (or meaningfulness, for that matter) but it will result in several of the derivational rules failing to preserve truth. As a result, the structural relationships identified by the derivational rules will no longer correspond to valid consequences, and the resulting theory will be unsound. This lack of soundness, which is taken to be a necessary feature of any logical proof theory, cannot be characterized in syntactic terms since (as our example shows) the difference between a sound and unsound theory can hinge exclusively on denotation. Thus, the requirements of truth preservation and soundness therefore impose a considerable constraint upon the semantic reinterpretation of a language, over and above that placed on it by the grammatical rules. Nothing, of course, forces us to give up or change the derivational rules, even under very wild interpretations, but they will have lost their original purpose as a logical theory and will have unpredictable semantic results.[40] All of this is just to say that the formality of a proof theory, if it is indeed formal, is distinct from the type of formality we characterized above for the structure of well-formed expressions (what we called "sentential formality"). Although a proof theory is clearly sensitive to the ordering of symbols in well-formed expressions, it is also sensitive to changes in denotation of these symbols (which, of course, the order of the symbols is not).

Likewise, neither does proof theory exhibit grammatical formality. If we assume that it does, the following should be the case. We can take all sentences of the language and place them into sets determined by which derivational rules can be applied to them. Just how we do this does not matter. When we consider the class of transformations germane to grammatical formality, it *should* be the case that no sentence is transformed to a sentence which belongs to a different set (or sets). But since every sentence is transformed to every other, this is guaranteed to occur. Hence, the derivational rules are also not formal in the

[40]The syntactic derivations will be just as predictable as ever.

purely grammatical sense.

However, there is a certain analogy between a proof theory and the idea that logical formality is insensitive to the particular character of objects, as there is a clear sense in which derivational rules are insensitive to much of the structure of the sentences on which they operate. This is clear from the fact that derivational rules are not defined for each and every sentence of the language to which they apply, but are instead defined generically for classes of sentences. This is usually accomplished by means of derivational schemata in which Greek letters stand for unknown expressions.[41] Thus, for example, the rule of detachment does not distinguish between the sentence "$P \rightarrow Q$" and "$(\forall xFx) \rightarrow (Fa\&Fb)$," since these each satisfy the structure $\alpha \rightarrow \beta$. That a sentence has this structure is a requirement for the rule of detachment to apply, and the rule applies in exactly the same way to every such sentence. This is what we mean by saying that the rule does not distinguish between these sentences — it is insensitive to substitution of members within the class. The rule is likewise insensitive to differences between sentences to which it does not apply. Of course, it *is* sensitive to substitutions between these classes.

[41] Although we have generally avoided discussion of "variables," a certain note on this topic is owed here. It is common logical parlance to refer to the use of Greek letters in derivational schemata as "metamathematical variables" or just "variables," and it may seem odd that we do not. The reason we do not is that schematic letters do not denote an unknown referent, "any such referent," "all referents," etc. Hence expressions containing these schematic letters are not evaluable from the perspective of the semantics of the language without supplying the appropriate bit of language for which they stand — by "filling in the blanks," as it were. True variables, or what we might call "semantic variables," do not require substitution for evaluation, but rather binding by a quantifier. Furthermore, even if we did bind schematic letters with quantifiers (say "for all sentences"), it is not at all clear that the resulting "derivational rules" would mean the same thing that they do in schematic form, or even be part of the language in question (rather than some metalanguage). At any rate, the main difference from our perspective is simply this: schemata pertain to language — schematic letters are blank spaces to be filled with bits of language; true variables pertain to semantics — they are the sort of thing that must be bound (implicitly or explicitly) by a quantifier and rely on this quantifier for their semantic values. This distinction is similar to that made by in many places by Quine, but most clearly in [Quine, 1986], pp. 66-68. A major difference is that Quine considers predicate letters to be schematic, and this is one reason for his rejection of second order logic, whereas we are focused here only on schematic letters occurring in explicitly metalinguistic derivational rules. Developing a position on the issue of predicate letters and second order logic would be out of place here, but the GTOF ultimately suggests a more-or-less pluralistic response — predicate letters can be taken either as schematic or as bindable variables depending on the application of the resulting theory (and the underlying semantics). See also [Quine, 1981], pp. 165-168, and [Quine, 1976b].

In general, a derivational rule will not distinguish between any n-tuple of sentences which satisfy it, or between any n-tuple which does not, but it *will* distinguish between these two groups. That is, a derivational rule is either applicable to a sentence (or set of sentences), or it is not. It is not capable of making any further distinction than this, since it functions the same way for every sentence (or set of sentences) to which it applies, or not at all for those to which it does not. And all of this *does* suggest that derivational rules are somehow invariant with respect to transformations defined on sentences, but more structure seems needed in defining these transformations.

Let us focus for a moment on just one derivational rule, simplification. Now, the schematic form of simplification is generally presented thus:

$$\frac{\varphi \wedge \psi}{\varphi}$$

Also, let us make the simplifying assumption, usual in treatments of first order logic, that simplification only results in the derivation of the left conjunct (a combination of commutation and simplification being required to derive the right conjunct). From this, we can create a set of ordered pairs of sentences such that the first sentence satisfies the premise schema and the second is the appropriate sentence to the inference, namely the left conjunct. Let this set of ordered pairs be the domain of invariance, and let the class of transformations be such that each pair is mapped to every other pair in the domain. Now, trivially, the relation defined by the simplification is preserved under these transformations. The domain essentially consists of this relation. However, it seems clear that certain structural (grammatical) features of the ordered pairs will also be invariant. Namely that the first sentence contains a conjunction as its main operator, and the second sentence is the sub-sentential string to the left of this main operator (which will also be a sentence). It is precisely this structural relation that is expressed by the derivational rule of simplification. And since this structural relation is invariant under transformations between instances of the relation we have defined (the ordered pairs), it is formal with respect to these.

It is easy to see that there is nothing unique about simplification that it affords this

result. The very same process can be carried out for each and every derivational rule of logic, and the result will be that each particular rule exhibits formality with respect to an appropriate domain, consisting of a relation between an n-tuple of sentences which satisfy the antecedent of the schema and the appropriate sentence satisfying the consequent. It should also not be surprising that, again in every case, we are faced with the familiar circularity that the domain of invariance is defined in terms of a relation which ends up being invariant under those transformations. And again this is not a problem because the relation that constitutes the domain not only need not be constructed by means of the derivational rules, but it is wildly implausible that it could have come about merely by recognition of some structural connection between, for example, a conjunction and a sentence identical to its left conjunct. There are plenty of these types of structural relations to be found in first order logic which are *not* included among the derivational rules.

This point underscores the weakness of derivational formality for being individually distinctive of logicality. For it turns out that derivational formality holds for *any* derivational rule formulated in a metalanguage, whether it is logical or not. This goes for the complete class of structurally related expressions (in terms of the grammar of the language), for example:

$$\frac{\varphi \vee \psi}{\varphi}$$

as well as "derivational rules" which have no ground in structural similarities, for example:

$$\frac{Pa}{Qb}$$

Clearly, neither of these rules are truth preserving, and so they are not what we would call logical. The second is so clearly non-logical that it seems wrong to call it a derivational rule at all, and yet from the perspective of symbol manipulation, both can be used to derive new linguistic expressions from old ones. Furthermore, just as with the rules of logical derivation, these rules are invariant over relations between premise and conclusion

sentences. In the case of the second rule, the extension of the relation contains only the pair (Pa, Qa).

The generality of derivational formality, therefore, raises the question of why certain derivational rules are "logical" rather than others. It also evinces the weakness of derivational formality to independently characterize logicality. More information is needed than is provided by the metalinguistic definitions of valid inference, and the obvious thing to say is that what makes a derivational rule *logical* is that it captures or characterizes *semantically* grounded families of logical consequence relations in terms that make them syntactically manipulable. This is a significant point, since it is the seam along which the formal aspects of the syntax of first order logic are sewn to the formal aspects of the semantics of the language. As we will see in the next section, they together result in many of the features that make first order logic interesting and useful as a language.[42]

[42]It is worth noting the relationship between this claim and the position put forward by Gentzen, Prawitz, and later Dummett (the "GPD" tradition). According to the classical, model-theoretic approach, which we are following, the selection of derivational rules is something in need of explanation. These rules are not implicit in the selection of characteristic constants or the grammar more generally. The GPD tradition, on the other hand, takes a basic set of derivational rules *as primitives*. The constants involved in such rules, then, derive their semantics from the derivational role they play. This approach has consequently been called "proof-theoretic semantics" or "inferential role semantics." Because the derivational rules just are the semantic primitives, there is no questioning their relationship to the semantic primitives of the language.

In fact, the GPD approach is quite interesting from the perspective of the GTOF as it effectively reinterprets what it means to be a logical language. Instead of a language consisting of symbolic constants combined to form truth evaluable sentences, GPD holds that the language consists of primitive derivational rules which are combined into *derivations*. The operant semantic notion, then, is that of validity, which GPD also defines in terms of these primitive elements (derivational rules), rather than satisfaction or truth.

Although a detailed examination of GPD formal semantics is beyond the scope of our present inquiry, some interesting questions arise. In particular, [Prawitz, 1971] is clear that derivational primitives correspond to non-formal, intuitive inferences. This is similar to the claim we make below regarding model-theoretic semantics, but the identification of distinct characteristic atomic elements (derivational rules) and composites (derivations) makes us wonder whether the two approaches can really be said to be about the same language at all, and more generally if a comparative examination in terms of the GTOF might further illuminate relationships between the two approaches. For more on proof-theoretic semantics, see [Gentzen, 1964] and [Dummett, 1991].

3.4 Logic as a Formal Language

We started out with a question about whether or not standard first order logic turns out to be "formal" according to our theory of formality, and also in what way. We were worried about whether objectual formality could really capture what logicians have historically meant in saying that "logic is formal." Now we have seen that logic is in fact formal in several respects. It surely exhibits some kind of objectual formality (although just which of these is properly "logical" is not a question we will address here). It is also defined by a grammar, which is grammatically formal, and that works in tandem with sentential formality to provide what we usually understand as "syntax." Furthermore, its derivational rules, as such, are each formal with respect to an underlying relation which defines them. This makes it seem reasonable to say that first order logic is formal according to the GTOF in virtue of the fact that nearly every significant characteristic turns out to be directly connected to some type of formality. We could then leave it to other philosophers and logicians to decide amongst themselves just which of these properly makes logic formal.[43]

However, as our inquiry has moved along, we have noted two important points. First, that the types of formality associated with syntax are very general in nature. So general, in fact, that on their grounds alone we could not distinguish between first order logic and any other symbolic system, or even some fragments of natural language. Although this, in itself, does not undercut the idea that logic is formal, it belies the intuition that logic involves formality in some deeper sense. The second point we have noted is that the only plausible grounds for the particular characteristics of the grammatical and derivational formalities of first order logic seem to rely heavily on semantics.

In the case of grammatical formality, recall, we needed a way to distinguish the sentences from non-sentences without appeal to grammaticality itself. The only other alternative is to employ the idea of *meaningfulness*, which of course is deeply semantic.

[43] A task similar to that undertaken in [MacFarlane, 2000].

What this means is that, for all of their formal rigor, the grammatical rules of a formal language are constrained by the rule that their application must always result in meaningful (truth-evaluable) sentences. This is only possible when the symbols of the language, in particular its characteristic constants, have semantic values which correspond to certain structures of the underlying semantics.

What are these constraints in the case of first order logic? We can begin with atomic sentences. The grammar requires that these consist of a predicate letter paired with a function symbol (possibly a constant function evaluating to an individual). The constraints are fairly straightforward under the standard extensional interpretation of first order logic, which takes predicate letters as denoting sets of primitive elements, and function letters as denoting functions which evaluate to primitive elements. Since predicate extensions contain *only* primitive elements, they cannot be applied (where application is represented by concatenation) to anything that does not resolve to an object. In particular, unless the logical constants denote primitive functions (and they do not), a predicate applied to them will result in a nonsensical, non-well-formed formula.

Predicates may also, of course, be applied to object variables, however the grammar in this case is more complicated. It is a general rule of first order logic that, while a predicate symbol followed by a variable is considered "well-formed" it is *not* considered a sentence. That is, an expression like Fx is a well-formed *sub-sentential* expression, but it is not itself a sentence according to the grammar. Why? Following [Quine, 1960a] and [Quine, 1976b], this is because the semantics of a variable comes essentially from its quantifier. So, rather than being nonsensical, an unquantified predicate expression is incomplete.[44] It is, therefore, a semantic constraint that object variables be bound by a quantifier to form a complete, meaningful sentence. It should also not be surprising, given the semantics of predicate symbols, that the quantifiers denote notions which are themselves sets of primitive elements — the range of possible values must be such that the

[44]Of course, many accounts of the semantics of first order logic include a rule that variables lacking a binding quantifier are bound implicitly (usually universally).

application of any predicate constant to each value will result in a meaningful (though quite possibly false) statement.

Next we consider the case of logical connectives in a quantificational context — when all arguments of the connective, taken in isolation, contain unbound variables. The grammatical rule in the standard formulation is that the binary connectives are interposed between their arguments, and the unary connective is prefixed. This very division makes the semantic constraint evident, in that the distinction between binary and unary connectives is driven by the underlying denotations. According to the objectual view we are following, these connectives denote set-theoretic operators, and it is not surprising that these operators match their symbols in terms of arity — the negation symbol denotes a unary operation on sets, the others binary operations.

Finally, we come to the connectives used in sentential contexts — when at least one supplied argument is a sentence in itself. These follow similar grammatical rules as in the quantificational context, in that the grammar distinguishes the arity of the connectives. However, in this context the connectives cannot denote set-theoretic operators, since at least one of the supplied arguments is itself a sentence. Instead, the negation symbol denotes a unary operation on truth values, while the others denote binary operations. In the case where a logical connective is used to produce a new formula only from existing sentences, the resulting formula is itself a sentence. In the case where a binary logical connective is supplied with a sentence and a well-formed formula containing free variables, the result is a well-formed non-sentence. There are two ways such a formula can be turned into a sentence: the argument with free variables may be bound with a quantifier, or appropriate binding quantifiers may be prefixed to the entire formula. However, since the quantifier has no effect on the argument which is already a sentence, the two collapse to the same solution — the non-sentence formula must be rendered truth-evaluable (turned into a sentence) in order for the entire formula to be truth evaluable. Since the semantics is such that the denotation of every sentence composed of a connective and one or two sentences is also a truth value, the semantics permits recursive application of these connectives (e.g.

to arguments which are also compound expressions). Additionally, the existence of notational variations, e.g., polish notation, makes the constraint clearly evident in that none of these notational variants change the arity of the connectives. This makes sense because the underlying objectual notions remain the same.

What enables the same symbols to denote truth-value operators in one context and set-theoretic operators in another is that the grammatical rules guarantee two things. First, that it is never ambiguous which meaning applies, because the type of every supplied argument is always clear, and second that all sentences of the language evaluate to truth values, regardless of the symbols out of which they are composed. Although not strictly a constraint of the language, we can see that it is the semantics that facilitates the denotation of two types of operators by a single set of symbols. If, for example, the two sets of operators were capable of taking an overlapping set of arguments (especially if their outputs differ in value or type), any language built on such a foundation could not make use of a single set of symbols. It is even clearer that this would be the case if the two sets of operators produced values of different types. If so, distinct operators would be needed to ensure that no nonsense or unevaluable expressions result.

Again, when we were considering derivational formality, we recognized the need for an independent criterion to define the relations between sentences which form the invariance domains of the derivational rules, thus effectively selecting which rules would count as logical. We suggested there that the semantic notion of logical consequence would fit this bill rather nicely. It is, in fact, quite plausible that derivational rules are generally constructed with certain logical consequences in mind. It is difficult to see how it could be otherwise, given the considerable number of "purely structural" relations between expressions that play no part in derivational rules. And, just as in the case of grammaticality, the addition of a proof theory to a language effectively places further constraints on the relationship between the syntax and semantics of the language. The difference is that the constraints of the derivational theory are much stronger. The reason for this is that, whereas the underlying semantic constraints for grammaticality stem merely from

a consideration of meaningfulness, the derivational theory of first order logic has a preoccupation with *truth preservation* and, more generally, *semantic derivability*. Not only must the interpretation of the language be such that all well-formed expressions come out meaningful, but it must also be the case that the derivational rules all turn out to be truth preserving and allow the derivation of new, *meaningful* expressions from existing ones (the relations characterized by Sher as "formal laws"). This constraint makes it exceedingly difficult to reinterpret the logical constants.

Consider, for example, the simple case from before in which the denotations of "$\vee$" and "$\wedge$" are exchanged. While causing no *grammatical* problems (all expressions are still meaningful), the result is that the derivational rules of addition and simplification (along with most others) are no longer truth preserving, and the proof theory as a whole is rendered unsound. This is so because the meanings of sentences, related by the derivational rules purely in terms of syntactic structure, have been changed. Thus the underlying relations between propositions have been changed, and the derivational rules are no longer grounded on logical consequence relations. The situation can be resolved easily enough by changing the derivational rules, but the point should be clear. A derivational theory is constructed in light of semantic considerations, and the derivational rules capture the fact that the sentences of the language which represent certain propositions related to one other by means of a certain consequence relation also have a certain syntactic relationship within the language. Furthermore, the very reason that these expressions are related structurally (and they need not be, for a logical consequence, in the semantic sense, to hold) is grounded in the fact that the grammar of the language was designed with these semantic considerations in mind.

This does not mean, of course, that it is impossible to undertake a radical reinterpretation of the language of standard first order logic, such that every symbol is assigned a non-standard denotation but where the grammatical and derivational rules are preserved. It does mean, though, that the requirement to preserve the grammatical and derivational rules places considerable constraint on doing so. In fact, the best way to accomplish such

a reinterpretation would be to look for structural analogs to the standard denotations, particularly the denotations of the logical constants. It is also fairly easy to see that certain parts of the language are easier to reinterpret than others. These are the non-characteristic constants — those which do not feature essentially in the grammar or derivational rules. In the case of first order logic, the validity of derivational rules hinges entirely on the preservation of truth. Truth, in its turn, is generally spelled out in terms of the satisfaction of sub-sentential predicate expressions. However, the derivational rules are not sensitive to what satisfaction means or how it is accomplished, and they are also not sensitive to the identities of particular predicate and object constant symbols.[45] Hence there is considerably more semantic flexibility with respect to the interpretation of predicate and name symbols, and this is one principled reason a standard "interpretation" of first order logic does not involve a reinterpretation of its characteristic, logical constants. The practical reason, of course, is that one is usually choosing to work with the language of first order logic out of interest in logic, and repurposing the "logical constants" to denote non-standard notions would be counter-productive.

So, although the language of first order logic exhibits multiple distinct types of formality, all of these (with the possible exception of sentential formality) are tailored to the semantics, which is itself formal. Although this conclusion is suggested by Sher, the application of our GTOF not only confirms the position but also shows that first order logic comes out as formal, just as we had hoped. Furthermore, this relatively novel way of looking at first order logic helps us to understand its utility. Because the syntax is structured to effectively mirror the semantics of the language, mere syntactic manipulation stands for — even *models* — corresponding relations and manipulations in the associated semantic domain. This enables us to easily conflate the sense of "logic" as a theory about

[45]This is itself a consequence of the focus of the language on invariance with respect to the identities of individuals and membership (i.e., set theory). Since the logical constants denote objectually formal operators, and all of the schemata for derivational rules explicitly include only logical constants (using schematic letters for the rest), it is no surprise that these rules are themselves insensitive to the non-logical constants (symbols denoting individuals and sets) to which the logical connectives are also insensitive.

certain relations in an objectual domain with the sense of "logic" as a formal language and proof theory.

This is an interesting and enlightening result in and of itself, but much of what we said above was either quite general or could very easily be made so. This suggests a conclusion regarding formal languages more generally — namely that as a rule they involve the structuring of the formal characteristics of the syntax of the language to match formal characteristics of the underlying semantic domain, and that derivational theories, where present, will have some additional preoccupation (often with truth, but not necessarily so). This idea, if true, could serve as the foundation of a much more robust account of formalized languages, one which recognizes that such languages are neither created *ex nihilo* nor essentially limited to abstract subjects like logic, mathematics, and geometry. It is also reminiscent of the somewhat radical position held by Richard Montague, that the semantics of all languages (formal and natural) could be accounted for in model theoretic terms.[46] While we are limiting our focus to formal languages, the claim we wish to entertain does involve the formal nature of the semantics of all formal languages. The main difference, however is that the GTOF, distinguishing as it does formality from logicality, does not oblige us to invoke model theory or any other specific type of formality.

Unfortunately, we cannot easily conduct a general investigation of such a sweeping idea any more than we already have, since for each case the relationship between syntax and semantics is quite specific and the semantic domains can vary wildly. Instead, in the following chapter, we examine whether this claim holds for a number of programming languages — formalized languages which are more applied, more complex, and less idealized than first order logic.

[46] See [Montague, 1974].

Chapter 4

Formality and Programming Languages

We ended our last chapter with a thesis very similar to that of Montague, except that the postulated connection between syntax and semantics is somewhat weaker, and there is no restriction that the semantics of the language turn out to be model theoretic. The basic thesis we are explaining, supporting, and exploring is this:

First, that though the syntax and semantics of a formal language do not completely determine one another, there is a strong relationship between the two based on what we have called "constraints," which are minimal standards which must be satisfied. These constraints may be considered from either the semantic or syntactic perspective, but are easiest to see when viewed from the syntactic side. If we have a definition of a language, but without any semantics, we can ask ourselves the question of just what could serve as a proper semantics for the language. We might have a range of choices, but these are clearly constrained by the characteristics of the syntax (as explained in Chapter 3). The semantic perspective is slightly more difficult to imagine, since it is not easy to conjure up a semantics from nothing (this, perhaps more than anything else, is what centuries of logical development has produced). However, as presented in Chapter 3, we could isolate the semantics of an existing formal language and ask the question, "Which syntaxes would work to express these semantics?" The result will clearly not be limited to the syntax of

the original language, but likewise not just any syntax will do.

Second, that the relationship between syntax and semantics, for a formal language, rather than being a relationship between the syntax and a model-theoretic semantics (as Montague supposed), is a relationship between the formal, invariant features of the syntax and the formal, invariant features of the semantics.

Third, that this close relationship effectively makes a language "transparent" with respect to its semantic domain, such that it is easy to ignore the linguistic aspect and take the representations and syntactic manipulations which occur *in the language* as though they actually are objects and manipulations in the semantic domain.

If this hypothesis is true, we would expect that all formal languages would satisfy it. Showing this requires either demonstrating that it holds for each and every formal language, or arguing that it holds from general and undisputed principles about formal languages. This second path is closed to us, because it is unclear what facts about formal languages we can appeal to, at least not which involve a relationship between syntax and semantics and are also undisputed.

This leaves us with the first path, which it is not possible to complete in the present context (if at all). Instead, we will examine a collection of formal languages to see if (and how) they satisfy this thesis. In focusing our discussion we should be guided first by a concern for clarity, but we should also have in mind generalizability. That is, we should choose in a way that our conclusions are likely to be generalizable to a broader range of formal languages with a high degree of confidence.

There are a number of classes of formal languages we could consider in this respect. The languages of generative grammars, game theory, those used in the natural sciences, or even the dance "language" of bees hold some promise for this. However, after logic and mathematics (or perhaps equal to them) the discipline most tied to formal languages is computer science. Computation does not, strictly speaking, require formal languages, but formal languages have proven to be an extremely effective tool for the discipline. And so it seems the most practical choice is to examine computer programming

languages. Not only does this constitute a very broad class of languages, but despite a common, naive opinion, they differ in significant and important ways from logical and mathematical languages. Furthermore, we have a fairly clear understanding of the issues driving the development of early programming systems, which places us in a position to understand the motivations and goals underlying particular decisions made in the design of these languages. This proves invaluable in understanding the connection between a language and its semantics. In particular, we will follow the development of programming languages as their focus shifts from machine-oriented semantics to problem or domain-oriented semantics.

In what follows, we must navigate some difficult waters. We must provide some depth of understanding regarding the technology in question, but at the same time we cannot hope to provide a comprehensive treatment of the languages we will be considering. Since this is a work of philosophy, and we assume that our audience has a primarily philosophical background, the discussion is tailored in a philosophical way with a focus on concepts and principles rather than the technicalities of implementation and details of particular hardware. We have endeavored to only explain those aspects of the technology strictly necessary to understand the philosophical points to be made, and this results in some aspects of languages and technology receiving short shrift, in particular relatively simple ideas whose implementations are practically challenging but which lack substantial ramifications from our perspective.[1] We have endeavored to provide enough in the way of detail, however, to prevent our explanations from being cryptic or overly schematic.

The following discussion is broken down into two primary sections. The first and more substantial section takes as a starting point the machine-oriented nature of low-level and early programming languages and focuses on the innovations that permit increasing

[1] A nice example of this is input and output (I/O) facilities. While technically challenging, I/O is not particularly philosophically interesting, and we feel at liberty to mention it only in passing. The same is true for most kinds of optimization. While an important area of ongoing research and a field rife with technical challenges, most kinds of optimization do not have independent philosophical interest and can be safely ignored. Even a cursory look at much of philosophy will reveal that we have historically had little care for optimization or efficiency.

degrees of "independence" from these machines and their structures. For this purpose we first consider machine codes (generically), and then look at one of the earliest genuine programming languages, developed for a machine called "EDSAC". We conclude this section by looking at one of the earliest high-level languages — FORTRAN. The second section considers developments which take into account non-machine oriented semantic domains. To this end we consider the developments of structured programming, object-oriented programming methods, and what has been called "descriptive" or "functional" programming. This final topic permits a few brief comments on the relationship, and inequality, of logic and programming languages.

4.1 Hardware and Machine Independence

The first subclass of programming languages we look at are those whose design enables some degree of what we call "hardware independence." Intuitively this means these languages enable essentially the same process or operations to be carried out in ignorance of the underlying physical system. The idea can be more strictly, and conveniently understood in terms of invariance. Put simply, a program or programming language is "independent" of a hardware facility if it is insensitive, or invariant, to changes (especially implementations) of this facility. Hardware independence can differ both in terms of which hardware facilities the program or language is insensitive to, and also in degrees, understood in terms of range of variation over which the program or language is invariant. Languages which are highly insensitive in both respects are called "machine independent."

In this section we examine three languages with varying levels of machine independence. The first, machine code, is by all accounts the least machine independent language. The second, EDSAC code, has some important features which make it independent with respect to certain machine features, but its hardware independence is still severely limited. Finally, we will look at one of the earliest and best known high level programming languages, FORTRAN, which makes a much stronger claim for machine independence.

4.1.1 Machine Codes

The association between programming languages, programming, and computers is so close that their coexistence is often taken for granted. This was not always the case. Very early digital computers (both electronic and electromechanical), though programmable, did not use programming languages of any appreciable kind. Programming these early machines was essentially a matter of rewiring the device (by means of plugs and switches) to control the processing flow. A nice example of this is ENIAC, created in 1946 by the Moore School of Electrical Engineering at the University of Pennsylvania.[2]

The use of programming languages with computers, although postulated and attempted much earlier by Babbage and Lovelace, was facilitated by the introduction of a distinct electronic unit dedicated to processing "instructions" (written in a language) and thereby directing the flow of computation. Such machines were known as "stored-program" computers on account of the fact that they also involved a memory component in which the program commands would be loaded for execution. Nearly every contemporary computer is a stored program computer, although there are a number of different types.

The most common approach to implementing a stored-program computer makes use of what has come to be called the "von Neumann architecture." One of the first descriptions of the von Neumann architecture, named after the logician John von Neumann (who contributed to its invention), can be found in [von Neumann, 1945]. In this description, von Neumann tells us that the machine should consist of three parts. The first of these he called the "central arithmetical part" or "CA."[3] The CA consisted of the physical devices which actually perform the basic operations involved in arithmetical computation (adders, multipliers, etc.).

In addition to the CA, a von Neumann machine also contains a distinct "central

[2] A very accessible discussion of ENIAC can be found in [Goldstine, 1972]. It is worth noting that ENIAC was eventually altered to permit more advanced, flexible programming approaches.

[3] [von Neumann, 1945], p. 18.

control" mechanism which is responsible for determining which simple operations to perform on which stored values and in what order. Again, this is the most significant point from the perspective of programming language use, as it removes the need to physically rewire the machine. The control mechanism essentially "rewires" the machine based on the stored input commands.

The third component of a von Neumann machine is a memory store. It is easily seen that every stored-program computer will need some kind of memory in which to store the program. What is unique about the von Neumann architecture is that it involves only one memory store, which is used for both program and data (numerical value) storage.

In principle, we should be interested in all computers which involve some kind of central control unit, including those with varying memory architectures (e.g., the Harvard architecture) This is because a central control unit is all that is required to facilitate the use of programming languages. In practice, however, we will focus only on von Neumann type computers. This is in part because the vast majority of modern computing systems implement the von Neumann architecture, and also because a broader focus would take us too far afield. While there are some interesting consequences for programming language syntax and semantics that result from these architectural differences, this is a matter for future inquiry.

A simple von Neumann machine is programmable by means of something called a "machine code." Put simply, a machine code program is the sequence of commands interpreted by the central control mechanism that results in the carrying out of a complex computation. A machine code is the "language" in which these commands are written. The best way to understand the basic principles of machine code will be to discuss the operation of an extremely simple von Neumann computer. The machine we will consider will be fairly schematic, but it is similar to the EDSAC, one of the first operational stored-program computers.

Our simple machine has a central control unit which treats memory locations as though they contain commands. The central control unit always begins processing at

memory address 0 and proceeds serially until reaching a control transfer command, a "halt" command, or the final memory address, for example 511 (but of course, it could be of any finite size). The arithmetic control unit of this simple computer also reads values from memory locations, but unlike the central control unit, it treats these as numbers. It has components for performing basic arithmetic operations on these values, as well as some auxiliary storage locations (distinct from the main memory) for storage of temporary values. One of these auxiliary memory locations, called the "accumulator," is the "output" location for the arithmetic operations. The main memory consists of a certain number (e.g., 512) of memory locations or "addresses," each of which can store a binary string of a specified, finite number of digits (e.g., 17).

Machine codes are generally quite simple, and as such we only need to consider a relatively small fragment to have a clear picture of how they function. Table 4.1 shows a fragment of the machine code for our simple computer, involving just enough to permit a simple program which adds two numbers.[4] Alphabetic references have been included for convenience, but it is important to remember that this is not how the language is represented within the machine.

With this fragment of machine code, we can consider a simple example of what machine code looks like and how it works. Let us consider a program which adds the numbers -12,743 and 9,957. With large numbers and many carries, it is possible that we might want such a program. The program in human-readable form is given in Table 4.2. Of course, in this representation, we have distinguished between numerical values and commands, but in a machine consumable form the only difference between a command and a numerical value is how the value is being interpreted at a given time. Also, for practical reasons related to hardware implementation, all numbers are represented as being between 1 and -1.[5] This means that in order to input the indicated values they need to be scaled appropriately. We can do this by multiplying each by 2^{-16}.[6] The true representation

[4] Adapted from [Wilkes et al., 1951], p. 5.

[5] See [Wilkes et al., 1951].

[6] This value is used because the input values are integers. Non integer values would require different

Table 4.1: Fragment of Simple Machine Code

Alphabetic Reference	Machine Code Value	Description
A	1.1100	Add the number in the indicated storage location into the accumulator.
T	0.0101	Transfer the contents of the accumulator to indicated storage location and clear the accumulator.
U	0.0111	Transfer the contents of the accumulator to the indicated storage location and do not clear the accumulator.
Z	0.1101	Stop the machine. The address field is not used for this command, and consequently may be any value, including zero.

of this program in machine code (including the final value at address 6) is given in Table 4.3.

Table 4.2: Human-Readable Machine Code Program

Address	Command Field	Address Field
0	A	4
1	A	5
2	T	6
3	Z	0
4	-12,743	
5	9,957	
6	0 [Will contain result]	

Table 4.3: Machine Code Program

Address	Value
0	1.1100000000000100
1	1.1100000000000101
2	0.0101000000000110
3	0.1101000000000000
4	1.1100111000111000
5	0.0010011011100101
6	1.1111010100011101

When run, the central control unit will cease processing commands when it reaches memory address 3, which contains the halt command. At this point, the resulting value

scaling factors in order to fit between 1 and -1. The resulting value must, of course, be scaled as well, "undoing" the original scaling. It is easy to understand how complicated this kind of scaling can become. As computational speeds increased, such "fixed-point" operations (so called because of the fixed location of the binary point) have given way to "floating-point" operations (wherein the location of the binary point is coded in the overall value), which eliminates the need for additional scaling.

of 1.1111010100011101 will be located at address 6. Properly scaled and converted to a decimal value, this is -2,786.

In order to carry out complex computations, machine languages also need to be capable of directing the central control unit to "repeat" a sequence of steps a specified number of times or until a certain condition is satisfied. This is accomplished by introducing commands which, instead of directing the arithmetical unit to carry out operations on the contents of a memory address, result in a "transfer of control" within the control mechanism to a certain memory address. What these commands essentially do is interrupt the central control unit's normally linear execution pattern. If address n contains a control transfer command, the next address read by the central control unit will most likely not be address $n+1$, but some other address specified in the command. The simplest control transfer command is a "go to" command. When the central control unit processes a "go to" command, the result is that the next command processed by the central control will be whatever address is specified by the command, with execution proceeding serially from there. This command can be used to create a simple, but non-terminating, loop by transferring control to any preceding address.

What is required to make "go to" an effective command is an additional feature, often called "conditional branching." This amounts to a command which has two possible outcomes depending on a certain condition (often whether a particular numerical value is negative or non-negative). This is one of the minimal conditions for a programming language to be "Turing complete."[7] The need for this is easy to see from considering the

[7]It is a classic result of computability theory that machines which have a certain minimum level of complexity are capable of processing or expressing, respectively, all of the same *functions*. These are called the "computable functions," and it is also a result of classical computability theory that these are equivalent to the class of partial recursive functions. Machines having this degree of complexity are called "Turing complete." Strictly speaking, a "Turing complete" machine is one capable of computing all computable (partial recursive) functions given unlimited time and memory. That is, the machine would be computationally equivalent to a "universal Turing machine." Of course, actual machines do not have these features, and in this application "Turing complete" is used to denote machines which *would* be strictly Turing complete if they did have unlimited time and memory. The classic description of the universal Turing machine is [Turing, 1936].

structure of recursive functions. For such a function to be interesting and useful, from a computational perspective, it cannot simply apply to itself *ad infinitum*. It should terminate eventually in some primitive element (often zero or the empty set). So, for such a function to be representable in a language, the language must have the ability to express a conditional transfer which effectively permits recursion to terminate and produce a meaningful value. An example of a conditional transfer command for our simple machine code is described in Table 4.4.

Table 4.4: Control Transfer Commands

Alphabetic Reference	Machine Code Value	Description
E	0.0011	If the number in the accumulator is greater than or equal to zero, execute next the order which stands in the storage location indicated; otherwise proceed serially.

This, together with appropriate memory management commands such as those in Table 4.1, renders the language "Turing complete." And, although we certainly have not presented anything close to a complete machine code, we have presented examples of the types of commands that one would contain: arithmetic operation commands, memory transfer commands, and control transfer commands.

Analysis

The first thing we should say regarding the syntax of machine codes is that it is very regular and simple. This is why we are able to only consider a small fragment of a machine code. This is probably also part of the reason that machine codes do not generally have explicit syntactic rules defined for them, in contrast to logic and other more familiar

programming languages. To anyone working with one of these languages, the syntax is more or less obvious from the physical construction of the machine. This is a significant point from our perspective, since we are interested in outside constraints placed on syntax. There is, however, one small matter to address first.

The syntactic waters of machine codes are somewhat muddied by the fact that the value stored at a given memory address has two distinct interpretations — as a number and a command. While distinct interpretations are generally a semantic issue, the trouble is that the syntax of the contents of a memory address are construed in two very different ways by the two interpretations. Taken as a number, the string of digits is syntactically atomic.[8] Taken as a command, on the other hand, the string of digits is syntactically composite.

The first question to answer, then, is what the relevant syntactic unit of the language is. It is tempting to say that the entire "vocabulary" of the language for the machine will consist of fixed length (e.g., 17 digit) binary "words" ranging from 1 to -1, and that these are interpreted differently by different parts of the machine. We think, however, that this is not an accurate picture of what a machine code really is. In particular, it does not

[8]There may be questions over whether the inclusion of a signing digit (indicating whether the number is positive or negative) makes the resulting number compositional. Saying this is so is equivalent to saying that "-1" is composite from "$-$" and "1". There can be no doubt that "-1" is lexically composite, being formed out of those two symbols. However, whether or not the expression is syntactically composite depends on whether one takes the expression as the name of a number (negative one), or as the evaluation of a function (denoted by "$-$") which when applied to the number one evaluates to the number negative one. The difference is easy to see by considering the true arithmetic expression "$-(1) = -1$". In this expression, the first "$-$" is clearly the name of a function (i.e., the whole expression is syntactically composite). We know this because "$-(6-5)$" is permissible, even though "$6-5$" is not an atomic expression. If the symbol "$-$" receives the same interpretation on the right-hand side, then the equivalence is true in virtue of the syntax of the language, e.g., by means of a rule which allows the omission of parentheses when the expression within is atomic. If, however, "-1" is taken as a primitive designation of negative one, the truth holds in virtue of semantic facts, e.g., facts about the function denoted by "$-$" on the left-hand side. Just which interpretation is correct (or we choose to operate with) will make little difference from an arithmetical point of view, and perhaps for this reason the issue rarely comes up. We think the most natural understanding is to take "-1" as a primitive name in large part because "$+$" can be used in a similar fashion to clarify that a value is positive, and it seems exceedingly doubtful that any functional interpretation is at play in such cases. It is nevertheless still the case that the primitive names of negative numbers are formed by means of a lexical function, and in this sense have an air of compositionality. By the same turn, however, "10" is a composite of "1" and "0".

seem plausible to think that the full "word" stored at a memory address, when taken as a single number, should count as a statement of the language at all. The first reason for this is that, if so, they have no syntax, because they are atomic. Second, it is dubious that they are interpreted at all, rather than being merely manipulated. That is, they do not have any effect on the functioning of the machine. But the final reason is that, as a *coded* form of orders controlling the machine, machine code is *interpreted* by the central control unit. The central control can direct the arithmetical unit to perform an operation on the numerical value stored at a certain address, but the central control unit is never in a position to interpret the value of an address as a number. It is notable, in fact, that numbers themselves never actually occur explicitly within a machine code program. That is to say, there is nothing equivalent to "1 + 1" in a machine code. Instead, one must add the contents of two memory addresses. Just what these values are does not matter from the perspective of the central control. As far as the central control is concerned, every string between 1 and -1 is a potential command. It is these combined command-memory address codes that form the basic syntactic "sentences," or "statements" as they are normally called, of machine code.

The basic vocabulary, then, consists of command codes and address codes. Given the technical, physical constraints of machine construction, we know that, in fact, every permissible expression of the language consists of a command code followed by an address code. This suggests two plausible syntaxes for the language. First, the syntax might consist of a separate rule for each command code, stating that the command code followed by an address code is a permissible expression, and that nothing else is a permissible expression. Alternatively, the syntax might consist of a single rule based on the lexical categories of command codes and address codes. The rule would simply state that *any* command code followed by an address code, and nothing else, is a permissible expression.

According to the first view, the command codes would be seen as characteristic constants of the language because they are assigned an explicit role in the syntax (as discussed in Chapter 3). On the second view, the entire vocabulary consists of constants,

none of which are given a unique, explicit syntactic role, and hence the language would not have any characteristic constants.

We feel that the second possible syntax is the most plausible for two reasons. The weaker reason is that it is the simplest adequate syntax for the language. The more important reason stems from the fact that we can easily imagine "extending" our machine code by adding additional commands. This preferred syntax is easily amenable to such additions. In fact, the syntax need not be altered in any way. If we imagine adding a command which would require changing this syntax, say a command code which requires two address fields, we find that this new command exceeds the limitations imposed by the machine. Hence, any command which *could* be added to the language will satisfy this syntactic rule. We deem this syntax more plausible, not only because it is simpler, but because it is also maximal in the sense of being adequate for every *possible* statement of the language under arbitrary extension without alteration of the machine itself.

The only syntactic rule of the language, then, is that a command code is always followed by an address code. Just what is done with this address (if anything at all) is not indicated syntactically, it is part of the semantics of the command code.[9]

The purpose we have set is to consider carefully the constraints placed by semantics on the resulting formal characteristics of the language, and the hypothesis is that these semantic constraints, in the case of a formal language, will be formal as well. The machine code we have been considering is somewhat unusual in that it wears its semantic constraints somewhat on its sleeve. This is a common trait of machine codes, for reasons we will explore. So much so that, in fact, formal descriptions of syntax are often unnecessary for machine codes. This is because the constraints are primarily due to the underlying

[9]The word "semantics" is important in both philosophy and computer science, and in many cases the term is used differently in the two fields. In philosophy, the "semantics" of an expression consists of its meaning (which may itself consist of different elements, e.g., intension and extension, sense and reference). This is how the word was used in the preceding chapter. In computer science, the semantics of an expression is sometimes taken to consist in the transformation the expression brings about on the underlying data. Here, and in many places, the two notions coincide. In those places where they come apart, we will take care to clarify the point.

technology, the hardware, for which the machine code was designed. If we understand the hardware, the syntax of the associated machine code is to a high degree self-evident. This is especially so if one adopts the view that machine code syntax is based on classes of constants rather than particular constants.

At a general level, it is easy to see that the hardware constrains the permissible expressions of the language to those representable in at most 17 digits (or however many digits are permitted on a given machine). In fact, unlike some other languages we will consider, hardware imposes an absolute limit to the semantic expressibility of machine codes in that there can be at most 2^n (2^{17} in our example) unique statements of the language. If expressions do not influence the semantics of other expressions (that is, if the semantics is not context sensitive) then this is the upper limit on the number of meaningful expressions that can be formed in the language, although, of course, most machine codes are not so semantically rich.[10]

However, our hypothesis bids we look for a closer relationship. This is because, insofar as machine codes have a "subject matter," they are about the machine for which they are codes. That is to say, they direct or prescribe (and also effectively describe) the functioning of the machine for which they are designed. If our hypothesis holds, we should expect a close relationship between the syntax of machine codes and the machines on which they are used.

On our preferred understanding of machine code syntax, there is but a single rule — that a command code should be followed by an address code. This means that all permissible expressions have exactly the same syntactic structure. Even on the alternate interpretation, the syntax of the language is comparatively quite simple, as, for example, there are no constants which can transform a well-formed statement into a well-formed statement. Again, this is so because allowing such a rule in the syntax would permit well-formed statements to exceed the physical bounds of the hardware.

The length constraint alone, though, does not explain why a machine code would

[10]In information theoretic terms, this is the maximum number of informative signals.

have any particular syntax. However, if we think in a similar way about the operation-address syntax that our machine code has, it becomes readily apparent (and is not at all surprising) why it is structured as it is. An operation-address syntax is a natural choice given the purposes and goals the language was designed to meet — to direct the operation of a computing machine. What this amounts to, essentially, is controlling the transformation of bits of data. Nearly every such "activity" undertaken within a computer is characterizable as a function.[11] So, it is not surprising that a syntax for a language describing or controlling such operations should follow a general function-argument pattern.

Furthermore, results from the lambda calculus let us know that a function taking n-many arguments may be alternatively represented as a set of monadic functions applied in series.[12] Because the purpose in designing a computer (the hardware) is the carrying out of these functions, the hardware will naturally be built in whatever way makes this easiest given the existing technology. It is not surprising, then, that computer hardware is not designed to carry out binary, ternary, etc. operations when monadic operations will do. Designing the machine to carry out a basic set of monadic operations is the simplest and most flexible approach. It is furthermore quite convenient from a linguistic perspective, since a monadic function-argument syntax (where functions cannot be arguments) is very simple for a machine to parse (and certainly simpler to parse than a syntax involving functions which take varying numbers of arguments). It is the most easily implemented compositional language capable of expressing, and thereby directing, the activity of the hardware in performing operations on data.

These syntactic constraints are essentially built into the hardware of the machine, and it is for this reason that an explicit syntax is generally unnecessary. The technology makes it such that expressions which are impermissible simply cannot be uttered (or in this case, placed into a memory location). When we say that every operation the computer carries out is a function which takes a single, non-function argument, we are essentially

[11] In fact, just about everything that occurs within a computer is characterizable as a function.

[12] [Hindley and Seldin, 2008]. The process is often called "Currying."

saying that this is an *invariant* of the machine itself. That is, if we take as our domain the set of operations the machine is designed to carry out, and as our class of transformations a class of functions which transforms each operation into every other, the most prominent invariant we are left with is that each operation affects only one memory address.

But this is perfectly analogous to the command-address syntax, which is the only grammatically formal characteristic of the language. Which is to say, they satisfy our hypothesis. The invariants of the syntax correspond to the invariance of the semantics. Also, just as we hypothesized at the conclusion of Chapter 3, it is this correspondence that lends a kind of transparency to the language. We can say of a machine code program that *it* "computes thus-and-such function," when to be precise we should say that it directs *the computer* to compute the function.

To see that this is so, we can imagine an alternative situation in which a different type of computer is able to receive input in a different language, say one of the languages of mathematics. Now let this machine, though it has just the basic operations of a Turing-complete von Neumann machine, be by some mysterious artifice able to read an equation in the mathematical language, carry out the appropriate computation, and provide a solution to the equation. In such a case, we would not say that the mathematical equation in any way *describes* or *captures* the operations the machine carried out to solve the equations. We would not say that the equation "computes thus-and-such function," as we can say for a machine code. It might *represent* the function, but it does not provide much in the way of how a solution can be found (and in many cases effective means of computation are obscure). In other words, there is no transparency through the mathematical language into the operations of the machine.[13] That there is such transparency for machine codes is therefore non-trivial and interesting.

Before moving on, we think it prudent to take a moment to clarify an important

[13]To be sure, there *is* an important relation between the two, and it almost certainly has something to do with invariance. It just is not the sort of relation that permits transparency, or the use of mathematical language as a formal language to direct or describe the basic functioning of a computer. We will return to this thought, and the example, in later sections.

point about what we have accomplished. We have shown that there is considerable similarity between first order logic and a basic machine code in terms of formality. This might be distasteful to those who hold that logic is somehow special in its formality. While we are to some degree challenging the idea that there is a difference in kind between first order logic and other formal languages (when it comes to their formality),[14] we are quite amenable to the idea that there may be considerable differences in what might be called "degrees" of formality. That is, we think that logic is specially formal in that it is *more* formal than other languages (and the theories that go with them). This idea is explored more fully in Chapter 5, but we can characterize it ostensibly here.

There happens to be a language with a function-argument structure which, under a certain interpretation, is equivalent to first order logic. This language is typed combinatorial logic (or an equivalent formulation of a typed lambda calculus). For the present purpose, the most important characteristic of this language is that there is one "operation" — usually called "application" — which is characterized by the mere juxtaposition of symbols. For example "FX" is the application of "F" to "X". It is fairly easy to see how this is similar to our machine code. Without going into too much detail, a major difference is that application in combinatorial logic is iterative. For example "GXY" means an application of "G" to the result of applying "X" to "Y". Our machine code only permits a single application in a given expression, which limits its expressiveness. What is more, our machine code involves *no variables*.[15] This is a further limitation. One can see that, even if we grant that machine codes are formal in the same sense that combinatorial logic and first order logic are formal, the claim that it has a lower or lesser degree of formality is sensible.

[14] By a difference of kind we mean a difference of what counts as a formal language and what does not count as a formal language, not a difference in the subject domain with respect to which the language is formal.

[15] Certainly, when writing a program in machine code there are steps which emulate the use of variables we might use to carry out the computations by hand, but nothing like this is a part of the machine language itself. Every statement in a machine code describes a fixed operation which is carried out on a fixed object (in this case a memory location). Even though a programmer may make use of variable expressions in determining how to write the program, there are no equivalent machine code expressions because everything must have a concrete instantiation at execution time.

The language, at best, "captures" invariances only over a small domain (just the machine), whereas the domains for typed combinatorial logic and first order logic are considerably larger. The idea of degrees of formality is implicit in the idea of degrees of machine independence which we invoke frequently below, but we reserve our explicit treatment of the subject to Chapter 5. The final section of this chapter will look more closely at the connection between programming languages, combinatorial logic, and standard first order logic.

4.1.2 Memory Location Independence

It should be obvious that nobody would want to write computer programs entirely in machine code, and certainly not complex ones. Very early on, programs were first written in a slightly more human-friendly form by substituting letter combinations for command codes and decimal numbers for addresses. Once completed, the program would be transcribed by hand into the machine code for execution. Similar "lower level" codes still exist today in the form of assembly "languages."[16] It is tempting to think of this as a translation from an alphanumeric code into a machine language, but we must be careful here. Generally speaking, alphanumeric codes (and assembly languages) of this kind do not involve anything in addition to either the syntax or semantics of the underlying machine language. They are "codes" in the proper sense. They are simply alternative representations of the same language, with rules for direct transliteration between them,

[16]We use the notions of "lower" and "higher" level languages in more or less their standard sense in the programming community. Machine codes are the lowest level languages because of their unmediated connection with hardware, and also their overall simplicity (limited number of constants, simple syntax, etc.). Assembly languages are of a higher level because these are transformed "downward" into machine codes. All languages "higher" than machine codes require some kind of transformation (assembly, interpretation, or compilation) which ultimately result in machine code commands. The terms "higher order" or "higher level" are generally used for languages which pass through at least one intermediary language (for example, an assembly language). "Low" or "lower" usually refers to languages at or below the assembly level. This sense of "level" and "order" should also be distinguished from the logical use, in which the "order" of the language usually indicates the types of quantification involved in the language.

rather than separate languages requiring rules of translation.[17] Nevertheless, programming in this way makes certain characteristics obvious which are important to the development of more powerful, higher level languages.

One thing recognized more or less from the very beginning was that programming involved a significant amount of repetition and reuse. We have already considered this with respect to the control transfer facilities of machine codes (enabling loops and recursion), but it is also easily recognized that entire, often lengthy, subsections or "segments" of programs perform the same series of transformations required by other programming tasks. It was recognized early on that the quality and efficiency of coding could be improved significantly if there were a library of pre-written program segments — "subroutines" — from which a programmer could simply select the desired parts.

Recall, however, that every machine code command contains a reference to a memory address, and that most of these commands make essential use of these addresses. This means that, though a programmer may wish to make use of the sequence of transformations described by a certain segment of machine code, a literal reuse is only possible if the segment is placed in precisely the same memory location as in the original program, and necessary values are likewise precisely placed. If this is not the case, as in most cases can be expected, then the memory addresses must be rewritten to reflect the segment's different position within the current program. This significantly undermines the motivation for having a subroutine library in the first place.

A clumsy solution to this problem would be to expand the library to include ver-

[17]The difference here between transliteration and translation is subtle but important. Transliterations are only possible when the differences between the languages consist entirely of different *symbols* for the same semantic units. Thus, a transliteration can be defined as a bijection from one language to another which preserves semantic values. If no such function can be defined between two languages, transliteration is not possible. Translations, on the other hand, only require that the target language have at least the expressive power of the source language. Consequently, translation is often possible even when transliteration is not. A simple example (out of multitudes) can be developed from first order logic by having one language include "$\exists$" and the other "$\forall$", in addition to the usual constants. No transliteration will be possible between these languages, since every bijection results in at least one mapping which does not preserve semantic values. However, because these expressions are semantically inter-definable (by means of "$\neg$"), we can easily establish rules of translation.

sions of the subroutines appropriate for every possible position in memory. This is workable but obviously inefficient. The obviousness of the inefficiency is interesting and telling. We recognize, without even seeing an example, that two subroutines which differ at most in being appropriate for different memory locations are two *copies* of the same subroutine. At least, that is what we want to say. The semantics of machine code, however, will not allow this identification. What we have recognized by this is that, at least for most subroutines, the actual memory locations make no difference to the underlying computations that are carried out. What does matter are the relations between the memory addresses used. If, for example, a program segment begins at address 50, and a crucial value is stored at address 85, the critical fact is not that the value is at 85, but that the value is stored 35 memory locations later than the beginning of the sequence. If we symbolize the initial memory location (50) by the constant a, it is easy to think of the memory location of the value in question simply as $a+35$ (that is, $50+35$). We will call values like a an "offset" because they indicate the location of the subroutine relative to the origin (memory address 0). It is not difficult to see that the offset itself will not affect the functioning of the subroutine, provided that internal addresses are defined with respect to this offset value. The recognition that the offset value a does not affect the subroutine seems to warrant the treatment of a as not referring to the location 50, but possibly to any location whatever. That is to say, instead of treating a as a constant denoting memory address 50, it is tempting to think of it as a variable because, even were the denotation of a something other than memory address 50, the data transformations carried out would be the same. What we have just recognized is that the information process characterized by a segment of machine code is invariant with respect to memory locations, a fact which is clearly *not* captured by the machine code itself.

This is important to recognize and it legitimates the practice of rewriting segments of machine code merely by altering the address references to fit a new memory location. It does not legitimate the writing of program segments with variable expressions in place

of address references. No such expression is permissible in a machine code.[18]

The practical utility of having usable program libraries, and the technical problems with developing those libraries in machine code, drove the development of what is usually considered the first programming system, constructed for the EDSAC at Cambridge. The EDSAC programming system consists of more than just the input language itself, which we will call "EDSAC code," but also of a special program called the "Initial Orders," which was responsible for the transformation of programs written in EDSAC code into appropriate machine code instructions. The need for similar transformations in contemporary higher order languages (e.g. C++, Java) is so ubiquitous at present that they are merely called programming "languages" and not programming "systems." However, the fact that using any language of higher order than machine language requires additional software (especially compilers and interpreters) should not be overlooked as the transformations that take place (beyond the mere transliteration of assemblers) are of critical importance to our inquiry. We will be discussing the Initial Orders in some detail, as a simple example of the role that compilers and interpreters play with respect to the formality of higher order languages. In so doing we hope to avoid the need to delve into the depths of modern compilers (for which the principles have not changed), and request forbearance in this regard. In addition, what follows is not in any way an introduction to programming with the EDSAC code, as we will gloss over all but what is significant for our purposes.

For our purpose, the primary innovation of the EDSAC programming system is the Initial Orders program. What the Initial Orders does is treat the input lines of EDSAC code as data to be transformed rather than commands to be executed. The resulting output is machine code which could be executed on the EDSAC. Philosophically, what the

[18]It is a common objection to this claim that there are indeed variables present in a machine code, since the values stored at memory address may be changed. These, it is claimed, are variables. While it is true that machine code programs make use of memory addresses *as though* they are variables, the *language* has no capability of expressing or representing this. Instead, explicit reference is only made to constant addresses whose properties (values) may change in order to emulate variables. Though the addresses emulate variables, they do so indirectly, and the fact that they emulate variables has much more to do with the knowledge and understanding of the programmer than anything to do with the syntax or semantics of the language. On its own a machine code is restricted to the use of fixed constant symbols denoting memory addresses.

Initial Orders do is free EDSAC code from some of the constraints imposed by the hardware — something we will call "memory address independence." In effect, the Initial Orders allowed the EDSAC to execute programs written in a "non-native" language. A full description of the EDSAC machine can be found in [Wilkes et al., 1951].

A significant part of the role of the Initial Orders is similar to the transliterative process known as "assembly." The Initial Orders is responsible for taking commands, written in a human readable form, and placing the associated machine code commands at an appropriate place in the memory of the machine. For our purposes, this is the least interesting aspect of the EDSAC programming system, as it does not differ significantly from machine codes. Take as an example the EDSAC code statement:

A 6 D

After processing by the initial orders, the following machine code command is placed in the store:

$$\overbrace{1.1100}^{Command}\overbrace{00000000110}^{Address}\overbrace{1}^{L|S}$$

The numerical value assigned to A is -.25. The leading 1 indicates a negative value, and the following four digits, with decimal value .75, are the binary complement of the absolute value of A.[19] The following eleven digits indicate address 6, and the final digit indicates that address 6 is to be treated as a long memory location.[20] The numerical interpretation of this command is the same as in the example machine code program in table 4.1.

In discussing the EDSAC programming system, we must stress that EDSAC code, as a programming language, was never input directly into the machine. It was input as data

[19]Strictly speaking, A was assigned the value -4, which is then scaled by 2^{-4}. As all code values are so scaled, it seems simpler and more straightforward to speak of them as assigned their scaled values. The role of code letters, like "D" is discussed in more detail below.

[20]This simply means that the memory location in question (6) is to be treated as including also the subsequent memory location, effectively doubling the length of the binary string that can be read from the memory location. This is primarily useful when results of a higher precision than can be stored in a single memory location are required.

to the Initial Orders program, which then placed appropriate machine code commands in the appropriate locations. Given this, EDSAC code consists of two main parts. The first of these is called the "order code." This is the part of the language which is processed in the assembly method just discussed. The symbols are transliterated (with the exception of code letters) and the result is a machine code command placed in the EDSAC memory store for future execution. The second, more innovative part of the EDSAC code consists of commands called "control combinations" and "code letters" which together direct *the Initial Orders* on how to process the EDSAC code, but do not themselves directly result in machine code commands being placed in the store. There are in addition to these what are called "pseudo-commands." These are essentially numerical values masked by command syntax. They are the least interesting for our purposes, and what we say about them will be accordingly brief.

The EDSAC code contains fourteen code letters (one of which is "D" from the example above). Thirteen of these code letters are "terminal" meaning that their presence marks the end of an EDSAC order code statement. In fact, every well-formed order code statement in EDSAC code terminates with one of these terminal code letters. The fourteenth code letter (π) is the sole non-terminal code letter. It may immediately precede any terminal code letter, but may not precede itself. The most interesting function of code letters for our purpose is that when processed by the Initial Orders they result in the addition of a certain value to the address value specified in the command. Just which value is added is determined by the code letter (often in conjunction with one of the control combinations). To see how this works, consider that the final digit of the machine representation of "A 6 D" indicated that address 6 was to be treated as a long storage location. During input, the Initial Orders first computes the numeric value of the command code (A), then translates the address into its binary machine representation. The final step is to add a value to this address according to the terminal code letter. In the case of F, the correct terminal digit is 0, which is the default value. Consequently, the number added to the address fields is 0. In the case of D, the terminal bit must be set to 1, and this is done by adding 2^{-16}

to the numerical representation of the order code, which causes the machine to treat the address as a long memory value.

Of course, from the perspective of the machine, what is really happening is that the value at a certain memory address is being added to the numerical address value. The values corresponding to code letters are loaded into main memory along with the Initial Orders program before program input commences. When the Initial Orders receives a command code terminated with "F," the machine is directed to add the contents of memory address 41 to the address value of the order code. During execution of the Initial Orders, address 41 always contains the value 0. "D" functions the same way, but is associated with memory address 43, which has the constant value of 2^{-16}. It is appropriate to think of "F" and "D" as constants of EDSAC code for the following reasons. First, even though they *could* be assigned different values (by changing the values at addresses 41 and 43), doing so would prevent nearly all EDSAC code programs from being executable. That is, the *language* of EDSAC code assumes and depends that these values are constant. On the other hand, "F" and "D" could be fairly easily "remapped" to addresses other than 41 and 43, by means of appropriately revising the Initial Orders program itself. All EDSAC code programs would continue to function just as well as before. This is why, of the two plausible semantic values for "F" and "D" (memory address and numerical values), it is most appropriate to think of "F" and "D" as constants denoting 0 and 1 respectively.

What we have just considered is the first way in which the use of the Initial Orders enables certain features of the language to exhibit memory address independence. The semantics of the code letters are freed, by the internal details of the Initial orders, from the hegemony of the underlying hardware.[21] This is a significant step toward the capability to produce a subroutine library, but it is incomplete. What is needed is a means of specifying

[21] It should be noted that the role of these constants, so far described, is limited to this specific function with respect to the memory address component of a command code. Though it might make sense to think we could add "D" to itself, to arrive at the value "2", doing so within the language of EDSAC code would require appropriate syntactic rules, which were never included in the language. We will, however, consider an additional use of code letters which is a step in this general direction.

the offset location of the subroutine within the larger programs in which it is included. A constant *could* be used for this, if say "Q" denoted the value 52, but this would not improve matters much as it would require that the subroutine always begin at memory address 52. What is needed is a symbol which denotes the offset regardless of where the subroutine is placed. That is to say, a variable.

The EDSAC code letter reserved for this role is "θ". In first order logic, the inclusion of variables is facilitated by stipulation. In the syntactic rules of the language, certain terms are reserved as variables (usually $x_1, ..., x_n$), and their semantics is likewise stipulated in interpretations of the language.[22] In the case of a programming language, it should be clear that this method is not possible, unless "stipulation" is taken to include the construction of mechanisms to facilitate the transformation of statements using the expressions into machine code. We cannot simply declare that the machine must treat θ as a variable, it must be enabled to do so. For the EDSAC code this is accomplished by means of code letters together with the aforementioned control combinations.

Control combinations are essentially directions which tell the Initial Orders program how to treat subsequent EDSAC code input, but do not themselves result in code being placed in the store of the machine. As an example, consider the control combination "E *m* K P F". Before any program runs on the EDSAC, the Initial Orders must be executed to transform and place the EDSAC code program in the store in executable form. However, the Initial Orders is a program itself, so at the end of processing, execution must be passed to the first machine code command of the input program. "E *m* K P F" results in a transfer of control to memory address *m* (without a corresponding control transfer command being added to the target program), where *m* is the predetermined starting location of the program .

This illustrates that control combinations play a number of roles in addition to facilitating the use of variables. Although control combinations appear compositional, we

[22] As we mentioned in Chapter 3, the semantics of variables is strictly speaking entirely dependent on quantifiers. An analogous point, discussed below, also holds for "θ".

argue below that they should not be thought of as such. In fact, despite the appearance that many additional control combinations can be formed, they are all effectively summarized in Table 4.5. Here, we will only discuss the one which directly concerns the treatment of θ as a variable. A more comprehensive treatment follows in the analysis section.

Table 4.5: EDSAC control combinations.

Control Combination	Description
T *m* K	This causes the next order processed to be placed in storage location *m*.
G K	This causes the address value for the next command location to be placed at address 42 (which corresponds to the variable code letter θ).
T Z	This causes the value at address 42 (which in most cases is an address value) to become the next location at which the next processed command will be placed.
E *m* K P F	This stops the reading of orders (that is, it halts the Initial Orders program), and causes control to be transferred to address *m*, with the accumulator cleared (effectively executing the input program).

The Initial Orders takes EDSAC code input and places lines of machine code in the store by means of internal transfer orders specifying the location at which the next processed command will be placed. Generally speaking, the address in this transfer order advances by one for each EDSAC order code processed (starting with the first memory address not required for the functioning of the Initial Orders). Consequently, translated

orders are placed in the store serially, as they generally should be.[23] Many of the control combinations are designed to manipulate the value in this transfer order, thereby altering the location at which the next order will be placed. The control combination of particular interest to us is "G K".[24] To understand how this control combination results in θ functioning as a variable, we will consider the simple subroutine shown in table 4.6.

Table 4.6: Example EDSAC Subroutine (pseudo-code).

Order Location	Order Code	Explanation
m	S 0 F	Subtract value at address 0 from the accumulator.
m+1	G *m*+4 F	If value in accumulator is less than zero, transfer control to *m*+4.
m+2	T 0 F	Transfer value in accumulator to address 0.
m+3	E *m*+5 F	If value in accumulator is equal to or greater than zero, transfer control to *m*+5.
m+4	T 1 F	Transfer value from accumulator to address 1.
m+5	...	...

This is a simple but sensible algorithm we might actually make use of. It is fairly

[23] Care should be taken not to confuse these transfer orders, which are internal to the Initial Orders and involved in its processing of an input program, with transfer orders that may be part of the program being processed. The latter have no effect on the ordering of commands in memory, but they do affect the order of execution of these commands.

[24] Technically speaking, the control combination is "G 0 K", but a convention of the EDSAC code is to omit leading zeros, including even the singular zero value. In the case of "G K" the omission is particularly appropriate, as any "G *m* K" where *m* is any numerical value will be effectively synonymous with "G K". The numerical value has no effect whatsoever.

easy to see how it is supposed to function, and how memory addresses can be defined with respect to the offset value *m*. Unfortunately, this is *not* EDSAC code, and consequently it is not executable on the EDSAC system (the Initial Orders will not know what to do with an expression like "$m+4$"). The control combination "G K" is used to solve this problem. When the Initial Orders receives "G K" the location in which the next command is to be place is stored at memory address 42. The command code "θ" functions just as "F" and "D" — it results in the addition of the value stored at 42 to the address component of the command. Just as in the case of "F" and "D" there is nothing significant about the link between θ and address 42 — any address would do, provided the Initial Orders are configured to use that address with both "G K" and "θ" in the appropriate way. However, unlike code letters "F" and "D", the value of "θ" is not known at the time a subroutine is written, and in fact "θ" will take different values under different circumstances. In every case, "θ" effectively denotes the offset value of a subroutine, provided that the subroutine is prefixed with the "G K" control combination. The corresponding EDSAC subroutine, which is executable, is the following:

In calling this subroutine "executable" we mean two things. First, that the subroutine may be included within a program at any memory location. More importantly, however, we are saying that the subroutine is well-formed in the language of EDSAC code, despite the fact that the value of "θ" is unknown. In saying subroutine is well-formed, we mean not just that it satisfies the grammatical rules of the language, but also that it is *meaningful*. And this brings up the question of the semantics of "θ". In regard to this there are two possible interpretations. The first is that "θ" in some way denotes an *arbitrary* memory address value. Meaningfulness then comes down to the fact that the subroutine is executable for any value which "θ" might take. Alternatively, we can say that "θ" denotes the full range of its possible values, in which case the meaningfulness of "θ" depends on the fact that the subroutine is executable *for all* values it might take. There is a straightforward analogy to be made, then, between "θ" and a first order variable. The first interpretation, when applied to a first order variable, legitimates the derivational rule

Table 4.7: Example EDSAC Subroutine (executable).

Order Location	EDSAC Code	Explanation
N/A	G K	Causes address in current transfer order (m) to be placed at location 42.
m	S 0 F	No change.
$m+1$	G 4 θ	Causes value at 42 to be added to the address $(4+\theta)$.
$m+2$	T 0 F	No change.
$m+3$	E 5 θ	Causes value at 42 to be added to the address $(5+\theta)$.
$m+4$	T 1 F	No change.
$m+5$	...	...

of universal generalization (if F holds for an arbitrary a, then $\forall xFx$). The second interpretation is the intuition underlying the semantics of the universal quantifier. And, of course, in every context in which it matters, the two interpretations are effectively equivalent, both for first order variables and for "θ". Given both the similarity of role and semantics between "θ" and a first order variable, it seems more than reasonable to consider "θ" itself to be a variable in its own right.

The variable code letter θ and the control combination "G K" therefore make it possible to write executable subroutine segments of EDSAC code which make no reference to absolute memory store locations. They are effectively memory address independent. Such code may consequently be inserted anywhere (at any memory location) within any EDSAC program, and be both executable and reliable in carrying out the same transformations on values. This is precisely what is needed in order to construct subroutine libraries.

The inclusion of genuine numerical constants and variables in the EDSAC code fa-

cilitates another useful feature that is of interest from a formal perspective. A major theme of the development of programming languages is a push to increase the range of contexts in which a program or subroutine is executable and useful. A subroutine is more valuable if it is useful in many contexts. We have so far been considering "context" in terms of locations within the memory store. Taken this way, memory address independence is a kind of context independence.

But what counts as a context can nearly always be thought of in a number of different ways, and it should not be surprising that there are others worth considering here. Consider a subroutine which only manipulates numerical constants of the sort described above (e.g. "D" and "F"). Such a subroutine could certainly be placed anywhere within the memory store, which is excellent, but it is still of rather limited use. As a function which only ever operates on fixed, constant inputs, it will always provide the same, fixed, constant output. The "inner workings" of such a subroutine make little difference, and the whole thing may as well be replaced by a single constant value.

Useful subroutines are those which are capable of receiving a range of inputs and producing a range of outputs. In machine codes, this is accomplished in practice by altering the values at a particular memory address, which is the approach the EDSAC code was designed to avoid. The control combinations and code letters together allow subroutines to be written which may accept different arguments at different times without any direct alteration of their code. The act of inputting a particular value (out of a range of values) into a subroutine is generally referred to as "passing a parameter." A parameter is similar to a variable in that, where a variable may be generally understood as a placeholder for *some* value (often unknown), a parameter is something like a receptacle for *some* value which has not been predetermined. Whereas a variable is always assumed to have *some* value, a parameter must be "passed" a value before it is proper to speak of it as having any value at all. The difference is subtle and, practically speaking (and this is all the more true in contemporary computers), parameterization makes use of the same techniques as variables.

The first method of passing parameters within EDSAC code is to preset them prior to processing by the Initial Orders. This is to say, a subroutine may be written which includes one of the code letters functioning as a variable. The subroutine itself does not actually include the input value, but instead uses the code letter to stand in as a "receptacle" for some unknown value. When the subroutine is to be included in a larger program, the programmer may prefix the subroutine with a command which places a specific value at the memory address linked to the code letter. On execution, this will be the value of the parameter (code letter) available throughout the subroutine.

Preset parameter passing occurs during processing by the Initial Orders, which means that preset parameter values are static during execution. It is fairly reasonable to expect, however, that a subroutine might be used repeatedly within a single program, and that each time there is a desire to pass different parameter values. If, for example, one wishes to produce a table of square roots, one will wish to invoke a square rooting subroutine multiple times and pass a different parameter value each time. With preset passing, this would require a separate copy of the subroutine for each parameter value, if these values could even all be known in advance.

The second method of parameter passing is designed to solve this problem. It effectively allows a parameter to be passed into a subroutine each time the subroutine is executed, and thus a different parameter value may be passed at each execution. A technical explanation of how this is accomplished, however, would go far beyond our interest in the topic. Generally speaking, the technique involves the use of multiple variable code letters, in addition to θ, which cause the Initial Orders to calculate, first, the location of the subroutine itself, and then the value of the address immediately following the command which transfers control to the subroutine. Then a variable, say, H, may be set to contain the calculated address. H may then be used within the subroutine to invoke the value at this address, albeit somewhat indirectly, in a way which is insensitive to just what address is used (which will depend on the subroutine's location in memory).[25] The net effect is

[25]For those familiar with the technical nomenclature, H functions essentially as a constant pointer.

that a subroutine may be written which takes as a parameter value whatever value is stored at the memory address immediately following the command passing control to the subroutine. The use of code letters and control combinations makes it possible to do this without explicitly specifying any parameter values *or* the address where the parameter value can be found. Consequently, an EDSAC code subroutine may be written which is independent both of memory location and parameter values.

Analysis

At the end of our discussion of machine codes, we put forward a thought experiment designed to show that machine codes have a distinctive "transparency" with respect to the operation of the machines for which they are designed. In this example, we considered an idealized machine capable of receiving equations input in a mathematical language and producing solutions to these equations without further aid. In the previous context, the idea was supposed to highlight the difference between a machine-transparent language, and one which is not. But saying that a mathematical language is not transparent with respect to the operation of the computer is not at all the same as saying that the language may not be transparent with respect to something else. In particular, if the governing hypothesis of this chapter is true, and if the languages of mathematics are indeed formal, then we would expect these mathematical languages to also be "transparent" with respect to the objects for which they are constructed.[26] This is not a point we have the time or energy to argue for, nor is it one on which the following discussion rests. It is interesting in this context, however, for two reasons. First, the ideal machine in the thought experi-

[26]Though we do not have time here to present detailed arguments for this point, we do think that it is very plausible from an intuitive perspective. If we take a run-of-the-mill user of our standard arithmetical language, and start asking them questions about what they are "doing" as they carry out the manipulation of symbols, they will almost certainly respond that they are carrying out a certain operation or set of operations. If asked what the number five is, they will almost as certainly present the Arabic numeral "5" and, more often than not, insist that it is, in fact, the number five. It is not until one begins learning about arithmetical systems with different bases and symbols that one begins to distinguish *the number five* from *the numeral "5"*. In thinking this way, we follow [Frege, 1960].

ment was, and to some extent still is, an ideal goal for programming language designers. The obfuscatory nature of machine code with regard to what is going on *mathematically* is more of a hindrance to writing useful programs than the transparency into machine operations is a benefit. Second, our claim in what follows is that EDSAC code is an important first step toward this ideal, and that certain important syntactic elements bear relationships to invariances in the mathematical domains of functions to make these syntactic features transparent with respect to this domain.

The EDSAC code has one foot in each of two different worlds, as it were. On the one foot, certain syntactic elements (e.g., the command codes) are still constrained by, and transparent with respect to, the operation of the machine. On the other foot, the innovative elements of the language (permitted by the use of the Initial Orders) break free from the constraints of the machine and move the overall representation *closer* to that of a mathematical language. These syntactic features are governed by semantic constraints of a functional nature. Not surprisingly, each of these obscures the other. When the order code is employed, it takes considerable work to understand what mathematical function is being computed. When variables like "θ" are employed, it is no longer possible to understand the underlying machine operations involved without a detailed understanding of the Initial Orders.

A large part of the discussion that follows focuses on the important role that the Initial Orders plays in enabling programs written in EDSAC code to break free, at least in part, of machine oriented semantics. Furthermore, in large part we will be focusing on the *differences* between machine code and EDSAC code, and the resulting difference in terms of invariance.

The syntax of EDSAC code is slightly more complicated than for a machine code, as is the case with all of the languages that follow. In the interest of expediency, we will limit our discussion of syntax to syntactic categories when possible.[27]

The syntactic categories are as follows (with the total number of symbols in each

[27]For more specific detail, see [Wilkes et al., 1951].

category):

1. Command codes (29)
    (a) Command codes taking numerical address arguments (14)
    (b) Command codes taking numerical arguments. (5)
2. Numerical constants (2048)
3. Standard code letters (15)
    (a) Terminal code letters (14)
    (b) Non-terminal code letters (1)
4. Control combination code letters (4)

In all, not counting numerical constants, the language consists of forty eight position specific symbols.[28]

All EDSAC code expressions follow the same general pattern. They consist of a command code symbol, followed by a numerical value, optionally followed by a non-terminal code letter, and terminated by a standard terminal code letter or a control combination code letter (all of which are terminal). Thus, every expression consists of three or four symbols. As noted above, some commands, most notably "G K", appear to diverge from this pattern on account of a convention of omitting all leading zeros. Hence the syntax of "G K" is implicitly "G 0 K", and likewise for other commands.

It is reasonably easy to see the similarities between this syntax and that of any machine code, and since the EDSAC order code has the most in common with machine code, it makes sense to begin our analysis with it. A statement is an order code if and only if it has the following form: [command code][numerical constant][(optional) non-terminal

[28] Although a few of the symbols are used in multiple categories, they always have distinct, unrelated semantics, and the grammar is such that it is never ambiguous which meaning should be invoked. They are effectively distinct elements of the language.

code letter][code letter]. Aside from the addition of code letters, we can see that order code commands and machine code commands share a lot in common syntactically. It is tempting to say that this is because the order code commands are transliterated directly into machine code statements, and that for transliteration to be possible, syntax must be essentially the same. Such a view overestimates the constraints placed on a source language (e.g., EDSAC code) by a target language (e.g., machine code). The use of intermediary processing by the Initial Orders provides numerous other possibilities. For example, EDSAC code might have consisted of a single, atomic expression for every possible machine code command. Although very inefficient, the Initial Orders would then need to look up the machine code command associated with each symbol. Or, more plausibly, the language could have been made to look more like a mathematical language (which we know is possible, since other languages do this). Instead, the EDSAC order code retains the general syntactic structure of functions being applied to arguments (with functional application represented as concatenation, just as for a machine code). This is not a choice which is forced by the syntax of machine code. It is instead the result of the same semantic constraints that are at play for machine code syntax. That is, the order code portion of EDSAC code is designed to direct or describe machine operations. Just as before, these operations are all interpreted as monadic functions, and thus the same constraints apply to the EDSAC order code.

There are, however, two significant differences we need to address. The first is that acceptable numerical constants for order code statements are constrained based on command codes. For some command codes, the values may range from 0 to 511, and for others from 0 to 2047. The first group we have annotated above as those "taking address values as arguments." This is not, strictly speaking, a syntactic rule. The correct syntactic rule would be to simply list those commands and stipulate that any statement with a command code from the list followed by a numerical constant with a value greater than 511 is not a valid statement. The fact that there is a reason for this, that the command codes denote operations taking address values as arguments, is a *semantic* fact about those

operations. It is, furthermore grounded on a *material* fact having to do with the specific hardware of the EDSAC computer, which only had 512 memory locations. As we will see in section 4.1.3 on "machine independence," this is a characteristic that makes EDSAC code "less formal" in a particular respect than higher level languages. However, it is still an important fact that this syntactic rule corresponds to an invariant fact about the EDSAC hardware — namely the presence of only 512 memory locations (numbered 0-511). In this case, the domain of invariance consists of all operations the machine does or can carry out on memory addresses. The class of transformations consists of functions which take every such operation to every other operation. It is an invariant of these operations that they only operate on memory addresses 0-511. Of course, this invariance is grounded in material facts about the machine, which could be altered to have more than 512 memory addresses. In such a case, however, it is easy to see that there would be compelling reason to alter the syntax of the language to account for this change, and that the need for such a change would be driven by the semantics of the command codes, which is in turn altered by the change in material conditions.

A similar fact holds for the command codes which constrain the numerical constants to have values less than 2048. This restriction is grounded on the size of each memory address. That is, a certain number of digits in each memory location are taken up by the machine code operation command (transliterated from the command code letter). The remaining space is large enough to hold a numerical value no greater than 2047. Again, from a syntactic perspective the corresponding rule is merely that a certain class of constants can only be followed by a numerical constant with a value between 0 and 2047. In this case, the constraint does not come from the semantics of the command code, but from the general semantics of the statement as describing a machine operation. As such, the order code statement must express operations which are possible on the machine, and this requires that the supplied values be able to fit within the memory register of the machine. All of the command codes in this category denote scaling operations on whatever value is

stored in a privileged memory location.[29] It is tempting to think of this as a mathematical scaling operation, in which case the syntactic limitation has nothing to do with the semantics of the operation. There are no mathematical limits on how numbers may be scaled. The trouble with this view is that it strays too far from the intended semantics of the order code, which is to describe and direct operations *on the machine*. A scaling operation on the EDSAC, bearing whatever resemblance it may to general mathematical operations, is limited in certain ways by the hardware. In this case, if we take as our domain of invariance all memory locations on the machine and we take a class of transformations which maps each memory location to every other, an invariant will be that each memory location is at least a certain length (e.g., the length of the shortest memory location). In nearly every case, all memory addresses have the same length, so this length itself is invariant. In any case, since the EDSAC syntax does not contain rules based on particular memory addresses, it must be possible for every permissible command to fit within every memory address, and so there will always be an upper limit to the values of these numerical constants, otherwise the commands would be too long (and hence no longer describe a machine operation).

From our perspective, one of the most interesting differences between EDSAC code and machine codes is the introduction of code letters. The function of code letters, as stated above, is the addition of a value stored at a memory location to the value of the numerical constant of the command. We have already argued that the semantics of these code letters is best thought of in terms of numerical constants and variables. This is because the role that they play in EDSAC code, from a linguistic perspective, is entirely independent of — that is, invariant with respect to — the memory address linked to the symbol. Below we will address specific details about the treatment of code letters as variables. At this point, it will be useful to focus on their role as denoting numerical values (whether constant or variable).

The first thing to note is that the ability for code letters to denote numerical values is entirely due to the intermediary role played by the Initial Orders. Good evidence for

[29] This location is called an accumulator. See [Wilkes et al., 1951].

this is that no corresponding element occurs on the machine code side. Code letters are effectively lost during the translation process. This also indicates that code letters have a different scope than, for example, the command code portion of an order code. This is because, as we have seen, the semantics of the command codes requires that they take only a single value as input. So, the code letters must be evaluated within a narrower scope than the command code. We can represent this implicit scoping by using parentheses as in a mathematical language: [command code]([numerical constant][non-terminal code letter][code letter]).[30] This means that we only need to focus on this smaller scope to understand the relation between the syntax and semantics of code letters, both of which are very straightforward. The syntax is the concatenation of a code letter to a numerical value. The semantics is that the value denoted by the code letter is added to the numerical value. We traditionally conceive of addition as a dyadic operation denoted by the symbol "+". This is necessary in standard arithmetical languages to both distinguish the operation of addition from others, and also to distinguish arguments from one another (other approaches will work, but concatenation is not one of them). In this context, however, there are no other operations in need of denotation within the scope we are considering, nor is there any danger of not being able to distinguish arguments if they are concatenated. So, concatenation plays the denotational role that the "+" sign plays in arithmetic.[31]

The syntax of the expression governed by this addition operator is: [numerical constant][code letter]. A classical approach to number theory is to define various classes of numbers in terms of equivalence classes based on other classes of numbers, ultimately all based on natural numbers. Equivalence classes are, themselves, defined in terms of invariant features (e.g., those features which make members equivalent). The classical ap-

[30]Of course, EDSAC code has no need of grouping symbols, since there is no possibility of significant syntactic ambiguity. The only syntactic ambiguity occurs when the non-terminal code letter is employed, and the associated semantics cause every scope interpretation to result in the same evaluation. In practice the Initial Orders first adds the value of any non-terminal code letter to that of the numerical constant before processing the code letter.

[31]Of course, functional application is also denoted by concatenation of a command code to a numerical value, but this is syntactically position-specific and is also in a different semantic scope, hence no ambiguity arises.

proach to defining natural numbers is in terms of sets of sets with specific cardinalities. In each case, the cardinality is a stipulated invariant, and it is invariant under transformations between sets within each cardinality class. The code letters and numerical constants functioning in the addition context, therefore, also denote semantic invariants.[32]

Addition itself, on its various common definitions, is an arithmetical invariant. If addition is defined by means of a successor function "$^+$" and recursion, we have: $0 + a = a$ and $a + (b^+) = (a + b)^+$. This characterizes a relation which is insensitive to the particular identities of natural numbers, which is to say it is invariant over transformations from any pair of natural numbers to any other. Alternatively, addition may be defined set theoretically as the cardinality of the union of two disjoint sets. In this case, addition characterizes a relation between the cardinalities of two disjoint sets and the cardinality of their union. Again, since the definition is set up without reference to any particular sets or their properties, other than that they are disjoint, the relation is invariant with respect to all transformations between disjoint sets. Furthermore, for any two disjoint sets with cardinalities α and β, the cardinality of their union will be the same. Ergo, the relation is also invariant with respect to transformations between equipolent sets. On either account the addition operator is an invariant.[33]

[32]It is worth noting that address values have different semantic denotations based on syntactic context. As address values, they are proper names of addresses, and as such are not appropriate arguments for any arithmetical operator. For any list of names, however, an operator could be assigned which would take that name and an integer value as arguments and map these to other names. For example, $f(Jina, 2)$ could map to *Josephine*. Such an operator could be denoted by "+" and have the same syntax as the arithmetical symbol. If numerical names are also assigned systematically, for example serially, then the syntactic representation would *appear* the same as that of addition. This, for example is what occurs with "pointer arithmetic" in many programming languages. It would be easy, then, to confuse this function with addition. However, not only do the semantics differ completely, but the address-addition operation will only be defined for a relatively short list of numerical names and integers. In the present context, we can be sure that the addition operation denoted by concatenation is addition, and not address-addition, by the fact that it takes arguments which are not address values (e.g., 512-2047) without any problems, and produces outputs which in many cases are not valid memory addresses (names). The resulting value is then treated either as a value or a name by the command code, according to its own syntax (regardless of whether it is a valid name or not).

[33]There are, of course, many functions which are invariant in this same way. Just as was the case for logical notions, there are many more arithmetical invariants than are denoted by constants in our arithmetical languages. Fortunately, our claim here is only that the syntax of a formal language is constrained by invariances in the underlying semantics, not that it exhaustively captures all of these. That is a question

The syntax of order code concatenation is then constrained by the need to represent addition, and as such it is not syntactically restricted to particular arguments based on the final evaluation. We have an operator denoted by concatenation, and the two requisite arguments denoted by numerical constants and code letters. Without these characteristics, addition could not be represented in the language.

We can tell that code letter concatenation is arithmetically oriented (rather than machine oriented) because of the aforementioned lack of syntactic limitations. For example, there is no rule preventing the concatenation of "D", which denotes the number one, to a numerical value of 2047. Within the scope of the arithmetical operation, this is valid, and it is technically a well-formed EDSAC statement. It will not, however, cause the machine to function in the expected manner. The same thing can be done by concatenating other code letters to memory address values to reference non-existent memory. Thus, use of these code letters may result in problematic machine code commands. These are facts that the programmer herself must keep track of.

These potential problems are the direct result of mixing invariance domains. On the one hand, the machine-oriented syntax is structured with respect to invariant features of the machine, its components, and its operation. The machine oriented elements of the syntax are as constrained as possible by the hardware of the underlying machine, but this constraint is in direct conflict with the semantics of code letter concatenation as arithmetic addition. Furthermore, the syntax (or semantics) of code letter concatenation could not be further constrained without sacrificing the original purpose of allowing the construction of address-independent subroutine libraries.

When it comes to the second component of the EDSAC code, control combinations, [Wilkes et al., 1951] is not particularly forthcoming regarding syntax. Instead, a list of control combinations is provided. Some guidelines are also given for completing certain control combinations, but these shed little light on any internal structure these commands may have. There are, however superficial reasons for thinking that control combinations

more closely related to definability in a language, which we are not taking up.

have some internal structure. First, they are called "combinations," which makes us think that they must have parts which are combined in some way. Second, control combinations, strictly speaking and purely from the perspective of symbol ordering, share the same structure as order code commands. They consist of a command code, followed by a numerical value between zero and 2047 (often zero), follow by one or two code letters. Finally, the surface appearances of similar control combinations generally have systematic similarities in the provided descriptions of what the control combinations do, and what differences there are seem to track structural differences.

A deeper reason, though, for thinking control combinations may have an internal structure is that in several places in [Wilkes et al., 1951], "K" and "Z" are referred to as "code letters." For example, "It may be noted that the operation of code letter Z is always equivalent to that of K and θ combined."[34] What is interesting about this is that "K" and "Z" are not included in the list of code letters (along with "θ") presented earlier in the text, nor is anything else mentioned about their role other than the above and that they are "terminal" code letters. However, some of the example uses of control combinations makes it appear as though they do not terminate with either "K" or "Z". For example, "E Z P F".

Since [Wilkes et al., 1951] is not explicit about whether these control combinations should be treated as atomic constants or as compositional, in most places suggesting the former but sometimes implying the latter, understanding which is the case requires some further investigation. In particular, we know that the semantics of programming languages are compositional. That is, if a statement in the language is compositional, the meaning of the statement is also compositional in terms of the semantics of the particular atomic constants. This being so, one way to investigate the syntax of control combinations is to investigate their semantics, which in this case means a careful examination of the Initial Orders program, fortunately published in its entirety in [Wilkes et al., 1951]. We have undertaken this investigation, but the details are far to tedious and digressive to recount

[34] [Wilkes et al., 1951], 161.

here. The results, however, are as follows.

The most significant internal detail of the Initial Orders is that it distinguishes between command codes and control combinations based on the symbol located in the terminal (code letter) position. It is primarily for this reason that "K" and "Z" are referred to as "code letters," i.e., because they occupy the terminal position which indicates whether the complete statement is an order code or a control combination. This parsing rule means that *any* statement terminated by a control combination code letter will be interpreted *as if* it were a control combination, regardless of what other symbols occur in the statement. In the case of command code letters which are assigned negative values by the initial orders, the particular command code does not matter (only the sign digit is treated as significant). "G" (familiar from the important control combination "G K") is one such command code. It seems to have been chosen arbitrarily from the group of nine negatively valued command codes (all of which have effectively the same semantics).

The same cannot be said of the nine other command codes with positive numerical values. In these cases the Initial Orders uses the supplied command to partially rewrite one of the machine code commands which makes up the Initial Orders. The resulting command is not part of the input program, but will eventually be executed as part of the Initial Orders, and as such it has a dramatic effect on the functioning of the Initial Orders. Many of these are quite dangerous in the sense that they effectively destroy the ability of the initial orders to process the input program, and result in nothing useful. It is for these reasons that Wheeler only ever lists control combinations beginning with "T" and "E" (in addition to "G"). So, from a semantic perspective, there are only three possible components to occupy the command code position of a control combination, the semantic component denoted by "G" (and eight other code letters), together with those denoted by "T" and "E".

The numerical component of a control combination is always interpreted as a reference to a memory address, and hence can range from zero to 511. The non-terminal code letter "π" may be used to indicate that the address is to be treated as long. In this

context it is no different from that of the order code, where the syntactic construction of a numerical constant followed by "π" denotes an arithmetical addition operation.

Every presented control combination then terminates with either "K" or "Z". Our examination revealed that two other symbols would result in an expression being processed *as if* it were an order code, but that the results would be either dangerous or meaningless (e.g., not result in any machine operation).

Unlike the case of standard code letters, "K" and "Z" do not directly denote numerical values, and their concatenation to complete a control combination cannot be interpreted as arithmetic addition. This is so even though, as the quotation above indicates, "Z" functions in a semantic sense as a combination of "K" and "θ".[35] The code letter "K" serves only to indicate that the statement is a control combination, while "Z" both indicates a control combination and causes the value stored at address 42, i.e., the value denoted by "θ" to be added to the address value where the command will be placed.

The final result of all of this is that the control combinations which can be thought of as "meaningful" from the perspective of the Initial Orders (the consumer of the language) are actually quite few in number. They consist essentially of the list Wheeler provides in [Wilkes et al., 1951], reproduced in Table 4.5 above.

In addition to those commands, there could in principle be control combinations in which "G" is replaced by any of the eight other negative valued command codes, and also combinations of the form "G *n* K" and "T *n* Z" in which "*n*" is non-zero. However, these expressions are all synonymous with "G K" and "T Z", respectively, as neither makes any use of the supplied value.

Apart, then, from being able to construct synonyms for a few of the listed commands, there are no other legitimate control combinations. This suggests that, although the symbolic structure of these statements makes it appear as though they have a compositional grammar, the way that the statements are processed by the Initial Orders means

[35] It cannot, of course, function this way in a syntactic sense because no valid statement of EDSAC code can have more than one terminal code letter, thus statements cannot end in "θK" or "Kθ".

that any appropriate grammar would effectively list the above expressions as characteristic constants. The only variability, then, would be the numerical values supplied to some of them.

It is our opinion, therefore, that these control combinations are best treated as atomic constants of the EDSAC code. A control combination statement would then consist of one of these constants and a numerical value infixed within the string of symbols that makes up the constant (essentially for reasons having to do with machine parsing).

There are some apparent anomalies from this description, as Table 4.5 contains commands "E *n* K P F" and "E Z P F". One important characteristic of the Initial Orders is that it reads the symbols of an input program in sequence, and does not pay any attention to line breaks. Each of these apparently anomalous commands actually consists of two distinct, well-formed EDSAC code statements. The first recognizably consists of "E *n* K" and then "P F", which plays a special role discussed below. The second one is similar.

These sequences of statements are illustrative of one of the most distinctive characteristics of the EDSAC code. Control combinations introduce syntactic rules governing the composition of groups of statements, not just individual statements. This contrasts with most logical languages, in which syntactic rules only govern the well-formedness of sentences, not groups of sentences. Of course, the derivational rules of logical languages *do* involve groups of sentences, and derivations can of course be characterized syntactically, but there is no sense in which a derivation can be ill-formed. A sequence of sentences either is a derivation, or it is not (e.g., it is invalid), but a sequence of well-formed sentences cannot itself be ill-formed in a grammatical sense. This is not so with sequences of EDSAC statements involving certain control combinations.

The simplest example is "T *n* K". The effect of "T *n* K", as stated by the description given in Table 4.5, is to place the next input command at memory location *n*. Any "T *n* K" command is only meaningful when followed by a valid order code. "T Z" functions in a similar manner.

A different set of rules governs the use of "G K", which results in the placement

of an address value at address 42 (denoted by "θ"). The rule described by Wheeler is that "G K" must be prefixed to the start of every library subroutine fed into the Initial Orders. This is because library subroutines rely on "θ" to indicate the offset of the first subroutine command from memory address zero. "G K" could be placed elsewhere in code, but in most cases it would have no effect at all. At worst, it would result in an unpredictable, "broken" program. Furthermore, EDSAC subroutines will not function without this prefix, as the resulting offset would be zero (in the best case) or whatever value is currently denoted by "θ". Thus, EDSAC subroutines are only well-formed if they begin with (or are prefixed with) the control combination "G K". "G K" has no other role in the language.

Finally, the compound control combinations "E *m* K P F" and "E Z P F" each contain a special "pseudo-command," "P F". Each of the "E" commands results in the EDSAC executing next a certain statement, stored at memory address *m* or 42, respectively. "P F" is a very special expression of the EDSAC code, as it is the only expression with context based semantics. In general, "P F" is a "pseudo-code" (discussed below) resulting in a value of zero being placed at a memory address. When "P F" follows a control combination (and in other specific contexts) it instead causes the Initial Orders to prepare the arithmetic memory registers for program execution by setting them to zero. Thus, although the "E" commands could be used in principle apart from "P F", it in general does not make sense. Although "P F" itself can be used apart from control combinations, it has a different meaning.

We will not elaborate further on the syntactic characteristics of the order code, as we feel we have explained enough to address the connection these syntactic features have to semantic invariants. For this purpose, the control combinations can be divided into two categories based on their semantics. The first group contains those commands whose semantics cannot be interpreted as other than about the functioning of the Initial Orders. That is, these commands direct or describe the functioning of the Initial Orders in much the same way as order code expressions direct or describe the functioning of the

EDSAC machine. This group consists of every control combination except for “G K” and its synonyms. Each of these statements consists either of an atomic constant expression (“T Z”, “P F”) or a constant command expression with an infixed address value argument (“T *m* K”, “E *m* K”).

The internal syntax of the non-atomic statements is the same function-argument syntax we saw for the order code. The atomic statements, even including “G K”, have no internal syntax essentially because they are bound to a constant address (often 42). This is why “T 511 Z” is synonymous with “T Z”. The constant address is used regardless of the numerical value supplied, and so the value results in a synonymous command.

Because these commands involve an address value, it might be tempting to think that they denote operations that the Initial Orders carries out (as opposed to the hardware itself). In a certain sense this is true, insofar as the commands direct the Initial Orders to place machine commands at certain addresses. But it is important to recognize that this kind of direction does not go very far. In the case of machine code, the computer will do *nothing* without commands. The commands direct the entire operation of the machine. Control combinations are far from this level of detail. For the most part, the Initial Orders is capable of functioning on it’s own. There is no need, for example, for the programmer to tell the Initial Orders what numerical value to associate with a certain command code, or to tell it to process the next command. All of this is undertaken autonomously.

What the control combinations, besides “G K” and “P F”, effectively do is provide necessary information to the Initial Orders. For every command that is input into the Initial Orders, the program needs to know *where* to place the command in the store. This is the role of “T *m* K” and “T Z”. Then, for every program input into the Initial Orders, the program needs to know *where* to begin execution. This is the role of “E *m* K” and “E Z”. These commands are essentially providing locational information to the Initial Orders with respect to the memory store. In saying which semantic invariances are denoted by these constants, we are faced with a seeming overabundance of related possibilities, owing to the fact that programs must ultimately be placed somewhere in the memory store of the

computer (or translated into commands in another language which are placed there), and that execution must begin at one of these locations. These are essential invariances of the way the hardware is built, specifically the memory and central control unit. They are also invariances in terms of the operation of the Initial Orders (and the information it needs). Finally, they are invariances of every program placed in the memory store (in machine code).

The last of these is clearly inappropriate, since our focus in specifically on the language of EDSAC code, and we know that programs constructed in EDSAC code can be written in a memory address independent way (by means of "G K" and "θ"). Furthermore, many modern languages exist which have no need of locational specifications (because the compilers take care of this entirely). The best thing to say is that these constants denote invariances in the operation of the Initial Orders, and that these invariances in the Initial Orders are themselves responsive to the invariances of the hardware.

In terms of supporting our overarching thesis, one can see that the semantics of these constants constrains their internal syntax (or lack thereof). In the case of the atomic statements, it is because they denote themselves all of the necessary locational information (albeit indirectly). In the case of those which take an argument, they do so because a location must be supplied. The Initial Orders, of course, takes care of a significant amount of locating commands by defaulting to the next memory location in the sequence. Thus, these location-based commands need only be invoked when the default needs to be overridden. This default operation, however, is merely a *convention* and does nothing to alter the fact that the Initial Orders requires a location for every order code it processes. That is, if we take as our domain of invariance the order codes processed by the Initial Orders, and consider the class of transformations that takes each of these to every other, in addition to the syntactic characteristics of the order code itself, an additional invariant is that every order code is mapped to a location in the store by the Initial Orders.

Furthermore, the locational semantics of these expressions clearly constrains the higher-level syntax. Namely, if the control combination specifies a location for a command

to the Initial Orders, there *must* be a subsequent order code to be placed at that location, otherwise the sequence of expressions is semantically nonsensical.

A similar thing can be said about "E *m* K" and "E Z", but in terms of EDSAC code programs processed by the Initial Orders. If we take this as the domain of invariance, and consider the class of transformations between each program to every other, the most significant invariant from the perspective of the Initial Orders is that the programs have a starting point, and these commands express this fact. Practically speaking, very little else will remain invariant under these transformations, except for very general facts about being a program, being an EDSAC code program, etc.

We have come to a point where we can discuss the special control combination "G K". Given the preceding discussion, we know that every program and subroutine must receive some location in the memory store. We also know from our general discussion in the preceding section that the code letter "θ" can be used to write subroutines which are reusable in many different programs (and hence in many different locations). "θ" can be thought of as analogous to the English indefinite pronoun "somewhere." So, subroutines are written entirely by means of references to "somewhere." But it is clear that this alone will not result in a meaningful program placed in the memory store. The indefinite "somewhere" needs to become a definite value at some point in order for this to happen. The control combination "G K" is responsible for this. When the Initial Orders processes "G K" it takes the address of whatever the next memory location is (which will receive the first statement of the subroutine) and stores this value at address 42, effectively assigning that value to "θ". Thus, "somewhere" becomes a concrete location by means of "G K". This process is called "binding" in computer science. That is, "G K" *binds* "θ" to the address value of the first argument of the subroutine.

It will be clear to anyone familiar with logic that the role "G K" plays with respect to the variable "θ" is remarkably similar to the role played by quantifiers. That is, variables in first order logic must be *bound* by a quantifier in order for the complete expression to be meaningful. The quantifier essentially tells us how to treat the variable when it comes

time to evaluate the expression. In fact, the need for variables to be bound holds for every logical and mathematical language that uses them. Very often, however, every variable is bound in the same way (e.g., algebra) and hence no symbol is used, but there are still clear rules of interpretation of the variables. It is even worth noting that in [Quine, 1976b], Quine characterizes variables in terms of indefinite pronouns, just as we have done above for "θ".

Our claim, then, is that the semantics of "G K" is that of a binding quantifier statement which explains how to evaluate statements containing "θ" when the time for their evaluation has come. We might think of it as saying "for all memory locations . . .," thus providing "θ" with its semantic value. Then, wherever the subroutine actually is, the Initial Orders can carry out an operation analogous to "universal instantiation" and bind the variable to a specific value. It does this by storing the numerical name of the memory address of the command *immediately following* "G K" at memory address 42. Consequently, whenever "θ" is processed by the initial orders the memory address stored at address 42, which is the offset of the start of the subroutine from memory address 0, is added to the values of the numerical constants of commands within the subroutine, which are themselves offsets from the first command of the subroutine. The result is that the Initial Orders is able to place the correct machine code command, complete with an accurate memory address value, in the memory store. All of this is necessary to resolve the memory-location-independent subroutine into hardware-dependent machine code.

Furthermore, the corresponding semantic invariance captured by "G K", together with "θ", is that the numerical operations carried out by sequences of statements are memory location independent — the operations they carry out are the same wherever they are placed in memory. More strictly, we can take as our domain of invariance all subroutines of EDSAC code written for every possible position in memory. We then take the class of transformations which maps each subroutine to every other which differs at most in its position in memory. What is invariant across these will be the operations carried out by the sequences of commands. By using a single symbolic variable to stand for the

complete range of trivial variances, this variance is "washed out" of associated linguistic expressions. What without the variable would have required a large number of different statements or subroutines with minor differences can effectively be expressed by a single statement or subroutine. With the variance washed out it seems appropriate to say that the resulting statement or subroutine represents an invariance — in this case the invariance of data operations of sequences of commands placed in different locations in memory.

The semantics of "G K" as a binding quantifier also places the syntactic constraint that it must indicate scope. There are, of course, a number of ways of syntactically indicating scope, but given the relative simplicity of the language (no scope nesting, no multiple binding quantifiers, etc.) the most practical thing to do is to have "G K" precede all statements within its scope. A new "G K" marks the end of the preceding scope and the beginning of the next.

The final control combination to address is "P F". The effect of "P F" is to place the arithmetic registers into a consistent (empty) state for execution. The statement therefore denotes an invariant state of the arithmetic register at program execution time. That is, if we take transformations between all EDSAC programs, an invariant assumption of every program is that at execution the arithmetic unit begins in the same (empty) state. "P F" denotes this state, and this denotation makes it appropriate immediately preceding program execution and nowhere else. Hence, "P F" is generally the final statement of any EDSAC program.

This all but completes our discussion of the EDSAC programming system. We will touch only briefly on the final component of the EDSAC code, the "pseudo-commands." Pseudo-commands are EDSAC code statements which begin with the code letter "P". When processed by the initial orders, "P" transliterates to the value zero. No EDSAC machine code command is designated by zero, hence the result is the placement of a numerical value in the memory store. "P F", when not functioning as a control combination, resolves to the value zero, "P 2047 F" to 2047, etc. Pseudo-commands are essentially atomic constants which designate numerical values, with the exception that code letters

are processed in terms of arithmetic addition (just as for the order code). They are essentially data, rather than commands, and hence do not fall within the scope of our discussion.

4.1.3 Degrees of Machine Independence

Early on, when the number of electronic computers was quite small and each new machine represented a significant advancement over its predecessors, programming systems were usually considered to be a technology which accompanied particular machines. Even so, early programmers had a keen awareness that much of the work of programming involved redundancies. We have already seen how the innovations of the EDSAC code enabled a library of reusable subroutines to be created for execution on the EDSAC. It was furthermore becoming clear that, even as the cost of computer hardware was coming down (relative to computing power), the cost of software development was on the rise. Any opportunity for reuse would save time and money. Given that computers were all created to address the same kinds of tasks, it became desirable to be able to reuse programs not just as subroutines within other programs, but on entirely different machines.

If it is possible to run programs written in a certain language on different machines, the programming language is said to be "machine independent." It will not be surprising, given the previous section on memory address independence, that machine independence is characterizable in terms of invariance. We can understand it intuitively in the following way. Given even the rudimentary understanding of the technology developed above, it should be reasonably clear that if two machines are similar in the right way — they implement the same operations in the same way, have equally sized memory stores, etc. — any program that runs on one of them will run on the other. This could be seen as a kind of "machine independence," and it is a property that programs written in EDSAC code have. That is to say, any executable EDSAC code program would also run on any machine which is appropriately similar to the EDSAC. This much is true even of machine code, but it is not what is generally meant by "machine independence." What is meant is

rather that a program written in the machine independent language may be executed on machines which are not similar in relevant ways. It is best to think of this in terms of a distinction between tokens and types of machines. By a "type" of computer, we mean a class of machines that are identical to one another in terms of characteristics germane to computation. A token machine of a given type, then, is any computer belonging to this class. Two token machines could be painted different colors and still be of the same type if they implement the same operations in the same way, have equally sized memory stores, etc. The more limited sense of machine independence just discussed is independence with respect to tokens. The machine independence we are really interested in, however, is that which occurs with respect to types. Henceforth, we use the singular terms "machine" and "computer" to denote *types* rather than tokens. Likewise, proper names like "EDSAC" and "IBM 704" should be taken as denoting the type to which they belong (and which they define), rather than the token machines themselves.[36]

There is, however, one sense in which EDSAC code has a small degree of this type of machine independence. From a practical perspective, the Initial Orders could function equally well on similar machines with larger memory stores. It follows that any program executable on the EDSAC would also be executable on such machines. This independence does not go very far, however, because the converse does not follow — all programs executable on machines with larger memory stores are not executable on the EDSAC. This failure is due to the fact that a programmer must know details about the memory store in order to write a program executable on both machines (that is, she must avoid using any of the additional memory addresses).

The preceding point does serve to illustrate, however, that it is generally not enough to say simply that a programming language is "machine independent." This must be char-

[36]Strictly speaking, this means that when we say a program in a certain language is executable on different machines we should really say that it is executable on tokens of different machines (types). In the interest of simplifying our explanations, however, we will continue with the simpler language with the understanding that the more complex idea underlies it, except where doing so would lead us into error. We will treat named types (e.g., IBM 704) in a similar manner.

acterized against a class of machines (a class of types) across which the program is executable, and this *de facto* means that it is characterized against those features of the machines to which the language is insensitive, which is to say invariant. It is already assumed, for example, that "machine independence" obtains with respect to electronic computers, but not with respect to automobiles or toasters (even programmable ones). Likewise, we have already suggested that full-blown machine independence is something broader than mere "EDSAC-like-machine independence."

Not surprisingly, the most machine *dependent* programs are those written exclusively in machine code. Because these programs make constant and explicit reference to the facilities of the machine (memory addresses, etc.), they are only executable on the machine for which they are written. The technology that made memory address independence possible was the intermediary program called the Initial Orders, which affected a translation from EDSAC code into machine code. The Initial Orders is the first historical example of an "implementation layer" for a language. An "implementation layer" for a language consists of a program (or programs) which makes it possible for programs written in a higher level language to be rendered into machine code for execution on a given machine. They are intermediaries between the language itself, which may look nothing like machine code, and the commands the machine requires for execution.

There are two main approaches to implementation - compilation and interpretation. The main difference between the two can be understood roughly in the following way. A compiler is a program which takes a source program (written in a high level language) and processes it into a form which is executable on the machine (or an intermediary form).[37] At this point, the compiler's job is complete, and the resulting program may be executed at will, or even transferred to a different machine (or token of the same type) for execution.

[37]For a given higher order language, there may in fact be a number of implementation layers (in some cases including even an operating system). However, because we are concerned only the extrema of the process — programs written by humans in a higher level language and binary machine code which is consumed by the computer — the number of intermediaries involved makes no difference. In any case, there is nearly always at least one intermediary assembly language.

An interpreter, on the other hand, carries out this process at so-called "run-time." This means that an interpreter processes the source program into machine executable commands *when the program is executed*. An interpreter must therefore do its work every time the program is executed and must always be present for execution to be possible. In this sense, the Initial Orders is more similar to an interpreter than a compiler.

There are significant technical trade-offs between compilation and interpretation, and there are even ways of combining them. From our perspective, however, they accomplish the same goal. They play the role of intermediary between the machine independent language and the extremely machine dependent machine code which is ultimately executed. Thus, we can say that a programming language "has been implemented" or "has an implementation" if some such intermediary program (or series of programs) has been written which allows programs in the language to be executable on a given machine. In what follows, we use the term "implementation layer" to refer generically to the class of such intermediary programs (including compilers, interpreters, and assemblers).

An interpretation layer, then, enables the machine to execute a program written in the higher level language in question. In so doing, the interpretation layer makes use of the aspects of machine code which machine independent languages need to avoid. It will, for example, be aware of how the memory of the machine operates and will make use of this. It will also contain instructions for interpreting the commands of the source language as operations (or combinations of operations) in the target machine language. This means that implementation layers are generally *not* machine independent (although parts of them may be).[38]

[38]Although it is worth making clear that there is nothing preventing an implementation layer (or its parts) from being written in a machine independent language. In fact, it is standard practice for most compilers to at some point be written in the very language which they are designed to compile. The important thing to recognize is that an implementation layer takes a program in a source language as input (data), and outputs a program in a target language. This does not require the implementation layer to *invoke* or *use* the machine dependent target language, but only to transform the original program into one executable on the specific machine — into a sequence of commands which *themselves* are in the target language. Of course, compiling a compiler written in the source language it is designed to compile *does* require either an antecedently existing implementation layer for the language *or* the use of a technique called "bootstrapping," in which

Although implementation layers are often quite complicated, and even different implementation layers for the same combination of language and machine may differ significantly, we can say something about the process in general. First, we recognize that every high level programming language, just as every formal language, consists of atomic constants and variables. At the most basic level, implementation consists in processing and resolving these two groups.

With respect to variables, the implementation layer determines the amount of memory required to hold a value of the type specified, and ultimately the variable is assigned a location in memory. In most cases, it does not matter where in the memory the variable is "located" as the implementation layer takes responsibility for keeping track of these and supplying the correct memory address whenever it is required.

In most higher order languages, there are effectively two types of constants, those denoting constant values (data) and those denoting operations. In the first case, the implementation layer will either supply the denoted value or a reference to a memory location in which this will be stored. The implementation layer translates operational constant symbols into whatever machine code segment *implements* or *instantiates* the *intended semantics* of the constants. This may be very simple and direct, as in the case of basic arithmetical functions, or a complex segment consisting of many lines of machine code. The underlying principle, though, is that *however* different implementation layers resolve given constants (and variables, for that matter), they must all instantiate the same function and have the same effect. The end result being, of course, that a program written in the higher order language will work in (ideally) exactly the same way for any machine with an implementation layer for the language — it will be machine independent.

In what follows, we will examine machine independence and its relationship to formality more closely, again with a particular focus on the relationship between the syntax and semantics of higher level languages. Because we cannot examine every higher level

---

a small compiler is written (not in the source language) to compile a portion of the compiler (written in the source language) sufficient to compile further portions of the main compiler, and so on until the entire compilation is complete.

language (for rather obvious reasons) we will consider one historically notable language — FORTRAN.

FORTRAN is one of the earliest and best known high level programming languages. The language was originally created with a focus on scientific numerical calculation but was subsequently expanded and used in considerably wider-ranging applications. This original focus, though, should bring to mind the imaginary machine considered above, which is capable of receiving and executing programs written in a purely mathematically oriented language.

The first version of FORTRAN, known as FORTRAN I, was designed for use with the IBM 704 series of computers, and it was also first implemented on these machines. Many versions of FORTRAN have been published and implemented since its initial release by IBM in 1958. We have opted to consider here this first version, and in the following section on "sequence independence," note some of the changes made for the second language standard, FORTRAN II.

FORTRAN I makes a nice example in part because of its simplicity, but also because of its flaws from the perspective of machine independence. Although it certainly involves all of the major innovations of higher level languages, because it was first targeted at only a small range of machine independence — the 704 series — it provides examples of both what features facilitate machine independence and what features restrict it. The sense in which the language is "simple" is that it involves relatively few constants and its syntax is relatively simple when compared to more modern languages. It does not mean that FORTRAN programs are simpler than similar ones written in more modern languages, which is rarely if ever the case. This simplicity is an asset from our perspective because it will simplify our discussion in much the same way axiomatic formulations of first order logic simplify proving meta-theorems (and in both cases, the simplicity on the language side makes working within the language more complicated). Furthermore, our focus on FORTRAN will not require the detailed examination we undertook for EDSAC code, since it centers around how a language achieves machine independence and the gen-

eral effect this has on its syntax. These features are in no way peculiar to FORTRAN itself.[39]

The basic anatomy of a FORTRAN program is similar to an EDSAC code program in that both consist of a single, linear sequence of well-formed statements of the language. It is standard to refer to each of these statements as a "line" of code. The nature of a FORTRAN sequence differs slightly from that of an EDSAC code program owing to the fact that FORTRAN is agnostic with respect to memory addresses.[40] In the case of EDSAC code, each command code had a means of reference automatically assigned to it in virtue of being placed at a certain memory address. Sequences of code could then be referenced by the initial memory address of the sequence or by means of a variable denoting this address (whatever it may be). In FORTRAN I, there is still a need to refer to particular lines of code, but owing to memory address independence, memory addresses will not work for this (because they are unknown). Instead, important lines of code may be assigned numerical "names" called "statement numbers." Besides being within a certain range, there are no restrictions on statement numbers, and in particular these do not necessarily have any relationship to the lines' positions in either source or object code. For example, the fifth line of source code could easily be assigned the statement number 543. Furthermore, not all lines need be assigned a statement number; this is left to the discretion of the programmer.

A complete list of the 32 types of statements which are permissible in FORTRAN I can be found in [Backus et al., 1956]. These 32 types of statements can naturally be divided into four categories. The categories are: arithmetical statements, control statements, input/output statements, and specification statements.

[39]For more detail, or to check our veracity, we direct you to [Sammet, 1969] and [Backus et al., 1956]. For those who may wish to dabble in FORTRAN I programming, [Mitchell, 1957] and [Backus et al., 1956] make a nice starting point.

[40]At one point, the ability to "descend" and include machine code directly in FORTRAN programs was added (FORTRAN III), but the unsurprising result was that object programs were too closely tied to particular machines (a lack of machine independence) and the feature was drop from subsequent versions. See [Sammet, 1969].

All arithmetical statements have the form $a = b$ where a is a variable letter, and b is a string of symbols which evaluates to a numerical value. The syntax of arithmetical statements is intentionally similar to standard mathematical notation, and this is particularly true with respect to statements involving numerical constants, standard arithmetical operators ($+,-,*,/$), and grouping symbols. The only difference in this case is that "=" functions in an assertive rather than an evaluative role. It is called the "assignment" operator (and not the "equality" operator) because rather than returning a truth value based on the equivalence of the operands, it sets the value of the left-hand side (a) equal to the evaluation of the right-hand side (b).

In addition, FORTRAN I includes a small number of numerical function constants which take arithmetical expressions as arguments and, since they return numerical values, themselves constitute numerical expressions. This is accomplished by a now familiar function-argument syntax. If $SOMEF$ is a function taking two arguments, and c and d are arithmetical expressions, then $SOMEF(c,d)$ is an arithmetical expression, and $a = SOMEF(c,d)$ is a valid arithmetical statement. The only other limit on arithmetical expressions comes in a requirement that the resulting statement contain at most 660 characters (for hardware oriented reasons). [41]

One of the things that makes EDSAC code highly machine dependent, and also error prone, is that control statements always take direct references to memory locations as arguments. This is true even in the case of included subroutines, in which control statements must refer to relative memory addresses within the subroutine. This means that insertion or deletion of a command requires extensive revision of control statement arguments. FORTRAN I, on the other hand, makes use of the aforementioned statement numbers for this purpose. Since a statement number is essentially a numerical name which may be arbitrarily assigned, this name can be supplied to a control statement. Every FORTRAN

[41] The class of arithmetical expressions could, in some sense, be expanded by the addition of defined function symbols. In FORTRAN I, however, this was only possible by adding the desired function to the compiler in the form of low-level code. There was no facility for writing these in the FORTRAN I language itself.

I control statement requires at least one such statement number argument.[42]

The syntax of input and output (I/O) commands is best understood by directly consulting [Backus et al., 1956]. The only thing important to note about the syntax of FORTRAN I I/O commands is that they syntactically distinguish between multiple types of input and output facilities and data formats. There is a close relationship between the syntax of these commands and the characteristics of the underlying I/O hardware, and we will return to the significance of this below.

There are three final FORTRAN I statements to discuss, the so-called "specification statements." These statements are most closely analogous to the control combinations of EDSAC code, as they are not executed but instead direct the FORTRAN compiler to process commands in a certain way. For example, a "FORMAT" statement is never executed, but is taken as part of a read or write command to direct the machine on what format to expect from input or impose on output. The syntax of these commands derives directly from their semantics and so will be discussed in that context.

Analysis

FORTRAN I was originally targeted, as we have said, at the IBM 704 series of computers. This has two important consequences. First, the first FORTRAN I compiler was written in the machine code for 704 series computers. In fact it was designed to be executable on a 704 with a relatively small memory core. Even so, the compiler was designed to be capable of producing programs executable on 704 machines with much larger memory stores.[43] Over time FORTRAN came to be implemented on many different machines, and FORTRAN compilers can still be easily found despite a dearth of available IBM 704's. At the very least, though, we can say that FORTRAN I, and any program written in the language, is "IBM 704 independent" — meaning that such a program will

[42]For a more detailed discussion of control statement syntax, see [Backus et al., 1956].

[43]In principle, any 704 machine could execute a large program if the program were segmented into appropriately sized pieces.

execute accurately and reliably on any 704 series machine. This means that the execution of the program will be exactly the same whatever 704 it is executed on, which is the same as saying that any FORTRAN I program is invariant across transformations from any 704 machine to any other (both in terms of types *and* tokens).

We can go one step further, though, and say that the invariance of programs, which are themselves determined by the syntax of their constituent statements, rests upon the fact that the syntax of the FORTRAN language captures features which are invariant across 704 machines. Furthermore, just as we saw in the case of EDSAC code, the invariant features captured in FORTRAN I fall into two categories, those which are oriented toward the operations of the machine (and tend to restrict the machine-independence of programs) and those which are oriented toward the expression of mathematical functions.

The arithmetical statements, central as they are to the expression of mathematical functions, have the strongest claim to being machine independent. An easy way to see this is to think of what is required, minimally, of a machine to effectively implement these statements. Obviously, the machine must have a memory store in which the resulting values may be located. This is captured syntactically by restricting the left hand side of arithmetical statements to variables only, which are tied to such memory locations. The only other requirement is that the underlying machine be capable of sufficient basic operations to implement the operations and functions denoted by arithmetical constants and function names. It happens that the numerical functions denoted by the operators and constants employed in the FORTRAN I arithmetic expressions are all computable.[44] Hence, the constraint imposed on any underlying machine is that it must be Turing complete. Aside from practical considerations of efficiency, it does not matter how these functions and operations are implemented, either in terms of basic materials (electronic circuits versus

[44]In terms of classical computation, of course, a language with constants denoting uncomputable functions would not be implementable on any machine. This constitutes neither machine dependence *nor* machine independence. Such a language would be, we may say "off the map" of our present discussion and is, consequently, *not formal* with respect to computing machines. The difference between "formal," "informal," and "non-formal," will be discussed in greater detail in Chapter 5.

human beings) or basic components of the arithmetic unit (provided they are rich enough to compute all partial recursive functions).

The arithmetical statements of FORTRAN, taken on their own, have a considerable degree of machine independence. A major gain over the EDSAC code, in particular, comes from the noticeable absence of memory addresses and the direct inclusion of numerical constants and variables into the language. The relationship between the syntax of the arithmetical expressions and the invariances involved in computable numerical functions is straightforwardly the invariances captured by basic arithmetic and set theory.[45]

What is also quite clear, especially in comparison with EDSAC code, is that the arithmetical statements are almost exclusively focused on the representation of mathematical functions, not the processes by which these functions are implemented by the underlying hardware. Those are details handled by compilation. The arithmetical statements, together with the "core" group of control statements (GO TO, ASSIGN, IF (*a*), and DO), make the language of FORTRAN I itself Turing complete. And these syntactic features — numerical constants, variables, arithmetical constants, core control statements — capture the fact that the language is designed to express computable functions. This *seems* obvious, until we recognize that FORTRAN I need not be Turing complete to be machine independent. In fact, there are other machine independent programming languages which are not strictly Turing complete. Just as we would expect, they are designed with a different class of invariants in mind (some subset of the partial recursive functions).

In terms of machine independence, we may consider the core control statements in the same way as the arithmetical statements. What is required for a machine to implement these? It turns out only very basic capabilities are required. In particular, the ability to associate a numerical name or variable with a memory address, which is essentially

[45] A minor exception to this claim concerns the fact that variables and numerical values in FORTRAN I are "typed." This means that they are bound to different formats for storing values. The syntactic rules governing the mixing of fixed and floating point constants and variables can be best understood as capturing invariances of the ways numbers are stored in physical digital media. As such, this reins in the machine independence of the arithmetical statements somewhat, as any underlying machine must be capable of handling these types appropriately.

the same capability required to handle numerical constants and variables, the ability to compare values against zero, and the ability to transfer control to another line of code. All of these are basic capabilities of any machine satisfying the von Neumann architecture, and hence they will be invariant across all von Neumann machines (not just the IBM 704s).[46]

The case is quite different for the other control statements, which directly involve specific hardware of the 704 series of computers, e.g., sense lights and sense switches. From a syntactic standpoint, these commands do not differ much from the control statements and have considerable semantic overlap as well — they all involve the evaluation of a condition followed by a transfer of control. To this extent, the inclusion of these constants in the language is consistent with its intended ability to compute partial recursive functions. However, an additional semantic element is that they refer explicitly to hardware features which were only really implemented on the IBM 704 series of computers (and even then, not all of them). Thus, these statements are only invariant across a small subset of machines. Fortunately, the commands are entirely redundant from a computational perspective, and quickly fell out of disuse (e.g., FORTRAN I programmers would use only a fragment of the full FORTRAN I language).

In general, input and output (I/O) tends to be the most machine dependent class of operations. This is understandable as I/O commands are designed to operated directly with physical media that store information in different ways. FORTRAN is certainly no exception. Ideal machine-independent I/O commands might simply be "READ" and "WRITE," allowing the implementation itself to manage the reading and writing. In practice, however, different technologies were developed involving significantly different data formats. The first, and oldest, of these involves reading values from paper tapes or card in which holes were punched. The other was a kind of magnetic drum — a precursor for contemporary magnetic drives. The two kinds of media had to be treated in different ways, and in many cases machines were capable of reading both. The makers of FORTRAN

[46]In fact, the invariance in this case is more thoroughgoing. In principle, all that is required is that the underlying machine be Turing complete, and the arithmetical and core control statements be implemented.

dealt with this problem by including sets of analogous commands for each kind of I/O device, and even for two different tape data formats. The syntax of these commands follows straightforwardly from the characteristics of the associated devices in the same way that the function-argument syntax of machine code follows from the underlying machine facilities. For example, because the IBM 704 had four tape readers, an integer value must be supplied to the read/write commands in order to specify which of the tape readers to use. The need for certain commands, like "REWIND", "BACKSPACE", and "END FILE" follows from the particular characteristics of tape readers, as evidenced by the lack of analogous commands for operating the magnetic drum (whose commands instead require location values to be specified). Obviously, to implement these commands in a way consistent with their syntax, a machine would need four tape readers and a magnetic drum. The inclusion of these commands, again, restricts the machine independence of the language and at the same time the size of the class of machines with respect to which it is invariant.

Given the nature of input and output — that it involves the transference and retrieval of digital data onto and off of physical media — it has always been, and will likely always be, a significant hindrance to machine independence, invariance, and formality. This is not particularly surprising, as we may think of it as something of an impingement from the lowest level (physical operations) into the higher-level world of expressing computable functions.

FORTRAN I, taken as a whole, succeeds in achieving a small degree of machine independence — with respect to IBM 704s. What is clear from the brief look at the fragments of the language presented above is that certain fragments have potential for a greater degree of machine independence, while other fragments serve to restrict this dependence. This brings into view several interesting points.

The first of these is to note that, if we take a particular fragment of FORTRAN I, call it FORTRAN*, we can have a formal language which has a higher degree of machine independence than FORTRAN I. In particular, if we keep only the arithmetical state-

ments, the core control statements, and nothing else (ignoring the need for I/O) we have a language which is implementable on machines with a memory store, sufficiently rich arithmetical capabilities, and appropriate control capabilities. This essentially amounts to the requirement that the machine instantiate a Turing complete von Neumann architecture. These three components are features that remain invariant across transformations between all computers which implement a von Neumann style architecture. In general, machine independence is understood to mean independence with respect to this particular class of machines (as most contemporary computers are von Neumann machines).

This entails that programs written in FORTRAN* are invariant not just with respect to IBM 704 machines, but with respect to all von Neumann machines (assuming FORTRAN* is implemented for those machines). By contracting the complexity of the syntax of the language (primarily by dropping constants), the range of invariance of the resulting language has increased. This is analogous to the structured relationship between the geometrical systems of concern to Klein, where alterations to the number of stipulated invariants caused contraction or dilation of the associated class of transformations.

We may, consequently, expect that machine independent languages will involve the same basic syntactic features of FORTRAN*, because the purpose of these languages are all to direct the functioning of a von Neumann computing architecture in computing partial recursive functions. Results in computational theory suggest that just one such language should be enough, since anything expressible in one Turing complete language is expressible in all of them. However, we know that there are other high level languages. If we examined these, we would soon recognize that, beyond a certain minimal basis (grounded on the desire for Turing completeness) the languages involve numerous other constants and syntactic constructions that, from a purely computational perspective, are unnecessary. Although the addition of these "unnecessary" constants and syntactic "sugar" does not have any effect on the generality of the resulting languages, there is a sense in which the these languages are more *specific*, or are tailored to a particular range of problems. This is not to say that the same tasks cannot be accomplished with different

languages, but that a particular language, consisting as it does of certain constants rather than others, can have practical consequences when using the language to solve certain problems.[47]

It is significant, and crucial, to note that this specificity is completely orthogonal to the generality gained in being machine independent. Any such language still only contains constants denoting operations executable by all von Neumann machines (though surely not every possible constant), and hence are invariant over the same set of transformations (from machine to machine). The languages are equally formal with respect to the machines they direct, but not necessarily so with respect to specific classes of computational tasks and problems they might be used to solve. The remainder of this chapter considers ways in which modern programming languages have been tailored to capture invariances within specific problem domains.

4.2 Beyond Machine Independence

We have been arguing that the underlying semantics constrains the resulting syntax of a programming language. We have argued that the core of this relationship is that the invariances of the language (the syntax) correspond to, or capture, invariances in the associated semantic domain, and that this lends a kind of transparency to the language such that it can "stand in" for the real thing in important contexts.

An ongoing theme has been the transition of language development from a focus on the underlying hardware to an increased focus on the numerical problems computers were originally created to solve. We have also generally characterized the important role the implementation layer plays in making this possible, and we have seen how some languages

[47]It should be clarified that these language additions differ fundamentally from what we might call the "defined constants" to be encountered in the next section (e.g., named subroutines). In the case of an added primitive constant or syntactic structure, however complex the underlying operations which realize it may be, the implementation interprets the symbol or structure in a way which goes beyond syntactic manipulation of the source language. This is not so with defined constants, which stand in for other statements in the source language and are usually replaced by these prior to true compilation.

incorporate syntactic features which capture invariances in different subject domains.

In the following sections, we shift our focus to ways in which structural characteristics of modern programming languages are tailored to non-machine oriented semantic domains. The first of these concerns the structure of programs themselves, and the final two address the characterization of problems (especially those which are non-numerical) and solutions to these problems. We will, however, no longer be focusing on specific languages, but instead on more general developments.

4.2.1 Structured Programming

Recall that one of the important goals behind creating the EDSAC programming system was the construction of memory address independent subroutines. This was accomplished by introducing variables and a means of binding these variables to whatever memory location the subroutine would actually occupy. Although this does provide a degree of invariance to the subroutines themselves, it does not extend very far. In a program which uses a subroutine, "G K" must be prefixed to the subroutine, directing the Initial Orders to calculate the memory references within the subroutine. Doing this still requires making implicit reference to memory addresses. Likewise, control transfer commands all require specific memory addresses as arguments, even if these may be expressed as a value to which an offset is added.

The net result is that all EDSAC programs are expressed as a single linear sequence of commands (even though control transfer commands mean it is not executed as such). In particular, even a library routine must appear within the main program sequence. If a subroutine is to be invoked twice, it must either be included in the sequence two distinct times, or an appropriate control transfer, with a reference to a memory location in the sequence, must be given.

Programs written in subsequent languages were constructed in a similar fashion, despite the fact that their high degree of machine independence provided freedom from

referencing any memory addresses at all. Statements would be arranged in a single, linear sequence (with library functions possibly referenced and inserted at compilation). This sequence would be numbered, and control transfers (generally by means of something similar to FORTRAN's "GO TO" command) would be achieved by references to specific positions in the sequence.

A similar observation is to be made here as was in the case of library subroutines earlier — namely that if a sub-sequence of a program realizes a conditional or loop structure, this structure is conceptually independent from its location within the sequence, and it is dependent only on the commands it contains and the conditions it involves. However, reliance on control transfer by reference to locations in the sequence forces loops and conditional branching to be tied to specific locations in the overall sequence. Practically speaking this means that the insertion of a new command anywhere prior to such a control transfer could easily result in a malfunction unless all affected control transfer statements are updated accordingly (which could even include all subsequent control transfer statements). This makes writing recursive procedures particularly difficult and error prone.

The problem can be characterized intuitively in the following way. What is desired for a control transfer is really something like a demonstrative pronoun. For example, "If condition X, then go to *this* command," where "this" somehow points at a specific command. In the case of a recursive function, "this" refers to the very sequence of which the transfer command is a part. The facilities provided by many languages, however, are in no sense demonstrative but rather proper names of locations in the sequence. What is needed is a "sequence independent" way of indicating commands and sequences directly, rather than indirectly by means of locations within the sequence.

Languages which facilitate "structured programming" do precisely this by adding two new syntactic components. The first of these consists of new constants which serve essentially the role of demonstrative pronouns by bracketing a sub-sequence into a block. For example, an unstructured conditional sequence might look like this:

1. IF $x = 5$, GO TO 4

2. $x = x + 1$
3. GO TO 5
4. $x = x^2$
5. PRINT x

This program prints "25" if x equals 5, otherwise it prints the value of $x + 1$ (and it never prints "6"). It is easy to see that, if one inserts a new command immediately after line two, the result will still be to print "25" if $x = 5$, but otherwise $(x+1)^2$. To verify, compute the following with $x = 5$ and $x = 6$.

1. IF $x = 5$, GO TO 4
2. $x = x + 1$
3. CONTINUE (anything could be here)
4. GO TO 5
5. $x = x^2$
6. PRINT x

The structural counterpart does not suffer from this problem, and can be summarized as follows:

1. IF $x = 5$ THEN
2. $x = x^2$
3. ELSE
4. $x = x + 1$

5. END IF

6. PRINT x

We include line numbers for ease of reference and to make it explicit how the statements in the sequence depend in no way on these for their semantics. Instead, there are new constants demarcating sequences (called "blocks") of code syntactically, playing essentially the role of demonstratives.

If x equals 5, then set $x = x + 1$, otherwise set $x = x^2$. Finally, print x.

It is also clear that inserting or deleting commands will not have the same tragic effects as in the non-structured example. Modern languages usually contain similar delimiters for loops as well (usually of more than one kind).

Statement numbers in FORTRAN I provide a different, more limited solution to this particular problem. These play the role of names (i.e., rigid designators) of statements rather than demonstratives. Hence, they will always refer to the same statement in code. This solves the preceding problem, but at the cost of using many, possibly hundreds or thousands, of additional names. These are not only difficult for programmers to remember, but also increase the risk of critical typographical errors.

What is often lost in a language oriented toward structured programming is the ability to unconditionally transfer control to a particular line of code in the sequence (e.g., FORTRAN's unconditional "GO TO" command). This would seriously hamper programming were in not for the second new syntactic component of structured programming — the ability to assign a name to a particular sequence of commands.[48] These sequences become effectively the same as library subroutines, except that they exist independently of the main sequence of execution. They are not generally included in-line at the point in the main sequence where they are required (as happens in EDSAC code). Instead, they may be appended to the main sequence or even stored in a separate file. When required by the

[48]This is in contrast to FORTRAN I, in which names are assigned to particular statements only

main sequence, only the name is included in the statement (called an "invocation") and the implementation layer manages how this is ultimately represented in machine code.

The name of a subroutine is essentially a defined constant. When the name occurs in the main program, this should be understood as intuitively as saying "now execute *XXX*," where "XXX" is a constant name. In unstructured programming, the primary use of unconditional transfer commands is to re-execute a particular sequence of code. With structured programming, this sequence is invoked by name whenever necessary. Importantly, changes to either sequence of code (the primary or subroutine) will not disrupt the other. It furthermore makes recursion very easy, as a subroutine may simply invoke itself.[49]

The result of this is code which is insensitive, or invariant, to facts about the code itself — namely, relative locations. This is a relatively simple, syntactically-oriented formality, not unlike the use of "(" and ")" to indicate scope in most formal languages. For example, a logical quantifier *could* instead be sub-scripted with the number of symbols within its scope. Instead of "$\forall x((Fx \wedge Gx) \rightarrow \forall x(Gx)) \rightarrow \forall x(Fx)$" we could have "$\forall x_9 Fx \wedge Gx \rightarrow \forall x_2 Gx \rightarrow \forall x_2 Fx$". Such numbers would require constant updating, but suitable rules could be formulated. Likewise, names (often abbreviations) can be assigned to stand in for more complex expressions. Neither of these alters in any way the expressive power of the logical language or requires a robust semantic interpretation, being merely demonstratives and names for already defined sequences of symbols or statements. They do, however, free up the syntax from needing to track irrelevant information about the composition of expressions.

It should be no surprise that this technique is called "structured" programming, as it provides the syntactic facilities to express structure that is present but not captured by unstructured programming languages and techniques. These structures are invariant characteristics of the way certain blocks of code (e.g., conditionals, loops, and subroutines) are used. The "gain" in terms of formality is not with respect to any subject matter other

[49]FORTRAN II included fairly robust facilities of this kind for subroutine construction. [Sammet, 1969]

than the code itself.

4.2.2 Object-Oriented Programming

If we were to present a problem or programming task to a number of programmers and have them come to agreement on what the best way to solve the problem is, and then ask them to go off separately and write programs to solve the problem using the same language (e.g., FORTRAN), the resulting programs would almost certainly differ from one another. Even assuming that each exactly solves the problem (in the sense that neither program does anything in addition to what was originally asked of it), the resulting programs may yet differ in significant ways. There is a clear sense in which each programmer has written a different program. The sequences of operations their programs describe differ. However, it is also reasonable from a certain perspective to think that the programs essentially "do the same thing." They instantiate (though differently) the same, previously agreed upon, solution to the given problem.

An important theme thus far has been the development of programming languages more closely tied to problem domains than the underlying computer hardware. Recall that an explicit design consideration of FORTRAN was that it should be convenient for the representation of numerical calculation, as it was designed for use in solving scientific, numerical problems. However, computers have turned out to be useful for solving many other kinds of problems as well, and though in principle everything a computer does is numerical, there is a clear sense in which many of these problems are not numerical (or at least not exclusively so). This has naturally given rise to further innovation in programming language design, and in this section we will consider one of the most successful and influential developments — "object-oriented" programming. This development is in some ways similar to the development of structured programming in that it involves the way code is structured. It is importantly different in that this structuring is designed, not with respect to the code itself, but with respect to a certain class of problems the associated

techniques and languages address.[50]

Object-oriented programming involves adopting a new perspective on the structure of a program. Instead of consisting of segments of code which are executed in some order or other, programs are seen as consisting of software "objects" which have properties and perform actions. A software object is really nothing more than the sum of its properties and actions. When the implementation layer allocates memory for a software object, it creates the object from a pattern, called a "class," which is what is actually created by the programmer. A class is a grouping of code which includes variables (properties) and subroutines or "methods" (actions) which apply to every object created from that class. The class itself is given a name, and this name is treated as the type of the objects created from that class. It is common practice to give classes descriptive names based on the properties and methods of the class, which often mirror real world objects. How this works can be best understood by means of an example.

Imagine that we have been tasked with creating a system for managing student registration. From an object-oriented perspective it makes sense to have, among other things, a class named "student." Now, each object instantiated from this class will be a *student* type of object. Each *student* will have certain properties: a "name" property; a "student ID" property; perhaps a "GPA" property; and a "class standing" property. Each student object will also have certain actions it can perform. In this case, "add a class," "drop a class," and whatever other actions are required during the process of registration. Many of these methods may be run-of-the-mill structural subroutines, or they may reference properties and methods of other objects.[51]

---

[50] It is worth noting that the predominant attitude in the field is that "object-oriented" programming is a method or style rather than a type of language. This is because, with some ingenuity, "object-orientation" can be achieved in purely structural languages, and structural programming is used within languages designed to be object oriented, and may even be so used to the exclusion of object oriented techniques. We feel that the question of whether a language is to be described as "object-oriented" (or anything else) should not hinge on what *can* be done with the language (in this respect all Turing complete languages are equal), but on what it has been designed to do well, more easily, etc. Which is ultimately based on the particular constants and syntactic rules of the language.

[51] Although chains of method calling may be constructed (e.g., methods calling methods, calling methods,

An object is "created" as an instance of a class by declaring a name for the object (a variable called a "reference") and stating what class it instantiates. This is done in another piece of code, which may well be a method of another object. Once instantiated, an object is bound to a location in memory and may have its properties set to values and have its methods invoked.

Returning to our example, imagine a flesh-and-blood student attempting to use our registration program. The program requests the student ID number, which the student dutifully supplies, and indicates that she would like to proceed. Program execution begins by instantiating a *student* object corresponding to the actual student in terms of certain relevant properties (these values having been retrieved from a database and assigned to the appropriate variables). This *student* object will perform appropriate actions as its methods are invoked by the main program. If we have made a sensible program, this will usually be in response to some action on the part of our flesh-and-blood student, and will hopefully result in successful registration.

Of course, in practice things are generally more complex than this, and even a simple program may involve dozens of classes and other related components. There are many practical virtues of object oriented programming, not the least of which is that it is less prone to error and enables much more effective reuse of existing code. The student class from the program just considered could also easily be included in a grade management application. From our perspective, object oriented programming has one theoretical advantage which the previous example makes apparent. The structuring of the code, which has been freed from dependence on the structure of the machine, has taken on the structure, the form, of the very problem it was designed to solve.

The program becomes a model of the real-world activities that are to be carried out. The advantage in this is that, once the problem has been modeled, it is fairly straightforward to construct a program to solve it. It is certainly more straightforward than un-

---

and so on), any such chain must terminate at some method which is a procedural (non-object oriented) subroutine (or several of them). Otherwise either nothing will happen, the computer will run out of memory, or processing will be stuck in a non-terminating loop.

structured programming, especially if the problem is very complex.

This relationship between the structure of the program and the real-world problem it addresses is made possible by the addition of constants and syntactic constructions which capture invariances in a rather large sub-class of problems addressed by computer programs. Namely, those problems which are naturally modeled in terms of classed objects which have properties and can carry out actions. This quality, of being naturally modeled in terms of interacting objects, can be thought of as a stipulated invariant which defines an associated class of transformations between all problems of this type. Then the additional emergent invariances are that the objects in the models have properties and are capable of carrying out certain actions. And these are precisely what is captured by the syntactic constructions and constants which are essential to object oriented programming. So, again, the constants and syntax of the language, which constitute its form, are designed around mirroring invariances in the target domain.

More specifically, what we see in a given program also involves a kind of formalization, as does the underlying model. The model, if it is a good one, will capture the invariances of the particular problem (or set of problems). The model captures invariances of the particular types of objects involved. All students are treated, programmatically, as being the same (same features, same actions, etc.). These models can be accurate or not, and if not they will result in a poor program. For example, if the student class is defined such that all students have a "social security number" property, and this property is required for certain actions, there will be problems for non-resident aliens. The model asserts that the possession of a social security number is invariant across all students. Non-resident alien students, who by external criteria are in fact students, fail to have this as a property. The model fails by assuming an invariant property of a class which turns out not to be invariant.[52]

[52]What has really occurred can be understood in more detail as follows. We began by identifying the intuitive characteristic invariants of the real-world class of students. Presumably, this involved things like having matriculated and taking classes, etc. We then made an *assumption* regarding a certain property, the possession of a social security number, as being also invariant across this same class. This assumption may turn out to be false on *empirical* grounds, if there turns out to be such a foreign student. We might also

It is clearer in the case of object-oriented programming than any yet considered that these syntactic features render the languages transparent with respect to the domain of invariance. In the example above, we created a class called "student" which corresponds to the real-world class of students. It is then quite easy to conflate the action of student object with those of a real student. This is a primary virtue of object oriented programming techniques.

4.2.3 Descriptive Programming

The languages so far discussed are generally classed as "prescriptive," "directive," or "imperative" languages, as are the styles of programming associated with them. In general what this means is that the languages are designed to direct and control the operations of the machine, albeit in a relatively formal way and with an implementation layer handling most of the details. This is true even for object oriented programming, since method details are ultimately imperative in nature.

We began the preceding section with a thought experiment involving a number of programmers writing programs to solve the same problem. We said that there would undoubtedly be differences between the resulting programs. This results from the nature of prescriptive programming languages, which require the programmer to explicitly represent *how* a computation proceeds. Since nearly every solution can be constructed in code in many different ways, the same solution can be realized by more than one program.

A comparatively new trend in programming language design called "descriptive" or "functional" programming focuses more directly on the solution realized than the sequence of operations which realizes the solution. In short, it focuses on *what* gets done, not *how* it gets done.

From our perspective, descriptive programming aims at what we might call *im-*

think that it is false on *stipulative* grounds, since there is no principled reason there could not be such a student, regardless of existential facts. At any rate, the possibility of error in such a case depends on having an antecedently, clearly demarcated domain.

plementation independence. It takes the view that all computational processes ultimately realize computable functions. In particular, each computable function can be realized by many different computational processes. In being independent of these particular processes, a program would need to be expressed in a language free from reference to (and hence invariant with respect to) particular machine states, while at the same time being rich enough to express the computable functions. As a result, it would only need to represent these functions directly *as functions*, not as sequences of steps.

From a theoretical perspective, functional languages predate electronic computers in the form of λ-calculi and combinatorial logics.[53] Obviously, these languages have no need of references to machine states, nor are such references even possible in the pure forms of the language. They are, nevertheless, more than adequate for expressing all computable (i.e. partial recursive) functions.[54] In practice, these languages, as they stand, are not adequate programming languages. However, most descriptive programming languages are based on the principles of these formal systems, and so it will be worthwhile to consider one of them briefly.[55]

We will consider for our example the pure, untyped combinatorial logic (CL), as this is one of the simplest functional systems. CL syntax consists of one rule, usually called "application," symbolized by the juxtaposition of expressions. All well-formed expressions denote functions. The expression "*AB*", for example, reads "function *A* applied to function *B*." In addition, the language contains "(" and ")" for controlling the order of application, and a convention of left association allows their omission in many cases. Pure CL contains only variables and the constant symbols "*I*", "*K*", and "*S*". "*I*" denotes the identity function, such that $IX = X$. K denotes a function whose application results in a constant function, say KX, such that $KXY = X$. S is rather more complex, denoting a kind of composition function — $SXYZ = XZ(YZ)$. All variables are free (and, in fact,

[53] [Hindley and Seldin, 2008]

[54] [Barendregt, 1981], [Hindley and Seldin, 2008]

[55] A fact that will prove important in characterizing the difference between logic and programming languages, discussed below.

implicitly universally bound).

The motivation behind CL is the study of ways in which functions combine with one another. The domain of this language presumably consists of some class of functions (possibly all of them). It is obvious that application will be invariant across transformations from function to function, as it is part of the definition of a function that it applies to arguments. This is furthermore so because polyadic functions may invariably be expressed as compositions of monadic functions (via a process called "Currying"). Given that the language has the purpose of exploring the composition of functions — their applications to one another — this ends up being the only syntactic operator. As such it requires no symbol and is represented by mere concatenation.

There is an air of tautology in saying that I, K, and S are invariant across the domain of functions. In some sense, this is part of their definition. As it turns out, however, one of the important and interesting questions for CL (and λ-calculi as well) is what functions turn out to be representable in the language. From our perspective, this amounts to clarifying the domain over which the notions denoted by the constants are invariant. In particular, if a function cannot be defined in the language, this will be because the language does not contain constants which make a distinction between that function and others possible (it lacks the expressive power). In this sense, it treats these "undefinable" functions as of a piece with many other functions, because it is unable to distinguish them. This is to be expected, since we know that a smaller number of invariants will demarcate a broader class of transformations.

A language becomes richer as constants are added (or the syntax becomes more complex). In terms of definability, a richer language is able to distinguish more regarding its domain — it is able to *assert* more about the particular character of classes within its domain. There are, consequently, fewer transformations with respect to which the constants of the richer language, taken as a whole, are invariant, and so we can say that the result is less formal.

There are different variations of CL based on various reduction and equality rela-

tions, and typing systems. One of the interesting questions is which functions are definable (representable) in which of these languages. And the languages we are concerned with are those which hold promise for expressing the functions instantiated by computers. The largest such class is, not surprisingly, the computable functions. As it happens, there is a version of CL able to represent all partial recursive functions, which on the standard understanding of computability are just the computable functions.

Descriptive programming takes CL (or the λ-calculus) as a starting point. Just as with any higher-order language, one cannot simply input an expression of CL and have it count as a program. An implementation layer is required in order for the input expressions to be meaningful. In particular, the implementation layer must know how to treat variables and input values, and it is here that the differences between descriptive programming languages and CL (and also between programming languages and first order logic) become clear. In CL, variables stand for atomics — either functions or particulars which are arguments to functions. Because pure (untyped) CL is a study in function-composition, all variables are interpreted as denoting functions. In computational contexts, however, we are not generally inclined to think of input data as functions. We could try to rectify this difference by treating data as constant functions (mapping to their conventional values), but we would then still face the discrepancy that the purview of typed CL (which is most similar to a programming language) far exceeds that of any programming language. In particular, just what a function may be applied to (what its arguments may be) is not specified in CL. Types are formally *distinguished* but nowhere *specified*. Hence, expressions of the language may range over an extremely wide domain. Such functions might take actual elephants as arguments and map these to continents on the Earth.

On a computer, however, everything is ultimately executed in machine code, with no variables. Everything is ultimately given a concrete "existence" in the machine. The best that can be managed would be a function which takes a *representations* of elephants and maps these to *representations* of continents. These representations, though, must be constructed out of data types that the implementation layer is capable of handling. Fur-

thermore, were it even possible to feed an actual elephant into a computer as a datum, the implementation layer would need instructions on how to treat elephants. The same is, of course, true of more mundane things like integers, floating point numbers, strings, and the like. Setting aside the difficult notion of feeding objects as data to computational processes, it seems all but impossible that an implementation layer could possibly be created to handle the full range of types that variables in CL (and first order logic) range over.

Descriptive languages, like all programming languages, must distinguish between particular types of data so that they may be treated appropriately. Consequently, descriptive programming languages have constraints on the types of inputs which are permissible. They are designed to express functions taking input values from only a finite number of types. This is yet one more restriction on the semantics of the variables, and this limits the range of realizable functions to that subset of the partial recursive functions concerned exclusively with the appropriate types. In some cases, descriptive languages are designed with one particular type in mind (e.g., regular expressions).

In this respect the resulting languages are less formal than CL — namely that they have a target domain which is a subclass of the domain of CL. We can also see how this difference is forced by the *practical* use of the language. To be clear, while descriptive languages are free, or invariant, with respect to machines, and also with respect to the computational processes which realize the functions the language is designed to express, the language is quite far from being *completely* machine independent. At the end of the day, it must be executed on *some* machine in a hardware-bound machine language. Try as we may to "free" our languages from this fact, it is far from obvious how one could make, say, an expression of CL intelligible to a machine without a compiler to tell it what to do. This is particularly so as the range of CL (and that of first order logic), as standardly interpreted, presses the bounds of *human* intelligibility (some even suggest the extensions of first order predicates should include objects in other possible worlds).

That said, it should be almost as clear as in the previous section how the semantics of descriptive programming languages constrains their syntaxes to structures quite similar

to CL (or the λ-calculus). These syntactic invariants capture the invariant characteristics of the semantic domain of functions. This, furthermore, enables us to quite easily manipulate expressions of the language *as though* they are the functions themselves. Hence, the relation between the syntax and semantics of these languages renders them transparent, in accordance with our thesis.

At this point we can also say more to clearly distinguish programming languages from first order logic. The best argument for their supposed unity is grounded on what is called the "Curry-Howard Correspondence." This metalinguistic theorem holds that for every deduction in a first order language there is an analogous computation and *vice versa*. Hence a certain version of CL can be seen as coextensive with first order logic. And since the semantics of first order logic are exhausted by extension, the two collapse into different notational systems for the same subject.[56]

The consequences of the Curry-Howard Correspondence for formality really deserves its own study, and so we will not take it up further here. What our considerations of descriptive programming should make clear is that, at the very least, programming languages are significantly *less formal* than first order logic. The analogue of $\forall x$ in a computer language at best ranges over all permissible values of each permissible type of data, which are all finitely expressible. This is far from the characteristic universality of logic. Furthermore, there are programming languages so much further down this formal chain of being as to hardly resemble logic at all (consider the EDSAC code), yet these still exhibit the important formal characteristics that are our central focus.

The similarity between the two classes of systems should not be surprising given that the underlying *purpose* of programming languages is to direct numerical computations, and one of the earliest purposes of first order logic was the study of numerical objects.

[56]In fact, Quine's version of first order logic without variables presented in [Quine, 1976a] essentially amounts to a system of Combinatorial Logic. Tarski presents a similar system in [Tarski and Givant, 1987].

Chapter 5

Theoretical and Philosophical Considerations

We have so far focused primarily on applications of the GTOF. This was to establish initial plausibility for the theory and also to provide a clearer understanding of what it amounts to through some "hands-on" applications. It also provides some idea of how to go about applying the theory to various other "formal" systems and theories. We are hopeful we have adequately satisfied these goals, and we will now turn to focus more directly on the theory itself. In what follows we will consider some of the interesting and potentially problematic consequences of this theory of formality, as well as its relationship to other philosophical positions.

5.1 Theoretical Considerations

Formality, we have argued, is a quality of those invariants which emerge from the specification of a domain and a class of transformations on this domain (together with metaphysical facts about that domain). This definition leads to some interesting results which generally fall into two categories. The first are consequences which pertain to par-

ticular instances of invariance. Since they occur only with respect to a single domain, class of transformations, and class of invariants, we can call these "non-relational" consequences. The second category concerns "relational" consequences — relationships between different instances of invariance.

There are two non-relational consequences of the theory of formality that we will consider. The first of these we call the "ubiquity of form." Since the general theory of formality allows us to select *any* domain and class of transformations on that domain, it has the consequence that many things may be formal that are not normally so called. In fact, it might be the case that everything is formal in some respect — that is that form is ubiquitous. The second non-relational consequence concerns the possibility of what in mathematical language would be called "degenerate" cases. These are essentially extrema of the theory, and they have some interesting features we should consider.

Perhaps the most interesting and significant consequences are those which result from the fact that the general theory of formality permits distinct domains and classes of transformations — what we call "respects" or "types" of formality. Although in many cases, two respects of formality are entirely distinct — their domains and classes of transformation are entirely disjoint — some interesting things may be said when this is not the case. In particular, we are able to characterize more clearly what it means to say that formality comes in degrees.

Further relationships we will consider are the fact that the same "thing" (broadly construed) may be formal in a number of different respects, and what it means for there to exist a stipulated relationship between one type of formality and another (as we have suggested occurs in the case of formal languages).

5.1.1 Non-Relational Consequences

The Ubiquity of Form

We begin with the idea that everything might be formal. A strong version of this claim amounts to a kind of metaphysical monism about form: everything is formal, and nothing is non-formal. This runs counter to longstanding traditions in philosophy which endorse a metaphysical dualism, called by MacFarlane "logical hylomorphism," between those things which are formal and those which are not (though, of course, there are conflicting views about the nature of these categories and what properly belongs to each).

It is clear that we cannot endorse metaphysical dualism, since doing so would require us to rule some domains and classes of transformations impermissible or somehow outside the scope of formality. We have no principled reason for doing so, and it is not at all apparent how we should go about finding one. On the other hand, it also seems implausible to say that *everything* is formal, at the very least because formality is intuitively a contrastive notion, opposed to something, and most useful "formal" theories are also fundamentally contrastive.

Although it may not be immediately apparent, the general theory of formality is well equipped to resolve this tension in that it permits us to walk a moderate path between metaphysical dualism and a monism in which everything whatever is formal at all times. It does so by being essentially *relativistic*, in the sense that a given formality always obtains in relation to a given invariance structure. It permits us to say that, from the perspective of a particular context, there is always a clear distinction between the formal and non-formal. However, because this distinction is made relative to a given context, it is not a distinction which holds across all contexts, and it therefore does not distinguish between the "formal" and "non-formal" as distinct types of objects or properties.

To see this more clearly it will be useful to define what constitutes a context, or what we will call a formal "frame." Apart from domain selection, invariances are determined by the selection of a class of transformations. There are two methods by which

this may be accomplished. The first might best be called "collection," and it proceeds by directly specifying, or collecting, the desired transformations. This is the approach employed in the characterization of logical formality discussed in Chapter 3, and it is also implicit in the various types of formality discussed in Chapter 4.

The second method we call "stipulation," and this consists in first identifying specific features to be held fixed or invariant. The class of transformations then consists of all those transformations which hold these features invariant in the given domain. From these transformations emerge invariants, among which are the originally stipulated invariants. This method is employed regularly in mathematics and geometry, whenever a definition is presented and subsequent properties explored or proven. Consider the example of "circle." Usually, this is defined as a figure whose points are all equidistant from a given point (the center). From our perspective, this definition stipulates that the property of having all points equidistant from a certain point in the same plain must remain fixed. Under the resulting class of transformations, certain properties are variant — for example, the radius — but others are invariant — the ratio between the circumference and diameter (π). Having a particular radius, then, is not a formal feature of circles, but π is, in addition to the stipulated equidistance and many others.[1]

In either case, the resulting class of invariants is entirely determined by a domain, D, the underlying facts concerning the features of individuals, and a class of transformations, T. It is this structure which we call a formal "frame" and denote by the ordered

[1]A certain analogy can be made here to methods of set and class construction. According to most set theories, sets must be built from either primitive ur-elements or existing sets (e.g., $\emptyset$). This is done to avoid certain well-known paradoxical results. Although every set is also a class, and so some classes may be constructed in this manner, classes may also be constructed by specifying formulae (or more loosely, criteria) for membership. The resulting classes may or may not actually be sets (i.e., constructable in a set-theoretic manner), and if treated as sets they may give rise to paradox.

In the present context, the method we have called "collection" is analogous to set construction, since it constructs a class of transformations from existing particular transformations or classes of transformations. "Stipulation," then, is analogous to class construction, since it effectively puts forward criteria for transformations to belong to the class. A notable difference, of course, is that the results of collection are not guaranteed to be sets. Although collection could easily be restricted to guarantee transformation sets, it is unclear what advantage this would have.

pair (T, D). Taking standard first order logic as an example, D will be the class of model-theoretic structures, for example $< A, a >$, $< A\prime, a\prime >$, etc., and T will consist of the class of functions between these structures which is thought to be characteristic of first order logic (isomorphisms, homomorphisms, etc.). In the case of machine independence discussed in Chapter 4, D is the class of machines with respect to which the program or programming language is independent (e.g., stored-program computers) and T is the class of transformations which takes each such machine to every other.

When we said earlier that formality was relative, what we meant was that in every case formality occurs only *with respect to* an associated frame. One important result of this is that, with respect to a frame a feature is either invariant or variant, but never both. Hence, with respect to a given frame there is always a duality of formal (the invariant) and non-formal (variant) features.[2] On the other hand, this duality is not a robust metaphysical dualism in that it is likely every feature will be formal with respect to some frames (possibly very many) and non-formal with respect to others (possibly even more).[3]

There remains the theoretical possibility of an all inclusive frame, consisting of a domain to which every other domain is a sub-domain and a class of transformations encompassing all transformations, *simpliciter*. The resulting invariants would then have the unusual property of being formal with respect to all frames. Whether or not this is

[2]We use the term "non-formal" as a generic for what is variably called "content," "matter," etc. in particular contexts. The basic idea of course is that the non-formal is that which changes, or is variant. We reserve the term "informal" for situations where a formalized system or process (e.g., logic) is generally employed, but for some reason is not. For example, for a logical or mathematical theory there is generally an accepted method for defining new concepts within the formal apparatus, but it is often convenient to also employ an intuitive or plain language definition, often for pedagogical reasons. We call such a definition "informal," regardless of whether or not it is invariant under the same class of transformations as the associated formal theory (which it often will be).

[3]There is, however, a sense in which the distinction is determined, and necessarily so. Recall from Chapter 2 that while we are free to consider any domain we wish, once we have selected the domain the features within this domain are *factually* determined — nothing short of altering the domain can change which particulars have which features. Furthermore, even though we may construct a class of transformations at will, once we have done so, the resulting invariants are also *factually* determined, because they share in the factual necessity of the features of particulars. This effectively means that the distinction between invariant and variant features, and hence formal and non-formal features, has a note of necessity about it, but only *after* the domain and class of transformations have been selected.

actually the case is a question we have no hope of addressing. The possibility does not, however, establish any respect in which formality might yet be absolute, for the resulting invariants would still be formal with respect to that particular "master" frame. This is because the GTOF holds that formality *only* and *always* obtains with respect to some frame or other. It is built into the very foundations of the view laid out in Chapter 2. Our theory, therefore, does not proclaim that everything *is* formal, but rather that for any given feature there is some frame with respect to which that feature will be formal.

Within this framework there undoubtedly remain some interesting, perhaps puzzling, possibilities. What we hope is that, for those yearning for a principled line between the formal and non-formal, this result is satisfying, or at least mitigates any disappointment or unease. It may, however, appear that this relativistic nature robs some of the mystery and splendor from a favorite formalism. Is logic (or another favorite theory) really formal in the same sense as, say, the moon, just with respect to a different frame? Our answer here is a qualified "yes." There are, in addition, some important relations between frames and respects of formality from which more meaningful comparisons may be derived, and which promise a more intuitively satisfying treatment of the situation. We take up the particular question of the special status of logic in section 5.2.1 below.

Degeneracy and the Limits of Formality

Degeneracy is an idea that comes from mathematics and geometry. Informally, a degenerate case is one in which a parameter of a mathematical or geometrical structure of a certain type has a value of zero, such that the resulting structure, though nominally still of the original type, is for all intents and purposes of a lower and simpler type. Classic geometrical examples include the degenerate circle with radius equal to zero and the degenerate ellipse with collocated foci. The first of these is equivalent to a point, the second a circle.

The important thing about degenerates is that they seldom have significant interest in terms of the more complex structures for which they are degenerate, since they are

effectively a simpler structure.[4] For example, when considered in isolation, it is intuitive to deny that the degenerate circle is in fact a circle, as it seems far more appropriate to treat it simply as a point (which of course, it is).

Degeneracy is of significance to the GTOF in helping to mitigate an intuitively awkward consequence of the theory — that we could construct a class of transformations containing only the identity function for the associated domain. In such a case every feature of every particular in the domain will remain invariant, and our theory therefore proclaims them all to be formal. In the case of a model-theoretic structure, not only are the logical notions invariant, but also the members of the model-theoretic domain, the extensions of predicates, denotation function, and every other feature of the structure. For structures which make use only of model-theoretic domains of arbitrary "objects" or mathematical objects (what we might call "pure" structures), the invariants are all of the extensional properties (since such structure can be extensionally defined).

The disturbing aspect of this case is that everything in the defined frame, according to the GTOF, is itself formal — nothing is variant. It would therefore appear as if there is nothing against which the formal is opposed, which runs counter to the historical and intuitively grounded idea that form is fundamentally contrastive — over and against "matter" or "content". It furthermore challenges our preceding attempt to preserve this very distinction.

To an extent, we are prepared to suffer this assault upon intuition (and history), for we cannot deny that this is a legitimate instance of formality under the GTOF. However, we would again mitigate the discomfort this may cause by suggesting that the situation be viewed as a *degenerate* case of formality, not a deep and meaningful theoretical consequence of it. It is degenerate in the sense that it contains no variance whatsoever. That is, if we consider the class of variants (the complement to the class of invariants) to be a parameter in the geometrical or mathematical sense, this degenerate case results from the

[4] In saying this, we should note that degenerates *are* often significant and important from other perspectives and for other reasons.

cardinality of this class being effectively zero as a result of the limitation on the class of transformations. As such, though this result concerning formality is technically accurate, it is fairly clear that a similar remark is warranted here as in the case of the degenerate circle. "Certainly," we want to say, "technically, this is a case of formality, but it isn't *really* what we mean by 'formal' anymore than a point is what we mean by 'circle.'"

The possibility of degenerate cases gives rise to a number of additional interesting questions. First, is it possible that a class of transformations might itself be empty, and if so, what would be the consequence for the associated class of invariants (if any)? Conversely, is there a limit to what functions may belong to a class of transformations, and if so, what are the consequences? Is it possible for the class of invariants to contract to the empty set? If so, how does this affect the corresponding class of transformations?

In addressing these questions, it will be helpful to have a more rigorous understanding of "transformation-contraction" and "transformation-dilation" than what we have already discussed in Chapter 2. To this end it will be useful to develop some additional technical apparatus. First, let MAX_T be a function defined on the classes of transformations such that for any transformation class T and its associated class of invariants I, $MAX_T(T)$ is the maximal class of transformations which leave I unchanged. We call $MAX_T(T)$ the *maximization* of transformation class T. An analogous function MAX_I can be defined on the classes of invariants such that for any class of invariants I and a class of transformations T which hold I fixed, $MAX_I(I)$ is the maximal class of invariants which are held fixed under T. We call $MAX_I(I)$ the *maximization* of invariant class I. Both MAX_T and MAX_I rely on the fact that each T and I define a class of the other type, say I_T and T_I respectively, but are also not necessarily the maximal classes which define these counterparts.[5] For ease of reference, and on account of their similarity, we will simply write $MAX(T)$ or $MAX(I)$ and let context determine the function we have in mind.[6]

[5]But note that $MAX_T(T)$ and $MAX_I(I)$ do.

[6]Denis Bonnay defines two related functions, *SIM* and *INV*. *INV* is a function which maps similarity relations (a more limited class of transformation classes) to the class of invariant operators each defines. *SIM*, conversely, maps a class of invariant operators to the similarity relation under which they

Furthermore, we call a transformation class *maximal*, T_{MAX} if and only if $T_{MAX} = MAX(T_{MAX})$. If we take a relation R such that for transformation classes T_1 and T_2, $R(T_1, T_2)$ if and only if $MAX(T_1) = MAX(T_2)$, we can easily see that R is an equivalence relation. Furthermore, it is easily proven that each T_{MAX} defines a unique equivalence class of transformation classes. Let there be two maximal transformation class, T_{MAX1} and T_{MAX2} such that $T_{MAX1} \neq T_{MAX2}$ and assume that $R(T_{MAX1}, T_{MAX2})$. By the definition of R, $MAX(T_{MAX1}) = MAX(T_{MAX2})$, but since $MAX(T_{MAX1}) = T_{MAX1}$ and $MAX(T_{MAX2}) = T_{MAX2}$, $T_{MAX1} = T_{MAX2}$. R therefore cannot hold between non-identical maximal transformation classes, and consequently each maximal transformation class defines a unique equivalence class under R. Analogous results can be obtained for classes of invariants, with each maximal invariant class I_{MAX}, where $MAX(I_{MAX}) = I_{MAX}$, defining a unique equivalence class under a relation analogous to R.

In Chapter 2 we defined a transformation-contraction as the elimination of one or more transformations from an existing transformation class to yield a new "smaller" transformation class. Likewise a transformation-dilation involves the addition of one or more new transformations to an existing transformation class to yield a new "larger" transformation class. What we said there was that a contraction of a class of transformations results in a dilation of the corresponding class of invariants, and conversely that a transformation-dilation results in a contraction of the class of invariants. Although this is generally the case, it is not true from a rigorous perspective as it permits of a number of exceptions.

The possibility of exceptions to this rule is implicit in definition of *MAX* and the equivalence classes created by R. This is because, by definition, a transformation class T is associated with the same class of invariants as $MAX(T)$. Each equivalence class is effectively associated with exactly one class of invariants, since every one of its members is associated with the same class of invariants. Consequently, any contraction or dilation which remains with a given equivalence class will *not* induce a change of the associated

---

are invariant. Restricted to the more limited scope Bonnay considers, for a given similarity relation S, $MAX(S) = SIM(INV(S))$, and likewise for a class of invariant operators K, $MAX(K) = INV(SIM(K))$. See [Bonnay, 2008], p. 12ff.

class of invariants.

Of course, the definition of *MAX*, though it entails the possibility of such exceptions, does not prove their actuality. It might turn out to be that each equivalence class contains only a single member. In fact, this is not so. The most easily recognized exception concerns identity functions. An identity function, by definition, leaves the domain for which it was defined *untransformed*. That is, application of the identity function results in no added variance. As a result of this, the identity function is included in every maximal transformation class, and dilation to include or contraction to exclude the identity function will have no effect on the corresponding class of invariants.[7]

A second, less interesting case, results from the expedient of defining transformation classes as classes of functions rather than binary relations. Dilating the class of transformations to include functions whose mappings include only mappings covered by functions which already belong to the class of transformations will, of course, have no effect on the class of invariants.

A third class of functions which may belong to a transformation class but not its maximization results from not requiring transformation classes to form groups. In particular, we allowed for transformation classes which are not closed under composition. This means that though a may be transformed to b by some function in T, and b to c by some other, this does not mean that T contains a function which transforms a to c. However, the definition of invariance is implicitly transitive. That is, if a feature f is invariant over the transformation from a to b, and also from b to c, then f will *de facto* remain invariant under a transformation from a to c, even though this mapping is not established by any function within the class of transformations T. If, therefore, the class of transformations is dilated to include functions all of whose mappings belong to the transitive closure of the mappings of the original class of transformations, effectively no new variance is added and no change results with respect to the associated class of invariants.

[7]This has an interesting consequence for the special case where a class of transformations consists only of the identity function. This is addressed below.

There may yet be more complicated exceptions in which the function in question does not fit into any of the preceding categories and yet, owing to the peculiar features of particulars, the specific mappings do not result in any effect on the invariants. This could occur in a case where existing functions do not, for whatever reason, map *a* to *b*, but as a matter of fact *a* exhibits every invariant feature and so does *b*. Any function which only adds such mappings would also have no effect on the associated class of invariants (and hence will belong to the maximal class of transformations).

So a more rigorous formulation of transformation-contraction and transformation-dilation is the following: Whenever a class of transformations T_1 is contracted or dilated to produce a class of transformations T_2 belonging to a distinct equivalence class ($(T_1, T_2) \notin R$) the associated class of invariants will expand or contract, respectively. Otherwise, there will be no effect on the class of invariants. Since the only interesting cases of contraction or dilation are those with consequences for the class of invariants, when we use these terms we always mean operations which are substantial in this way.

We may now turn first to the question of whether it is possible that the class of transformations might itself be empty, and what the consequences for the class of invariants are. If taken up directly, this is difficult to address. How can we make any distinction between variants and invariants when there are no transformations under consideration? This is where thinking in terms of transformation-contractions is useful. If we begin with an arbitrary maximal class of transformations T, what we know is that if we contract it the associated class of invariants I will dilate. Further contractions of T will result in further dilations of I. We already know that if we contract T down to the identity function (which, since T is maximal, it must contain), the result is that the class of invariants becomes co-extensive with the class of features of particulars in the domain. Of course, we need not contract to this particular function, but what the case suggests is that, as the cardinality of the class of transformations *approaches* zero, the class of invariants *approaches* coextensionality with the class of features of particulars of the domain. In the special case of contraction to the identity function, coextensionality is achieved. Furthermore, as sug-

gested above, the special character of the identity function is precisely that it leaves the domain *unchanged* — that is, it is a "transformation" which transforms nothing.[8] This intuitive result, if it holds up to scrutiny, strongly suggests an effective equivalence between a transformation class containing only the identity function and one which is empty — at least from the perspective of invariance and formality. This makes it plausible to think that a transformation class can in fact be empty, or even that the identity "transformation" is really just a convenient stand-in (or name for) the empty class of transformations. At any rate, the consequence for the class of invariants when the class of transformations is empty, then, is that it is coextensive with the class of features of particulars of the domain, which is to say that everything comes out to be invariant.

The second question effectively concerns whether there is a limit to transformation-dilations, and what consequences this may have for the associated invariants. The best approach to answering this question is to think of transformation classes as capturing, containing, or imposing a certain degree of *variance* with respect to the domain. This idea of variance is itself best understood in terms of the unique pairings of particulars that are characterized by the class of transformations. Every class of transformations imposes the same degree of variance as its associate maximal transformation class, since they all effectively establish the same pairings of particulars (including implicit pairings resulting from transitivity). Thus, transformation-contractions reduce variance, and transformation-dilations increase it. Taken this way, the question becomes whether there is a limit to the degree of variance which a class of transformations may impose. The answer is easily seen to be affirmative, for if a transformation class is such that it establishes pairings between every two particulars in the domain, no dilation can result in any additional pairings being created, and hence no class of transformations can impose more variance on the domain. Of course, any dilation of such a class of transformations cannot affect the associated class of invariants, since its maximization has no proper superclass. This is because the equiva-

[8] It should not escape notice that this makes the identity function something of a degenerate case itself, at least from our perspective.

lence class to which a class of invariants belongs is effectively determined by the pairings between particulars it establishes, and no additional pairings can be created. Hence, we call the maximal class of transformations which establishes pairings between every particular in the domain its *absolutely maximal* transformation class. Not only is it maximal in the sense defined above, it also imposes maximal variance on the domain, and as such it contains *every* function definable on the domain in question.

What, then, of the associated class of invariants? Clearly, it will be contracted to some kind of absolute minimum for the given domain. Intuitively, since every particular is being transformed to every other, the class of invariants will contain only those features which are common to *all* particulars in the domain. In general, this means that the class of invariants will tend to include (possibly among others) those features which formed the criteria for the domain to begin with. In the case of logic, for example, the absolutely maximal class of transformations will map *every* structure to every other. Clearly, very little will remain invariant. Essentially what remains are the semantic categories and types, including such general notions as "is an individual," "is a property of individuals," "is a 2-place relation of individuals," etc. Which is to say, the most significant invariants are those which characterize *what it is to be a model-theoretic structure*. These are the generic notions out of which models may be constructed.[9]

Though not strictly speaking a degenerate case, the preceding consideration gives rise to the possibility of another. Is it ever possible for the class of invariants to contract to the empty set? Since there is a limit to the amount of variance that may be imposed on a domain, whether this is possible will depend almost entirely on whether or not a domain may be constructed which is sufficiently heterogeneous such that the absolutely maximal class of transformations results in no invariants whatsoever. Given the extremely broad manner in which we have taken "features" it is exceedingly difficult to imagine this being possible. Even barring obviously trivial features, like that of belonging to the domain in question, it seems very likely that there will always be *some* feature, feature of a feature,

[9]This point is made by Sher in [Sher, 2008], following up on some remarks from [Bonnay, 2008].

etc. that will be invariant. Neither is it helpful to approach the problem from the angle of specifying the class of transformations as those which hold no features fixed. Since in many — possibly all — cases no class of transformations (not even the empty class) satisfies this criterion, it seems to be nonsensical. Yet again, this points to the composition of the underlying domain as the deciding factor in whether an empty class of invariants is possible, and we would welcome any demonstration of such a case.

So, the only clear degenerate case is that in which the class of transformations contains only the identity relation, which we take as equivalent to an empty class of transformations. And although the GTOF rules every feature of every particular in such a domain to be "formal," this is only so with respect to a frame in which effectively *no variance* is involved, and as a degenerate case should not be taken as significant to a characterization of formality generally any more than a point is significant to the characterization of circles.

5.1.2 Relational Consequences

Degrees of Formality

In Chapters 3 and 4 we suggested several times that formality can come in "degrees," such that one thing may be "more formal" than another. The intuitive idea is that one invariant is "more formal" than another if it is invariant over a larger range of transformations.[10]

First, assume two classes of transformations defined on some domain, D, denoted T_1 and T_2. According to the theory of formality, each of these gives rise to a corresponding class of invariants. Call these I_1 and I_2, respectively. Let $T_1 \subset T_2$. What is the resulting relationship between I_1 and I_2? Well, we may first conclude that $I_2 \subseteq I_1$. This follows from the intuitively obvious fact that any feature invariant over a certain class of transformations (T_2) will be invariant over any subclass of that class of transformations (of which T_1 is one).

[10]This could also be characterized in terms of domains, but even so what really matters are the transformations involved.

We could, for an example, let T_1 be the class of isomorphisms and T_2 be the class of homomorphisms defined on model-theoretic structures, as we have been considering in the context of first order logic. Now, clearly $T_1 \subset T_2$, since every isomorphism is also a homomorphism but the converse is not true. If a feature f is invariant under homomorphisms (T_2), that is $f \in I_2$, then it must certainly also be invariant under isomorphisms (T_1) since these are also homomorphisms, and hence $f \in I_1$.

The question, then, in any given case is whether $I_2 \subset I_1$ or $I_2 = I_1$. If $I_2 = I_1$, then there is no difference in degree of formality between any $x \in I_1$ and $y \in I_2$, with respect to D, because it turns out that I_1 (being identical to I_2) is invariant over T_2 as well, despite that $T_1 \neq T_2$. In fact, this means that T_1 and T_2 both have the same maximization, and hence T_1 and T_2 are associated with the same class of invariants.

The more interesting situation, which occurs for our logical example, is when $I_2 \subset I_1$. The invariants belonging to the subclass of I_1 which also belong to I_2 (the intersection) will have the same degree of formality as I_2 (because $I_1 \cap I_2 = I_2$). In the case of homomorphisms and isomorphisms, this intersection includes the logical notions denoted by the standard quantifiers ($\forall$ and $\exists$) and the standard logical connectives, but it does not include identity.[11] The class of features $Q = (I_1 - I_2)$, which is the maximal subclass of I_1 not containing any element of I_2, we call *less formal* than those which belong to I_2. The most notable element in Q in our example is the denotation of "=," which is invariant under isomorphisms (it belongs to I_1) but *not* under homomorphisms (it does not belong to I_2).

As a consequence, we consider the denotation of "=" to be *less formal* than the logical notions denoted by the quantifiers and connectives. This does *not* mean, however, that we take identity to be any less logical than the other notions. We maintain a distinction between formality and logicality which permits that notions differing in their degree of formality may yet be equally logical in nature. The question of logicality, from our perspective, hinges on the content of the theory, not its formal nature.

[11] [Feferman, 1999]

Furthermore, it is customary in the field of logic to say of systems with additional semantic resources (e.g., first order logic with identity) that they are *stronger* than those with fewer resources (e.g., first order logic without identity). This is grounded in the relative expressive power of the languages — those with greater semantic resources will be capable of saying more, and hence result in a theory consisting of more and stronger claims than theories with fewer resources. It may consequently seem backward that we claim a theory with greater semantic resources is *less* formal on account of these semantic resources — that we in fact make the weaker language the stronger. However, we again invoke the distinction between logicality and formality. Degrees of formality are as distinct from logical strength as formality is from logicality. Although in general it is true that a more formal system will be logically weaker (since invariance over a wider range of transformations will often impose stricter limits on what one can say), this is not always the case. In the case of logic, for example, the logical constants do not exhaustively denote the features invariant under either isomorphisms or homomorphisms. This means that stronger logical systems may be formed by including additional constants, without thereby having any effect on the formality of the resulting language. Likewise, certain constants could be dropped from a language resulting in a weaker system with no change in formality (although one must be careful, since as we have seen, dropping "=" *will* result in a change).

This way of thinking seems further warranted on account of the invariants of I_2 being stable over the broader range of transformations T_2 than those of Q, which are only invariant over the narrower class of transformations T_1. Hence, the elements of Q are *less invariant*, ergo less formal or of lower degree of formality, than the elements of I_2. The fields of mathematics and geometry are rife with systems related in this way.[12]

It is furthermore not necessarily the case that a more formal theory is better or more useful than a less formal theory. To see this, we can consider the chain of theories discussed in Chapter 2 during our generalization of invariance. This chain is depicted in figure 5.1.

[12] An interesting example of this is the "chain of formality" having as a less formal member field theory and as its most formal member category theory. Between this two lies a considerable range of interesting theories, which are ordered by the more-formal-than relation.

Our introduction to this hierarchy began with Klein's work on geometry. While we only considered a very limited theory, Klein's actual work extended to characterizing many types of geometry. The transformations Klein employed were bijective mappings of space onto itself, with an additional condition that some or other properties must be preserved. The objectualist approach to the semantics of first order logic takes this a step further, not requiring any particular properties to be preserved, but instead requiring certain structural features of universes as a whole to be preserved. Within this group, views differ over which structural properties are properly logical (e.g., isomorphisms preserve the cardinality of universes, homomorphisms do not), but the resulting views are in most cases related in terms of degrees of formality. From here, invariance was further generalized by Bonnay, who produced a number of classes of transformations which preserve even fewer structural features of model-theoretic structures. At the very top of this chain we reach the class of all transformations between model-theoretic structures, which we already encountered in our discussion of degeneracy above. As we said there, the corresponding class of invariants at this level consists primarily of semantic categories. What we did not discuss above, however, is whether or not these invariants are useful or interesting. For despite the fact that the semantics categories are the most formal features in this particular chain, they are in most respects the least interesting. At this very high degree of formality, the most that we can hope to learn from the invariants is the general nature of the domain of invariance itself — that is, what the criteria for its selection was. In this case the semantic categories tell us much about what it is to be a model-theoretic structure, but nothing more. But in order to even start down this path we needed to have a definition of a model-theoretic structure, or else we would not be able to characterize classes of transformations between them. Thus, the invariant features we find at the top of the chain of formality, when variance is maximized, have an air of tautology about them. We would not say that such features will *never* prove interesting in and of themselves, but they certainly do not have our attention in the way logical notions do.

On the other end of the spectrum we have speculatively suggested that at least some

of the special sciences are less formal than geometry, yet still within this chain of formality. This would be the case if the characteristic invariant features of these disciplines were invariant over transformations which belong also to the classes above them in the hierarchy, and in the case of logic this is not too difficult to imagine — with limited exception, large segments of the actual world can be characterized as model-theoretic structures. And while these scientific theories are certainly very interesting and informative, they lack the generality and scope that is so compelling about logic.

All of this is to say that, although we have certainly not considered it in as much depth as it deserves, formality may only be interesting up to a certain degree. As we consider theories with increasing degrees of formality, the breadth and generality of the features involved reveals to us significant and interesting facts and relationships. However, at the far end of the process, we end up learning little if anything more than what we began with. It thus may well be that, while logic is not the most formal theory in this chain, it is yet the most formal theory which is of considerable interest.

It is noteworthy that the GTOF is not included in figure 5.1, even though it involves a further generalization of invariance. This is because, as we said in Chapter 2, the sense in which invariance is generalized is that it permits the characterization of arbitrary types of invariance, not just those associated with model-theoretic structures. Thus, the depiction in figure 5.1 is just one of many possible "trees" of formalities that fall within the scope of the GTOF. What is also not represented in the diagram is that formality hierarchies may branch, such that two theories are each less formal than a third, but have no such relation to one another. This can be seen from the fact that, if transformation classes T_1 and T_2 are completely disjoint, no comparative judgment can be made. A similar result holds when T_1 and T_2 overlap without being coextensive, in which case both have a relation to a third class of transformations, their intersection, but not directly to each other. In such a case, the formality hierarchy branches, such that these two types of formality are each less formal than a third, but have no such relation to one another (a possibility not depicted in figure 5.1). The more-formal-than relation is therefore only a partial ordering relation, as

it only obtains when one class of transformations is a subclass of another.

A final thing to recognize is that if one theory or language, L_1, is more formal than another, L_2, this means that all of the features employed in L_1 are also invariant features for L_2. This is significant in explaining the mixture of formal languages that are sometimes used in mathematics, set-theory, and most notably the sciences. If the scope of a given field, say physics, is less formal than another theory (or is assumed to be so), for example mathematics, then physicists are justified in making use of mathematical invariants in constructing physical theories.[13] This gives rise to "hybrid" formal languages which contain fragments of languages with a higher degree of formality.

This also explains why, for example, cosmologists borrow from mathematics and not biology. They assume, and with good reason, that cosmology has a lower degree of formality than mathematics, and hence they are justified in employing mathematical invariants in their cosmological theories. The domains of cosmology and biology, so far as we know, do not share this relationship. They may not even belong to the same chain of formality.[14] The general principle that results from this is that a formal theory based on a certain set of invariants may make use of invariants from any other theory which is strictly more formal than itself.

[13]The case of mathematics is interesting and important for a reason in addition to its pervasive use in the sciences. When taken as a first order theory, a theory about numbers (whatever those may be), mathematics is not a particularly formal theory. If it is just a theory about transformations between numbers, for example, it is not at all clear how physics would fit below it on our hierarchy, or even how it is that mathematics plays any role in physics at all (unless physics itself is just about numbers). However, a second order perspective of mathematics, in terms of properties (or classes) of classes, does possess a considerable degree of formality and is the sort of theory physics and other sciences might fall under and could borrow from. It is therefore an interesting question what the relationship between first and second order mathematics may be. Although we cannot hope to answer this question here, we speculate that first order mathematics may be related to second order mathematics in much the same way that a formal language is related to its formal semantics. If so, first order mathematics would be a convenient, simplified apparatus for exploring and reasoning about second order mathematical objects.

[14]The phrase "chain of formality," though appropriate in many cases, could be misleading. If it is the case, for example, that both biology and cosmology are less formal than logic but that they share no relation to one another, perhaps on account of merely overlapping transformation classes, then the result would rather be a "tree of formality" than a chain.

Formality in Multiple Respects

In Chapter 4 we compared formal programming languages not only in terms of degree of formality, but also in terms of something we called "respects." Indeed, in even discussing degrees of formality, we have indicated that these are always made with respect to a certain frame. This idea is worth looking into a bit further, and we should, in particular, clarify how it can be that a language, denotation of a constant, or anything else might be formal in more than one respect.

In our discussion of degrees of formality, we used frames consisting of related transformation classes, defined either on the same domain of invariance or effectively involving a subdomain relation. Consider, for example, the relationship between the views which take invariance under permutations and isomorphisms to be characteristic of logic. Clearly, the two positions involve non-identical domains of invariance. The permutation view consists of a single universe, all of whose individuals are permuted to one another. The invariance domain at play with isomorphisms consists of model-theoretic universes. It is, however, easy to see that the universe involved in permutations (whichever it may be) is itself a model-theoretic universe, and also that permutations are included among the class of isomorphisms. Therefore both the class of transformations and domain of invariance on the permutation account are subclasses of their counterparts under isomorphic invariance, as depicted in figure 5.2. This is why the features invariant under isomorphisms are more formal than those invariant only under permutations. Such features are, in a clear sense, formal under both classes of transformations.

There is, however, also the possibility that features may be formal under more than one class of transformations which are not related to one another in this specific way. It is these features that we consider to be formal in "multiple respects," and they can be characterized as follows. First, for a class of transformations T to be defined on a domain of invariance D, it must be the case that every object in every transformation of T must belong to D. If we take two domains of invariance, D_1 and D_2, such that they are entirely disjoint, the result is that any classes of transformations defined on these domains, T_1

and T_2, respectively, will also be disjoint. Critically, it does not follow from this that the resulting classes of invariants, I_1 and I_2, will also be disjoint, and any $x \in I_1 \cap I_2$ will be invariant, and therefore formal, with respect to both frames. An example, also depicted in figure 5.2, will help make this clear.

Let D_1 be the domain consisting of all mice without genetic defects, and let D_2 be the domain consisting of all mangoes, likewise without genetic defects. Let transformation classes T_1 and T_2 be defined on these domains respectively such that each transforms every object of that domain to every other (mice to mice, mangoes to mangoes). Clearly T_1 and T_2 are disjoint. Furthermore, the resulting classes of invariants will not be equal, nor will either be a subclass of the other. But are these invariant classes disjoint? No. The property of having forty chromosomes turns out to belong to both classes of invariants. In such a case we can say that "has forty chromosomes" is formal with respect to the frame (T_1, D_1) and the frame (T_2, D_2).

We could, of course, try and create a "super domain" encompassing both D_1 and D_2, called D^S. But this won't change anything in terms of the invariants which are formal with respect to multiple frames, on account of the disjoint transformation classes. If we construct $T^S = T_1 \cup T_2$, the resulting I^S will be, not surprisingly, $I_1 \cap I_2$, which will include "has forty chromosomes" but be unequal to either I_1 or I_2. Rather than collapsing or merging the two original frames, we have produced yet another frame and a further respect in which the feature is invariant.

Thus, the expression of a program in an object-oriented manner, as described in section 4.2.2, can be formal with respect to the machine-oriented frame — consisting of a domain of machines and its absolutely maximal class of transformations — and also a problem oriented frame constructed around the class of problems the program is designed to solve.

It is important to recognize that being formal in multiple respects is decidedly different from the possibility of making use of features with a higher degree of formality, as discussed in the preceding section. Although in such a case the features *are* formal

in multiple respects, there is a relationship between the associated domains or classes of transformations. In the present context, no such relation is presumed, and the notion of formality with respect to a frame is therefore more general than, and implicit in, the idea of degrees of formality. It is also fairly easy to see that most features may be formal with respect to a great many frames. Just which frames are taken into consideration in a given context is primarily a matter of the interests involved in exploring the invariants in question.

Stipulated Relations

The final point to discuss concerns the manner in which respects of formality may be related to one another by stipulation. This occurs more frequently than we might initially think, as most (if not all) representation relationships involve this. The most significant case, of which we considered particular instances in Chapter 4, is that of a formal language and its semantics (or its constants and their denotations).

The basic framework for any such relation consists of two, possibly identical but usually distinct, formal frames. These frames each give rise to a certain class of invariants. A relation may then be defined between these classes of invariants. The thesis we explored in Chapter 4, that invariants in the language (characteristic constants, types, grammatical constructions) denote invariants in the semantic domain, can be understood as the claim that there is a function from the invariants in the language-oriented frame to the invariants in the semantics-oriented frame. Furthermore, although nothing prevents formal languages from containing perfect synonyms within the primitive, characteristic vocabulary, they generally do not. Hence, we may add the restriction that the function, which we may as well call a "denotation function," is injective. These functions should not be expected to be surjective, since formal languages nearly always involve a limited number of invariants and the associated semantic domain is usually considerably more complex.[15]

[15]Perhaps the most plausible context for finding bijective denotation functions would be to consider "term models" for languages. Even so, it seems likely that there may be invariant relations or higher order relations

The stipulated relations need not be functional, as they are in the case of formal languages. It just turns out that since formal languages usually involve denotation functions anyhow, functional relations are very useful. One non-linguistic situation where stipulated relations between invariants comes up is in the analysis and engineering of processes. In an analysis, the important features of things like agents, action, events, and products are characterized effectively by means of invariance. For example, instead of describing a process in terms of particular agents, by name or other reference, the process may simply describe a type of agent or role. This type is defined in terms of qualities and abilities which (it is expected) all agents of that type have. Since these qualities and capabilities are effectively invariants, it is easy enough to characterize these agent types in terms of invariance. The same will hold true for other components of the process. Once these invariant features have been identified, a process structure can be defined by specifying relationships between these types or certain invariant qualities, abilities, etc. [16]

In cases like this, of course, there may well be *real world* relations that correspond to the stipulated ones — for example if an analysis describes an existing process. Alternatively, a new process may be engineered by the specification of new relations. We consider both of these to be stipulative since even in the case of actual, existing relations these are not a direct consequence of the composition of the formal frames, and the relationship must be identified independently of considerations of formality. Hence, there may well be relations of this sort which are empirically determined or necessary. But because the relationship requires human characterization (e.g. it is part of the content of a theory), from a formal perspective these are also constructed or stipulated.

---

which do not belong to the image of the denotation function. Still, it might be possible in the case of a term model for a language explicitly designed to express such relations.

[16] It should not be surprising, in light of what we now understand of them, that formal languages have been developed to facilitate these activities.

5.2 Philosophical Considerations

We will now turn to some more broadly philosophical issues with the purpose of providing a better understanding of how our theory of formality relates to other philosophical work. These also divide nicely into two categories. The first consists of a prominent position on the association between logicality and formality put forward by John MacFarlane, and it is in some ways closely related to our GTOF. The second group of views feature invariance in a prominent way but connect it with something other than formality (and in general, have little to say about formality or logic).

5.2.1 John MacFarlane and the Specialness of Logic

In the title of his dissertation John MacFarlane asks, "What does it mean to say that logic is formal?"[17] His motivation for asking and seeking answers to this question is the identification of three distinct traditions in the history of philosophy, each with its own account of what makes logic distinctively formal. And though we do not ourselves seek an answer to this question, it is easy to see how the GTOF exacerbates the problem. Under the GTOF there are, quite literally, indenumerably many distinct types of formality, and we have already shown that there are a number of these on which first order logic comes out formal.

The GTOF, however, not only deepens the challenge of identifying the characteristic formality of logical systems generally, but to a significant extent it undermines the claim that logic is distinctively formal in the first place. For if logicality and formality are distinct, and everything can turn out formal in some respect, it is not enough to merely identify some way in which logic is formal, even if nothing else shares that particular type of formality, and say that it is thereby *distinctively* so.

MacFarlane's work presents us with two questions. First, whether any of the types of formality he discusses could themselves serve as a general theory of formality — either

[17] [MacFarlane, 2000]

identical to or in competition with the GTOF. Second, if we assume that the GTOF is true, and therefore that logic fails to be distinctively formal in a strict sense, might there still be a sense in which logic is somehow *specially* formal under the various notions of logical formality.

We must be clear, however, that our purpose is not to answer the question of what it is that makes logic special, since logicality is not our direct subject. Instead, we will suggest and explore ways in which logic can still be said to be specially formal, even if neither uniquely nor distinctively so. While in Chapter 3 we focused exclusively on the position that logicality consists of model-theoretic invariance, here we will look at a broader range of views, most of which are discussed by MacFarlane. It is our opinion, first, that none of these can serve on its own as a general theory of formality. Second, that each of these accounts involves a different respect of formality which could be rigorously characterized according to the GTOF. However, presenting detailed arguments to this end is a significant work in its own right, and doing so is beyond the present scope. Nevertheless, we provide some informal discussion concerning the role played by invariance, as for many of the views this characterization opens up a plausible sense in which logic could be considered specially formal.

MacFarlane begins by discussing three accounts of logic as formal that he considers "decoys" from the demarcational perspective.[18] With respect to two of these, syntactic and grammatical formality, we have already discussed their satisfaction of the GTOF (under the terms "sentential" and "grammatical" formality, respectively). The same reasoning used there explains why, in addition to being unfit for demarcational purposes, these types of formality cannot ground a general theory of formality or be seen as providing logic with any special character or place among our theories. First, these formalities are tied essentially to languages, and it is clear that any general theory of formality would need to extend beyond linguistic formality. Second, many, many languages may exhibit these types of formality, and many do so in potentially more interesting or basic ways than

[18] [MacFarlane, 2000], Chapter 2.

standard first order logic.

The third "decoy" is what MacFarlane calls "schematic formality." Schematic formality occurs, by nature, at the level of language. It is in this respect somewhat similar to syntactic formality, except that whereas syntactic formality involves the rules of formation of a language, schematic formality is based on other less fundamental structures. From the schematic perspective, logic consists of a collection of "argument schemata." The schemata are not themselves arguments, but they consist of a number of nearly complete statements with indicators (schematic letters) of how the statements may be completed to produce an actual argument. If done correctly, the schematic arguments should be such that every possible completion of the argument schema results in a valid argument.

First, a word about schemata and schematics more generally. It is easy to see the role that invariance plays in the case of logical argument schemata. Argument schemata, though not themselves valid, or even arguments, can be thought of as characterizing a collection of arguments — those which can be formed by finishing the schema according to certain rules. It is a straightforward step to recognize that the use of argument schemata suggests not only that the property of being an argument is invariant across transformations between all arguments within the class, but in most cases also that the property of validity remains invariant across these transformations (although there certainly can be invalid argument schemata, these are generally not taken as a proper subject of logic). Schemata are, therefore, another way of thinking of the structure of logical statements and arguments. It is distinct from syntactic formality in that the schematic letters are not really considered *part* of the schema, except insofar as they indicate a relationship between statements (e.g, specification of a "middle term").

This way of thinking can be easily extended to cover other kinds of schematics or uses of schemata. In *most* cases (for we do not wish to be drawn into a discussion of the nature of schematicity), schemata are associated with some representational system or other. This may be linguistic, as in the case of logic, but it also may be diagrammatic

(building schematics, flow charts), or possibly even pictorial or graphical.[19] What these schemata all have in common is being taken as incomplete in themselves, but as such they characterize the class of objects which are completions of the schema in question. Putting this in terms of invariance is straightforward.

MacFarlane rightly determines that schematic formality is not an adequate ground for the demarcation of logic. While he cites the possibility of non-logical schematic arguments as reason for this, what we have just said is even more compelling. Not only is there nothing essentially logical about schematic formality, there is also nothing essentially argumentative or even linguistic. On the other hand, schematic formality in some ways shares a close affinity to the GTOF, since an implicit understanding of the GTOF is that characterizing something in terms of formality or formal features is, at least in all useful cases, to characterize it *incompletely*. That said, schematicity, even if it does range beyond representational systems to encompass everything a general theory of formality should, is rather to be considered a *symptom* of formality rather than a general account of it.

None of this means, though, that there are no reasons for thinking that logic is special from a schematic perspective. For example, there are certainly logically invalid arguments which can be schematized and which are valid (or whatever lesser notion we would employ) with respect to a certain limited range of completions. However, it might turn out, and indeed it seems likely, that logical schemata hold over the broadest possible range of completions. Even though schematic formality is not distinctive of logic, logic may still exhibit the highest degree of schematic formality of any argument-schema system that we know of. If so, this would certainly warrant it a special status, especially if no other argument-schema system comes close. If true, it might consequently be thought that logic

[19]The last of these is probably not common, but is becoming increasingly so. One example can be found in what are sometimes called "memes" (in a non-technical sense). These consist of stock images upon which text is superimposed to provide a certain consistent effect (usually humorous or ironic), despite that the resulting composite graphics usually have distinct targets for their humor or irony. The text itself also frequently follows a schema, so the result comes about by completing a number of different schemata.

characterizes the most general class of valid arguments.

In addition to these decoys, MacFarlane seriously considers three respects in which logic is seen to be formal, unimaginatively named 1-formality, 2-formality, and 3-formality. We will examine each of these in turn, but one interesting general note MacFarlane makes is that "we can get at these three notions by construing 'formal' as 'independent of content or subject matter."' As we saw in Chapter 4, "independence" is often characterizable in terms of invariance, and so it is telling that MacFarlane connects the formality of each of the views with the idea of independence. Just what kind of independence, he recognizes, will depend on how "content" is understood.

"To say that logic is 1-formal," MacFarlane tells us, "is to say that its norms are *constitutive* of concept use *as such* . . . 1-formal laws are the norms to which any conceptual activity — asserting, inferring, supposing, judging, and so on — must be held responsible."[20] The content independence this is supposed to express is that logic is "applicable without qualification, in any domain." The idea is that the rules described by logic actually govern our use of concepts, hence any use of concepts, in any domain, will adhere to the rules of logic (and hence logic will be applicable there). Now, presumably, there may well be other rules that govern concept use. Particular domains may have rules about how concepts within that domain will be used, and even particular concepts may have rules of use. However, the 1-formal position holds that logic is about those rules which are so general that they actually *define* concept use, and hence there is no such thing as non-logical concept use.

The first thing to note about the position is that it clearly cannot serve as a general theory of formality since it is tied essentially to concepts and their use. It may well be that formality is involved in the use of concepts, but there are many cases where the formal system or features of concern have less to do with the concepts involved than other things (e.g., a particular language, physical properties).

That said, it does seem *prima facie* plausible that 1-formality can be characterized

[20][MacFarlane, 2000], p. 51.

in terms of invariance. The most promising way to do this is to characterize it in terms of rules which are invariant under transformations between uses of concepts. In fact, if we conceive of it this way, we can see that we will end up with a complex hierarchical structure of rules of concept use, defined in terms of the range of the rules' invariance — their degree of formality. If it turns out that there is a single set of rules which is invariant over all concept usage — i.e., rules which have the highest degree of 1-formality — we would be justified in thinking that these rules have a very special place among our theories, and if these rules are described or captured by our logical theory, then logic would indeed be special. We might even be justified in thinking, as those who hold that logic is 1-formal do, that these rules are special because they are *constitutive* of concept use.

On this account, the GTOF would not say that logic is special in being *distinctively* 1-formal, since many conceptual rules will have some degree of 1-formality. Instead it would say that logic is about those rules which have the highest possible degree of 1-formality. Even so, saying this presumes much. It requires that there is a single set of rules which governs all concept use *and* it requires that the rules characterized by logic be those rules. We do not know of conclusive results on either count.

What MacFarlane calls "2-formality" is something with which we are already familiar, as we discussed it at length in Chapter 3 as "objectual formality." Of course, this type of formality is explicitly characterized in terms of invariance, although there are many ways in which the associated class of transformations can be specified. It is equally clear that 2-formality cannot serve as a general theory of formality itself on account of its very specific relationship to model-theoretic structures. General though it is, it is not general enough.

Despite a generally critical stance toward most views in this tradition, it is still the one MacFarlane finds most promising for demarcational purposes, and his own position, which we discuss below, fits broadly under this heading. As promising as the position may be for demarcation, it makes it more difficult to say that logic is special from a formal perspective. This is because the classes of invariants used by any of the views in this

tradition do not, strictly speaking, realize the limiting case of "all transformations" under Klein's approach. This makes it appear as though it is not 2-formality that makes logic special and important, but the particular character of the transformations and invariants involved.

Some insight, however, comes from the ways in which philosophers with competing views on logic as 2-formal argue. In particular, contemporary philosophers do not seem to consider an extreme range of invariance (or "generality" as it is sometimes called), a point of contention. That is, they are not seeking a limiting case to Kleinian invariance — a position with the highest degree of invariance *simpliciter*. In certain cases the debates come down to whether or not a view is invariant enough or too broadly invariant (e.g., the question of whether identity is a logical notion can be understood as a question about how large the class of transformations should be). This suggests that there is some, albeit vaguely formed, intuitive understanding of the subject matter of logic. This further leads us to think that logic, whatever the correct view turns out to be, will be the theory with the highest degree of formality with respect to a certain subject matter.[21] Although we should avoid speculation about what this subject matter might be (since this is yet an open question in the philosophy of logic), we can with some confidence say that it includes everyday objects and objects in mathematics and the sciences which are "well-behaved."[22] Whether or not the domain of logic includes *all* objects is a question which has been raised by certain quantum mechanical phenomena. Nevertheless, on the suggested view, with respect to those objects and associated theories falling within the domain of logic, logic will certainly be seen as special in virtue of its formality. It will have preeminence over every other theory concerning objects which fall within its domain.[23]

[21] Similarly, it may be as we have suggested above, that logic is the most formal theory about a certain subject matter which is still interesting and informative.

[22] Of course, "well-behavedness" will need some definition which does not itself appeal to logicality. One approach would be to say that well-behaved objects are those which have properties satisfying certain rules. Just which rules these might be (e.g., rule of excluded middle) is a deep and interesting question.

[23] It is worth noting also that many of the purported virtues that have traditionally been attributed to logic, beyond generality, will either result from or have analogs which result from logic's high degree of formality in this respect. For example, the necessity of logic, with respect to the appropriate domain, is guaranteed

3-formality is by far the most conceptually challenging type of formality MacFarlane seriously considers. His gloss is that it "abstracts entirely from the semantic content or 'matter' of concepts — that it considers thought in abstraction from its relation to the world and is therefore entirely free of substantive presupposition."[24] It is, we are told, independent with respect to "semantic content." Just what this means is difficult to grasp because the idea bears a strong resemblance to 1-formality. MacFarlane's immediate discussion does little to clear this up, but elsewhere he does explain why 3-formality does not entail 1-formality, and this helps to understand the difference between the two. The crux of the issue is that, whereas 1-formality purports to be essential to the nature of concept use, 3-formality merely pertains to *ways* in which contentful concepts can be combined with one another. Hence, to say that logic is 3-formal is to say that it is *one* way in which semantic concepts may be combined. Other ways may be possible, we are told, and there is no constraint upon this from the world. No single way of combining concepts, for example logic, would be essential to concept use as such (although presumably that there is at least one way of combining concepts may be). In contrast, if logic is 1-formal, it is *the only* way concepts may be combined in thought, as it is constitutive of concept use. Logic taken as 3-formal, then, would be a framework within which ideas about the world could be combined, but would not itself pertain to the world or world-oriented concepts.

Thinking of 3-formality in terms of invariance actually provides a much clearer way to understand the difference between it and 1-formality. The formalization process for 1-formality begins by considering the ways in which we actually use concepts and

---

because logic itself was used to circumscribe the domain. This does not reduce logic to a triviality, however, for two reasons. First, whether or not an object or theory is "logical" is an interesting and important question. Second, even if logic is necessary over its domain in virtue of defining that domain, logic may still be very *useful* within this domain for reasoning about these objects. This is a point which also suggests a sense in which logic turns out *a priori*. If we define any set of rules and then gather together those things which satisfy the rules, we can say *without looking* at any particular object that it satisfies those rules. Now, if an awareness of these rules should happen to be somehow innate or learned at a very early age, and everyday objects happen to fall within the domain of the rules, then the rules could very well appear *a priori* in the traditional sense. An alternative account, wherein these virtues arise specifically from invariance under isomorphisms, is given in [Sher, 1999] and [Sher, 2008].

[24][MacFarlane, 2000], p. 51.

proceeds toward finding the rules for concept use which are invariant across all of these. That is, the domain consists of concept uses, and the class of transformations consists of all transformations between these uses. If logic really is 1-formal, it (or its characteristic invariants) should at least be among the resulting invariants. 3-formality begins instead with *stipulation* of a certain set of rules. Now, this stipulation could be involuntary (as Kant seems to have held) or voluntary (as Carnap certainly held). Nevertheless, once a set of rules (called a "conceptual framework" by logical positivists) has been selected, we may then proceed to either explore the framework itself (pure logic) or use the framework to combine concepts and make inferences (applied logic). In terms of invariance, the rules are stipulated invariants and the class of transformations consists of those transformations which preserve these rules. This is its main difference from 1-formality. Whereas 1-formality begins with a fixed set of transformations and seeks the associated invariants, 3-formality begins with a fixed set of invariants (rules) and explores the class of transformations and previously unknown co-invariants (e.g., theorems). A further difference between the two is that 1-formality presupposes a single transformation class for concept usage, whereas 3-formality does not presuppose that there is only one set of invariant rules. In this respect, unlike both 1-formality and 2-formality, 3-formality is not strictly normative (neither is it descriptive).

One thing that 3-formality does share with 1-formality is its inadequacy to serve as a general theory of formality, owing again to its close association with concepts and concept use. On both counts, it is difficult to see how any other instance of formality can be reduced to or explained by a presumably small and "content-free" set of rules for combining concepts.

Furthermore, although in principle 3-formality admits of degrees, just as the other types of formality we have discussed, because the position is not committed to a fixed foundation, there will be a much broader array of possible 3-formal systems. Some of these may be commensurable in terms of degrees of formality, but many will not be. It is furthermore difficult to see what difference degrees of formality would make on whether or

not the "logic" in question is considered special. One thing distinctive about 3-formality, however, is that anything having any hint of semantic content is excluded from being 3-formal. Hence, unlike every preceding consideration of the formality of logic, there is no 3-formal theory which does anything more than lay down rules for combining semantic concepts. Thus, when contrasted with other scientific theories, any 3-formal system will be formal in a way that the other theories cannot, by definition, be. Within the class of 3-formal theories, it may not be possible to grant a particular theory, say first order logic, any special place in virtue of its formality. However this is consistent with the views of many of those holding logic to be 3-formal, since they normally appeal to other reasons for explaining why we use first order logic rather than some other 3-formal system.

MacFarlane directs his most detailed criticisms at the champions of 2-formality. The specifics of these are not of interest to us here, but the general theory of formality makes it relatively easy to summarize. MacFarlane's primary concern is that sufficient reason is not provided for selecting a certain class of transformations (e.g., isomorphisms) in defining logicality, rather than some other. At best, 2-formality is endorsed as a demarcation of logic on intuitive grounds, which we agree with MacFarlane should always be suspect. In particular, MacFarlane is concerned with the limitation of the class of transformations to only transformations of objects, and not including other "presemantic types" (i.e., semantic denotations considered independently of language), specifically truth values. Truth values, to paraphrase MacFarlane in a vocabulary we are now familiar with, are implicitly fixed invariants that produce the correct results. It is this fixing that requires further justification.

MacFarlane's answer to this is that we choose the largest class of transformations which enable the resulting invariant notions to serve as a semantics for a language which serves certain pragmatic ends, and no more. The pragmatic ends MacFarlane identifies are truth and inference. According to MacFarlane, this also makes logic 1-formal, since the invariants which are necessary to support both truth and inference are those which are constitutive of thought *as such*.

There are many questions to be asked about MacFarlane's position. For example, is the resulting theory really constitutive of thought as such (1-formal), or is it perhaps constitutive of thought as it concerns truth and inference? Or, perhaps it is a theory about truth and consequence, but having little to do with what is constitutive of thought. These questions will lead us too far afield. What we must ask here is, first, does his position fare any better as a general theory of formality, and second, is there a sense in which logic can be said to be specially formal? To the first question we can easily answer negatively on account of its close relationship both to concept use and model-theoretic structures, for the same reasons discussed above.[25]

To the second question, however, we can answer affirmatively. First, we can appeal to what we said before regarding 1-formality. Although logic may not be distinctively 1-formal (because there are lesser degrees of conceptual formality), it may well be special in being maximal in terms of conceptual formality. In addition to this (or if MacFarlane is wrong about the resulting 1-formality of logic) we can appeal to what we said before about 2-formality, but with the identified subject matter of truth and inference. That is, logic would be specially 2-formal in being the theory about truth and inference with the highest degree of 2-formality.

Certainly what we have just said is neither rigorous nor complete, and we cannot therefore affirm with much certainty that the GTOF does not strip logic of every kind of special formality we may have thought it had. What we hope we have shown is that it is *prima facie* plausible to accept the GTOF and also maintain that logic is somehow specially formal. We have also tried to show that to a large degree this is independent of the view one holds about the nature of logic itself.

In nearly every case our approach rested on some appeal to the degree of formality of logic, which is perhaps to be expected. By nature, a higher degree of formality engenders an increased scope (of invariance) and corresponding broader applicability of

[25]Of course, presenting a general theory of formality was *never* one of MacFarlane's goals, and we in no way see his work as competing with our own. In fact, there are some significant points of agreement, not the least of which is the recognition that logic presupposes pragmatic preoccupations with truth and inference.

the resulting theory. What these virtues amount to will, of course, depend on what views one holds about the nature of logic, as this determines the respect in which logic will be thought to be specially formal.

Since we have no particular horse in the logical demarcation race, it occurs to us that we might be able to say something much broader than what we have said in these cases. In particular, whether or not logic is demarcated by any of the preceding types of formality, it certainly seems plausible to us that it possess several of them to a degree that warrants being considered formal in a special way, and it may yet possess more. If so, we could go beyond saying that logic is specially formal in a particular respect to saying that logic is specially formal in its "formal centrality." That is, although there may be theories that are more formal than logic in one respect or another (e.g., a theory of semantic types), perhaps no other theory can be said to be specially formally in so many respects. In particular, if logic is a highly formal theory associated with our use of concepts (either 1-formality or 3-formality) and is also a highly formal theory about things in the actual world (2-formality), then it would be very special in that it lies at the central intersection — so to speak — of our concepts and the world. A more conceptually formal theory may lack application to the world, and a more objectually formal theory may preclude combining concepts in useful and informative ways. Although this example is highly speculative, it is not wildly implausible.

5.2.2 Robert Nozick and Objectivity as Invariance

In relating our general theory of formality to other philosophical works we would be remiss if we left out Robert Nozick's *Invariances: The Structure of the Objective World*.[26] In this work Nozick puts forward some interesting theses centered around epistemic issues of relativism, objectivity, and subjectivity.

Despite the title, however, the book is not rife with discussion of invariance, which

[26][Nozick, 2001]

is sequestered almost entirely into a chapter dealing with objectivity and subjectivity in relation to facts and beliefs. True to form (that is, consistent with characteristics that have remained more-or-less invariant in Nozick's work across transformations of time and subject matter), even within the limited span of a mere forty pages or so Nozick leaves little time to ponder or ruminate as he darts from topic to interesting topic. We are taken from nothing less than a proposal for what objectivity is to a discussion of the reliability of scientific methods and the plausibility of scientific progress. We could not possibly address all that is of interest were it even within the scope of our current project. From our perspective, the key ideas concern the account Nozick gives of objectivity as invariance.

Even without further detail, we can see why this is a significant idea for us. It poses a challenge to the GTOF in that, if Nozick is right (at least in the way he sometimes presents it) *every* invariant might turn out "objective." This would mean that everything we have called formal would also be objective. Such broad coincidence, or identification even, seems suspect when it is purely the consequence of theoretical definitions.

On the other hand, we find the idea that objectivity could have something important to do with invariance interesting, plausible, and worth working out. Nozick himself takes some steps toward this end, but for the most part the discussion is quite general, wide ranging, and, dare we say, informal. This is particularly evident, for example, when he discusses what he perceives to be two distinct types of transformations with respect to which something may be invariant. The first one we are already familiar with — mathematical mappings or functions. The second, he claims, is the sense of "change" or "transformation" in which one object is altered or changes over time into a different state. What he seems to mean here is invariance over some kind of *process*.

What certainly is true is that mathematical functions cannot generally be thought of as characterizing processes, in the sense that they do not represent any sort of gradual or sequential change between the input and output values, and there is generally no commitment to preserving key identities (as is practically necessary to identify something as a process). It seems straightforward, however, that process transformations are really just a

special case of functional transformation when considered exclusively from the perspective of invariance. That is, when we say that x remains invariant under a physical process P, what we mean is that x is invariant under a mapping between physical states which are characterized by P. Now, P may be very complex, and in general is probably not arbitrary. It may be grounded on some causal concept or other, and it may easily be a class of transformations with trans-finite cardinality (e.g., infinitesimally small increments of change).

Nevertheless, there is no fundamental difference between the two classes when considered solely from the perspective of invariance. All that invariance takes account of is the mapping, not the definition or reason for that mapping. This does not, of course, mean that transformation classes may not differ in terms of conceptual provenance, and thereby also differ in terms of how interesting or useful they are. They often do.

Nozick's informal approach to the topic, while in some ways very accessible, poses a significant exegetical challenge if we want to understand his ideas in anything but the broadest strokes. In what follows, we present Nozick's view in the general terms he himself uses, pointing out some of the more serious problems as we go along. Finally, we propose a new position which we feel is consistent in spirit with much of what Nozick says, but which we feel is more consistent, plausible, and also resonates nicely with the GTOF.

Nozick begins by identifying three intuitive characteristics of objectivity — accessibility from different angles, intersubjective assent, and independence. These, he claims, are grounded in a fourth "more fundamental" characteristic. "An objective fact," he claims, "is invariant under various transformations. It is this invariance that constitutes something as an objective truth, and it underlies and explains the first three features (to the extent that they hold)."[27]

Nozick takes as his inspiration for this view the use physicists make of invariance. He claims that physicists "treat what is invariant under Lorentz transformations as *more*

[27] [Nozick, 2001], p. 76.

objective than what varies under these transformations."[28] Nozick goes on to point out the same circularity we have taken care to note ourselves: if the claim is that what is objective is what is invariant under certain transformations, it is not informative to define these transformations as those under which objective facts are invariant. It seems as though we need one or the other first if the position is to be at all illuminating.

It is Nozick's view that we do not begin with some *a priori* understanding of objectivity or objective facts. Instead he suggests an iterative process whereby the class of invariants and the associated class of transformations are continuously revised. Nozick reveals later that the starting point of this iterative process "is fixed by the phenomena and by the invariances that evolution has shaped us to notice and take account of."[29] Under his description, we begin with such a list of everyday invariances. The objectivity of these facts supports or entails the objectivity of other facts which are not on the list. These additional facts are themselves invariant under transformations which may not be in the original associated transformation class. Consequently, some of the original invariants may fail to be invariant under the new transformations, and are dropped from the list. The process iterates from this new list (with the addition of the new invariants), with each iteration producing a new list of invariants to be used in the next iteration. Nozick adds, "If well-confirmed theories that are our only explanation for certain phenomena fail to be invariant under certain transformations, these transformations will be dropped from the list of admissible ones."[30] Thus, the process as Nozick describes it might well result in invariance and transformation classes which contain none of the original members.

We will not linger too long on this story, since epistemology and philosophy of science are not our central focus, but it is worth pointing out that this iterative dynamic Nozick proposes runs into some difficulties if we think more clearly about invariance. Certainly, if we begin with a small class of invariants I (what we believe to be objective facts), this can be used to define a corresponding class of transformations. We may then

[28] [Nozick, 2001], p. 76. Emphasis added.
[29] [Nozick, 2001], p. 80.
[30] [Nozick, 2001], p. 80.

expand our class of invariants to include all facts which are invariant under this class of transformations ($MAX(I)$). All this says is that a class of invariants I may not be maximal, as defined above, and this is fine as it stands. The problem, however, comes in the next step. One of the two classes needs to change, but Nozick suggests that the reason for changing one or the other will come from within the class of invariant facts itself. These will, purportedly, suggest that other facts are objective, but these will somehow be associated with a different class of transformations. But we now know that there is a necessary link between a class of invariants and its associated class of transformations. How could these invariant facts suggest that another fact is "objective" (e.g. invariant) when that fact *is not* in fact invariant? If it were invariant, then it would already be included in the class of invariants and, more importantly, it would not result in a change in the class of transformations. This is not necessarily a fatal flaw, but Nozick owes us an explanation of how new facts are indicated as being objective. Is it some kind of entailment relation? If so, which one?

That said, the fact that such a process is seen to underlie the definition of "admissible transformations" helps explain why Nozick always stops short of explicitly stating what these transformations are, despite his frequent intimations that they are Lorentz transformations. This is because what counts as an "admissible transformation" can change with time as science progresses, which leads Nozick to conclude that "the only general statement we can make is *very* general: An objective fact is one that is invariant under all admissible transformations."[31]

This is the core of Nozick's idea of objectivity as invariance. He proceeds to offer some possible elaborations (e.g., adding theories or models to the process) and some speculative reasoning about degrees or levels of objectivity. It is nowhere clear whether Nozick is committed to any of these ideas, and the presentation makes use of poorly defined notions like "levels" and "objectivity-at-a-level." Are these levels of invariance? Of transformations? Of science? It is clear that Nozick has something specific in mind,

[31] [Nozick, 2001], p. 82.

since he even proceeds to speculate concerning the possibility of an infinite series of such "levels," but the idea is never made explicit.

It is tempting to think that Nozick is gesturing at something for objectivity which is much like our sense of "degrees of formality," except that a few pages later he explicitly puts forward an account of degrees of objectivity (very much similar to what we have done for formality). In what is perhaps the most detailed idea Nozick puts forward, he suggests that fact x may be *more objective* than another fact y (have a higher level of objectivity) if x is invariant under a class of transformations which is a proper superset of the class of transformations under which y is invariant. Nozick points out that the ordering thus imposed is only partial (just as for degrees of formality), and he even speculates on whether there might be a class of transformations under which nothing is invariant. What Nozick never does is make clear how, or if, this later idea is connected to the "levels" he discusses earlier.

In the midst of this discussion we also encounter a peculiar summary statement. Nozick writes, "Invariance under specified transformations explains the three marks of objectiveness mentioned earlier (namely, accessibility from different angles, intersubjective agreement, and independence from the observer and the theorizing mind)."[32] What is peculiar about this is that it is never made clear whether Nozick takes this to be something he has shown, something that he will show, or something that should somehow be obvious from what he has previously stated.

It is only several pages later, after the discussion of degrees of objectivity, that Nozick seems to address this claim. Even then, the only "mark of objectiveness" he takes up is intersubjective assent. His argument is brief and similar to a familiar argument from Plato's *Euthyphro*. Nozick reasons that it cannot be intersubjective assent which *makes* something objective, since nothing prevents a group of subjects from agreeing on actually false premises. Instead, it must be that intersubjective assent is possible *because* of something else about a fact which is intersubjectively assented to. That is, a fact is

[32][Nozick, 2001], p. 85.

not objective because we intersubjectively assent to it, we intersubjectively assent to it because it is objective (or has some objectiveness feature). This "objectiveness feature" is, on Nozick's view, "being invariant under admissible transformations." Nozick goes on to claim, "We need to find an objectiveness property such that we can see (at least sketchily) *how* that property produces (or tends to produce) intersubjective agreement. And with invariance as the objectiveness property, we are able to do this."[33] What is frustrating is that, although Nozick says we are able to tell this story, even "sketchily," it is not at all clear whether he has or intends to tell such a story. It is never made explicitly clear how broad invariance, e.g. under Lorentz transformations, enables intersubjective assent. It is also not clear how this assent is different from assent made possible by any lesser degree of objectivity (i.e., under fewer transformations).

After all of this, Nozick finally comes around to defining "subjective facts." We are told, "Why are some facts subjective? They are in our heads/minds/brains. We have de facto privileged access to them."[34] This is somewhat confusing in the context in which it is stated, as Nozick is not in general clear about the way in which "subjectivity" is opposed to "objectivity." Prior to this the difference appears to be relative, based on the "level" or "degree" of the invariance in question, but in the passage above it is stated as an absolute. If the relativistic formulation does not make a distinction between subjective and objective, but rather the absolute criterion, then everything Nozick has previously said seems more about *degrees* of objectivity or subjectivity than the distinction between the two. The *real* criterion of objectivity is that the fact is not "in" a head/mind/brain, and ostensibly this has little to do with invariance.

Furthermore, the very claim that subjective facts are "in" our heads/minds/brains is difficult to understand, as it runs against a standard intuition about "facts" — namely that, whatever they may be, facts are not the sorts of things that can be "in" anything or located anywhere. Facts, whatever they are, may certainly be *about* something or pertain

[33] [Nozick, 2001], p. 91
[34] [Nozick, 2001], p. 92.

to it — they display intensionality. So, it can make sense to say of a fact that it is "about" a head/mind/brain. One may also have a thought which is *of* or *about* a certain fact. One may believe or disbelieve, or love or hate, particular facts. There can also be facts about facts (we are discussing some of them). This is just to say that facts are also fit to play the role of intensional content, and as such it might make sense to say that a thought *of* a fact is "in" a head/mind/brain, because thoughts certainly are the sort of thing that can have a location (as long as it is in some head/mind/brain or other). Finally, a given fact is generally taken to be associated with (or identical to) a state of affairs which is what the fact is about. Thus, we can say of the state of affairs or any part of it, that the fact *depends* on it (or possibly is constituted by it). If, for example, the fact of concern is that Linda loves John, that fact depends, among other things, on a state of affairs involving Linda's head/mind/brain, and at least part of this state of affairs might be said to be "in" her head/mind/brain. None of these ways of thinking about facts, however, supports the idea that a fact itself might be "in" a head/mind/brain. A fact can be the subject of a thought which occurs in a head/mind/brain, it could be a fact about a particular head/mind/brain, or it may depend in some way on a state of affairs involving a head/mind/brain, but since facts themselves have no location, they cannot be simply "in" our heads/minds/brains.[35]

This suggests that Nozick must not mean us to take the idea that subjective facts are "in our heads/minds/brains" literally. A compelling clue to what he may have in mind is the statement that subjects have "privileged" access to these facts. Of the three possibilities just considered, only one involves facts which sometimes are accessible only to a given subject. This is the idea that a fact can be associated with a head/mind/brain state of affairs. If we think that some of these states of affairs are only accessible via subjective introspection, then not only may facts depend on these states, but only the owner of the

[35]A given fact may even involve all three, if, for example, the fact is that a certain head/mind/brain contains a thought about its own current state, which is that it is thinking of its own current state. Such a fact clearly is a fact about the head/mind/brain which is engaged in such introspection. It also depends on the head/mind/brain state of affairs that corresponds to the thinking involved. Finally, the head/mind/brain dependent fact is itself the object of the intensional state of thinking.

head/mind/brain (the subject) has access to these facts, since only the subject can determine which states-of-affairs obtain. On this view, though the fact may not be located *in* the head/mind/brain, the fact depends on at least one head/mind/brain state of affairs to which the subject has privileged access.

The virtue of such a position is that it is consistent with the intuitive idea of "subjectivity," which has historically been tied to the idea of privileged access. It is furthermore consistent with one of the more interesting passages of the text that is otherwise difficult to fit into the broader account.

> If, according to Nietzschean perspectivism, there were some constraints upon perspectives, and there were some things that are invariant across all the admissible perspectives, it is these things that would posses the greatest objectiveness. What is invariant under transformations of observers from humans to crickets is more objective psychologically than the features applying only to human observations, and than the features applying to cricket observations. Even if the two organisms divide up the world differently, some relations between human categories might map onto corresponding relations among the different cricket categories, and this isomorphic psychological structure would be more objective than what varies between the organisms.[36]

Those facts to which a subject has privileged access will be by definition inaccessible to other subjects. Thus, these will fail to be invariant under transformations between perspectives. Facts which are accessible across perspectives will be *less* subjective, and therefore presumably *more* objective.

Lending even further credibility to the claim that Nozick has epistemic access in mind is a subtle shift in the discussion from intersubjective assent or agreement to intersubjective access (which occurs, in part, in the passage above). Unfortunately, Nozick is never explicit about the relationship between the two, but given what he says elsewhere it seems reasonable to think that intersubjective assent is *evidence* for intersubjective access (without which, intersubjective assent would be astoundingly unlikely). Intersubjective

[36][Nozick, 2001], p. 93. Emphasis added.

access is itself understood in terms of invariance over epistemic or psychological perspectives.

The final point of interest for us is what Nozick has to say about objective beliefs. He tells us,

> An objective belief is one that is arrived at by a process in which biasing factors that tend to lead one away from the truth play no role.... [T]he [objective] belief tends to be invariant under a change in biasing factors, under a transformation in what desires, emotions, etc., the person possesses.... The judgments that result from the operation of subjective factors, many of which tend to be biasing, are not invariant under transformations in these factors, and hence lack objectiveness.[37]

Nozick takes this to be an "additional notion of objectivity," and thus distinct from the notion of an objective fact. However, it is fairly clear that the connection between the two is supposed to be invariance. Although Nozick himself says little else about this connection, there is an intuitive sense in which the kind of invariance described above is closely related to the notion of "independence" which is supposed to be a mark of objectivity (but about which Nozick says nothing explicit).

Before embarking on his discussion of the philosophy of science, Nozick offers a summary statement, which is important as it differs significantly from much of what Nozick has previously said. Nozick writes, "What is objective about something, I have claimed, is what is invariant from different angles, across different perspectives, under different transformations." This statement is odd and confusing for several reasons. At no point does Nozick explicitly discuss invariance with respect to either angles or perspectives (though we have tried to connect the two on his behalf), nor has he at any point elevated these to be equal to the more general account of objectivity *as* invariance rather than invariance of a particular type. Instead, we are told that the fact that objectivity just is invariance is supposed to explain these features.

[37] [Nozick, 2001], p. 96.

The confusion is made more robust in that Nozick seems to be of two minds regarding the fundamental relationship between objectivity and invariance. First, he says "An objective fact is invariant under various transformations." This is the formulation that looks most like our general theory of formality. Later, of course, Nozick modifies this to the claim that objectivity consists in invariance under "admissible" transformations. Despite the initial appearance that this limits the view, Nozick's subsequent discussion of "degrees of objectivity" and objectivity in different domains effectively means that objectivity is associated with *any* invariance. At the very least Nozick is unclear about which of these he intends or thinks most likely, instead making liberal use of speculative language.

The most serious problems for Nozick's view, however, concern the nature of the fundamental relationship he posits between objectivity and invariance. One of these we have already mentioned — Nozick never provides a distinction between objectivity and subjectivity explicitly in terms of invariance. Instead he suggests repeatedly that objectivity comes in degrees, based on the scope of invariance. This threatens to reduce the invariance account to merely an account of degrees of objectivity, rather than of objectivity itself. At the very least the matter is unclear.

Even as an account of degrees of objectivity, however, the view comes up deficient. While we do acknowledge an intuitive sense in which objectivity comes in degrees (as we explain below), the way in which Nozick formulates this is counter-intuitive. In particular, the type of the transformations involved does not matter on his view. So, for example, topology comes out "more objective" than Euclidean geometry simply on account of a broader range of invariance. Certainly there is a difference between the two, but we would never have been inclined to cite objectivity as part of this difference (for our part, we would cite formality instead). Without further evidence or explanation, the degrees of objectivity Nozick comes up with seem more of an artifact of tying objectivity to invariance *simpliciter* than a compelling reason to accept that objectivity is tied to invariance in this way.

Even if Nozick rejects the idea of degrees of objectivity, the other option he puts

forward is that what is objective is that which is the *most* invariant. But he provides no justification for thinking, for example, that facts invariant under Lorentz transformations are objective whereas the fact that the earth revolves around the sun is not. From an intuitive perspective these *both* seem objective. If Nozick wishes to claim something otherwise, he owes an explanation of how this is so. If he is being revisionary, then he owes us an explanation of how thinking of objectivity in this way is advantageous. The presentation of the view is so problematic that it is difficult to see what such a justification would be.

Despite its problems, Nozick's account of objectivity is not without intuitive purchase. First, the three "marks of objectivity" with which he begins seem both important to objectivity and to involve invariance in some manner. Second, it seems entirely plausible that invariance may play an important role with respect to science and scientific knowledge, and perhaps with respect to knowledge more generally. We explore the first of these below, and we take up James Woodward's compelling account of the second in the following section. We differ significantly from Nozick in that we do not see the two points to be deeply connected, other than that they each involve invariance, and hence formality. However, we do not think that we stray too far from the spirit of Nozick's work, and we think that the result has an interesting and fruitful relationship to the GTOF. In particular, on our account objectivity itself exhibits a kind of formality we might expect — it captures structural features about the relationships between subjects and objects, methods of observing and observations, and biases and belief formation.

The first mark of objectivity we will consider is the idea that objective facts permit intersubjective assent. Nozick reasons well, in our opinion, when he concludes that facts are not objective because they are assented to, but that they are assented to because of some other feature related to objectivity, which Nozick identifies as invariance. Nozick is never clear about what kind of invariance is required, but presumably he means invariance of the Lorentz transformation type. This step is doubtful for the same reasons we doubt the main view — just how invariance under Lorentz transformations enables intersubjective assent any more than many other kinds of invariance is totally unclear.

However, when Nozick shifts his discussion to the topic of intersubjective *access*, he provides the foundation for what holds promise as a characterization of the particular kind of invariance that is significant for objectivity. Let us imagine that each of us is the sole existent, perceiving being in a universe. Are there inaccessible facts? Possibly. But if there are inaccessible facts in such a universe they are not so due to any dependence on states of affairs in another mind. By fiat, there are no other minds. Any facts which are inaccessible are so for other reasons (which would likely make them inaccessible to every subject in the actual world as well). In such a world, the line between "objective" and "subjective" is effectively blurred owing to the presence of only a single subject.[38]

Now, let us introduce another human mind into this universe. This immediately changes the situation since there are now states of affairs (dependent on the other subject's mind) to which we do not ourselves have access. If we wish to communicate and get along with this other subject, we will be most successful if we stick to "common ground," i.e., those states-of-affairs to which we *both* have access. We cannot, for example, ask our new neighbor if the statement, "The item I am thinking of right now is blue," is true and expect to communicate successfully. And so, even at this stage we have a division between object and subject which we can easily put in terms of an invariance principle — intersubjectively accessible facts are those which depend only on states-of-affairs whose accessibility is invariant under transformations between subjects.

Nozick is led to see such intersubjective access as merely a "mark" of objectivity on account of a certain case in quantum mechanics where observations are only accessible to the subject, but where these observations are entailed by our best current scientific theory (which we agree to be objective, if anything is). We think this is an overreaction on Nozick's part because of an important difference between the two cases. In the case of quantum mechanics the expected observations, and indeed the very fact that they are observable by

[38]We do not bring into account "possible subjects" because we are merely making an illustrative point that does not hang on whether it makes sense to consider such "subjects." In actual reasoning about whether or not a certain fact or other is objective it may well make sense to bring possible subjects or perspectives into play.

only one subject, are entailed by facts which are accessible to multiple observers. The same cannot be said for head/mind/brain dependent facts, since head/mind/brain states of affairs are not entailed by any other intersubjectively accessible facts. This is what allows us to have expectations about the results of quantum mechanical experiments, even when our expectation is that these results are only accessible to a single subject, whereas we have few if any accurate expectations about the states of affairs of other minds.

To account for this we can simply take the closure under entailment of the previously defined class. That is, intersubjectively accessible facts are those which depend only on states-of-affairs which are accessible under transformations between subjects or are entailed by such facts. It is this particular type of invariance which permits intersubjective assent, since having access to the same facts (or their grounding states-of-affairs) is an important prerequisite for such assent (although unfortunately it cannot guarantee it will occur).

Nozick glosses the idea of "accessibility from different angles" as meaning that "access to it can be repeated by the same sense (sight, touch, etc.) at different times; it can be repeated by different senses of the same observer, and also by different observers. Different laboratories can replicate the phenomenon."[39] This is, as Nozick himself points out, closely related to the issue of intersubjective access in that it involves additional "access requirements" on objectivity. These are also, we feel, best characterized in terms of invariance over transformation classes.

The first way a fact can be accessible from multiple angles is for it to be accessible at different times. This easily turns into the principle that a fact is accessible from multiple angles if access to the fact remains invariant over temporal transformations — i.e., it is accessible at more than one time. Just which transformations or how robust the invariance must be is a question we will set aside for the moment.

Nozick tells us that a fact may also be accessible from multiple angles by being accessible to different senses of the same observer. While it is simple enough to say that a

[39][Nozick, 2001], p. 76.

fact is accessible from multiple angles if access to it remains invariant under transformations between sense modalities of the same observer, something more should be said given that sense modalities differ from one another in significant ways. That is, the case is not the same as intersubjective accessibility, where it is plausible to assume that other subjects access facts in a similar manner. If an individual sees, for example, a hot stove burner and then proceeds to feel it, it is not a straightforward thing that by touch she senses invariantly what she sensed when she saw the hot burner. What presumably goes on is that by each sense the observer is able to infer the same state of affairs — that the burner is hot. Of course, such inferences are routinely implicit in all but the crudest scientific observations, as much of science concerns phenomena which are inaccessible to the unaided senses. In our invariance principle, then, we should understand "access" in terms of inferred states of affairs, not that by different senses the observer receives precisely the same information.

Finally, an objective fact may be accessible from different angles by being accessible by different observers (including being replicable by different laboratories). But, of course, we already explored this idea in the form of intersubjective access, so we may simply say that a fact may be accessible from different angles by being intersubjectively accessible, which itself is grounded in invariance.

The final mark of objectivity, independence, involves essentially two desiderata. The first of these is that an independent fact holds independently of the mental state of any given (or all) subjects. The second is that it holds independently of any subjects' "observations or measurements." The first of these is easier to deal with, and Nozick himself provides the answer when he discusses objectivity of belief. Nozick calls a belief objective when the process that formed the belief is invariant with respect to transformations between different biases. We can simply add to this transformations between all mental states of each subject (that is, for each subject, transformations between its mental states). For the second desideratum, it is tempting to interpret independence from "observations and measurements" in terms of the actions of subjects. However, subjects' actions routinely and regularly change facts, and these do not therefore cease to be objective. What

is crucial for the actions of observation and measurement is that these are actions which *access* facts. What Nozick is suggesting is that if a subject accesses a fact, but such access causes a change in the very fact accessed (or its associated state of affairs), then this fact is not independent. In terms of invariance, we can say that an independent fact is invariant over transformations between pre-access and post-access states. That is, the class of transformations maps every pre-access state of affairs to its corresponding post-access state of affairs, and an independent fact (or its corresponding state of affairs) is one which remains invariant over these transformations. Regardless of what else may differ between the pre and post states, it is not the fact itself.

Thus, all of Nozick's intuitive "marks" of objectiveness can be naturally characterized in terms of invariance. Furthermore, we find that all of these marks (and their sub-marks) are related to one another and many of them can be united in a single notion we call a "perspective." We define a "perspective" to be an ordered quadruple consisting of a subject, a time, a mode of access (of the subject), and a mental state (of the subject). We can then use the class of perspectives to understand objectivity in terms of invariance. The issue of invariance under access (measurement and observation) is best kept distinct because of the special character of the associated class of transformations. It could in principle also be combined, but at the cost of considerable complication.

With this framework in hand, we can recover much of what Nozick had in mind for objectivity. First, we can say that it seems plausible that objectivity should have something to do with invariance under transformations between perspectives, since perspectives are constructed out of the sorts of things that intuitively confound objectivity — things which are closely tied to subjects.

We can also recover a more plausible sense in which objectivity admits of degrees, for facts may differ with respect to the perspective transformations over which they are invariant. These differences can occur along multiple dimensions, as the associated transformation classes will characterize a complete range of invariance for each component of the perspective. Clearly, if a fact is invariant over only a singleton transformation from one

perspective to itself, this fact will be "subjective" (if anything at all). If, on the other hand, a fact should happen to be invariant under transformations between all perspectives, we should say that such a fact is certainly objective, perhaps even "most" or "robustly" objective. There is considerable space between these extremes, and just what to say for certain cases may be complicated. If, for example, a fact turns out to be invariant under transformations between all perspectives for a single subject, what should we say? Probably that the fact is subjective. Other perspective components, e.g. mode of access, may not weigh quite so heavily in objectivity determinations. It could even be the case that "objectivity" and "subjectivity" form a continuum, much as many "vague" predicate pairs do, such that while there are certainly clear cases to be found, there are a vast number of facts in the "penumbra" between "objective" and "subjective" about which we are uncertain how to judge.

Although we find this idea fascinating, going further down this road would be an impropriety on our part. Not only would doing so be beyond the present scope, but also, despite a basic affinity for the view, we do not pretend to be experts on objectivity and do not wish to appear overly committed to this highly speculative position. Our purpose was only to show how we can make good sense of much of what Nozick says within the framework of the GTOF. Nothing we have to say about formality stands or falls based on what we have said here about objectivity.

That said, we should address the concern we had at the very beginning regarding the intersection of objectivity and formality. On the view we have just presented, the distinction between what is objective and what is subjective *is* a formal distinction. Or, rather, we have defined objectivity in terms of the formal characteristics of our intersubjective experience — the systematic relations between our subjective experiences which produce a framework we use to communicate and interact with one another and the world around us. This is not at all unlike what Alan Richardson takes to be an underlying motivation of Carnap's *Der Logische Aufbau der Welt*, that our ability to communicate and interact with one

another and the world is founded on formal features of our intersubjective experience.[40] This is not intended to be an ontological claim or commitment to idealism. It is rather founded on the basic notion of access to facts (by means of our senses) and characterizes the structural, formal similarities between our experiences (perspectives) as that which is objective.

Regarding Nozick's claim that objectivity *just is* invariance, Nozick has failed to provide convincing evidence for the position and the view is not independently compelling. This does not mean, however, that invariance itself may not play a significant role in science. We might see Nozick in this respect as concerned with those features of particular objects and the world which we might call most "fundamental." This is evident in his discussion of the philosophy of science, where he espouses the view that new laws which explain and subsume old laws are more "objective." Subsumption, though, is understood in terms of the older laws being shown to be instances of the new subsuming law, and from our perspective this can be understood in terms of invariance (e.g., the new law is invariant under transformations between all laws which it subsumes, and as a consequence all subsumed laws will satisfy the subsuming law). It is also natural to think of this in terms of the subsuming law capturing some common structural component of the subsumed laws. In being a more common structure, the subsuming law can easily be thought of as more basic or fundamental, in addition to being more general. Furthermore, as the law captures more broadly shared structural features, it is compelling to call the subsuming law *more formal* — it captures more general structures.

This is just one way of thinking of the role of invariance in science, pressed into the framework of scientific laws with which Nozick works. Fortunately, James Woodward has put forward a very interesting and more rigorous account of the role of invariance in scientific explanation, which we now take up.

[40] See [Richardson, 1997]. A major difference, of course, is that for Carnap "formal" means "logical," and this is a position we staunchly reject.

5.2.3 James Woodward and Causality as Invariance

We now take up a compelling account of the role of invariance in science and scientific explanation, put forward in [Woodward, 2003]. James Woodward is addressing what he sees as a number of problems with contemporary accounts in the philosophy of science centering on the notions of causality and explanation. The most significant problem with the dominant tradition of scientific explanation, which holds that such explanations must appeal to causal laws, is that it excludes many generalizations and causal principles employed in the special sciences from being explanatory. This is because these generalizations very often do not satisfy the relatively strict criteria for causal law-hood appealed to. Rather than pursuing a modification of what counts as a causal "law," Woodward presents an alternative account of causality which affords to these non-law-like generalizations causal status, and hence a role in scientific explanation. In what follows, we focus mainly on the content of Woodward's view and its relationship to the GTOF, since we have time to review neither the problems to which he is responding nor how he resolves them.

The reason Woodward's account is of interest to us is that, in his own words, "[I]nvariance is the key feature a relationship must possess if it is to count as causal or explanatory."[41] Similarly to Nozick's claim about objectivity, this appears as wide-ranging as our claim that invariance is constitutive of formality. Woodward takes care, however, to qualify that he does not mean by this that *everything* which is invariant in *any* way is causal or explanatory (which is good, since such a view would face considerable difficulty). At the same time Woodward employs a notion of "invariance" which is similar to our own yet in some ways much broader. These two distinctions from our own position warrant some additional explanation.

First, as we have explained it, the standard understanding of invariance holds that it is absolute and exceptionless. If even a single mapping alters a feature, that feature is not considered invariant. Woodward, however, "count[s] a generalization as invariant or sta-

[41] [Woodward, 2003], p. 239.

ble across certain changes if it holds up to some appropriate level of approximation across those changes."[42] This is clearly an expansion of the standard notion, yet Woodward provides little to explain how it might work apart from a few simple examples. It seems to us, however, that there are at least two ways in which a feature might deviate from standard invariance and yet be considered "approximately" invariant. The first way would be if a feature or relationship (Woodward's primary concern is invariant *relationships*) fails to hold precisely, but does hold under a certain degree of "error" or another well-defined factor which allows the relationship to be more permissive. Now, of course, a huge class of relationships could be made invariant in this way by simply increasing their permissiveness (just as any measuring apparatus can be said to be "accurate" to within a sufficiently large margin of error). This possibility is balanced by the fact that permissiveness will have a considerable negative impact on how useful a causal relationship is in scientific explanation. All other things being equal, a relationship which is less permissively invariant is to be preferred.

The second deviation from standard invariance occurs when the relationship permits of exceptions. As a non-scientific example, consider the generalization that "All birds have wings." This generalization expresses a relationship between the property of being winged and that of being a bird. The invariance which underlies the claim is that the property of wingedness is invariant under transformations between birds (or bird species). This relation fails of standard invariance, however, as Kiwis are wingless birds. Nevertheless, the generalization *is* invariant with the *exception* of Kiwis. Yet again, any generalization can be made approximately invariant in this way by allowing a sufficiently large number of exceptions (potentially everything). It is even more clear, though, that in this case the number of exceptions permitted will have a negative impact on the explanatory power of any resulting generalization. By definition such a generalization explains nothing about anything exceptional to it. All other things being equal, an invariant relationship permitting fewer exceptions is to be preferred.

[42] [Woodward, 2003], p. 239

Woodward's expanded notion of invariance encompasses all such cases, and this raises an immediate question not faced by standard invariance. How much deviation is too much? Is there a threshold degree of strictness a generalization must have to count as causal? Woodward provides no answer and it seems difficult to answer the question in general. Every candidate generalization occurs in its own scientific, explanatory context, and this context will determine much about the status of a generalization. In a new discipline, where there are few established general principles, perhaps more permissive generalizations will be viewed as legitimate (at least until stricter rules can be found). Very mature disciplines, e.g., physics, will likely be more strict, perhaps ruling against anything that is not strictly invariant.[43]

The second major difference between Woodward's account of causation and our account of formality is that Woodward is only concerned with a certain type of class of transformations. That is, unlike Nozick, who was concerned with what is invariant under the broadest class of transformations, Woodward is concerned with all classes of transformations that are of a certain type. Specifically, Woodward only concerns himself with invariance under transformations of background conditions (variables not explicitly involved in the relationship), interventions on variables explicitly involved in the relationship, and non-interventional transformations on variables explicitly involved in the relationship. In our opinion, this limitation is one of the primary virtues of Woodward's theory.

Although the details of these types of transformations do not figure into our present analysis, one thing that may not be obvious is whether interventions really are a type of transformation as we understand them, or if Woodward's notion of invariance is even more dramatically different from our own. Interventions, whatever else they involve, result in a change of the value of an explicit independent variable involved in the generalization. In the context of scientific explanation, what Woodward calls a "variable" is a means of

[43] Although even in this case, recent developments in quantum mechanics (indeed, the very advent of quantum mechanics itself) have caused a great deal of consternation about how strict a generalization must be to be explanatory. Of particular interest in this debate (among other, more esoteric issues) is the explanatory status of so-called statistical laws.

denoting a range of possible states of affairs (which are usually conceptually united and are almost always mutually exclusive). An intervention, from the perspective of invariance, is a change from one state of affairs to another, and since these changes can be characterized functionally (they are essentially ordered pairs), interventions really are transformations in the sense we have defined. Being "invariant under interventions," then, just means being invariant under a class of transformations, all of which are interventions.[44]

The distinctions between background conditions, interventions, etc. do not matter for our purpose. What matters is that Woodward has not identified a particular class of transformations as essential to causality or explanation. Instead, he has told us which class of transformations to look for in the context of a particular generalization. Thus, Woodward restricts his position to the family of transformation classes defined relative to a particular generalization (and therefore also relative to a scientific discipline).

This relativization of what underlies a causal generalization (i.e., invariance under a particular class of interventions) is seen by Woodward to be the chief virtue of his view. What it allows is for some generalizations to count as causal, and hence have an explanatory role, but within a potentially very limited scope. At the same time it permits for generalizations with much broader scope, so broad, in fact, that some of these will count as causal laws according to standard criteria. Woodward adds that "other things being equal, relationships that are more invariant (and hence more useful for purposes of manipulation

[44]Woodward puts forward a few additional points that are worth noting. First, he indicates that mere invariance under *some* interventions does not make for a convincing explanatory generalization. This is especially evident in the case where the generalization permits of a certain degree of error. Such a generalization is by fiat invariant under interventions which only change variables within this degree of error, and it is straightforwardly explanatorily useless. What is needed is for the generalization to be invariant under interventions which are significant enough to result in a change in dependent variables and for this change to be predicted by the generalization itself.

Second, Woodward holds (quite reasonably) that a generalization need not be invariant over *all* transformations in background conditions. This would defeat his goal in proposing a less strict account of explanatory generalizations since only scientific laws would satisfy such a restriction (if that). He notes, however, that the explanatory value of a generalization is influenced by its sensitivity to transformations in background conditions. All things being equal, a generalization which is invariant under a broader range of transformations of background conditions is to be preferred.

and control) provide better explanations." [45] So, he is still able to provide causal laws with a special place in scientific explanation. They are the ideal explanatory generalizations. They are not, however, the only explanatory generalizations, which is consistent with what actually occurs in most scientific disciplines.

Woodward elaborates on his view in a few ways, but the only one of interest to us is that he also recognizes that invariance comes in degrees. And while he does take steps toward incorporating the idea into his view, the result is comparatively limited in its depth. He only discusses the fact that generalizations may vary in how broadly invariant they are in terms of relations between their associated transformation classes. However, with the relativization of transformation classes to particular generalizations, there is no guarantee that the transformation classes associated with two generalizations will be commensurable in this way (or even at all). They may be entirely disjoint or overlap without one being a subclass of the other.

Despite this oversight, it does seem intuitively reasonable to say that a generalization could have more "explanatory depth," as Woodward puts it, than another. Degrees of invariance could be said to partially explain these kinds of judgments, in cases where one class of transformations is a proper superset of another. However, it seems clear that some judgments of this kind are not grounded in degrees of invariance. For example, there seems to be no doubt that the theory of general relativity has more explanatory depth than a generalization linking the size of fox litters to the availability of food. Yet both are causal and explanatory and neither explains anything the other does. The two claims share only a slight overlap in (mostly trivial) transformations, and since the generalizations share no variables in common, none of the shared transformations are interventions.

So, while what Woodward says about the impact of degrees of invariance on explanatory depth is reasonable and makes sense, it is far from a complete story. One thing Woodward never comments on, for example, is the relationship between generalizations with different degrees of invariance in terms of causal status. We are told that more broadly

[45] [Woodward, 2003], p. 243.

invariant generalizations provide better explanations, but we are never told whether they differ in terms of causality. Is one more fundamental? Real? Simple? While it is certainly possible that Woodward wants to grant all generalizations equal causal status, it seems doubtful that a more broadly invariant generalization is preferable merely from an explanatory standpoint.

Such is the role that invariance plays in Woodward's account of causality and scientific explanation. For the most part, we find the view plausible and very much in the spirit of the GTOF. More than anything else, Woodward's work raises some interesting questions for us.

First, what is the relationship between Woodward's theory of causality and the GTOF? Woodward's view does not fall cleanly under it, since the notion of invariance Woodward invokes is significantly broader than the one at the heart of formality. The most that can be said is that there may be some overlap. Insofar as there are generalizations which are invariant under an appropriate *unpermissive* transformation class *without* exception, then the general theory of formality certainly proclaims that such generalizations are "formal," as they would satisfy the general theory of formality.

Assuming Woodward is right, is this a result we should expect? We believe so, for such generalizations have much in common with more familiar formal things (other than that they are invariant). Take for example a first order system having a finite domain of objects, each of which is named. If we take an arbitrary sentence in the associated language containing a single universal quantifier, we can use this "master" sentence as a pattern to construct a set of sentences such that for each object in the domain we have a sentence identical to the quantified sentence in question but with the object's name in place of the variable and the quantifier symbol removed. Each such sentence may be thought of as an *instance* of the single master sentence.

What is the relationship between the set of instance sentences and the master sentence? The instances are effectively *contracted* to the master sentence by means of the quantifier. The quantifier enables the master sentence to express semantically (with re-

spect to the domain) everything that is captured by the collection of instance sentences. The master sentence itself expresses the fact that, with respect to the particular domain, the unique features of any particular object will have no effect on the truth (or meaning) of the sentence. This is the intuition logicians have sought to express in terms of invariance, since features which are insensitive to identity are invariant under transformations which do not preserve identity. Likewise, the master sentence can be used "predictively" to infer statements about particular individuals.

A similar effect holds true for scientific generalizations which exhibit strict invariance. They have the effect of contracting a huge number of individual statements into a single expression without any significant semantic loss. Of course, a major difference is that the subject domain is generally much more specific than what is used for logic, and the resulting generalizations are not generally assumed to have the usual characteristics attributed to logic (*a prioricity*, necessity, etc.). Still, what the generalization asserts is that, *with respect to the particular domain in question*, the identities of objects (or particular states of affairs, in the case of interventions) make no difference to the truth of the generalization.

Thus, there is intuitive ground for thinking that strictly invariant scientific generalizations have some degree of formality, and also that they have a lesser degree of formality than logical truths. This is just what our general theory of formality predicts, provided that the class of transformations over which the generalization is invariant is a subset of the class of transformations under which first order logic is invariant. This forces the domain of the generalization to be a subdomain of the logical domain, which is arguably always the case for scientific generalizations (what doubts there are mostly concern quantum mechanical phenomena).

If we are correct about the relationship between formal languages and their semantics, as characterized in Chapter 4, further compelling evidence for the formality of this class of scientific generalizations comes from the fact that they can serve, and often do, as foundations for formalized languages. Much of modern physics is carried out in a spe-

cialized, formalized language (borrowing much from mathematics), and the semantics of this formalized language is composed of the same invariances which are characterized by generalizations invariant under interventions.

What we can take from all of this is that Woodward's account of causal/explanatory generalizations in terms of what he calls "invariance" has only a relatively small overlap with our general theory of formality. This overlap concerns those generalizations which are truly invariant, in our stricter sense, and we have provided some reasons for thinking that it is appropriate to call these generalizations "formal."

Woodward's broader understanding of invariance, however, is an idea worth taking seriously in itself. We might, for example, wonder if our general theory of formality could or should be extended to include this broader notion as well. Doing so is certainly technically possible, and it is not intuitively implausible since there is nothing essential to formality that says what is formal must be without exception or permissiveness. Alternatively, we might distinguish some notion of "semi-formal" or even yet a further scale along which formal features might vary. Any of these options would add additional depth to the notion of formality. Not only might all of Woodward's causal generalizations count as formal, but it would make it possible to identify formal features in probabilistic settings, where samples cannot be expected to exhibit true invariance.

Whether there are compelling reasons for doing this, however, is beyond the scope of the present inquiry. Is it desirable for all causal generalizations to be formal? Do we want or need to appeal to formality in probabilistic contexts? Much hinges on the role (if any) of formality in the sciences and whether or not the broader formulation of formality (or a related notion of "semi-formality") is useful in any way.

The question of whether formality plays an important role in the sciences, and if so what that role is, is therefore a fundamental and important question. If it turns out that formality is just an incidental feature of some (or even all) causal generalizations, not much would hang on the choice between strict and loose invariance. It may well be that formality only plays a significant role in the so-called "non-empirical" sciences (e.g.,

logic, mathematics, geometry). But even if it is significant throughout the sciences, is it always significant in the same way?

In what we have accomplished so far, we feel we have presented some good reasons for thinking that formality is foundational to the "non-empirical" sciences. We even think we have pushed this a bit further with our work on formality and programming languages. We furthermore offer some speculation in the conclusion (section 6.2) regarding possible explorations in other sciences. We do not feel, however, that any such broad conclusions can be drawn from the present work. Indeed, even providing conclusive, comprehensive answers to our comparatively limited theses would require many volumes. Broader questions of this sort would require significantly more.

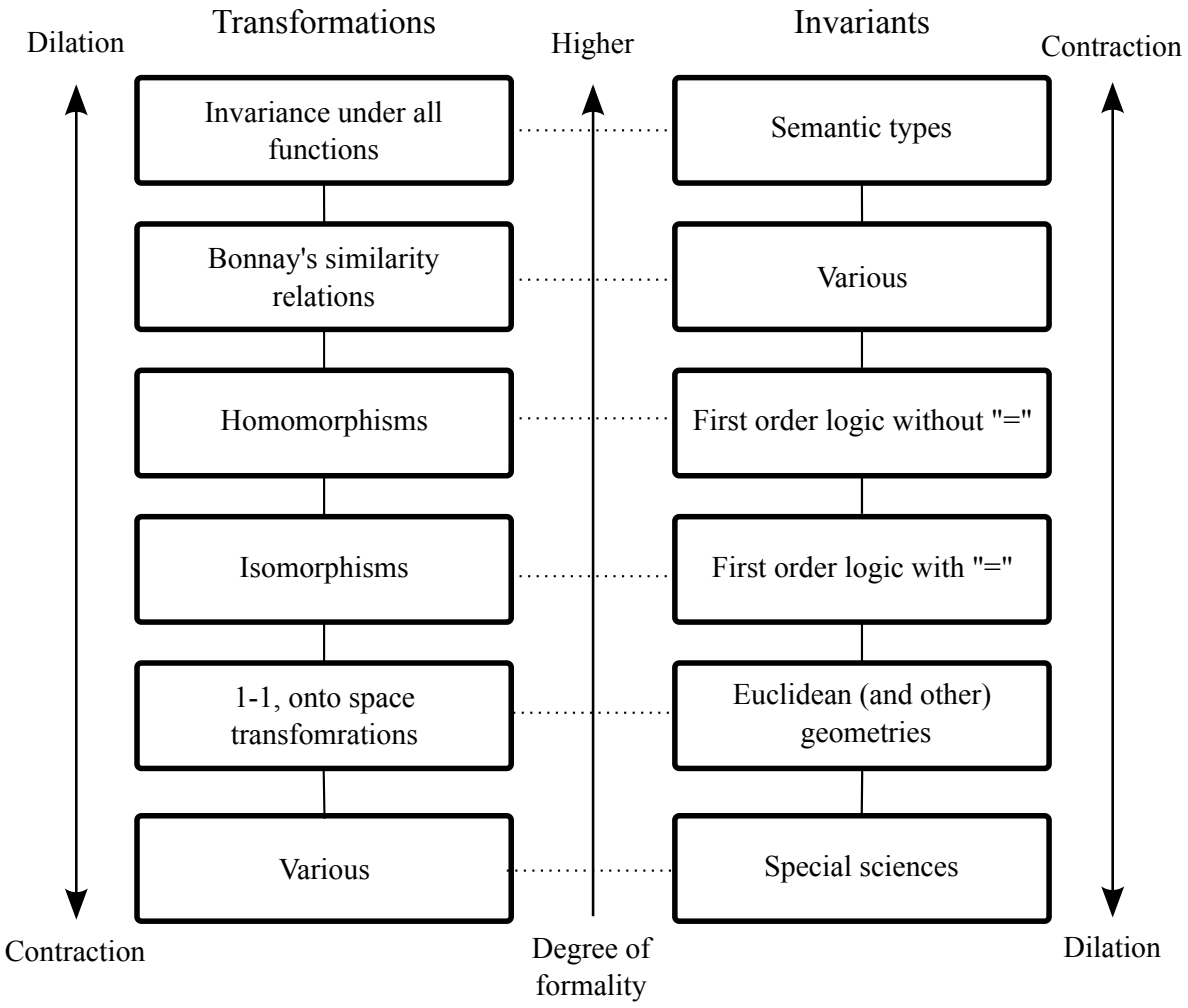

Figure 5.1: Formality hierarchy including first order logic.

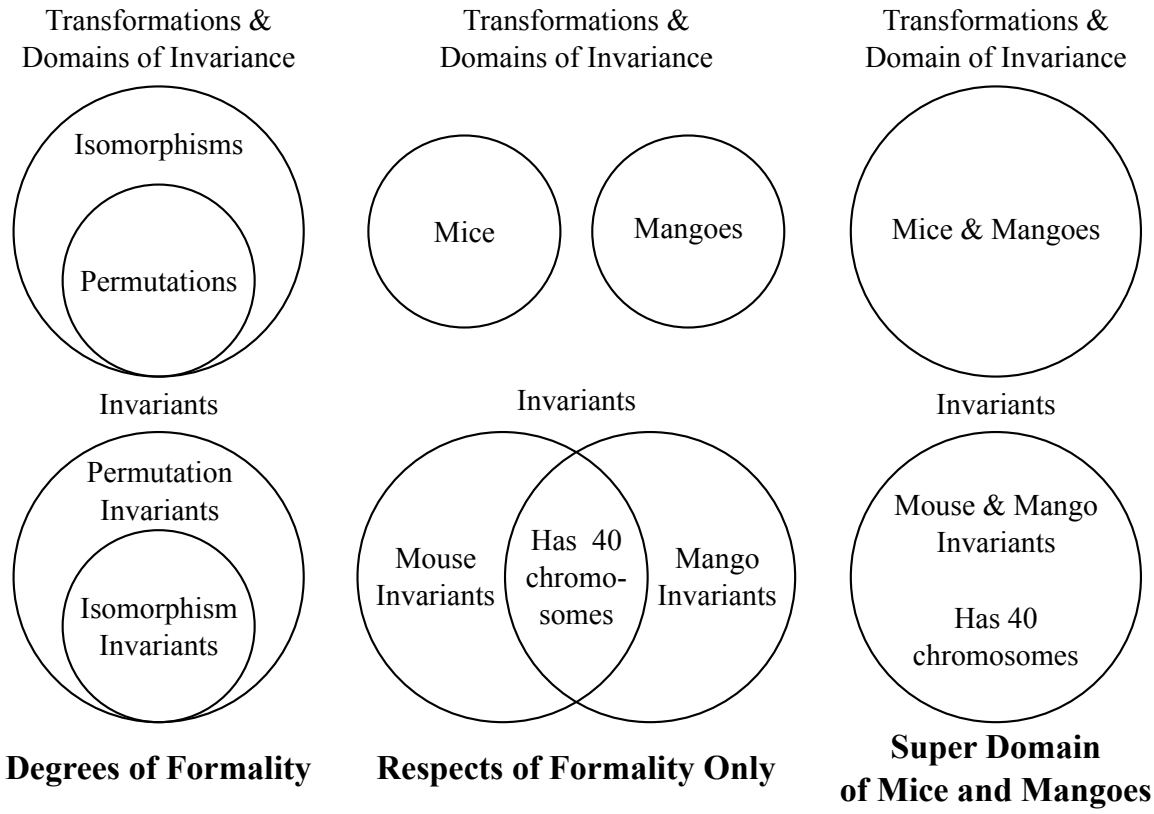

Figure 5.2: Respects of formality.

Chapter 6

Conclusion

The goal that we set in Chapter 1 was not to conclusively prove that the general theory of formality we have presented, what we have called the GTOF, is the correct theory of formality. Neither did we set out to capture precisely the meaning of the terms "formal" and "formality" as used by any or every language community. Instead our purpose was to show, first, that the GTOF works well for central and important cases of formality. This was the purpose of our examinations of geometry and first order logic. Our second goal was to show the promise the GTOF has for application beyond geometry and logic, and to this end we undertook an examination of various programming languages. These two goals stem from a desire to establish, within the limits of practicality, the *adequacy* of the GTOF as a general theory of formality. The third purpose was to establish, again within practical limits, the *utility* of the GTOF. Although results to this effect can be found in a number of places, the most notable result is that concerning the nature of the relationship between the syntax and semantics of formal languages — not that there is such a relationship, which is obvious, but that the relationship is founded on correlated classes of invariants.

Our reason for proceeding in this manner follows from the type of theory we hold the GTOF to be. That is, it is *not* a theory about what people now or in the past have meant by "form" and "formality," but rather a proposal for what we should mean when we speak

of formality *in general* (e.g., in abstract or interdisciplinary contexts). We find no fault with any current uses of "formality" or associated terms, nor do we advocate eliminating them from any discourse where they do not result in confusion or miscommunication. As a proposed definition then, and one for which there is as yet no contemporary competitor, there is little direct and clear evidence against which to evaluate the GTOF and its consequences. The most relevant criteria for accepting such a proposal is that it is adequate and useful. By "adequacy" we mean that the GTOF satisfies pre-existing intuitions about what counts as formal, and by "utility" we mean that the theory leads to additional insights, resolves known anomalies, etc. Both of these qualities come in degrees, and while an ideal theory would exhibit both perfectly, in reality acceptable theories often display trade-offs between the two. For example, a theory which is highly revisionary may yet be accepted on account of being extremely useful.

Unfortunately, practical limits have prevented us from providing a full account of either the adequacy or utility of the GTOF, but we do feel that *enough* of both have been established to warrant further interest and continued research. But before looking to the future, we should take a moment to review the key points so far established.

6.1 Summary of Results

6.1.1 Adequacy Results

In Chapter 2 we presented arguments to the effect that the adequacy of the GTOF gains intuitive purchase from its ancestral association with Klein's *erlangen programme*. The idea there was that, if anything has form, geometrical shapes do. We also suggested that it makes sense from an intuitive standpoint to broaden this idea to include also the shapes of physical objects. For simplicity we focused primarily on Euclidean geometry, and on account of the similarities between the GTOF and Klein's theory, showing it to be formal was straightforward. Furthermore, the success of Klein's own work suggests that

application to many other geometrical theories will be likewise straightforward.

In Chapter 3 we turned to what, aside from perhaps geometry, is arguably the theory most closely associated with "formality" — logic. We focused explicitly on standard first order logic, but many of our conclusions have broader implications. In our analysis we considered the two most prominent accounts of the "formality" of logic. The older of these is that logic is formal in virtue of its syntax and systematic proof theory — that is, on account of certain *linguistic* characteristics. The second and more contemporary position holds that the formality of logic arises from its insensitivity to the particular identity of objects — that is, on account of the semantics or *subject matter* of first order logic. The later view, which we named "objectual formality," was easily shown to be an instance of formality according to the GTOF, again on account of the role of Klein's work in both theories. Our analysis of the older position, however, found that "syntax" and "syntactic structure" have not always been characterized in a single way, and we were able to isolate three different ways of thinking about the "syntactic formality" of first order logic. Our first significant result with respect to the adequacy of the GTOF is that all three of these — what we called "grammatical formality," "sentential formality," and "derivational formality" — are themselves types of formality according to the GTOF. On account of these results, together with the objectual formality of the underlying semantics, we concluded that logic is not just formal according to the GTOF, but formal in a number of respects which are central the utility of logic itself.

One feature of the GTOF that we have brought up a few times is that it distinguishes logicality from formality. That is to say, in not building logicality into the definition of formality, the GTOF permits there to be a distinction between the two. As discussed in Chapter 5 one consequence of this is that it becomes difficult to hold that logic is distinctively or characteristically formal (although in section 5.2.1 we suggested how logic might yet be *specially* formal), which opens the possibility that the results of our application of the GTOF to logic might be directly applicable elsewhere. It was with this in mind that we undertook our analysis of programming languages and methodologies in Chapter 4.

What we found in Chapter 4 was that, to a considerable extent, much of what holds true of logic also holds true of programming languages and methods. It is almost trivial to show that programming languages exhibit syntactic formality (although they are not usually equipped with derivational systems). Our main focus was therefore on semantics and whether or not the semantics of particular languages are also formal in the same way as that of first order logic. What we found was that this is certainly so, but that the situation is somewhat more complex than for first order logic. To begin with, in every case the associated domain (or domains) of invariance are far more specific and limited than that of first order logic. While the semantics of first order logic involves a domain of model-theoretic structures, the domains we found associated with programming languages are much more humble — memory addresses, physical hardware, etc. Furthermore, although there are certainly relationships between the languages we have considered, and even functional equivalence between most of them, the underlying semantic domains of invariance differ between languages. Finally, whereas the characteristic constants and grammatical structures of first order logic all share a single semantic domain of invariance (i.e., they are all invariant under transformations between model-theoretic structures), many of the languages we have looked at include distinct characteristic constants and grammatical structures tied to distinct semantic domains. That is, they involve at least some notions which are invariant in multiple respects. We identified two broad categories into which these fall: semantics oriented around machine independence and semantics oriented around classes of problems. The first of these is primary, and involves the trend over time of languages to include only those characteristic constants which denote operations that could be expected of every machine of a certain class (and thus are independent from any particular machine in that class). The upper limit of this, which most modern languages realize, results in a language being Turing complete — having the expressive power to represent all computable (partial-recursive) functions.

The second semantic domain comes into play when languages include syntactic elements (sometimes called “syntactic sugar”) which do not contribute to machine inde-

pendence. Although strictly unnecessary, these features usually make a language easier to work with, and almost always so with respect to a certain class of problems. In particular, these additional syntactic features tend to denote features which are invariant over problems of a given type or class.

The semantics of programming languages, especially higher level languages, consequently come out as formal according to the GTOF. We furthermore saw a case where the intuitive degree of formality or formalization of languages (e.g., machine code vs. FORTRAN) turns out to be nicely tracked by the rigorous account implicit in the GTOF and explored more fully in Chapter 5. Machine code is less formal than FORTRAN in having constants which denote machine resources which are invariant across a narrower range of machines than FORTRAN (it is less machine independent).

Attempting to show the adequacy of any theory requires responding to its weaknesses and perceived counter-intuitive results. We have attempted to do so in Chapter 5, although given that the GTOF is a new theory, there may well be additional anomalies we are unable to see. To this end we considered the consequence of the GTOF that everything might be formal. Our response was that the GTOF is fundamentally relativistic, and thus there is nearly always a distinction between the formal and non-formal relative to a given frame. The only case where this fails is for a frame whose class of transformations contains only the identity transformation (or, in fact, is empty). In such a case, *every* feature comes out formal, and none are non-formal. Our response in this case was to argue that, while we accept the result, we should not take the case seriously since it is degenerate. From the perspective of the GTOF, formality is only interesting in the presence of *variance*, but the identity function (or the empty class) effectively impose *no variance* on the domain. Therefore the result, though technically true, should be dismissed as an uninteresting degenerate case.

The other main problem to which we responded in Chapter 5 was that the GTOF, by drawing a line between formality and logicality, robs logic of any sense of being *distinctively* formal. Our response there, apart from accepting the conclusion (toward which

we feel no animosity), was to show a number of ways in which logic could still be thought of as *specially* formal, covering a range of positions on the nature of logic.

While we certainly do not take these results to conclusively show that the GTOF has a high enough degree of adequacy (given its utility) to gain broad acceptance, we do feel that a good foundation toward this end has been laid, and that the most immediate sources of doubt have been addressed.

6.1.2 Utility Results

Not only did our inquiry in Chapter 3 result in the conclusion that first order logic exhibits formality in multiple respects, but it also led to the conclusion that this is not accidental. The formality exhibited in the semantics of first order logic imposes various constraints on the syntactic and derivational formality involved, such that there is a correspondence between the invariants on either side. It is nothing new to claim that there is a relationship between syntax and semantics, obviously. The novelty of the result is that the relationship occurs between the syntactic invariants — the logical constants and grammatical rules — and the semantic invariants — the logical notions. We reasoned that this correspondence is not accidental first on the grounds that the syntax and derivational rules of the language are constrained by what cannot be described as other than semantic considerations — meaningfulness and truth preservation. The requirement that well-formed expressions be meaningful effectively requires that the characteristic, logical constants denote semantic invariants. The requirement that derivational rules be truth preserving requires that they correspond to families of logical consequence relations, which are themselves semantic in nature.

A further reason for thinking that the correspondence between syntactic and semantic invariants is not accidental, however, is that it is precisely this relationship that permits the language of standard first order logic to be used as a tool for exploring semantic notions like logical consequence. The relationship permits the language to play the role

of a transparent medium for reasoning about and exploring the underlying subject matter (whatever that may be).

Again drawing upon the fact that the GTOF permits us to distinguish logicality from formality, we hypothesized that the same structure would hold for other, non-logical (or not strictly logical) formal languages as well. Exploring this claim was our second motive in our examination of programming languages in Chapter 4, and the conclusion was that the result does seem to generalize. That is, first, that there is a correspondence between syntactic and semantic invariants. Second, that the syntaxes of the languages are clearly set up with semantic constraints in mind. And finally, that the languages so structured function as transparent media for manipulating the function of computers and (in the case of object-oriented methods and functional programming) serve as models of "real-world" systems.

Again, we do not take ourselves to have established with any certainty that these useful consequences, together with the adequacy results described above, compel acceptance of the GTOF. We do hope that they establish that, in addition to being potentially adequate, the theory is interesting and fruitful.

6.2 Continuing Research

What we have thus far established are merely the most foundational and nearest in reach, but they are far from the most interesting and exciting aspects of the GTOF.

6.2.1 Adequacy Research

As we have already discussed, theoretical adequacy comes in degrees and it is not entirely out of the question for an acceptable theory to result in revisions to the way we might otherwise think about the world. This is perhaps more likely for theories like the GTOF, for which adequacy is primarily a matter of coincidence with antecedent intuitions,

as intuitions are generally easier to violate than other, more direct kinds of evidence. Even so, as we have already mentioned, some cases are more important than others. If the GTOF failed to rule that religious rituals and tuxedos are formal, that would clearly be more acceptable than if it fails to count logic as formal.[1] Continued research on the adequacy of the GTOF should therefore focus first on central, untested cases and gradually increase in scope.

The first obvious area of investigation is to expand the range of application beyond that of standard first order logic to include other logical languages. Of particular interest are intensional languages, especially modal and tense logics. These are significant in that they include operators which depart from the extensional (model-theoretic) foundations of first order logic. Furthermore, at least in the case of modal logic, the proof theories are much better understood than the semantics of the languages themselves. There are serious questions regarding the semantics of the modal operators and how these change with the addition or subtraction of various axioms. It seems possible that the GTOF might be able to shed greater light on what constraints these languages really impose on the semantics (i.e., the range of possible semantics for the modal operators of a given language) and how these operators relate to one another.

Another potentially fruitful line of thought concerns intuitionist (and other non-classical) logics, first to better understand their content or range of application, and second to characterize their interrelationships to other logical theories. One of the most interesting traditions is that based on the proof-theoretic semantics of Gentzen, Prawitz, and Dummet. In effect, this tradition sees logical derivation as a language itself, with basic rules (introductions, eliminations, etc.) as atomic elements and a grammar organized around validity. One of the tantalizing aspects of this approach is that its semantics is purportedly grounded

[1] We are speaking a bit loosely here, in that we have already accepted the consequence that everything is formal in some respect. How then, could the GTOF possibly fail? What we really mean is that the GTOF holds that the definition, topic, or concept involved involves formality (invariance) in an important or essential way. Logical truths are invariant to transformations between just about everything (even possible worlds), but it is not in virtue of this that the GTOF is considered adequate. What we showed in Chapter 3 was that logic is constructed on a formal foundation.

on informal, intuitive inference, rather than model theory. It is a very interesting question what the formal characteristics of such a system are and what its relationship might be to other logical systems. Such results have potential bearing on open philosophical questions concerning the nature of logic and which logical theory is generally "correct" or most appropriate for given purposes.

A final logical focus is higher order logics and their interpretations, again to learn more about their semantics and relations to one another. This bears on the controversial status of higher-order logics, the debate about which we feel often lacks a clear characterization of the assumed underlying semantics.

It would also be prudent to carefully consider the relationship between the GTOF and other potentially related concepts, looking for any conflicts or enlightenment that may so arise. Concepts of particular interest include types and tokens, concepts and instances, genera and species, and "matter" or "content".

The next area of focus is mathematics, more broadly construed than just geometry. Not only is mathematics as synonymous with "formality" as logic, but there are some important gaps in the work we have already undertaken that such an analysis would fill. For example, our analysis of the semantics of first order logic relied on a domain consisting of model-theoretic structures. These structures are themselves essentially part of, or defined by, set theory. Although nothing in our reasoning about logic requires that set theory (and its associated languages) should be formal, showing it to be so would certainly bolster that analysis. It would also be yet further justification for thinking that logic is specially formal. Our analysis of programming languages at many points also appealed to arithmetic notions, and though we did some informal work to suggest that these are themselves invariants, that work is far from rigorous or comprehensive. A complete analysis would facilitate a clearer specification of the relationships involved. Fortunately, invariance-based frameworks already exist for mathematical and algebraic analysis, and it is likely that these will rather easily turn out to be instances of the GTOF.

Of the empirical sciences, certainly physics is seen as the most formal, and is there-

fore of considerable interest from our perspective. One promising characteristic of current physical theories, other than their inclusion of large fragments of mathematical theory, is that they tend to focus on what are called "symmetries." Symmetries are interesting to us because they are themselves invariants, albeit of a much more specific type than we have characterized. Nevertheless, there is good reason to expect that many aspects of physics will turn out to be formal according to the GTOF, and also that an analysis of physics in terms of the GTOF will shed light on the complicated relationship between physical and mathematical theory.

The possibility of extending the GTOF's reach beyond mathematics and physics to, for example, chemistry or biology, is very exciting as these disciplines also involve formalisms, and sometimes even formal languages, to a considerable degree. Doing so, however, poses some difficulties with respect to the very sorts of generalizations of concern to Woodward.[2] That is, chemistry and biology (and most other sciences) make liberal use of non-law-like regularities, which as we outlined in Chapter 5, are not strictly formal according to the GTOF. We said there that whether or not it is worthwhile to modify or expand the GTOF to include less strict notions of invariance hinges on the usefulness of the resulting theory, which itself hinges on the overall role of formality in the sciences. Investigating invariance and "semi-invariance" in chemistry and biology may be one way of beginning to answer these questions.

If it turns out to be promising to open the GTOF up to less strict notions of invariance, a large number of fields become open to application of the GTOF. For example, one of particular interest to us is the formalized languages used for musical notation. Others include the study of various artistic "forms" (literary, musical, visual, etc.). But again, whether or not such analysis will be useful or enlightening is not something we can reliably forecast.

[2] See section 5.2.3 and [Woodward, 2003].

6.2.2 Utility Research

Some of the most exciting potential applications of the GTOF fall under the broad heading of "usefulness" — they are applications beyond merely accounting for what is generally taken to be "formal."

Perhaps the most promising new line of research stems from the successful extension of our hypothesis regarding the relation between syntax and semantics to languages beyond logic. This success suggests that it might well be possible to broaden the result to encompass *all* interpreted formal languages. As such, the GTOF could significantly improve our theory and understanding of the semantics of formal languages generally. Furthermore, the characteristic relationship between syntax and semantics could serve as a criterion for the successful interpretation (and reinterpretation) of formal languages, as well as a guide for the construction of new formal languages for specific purposes and domains.

A further consequence of the GTOF that has promise for being particularly useful is that its relational consequences, as discussed in Chapter 5, permit rigorous and interesting investigations into the relations between various formal theories in much the same manner as Klein's program permitted within geometry and mathematics. The extreme generality and flexibility of the GTOF does not require that every such formal theory be related to every other in a grand hierarchy, but rather permits more fragmented and complex possibilities (which there are good reasons to believe persist outside of mathematics). We have suggested some possibilities for this in the preceding section, but if we modify the GTOF with a weaker sense of invariance, as discussed in section 5.2.3, many possibilities beyond formal systems and theories would be open to analyses which may prove quite fruitful.

These future applications are more-or-less directly suggested by the work we have completed. There are, however, some less obvious, and significantly more ambitious, possibilities that we believe are worth looking into. We will discuss two of these here, one of which is historical in nature, and the other concerning the semantics of natural language.

The standard story about logic is that it was invented by Aristotle. Certainly there is some truth to this in that much of early logic made use of what Aristotle accomplished in the *Organon*. However, recent research has cast considerable doubt on the idea that a logical theory, a theory of what follows from what, is the purpose and focus of Aristotle's "logical" work. The matter is made more interesting in that Aristotle, following on Plato, made much of "form" without making a clear connection with his thoughts on syllogism and demonstration. There are, therefore, two related research projects of interest with respect to Aristotle. The first concerns the question of the purpose behind Aristotle's work in the *Organnon*. Since what Aristotle is engaged in looks much like a project of formalization, it stands to reason that the GTOF might be an enlightening perspective from which to view Aristotle's work, especially as it does not presuppose that formalization is essentially logical. Second, like us, Aristotle seems to have a view of form that is distinct from logic, mathematics, or geometry. In some ways it even superficially resembles the GTOF. It would be a worthwhile inquiry to gain as clear a picture as possible of what Aristotle's view of formality is, and what its relation may be, both technically and historically, to the GTOF.

The second ambitious research project concerns the semantics of natural languages. We have already mentioned that our results concerning the relationship between the syntactic and semantic invariants of formal languages could serve as a guide to the construction of new general purpose and domain specific formal languages. In addition to this applied investigation, the result also suggests a particularly intriguing possibility. At the end of Chapter 3, where we first suggested that the relationship might apply more broadly than just to logic, we mentioned that it bears a certain similarity to the semantic theory of Montague, who claimed that the semantics of natural language could be worked out in extensional (first order logic) terms. The similarity is primarily that both we and Montague hypothesize that a certain type of relationship will hold between a language and its semantics. We differ, of course, not only in our restriction to formal languages, but also in that we do not restrict the semantics to any particular domain, extensional or otherwise. Mon-

tague's work does, though, suggest the possibility of conducting the same sort of analysis for fragments of natural language that we have engaged in for formal languages. It does not seem implausible to think that the semantics of natural language might be grounded in some way on invariants, for example invariants under transformations between experiences (or experiences of certain sorts).

One reason for thinking that this might be the case is that some fragments of English (to take one language) exhibit inferential characteristics similar to logic, and which are sometimes even confused with logical inferences. Color language as applied to objects seems to have these characteristics. If, for instance, an object is entirely red, we can infer (vagueness not withstanding) that it is not blue, yellow, purple, or any other color. That no thing may be entirely two colors simultaneously is sometimes mistaken for a logical truth, or conversely the idea that something might be entirely both red and blue at the same time is taken to be "illogical." It is neither. Since first order logic is not sensitive to the identities of predicates, $Pa \wedge Qa$ cannot be logically false.

Where then does this intuitive principle of inference come from? We think the possibility that it may stem from an underlying experiential invariance to be an interesting and promising line of thought. In fact, such a pursuit yet again brings to mind what Richardson takes to be the guiding intuition of Carnap's *Aufbau*, to which we are sympathetic. The idea is that, despite that we do not have direct access to the minds of others and that our experiences (for all we know) might be very different from one another (e.g., inverted spectra), *something* must enable us to coordinate our activities and, more importantly, our communicative use of language. Carnap held (according to Richardson) that this intersubjective coordinating characteristic was formal in nature (i.e., formal characteristics of experience). Like all positivists, however, Carnap held that "formal" just meant "logical" and his view consequently had much in common with Montague. We have already discussed in the context of Nozick's account of objectivity how we can think of invariance across perspectives and individuals. And although Carnap rejected "objectivity," his quest for intersubjective features — features *common to all experience* — is similar enough to

be thought of in terms of invariance. Thus, by distinguishing formality from logicality, we open up the possibility of revisiting this compelling intuition with considerably more freedom than either Carnap or Montague allowed themselves.[3]

6.3 Closing Remarks

Our purpose in presenting such ambitious research projects is not to unduly aggrandize the GTOF, since of course none of these have been shown and many of them may yet fail. Instead we want to give an idea of the breadth of possibility for such a fundamentally simple theory. It is, we feel, one of the chief virtues of the GTOF that, though very simple, its extreme flexibility gives it the potential to be very informative. In this respect it breaks the general rule that simple theories do not, or cannot, say very much about the world. Strictly speaking, of course, the GTOF in itself *does not* say much about the world — it is just a definition of formality. It is in the *application* of the GTOF — as a guide and framework for the discovery and analysis of types of formality and their interrelations, particularly with respect to the relationship between languages and their semantics — that it has considerable potential. At the very least we hope that the GTOF inspires additional inquiry into the distinction between logicality and formality, the conflation of which, in our opinion, has led many legitimate and interesting ideas into obscurity. Then again, without an association between logicality and formality, and especially without those who associated logicality with invariance in the beginning, our general theory of formality would not exist.

[3] If such a position on the semantics of natural language could be worked out, it would almost undoubtedly need to make use of some less stringent notion of invariance.

Bibliography

[Backus et al., 1956] Backus, J., Beeber, R., Best, S., Goldberg, R., Herrick, H., Hughes, R., Mitchell, L., Nelson, R., Nutt, R., Sayre, D., Sheridan, P., Stern, H., and Ziller, I. (1956). *The FORTRAN Automatic Coding System for the IBM 704 EDPM: Programmer's Reference Manual*. Applied Sciences Division and Programming Research Department, International Business Machines Corporation.

[Bacon, 2000] Bacon, F. (2000). *The Advancement of Learning*. Number 4 in The Oxford Francis Bacon. Clarendon Press, Oxford.

[Barendregt, 1981] Barendregt, H. (1981). *The lamdba calculus: its syntax and semantics*. Elsevier North-Holland, New York.

[Bonnay, 2008] Bonnay, D. (2008). Logicality and invariance. *Bulletin of Symbolic Logic*, 14:29–68.

[Brading and Castellini, 2003] Brading, K. and Castellini, E., editors (2003). *Symmetries in Physics: Philosophical Reflections*. Cambridge University Press.

[Carnap, 1937] Carnap, R. (1937). *The Logical Syntax of Language*. Routledge.

[Carnap, 1963] Carnap, R. (1963). Intellectual autobiography. In Schlipp, P. A., editor, *The Philosophy of Rudolf Carnap*, pages 3–86. Opent Court Press, La Salle, IL.

[Dummett, 1991] Dummett, M. (1991). *The Logical Basis of Metaphysics*. Duckworth, London.

[Etchemendy, 1990] Etchemendy, J. (1990). *The Concept of Logical Consequence*. Harvard University Press, Cambridge, MA.

[Feferman, 1999] Feferman, S. (1999). Logic, logics, and logicism. *Notre Dame Journal of Formal Logic*, 49:31–54.

[Frege, 1960] Frege, G. (1960). *The Foundations of Arithmetic: A Logico-Mathematical Inquiry into the Concept of Number*. Harper & Brothers, New York, 2nd revised edition.

[Frege, 1967] Frege, G. (1967). Begriffschrift, a formula language, modeled upon that of arithmetic, for pure thought. In van Heijenoort, J., editor, *From Frege to GÃűdel: A Source Book in Methematical Logic, 1879-1931*, pages 1–82. Harvard University Press.

[Gentzen, 1964] Gentzen, G. (1964). Investigations into logical deduction. *American Philosophical Quarterly*, 1(4):288–206.

[Goldstine, 1972] Goldstine, H. (1972). *The computer - from Pascal to von Neumann*. Princeton University Press, Princeton, NJ.

[Hindley and Seldin, 2008] Hindley, J. and Seldin, J. (2008). *Lambda-calculus and combinators, an introduction*. Cambridge University Press, Cambridge, UK.

[Keisler, 1970] Keisler, H. J. (1970). Logic with the quantifier 'there exist uncountably many'. *Annals of Mathematical Logic*, 1:1–93.

[Klein, 1872] Klein, F. (1872). A comparative review of recent researches in geometry. *Bulletin of the New York Mathematical Society*, 2:215–249.

[Lindenbaum and Tarski, 1983] Lindenbaum, A. and Tarski, A. (1983). On the limitations of the means of expression of deductive theories. In Corcoran, J., editor, *Logic, Semantics, Metamathematics*, pages 384–392. Hackett, Indianapolis, 2 edition.

[Lindström, 1966a] Lindström, P. (1966a). First order predicate logic with generalized quantifiers. *Theoria*, 32:186–195.

[Lindström, 1966b] Lindström, P. (1966b). On relations between structures. *Theoria*, 32:172–185.

[MacFarlane, 2000] MacFarlane, J. (2000). *What does it mean to say that logic is formal?* PhD thesis, University of Pittsburgh.

[Mautner, 1946] Mautner, F. I. (1946). An extension of klein's erlanger program: Logic as invariant-theory. *American Journal of Mathematics*, 68:345–384.

[McGee, 1996] McGee, V. (1996). Logical operations. *Journal of Philosophical Logic*, 26:567–580.

[Mitchell, 1957] Mitchell, G. E. (1957). *The FORTRAN Automatic Coding System for the IBM 704 EDPM: Programmer's Primer*. International Business Machines Corporation.

[Montague, 1974] Montague, R. (1974). *Formal philosophy: selected papers of Richard Montague*. Yale University Press.

[Mostowski, 1957] Mostowski, A. (1957). On a generalization of quantifiers. *Fundamenta Mathematicae*, 44:12–36.

[Novaes, 2010] Novaes, C. D. (2010). Reassessing logical hylomorphism and the demarcation of logical constants. *Synthese*, pages 1–24.

[Nozick, 2001] Nozick, R. (2001). *INVARIANCES: The Structure of the Objective World*. Harvard University Press, Cambridge, MA.

[Prawitz, 1971] Prawitz, D. (1971). Ideas and results in proof theory. In *Proceedings of the Second Scandinavian Logic Symposium: Studies in Logic and the Foundations of Mathematics*, volume 63, pages 235–307.

[Quine, 1960a] Quine, W. (1960a). Variables explained away. In *Proceedings of the American Philosophical Society*, volume 104, pages 343–347.

[Quine, 1960b] Quine, W. (1960b). *Word and Object*. MIT Press.

[Quine, 1976a] Quine, W. (1976a). Algebraic logic and predicate functors. In *The Ways of Paradox*, pages 283–307. Harvard University Press, Cambridge, MA.

[Quine, 1976b] Quine, W. (1976b). The variable. In *The Ways of Paradox*, pages 272–282. Harvard University Press, Cambridge, MA.

[Quine, 1986] Quine, W. (1986). *Philosophy of Logic*. Harvard University Press, second edition.

[Quine, 1981] Quine, W. V. O. (1981). *Theories and Things*. Harvard University Press.

[Richardson, 1997] Richardson, A. (1997). *Carnap's Construction of the World*. Cambridge University Press, Cambridge, UK.

[Sammet, 1969] Sammet, J. (1969). *Programming Languages: History and Fundamentals*. Prentice Hall, Englewood Cliffs, NJ.

[Sher, 1991] Sher, G. (1991). *The Bounds of Logic*. MIT Press, Cambridge, MA.

[Sher, 1996] Sher, G. (1996). Did tarski commit tarski's fallacy? *Journal of Symbolic Logic*, 61:653–686.

[Sher, 1999] Sher, G. (1999). Is logic a theory of the obvious? *The Nature of Logic, the European Review of Philosophy*, 4:207–238.

[Sher, 2001] Sher, G. (2001). The formal-structural view of logical consequence. *The Philosophical Review*, 110(2):241–261.

[Sher, 2008] Sher, G. (2008). Tarski's thesis. In Patterson, D., editor, *New Essays on Tarski and Philosophy*, pages 300–339. Oxford University Press.

[Strawson, 1959] Strawson, P. F. (1959). *Individuals*. Methuen & Co. Ltd., London.

[Tarski, 1941] Tarski, A. (1941). *Introduction to Logic and to the Methodology of Deductive Sciences*. Oxford University Press, New York.

[Tarski, 1983a] Tarski, A. (1983a). The concept of truth in formalized languages. In Corcoran, J., editor, *Logic, Semantics, Metamathematics*, pages 152–278. Hackett, Indianapolis, 2 edition.

[Tarski, 1983b] Tarski, A. (1983b). The establishment of scientific semantics. In Corcoran, J., editor, *Logic, Semantics, Metamathematics*, pages 401–409. Hackett, Indianapolis, 2 edition.

[Tarski, 1983c] Tarski, A. (1983c). On the concept of logical consequence. In Corcoran, J., editor, *Logic, Semantics, Metamathematics*, pages 409–20. Hackett, Indianapolis, 2 edition.

[Tarski, 1986] Tarski, A. (1986). What are logical notions? *History and Philosophy of Logic*, 7:143–154.

[Tarski and Givant, 1987] Tarski, A. and Givant, S. (1987). *A formalization of set theory without variables*. American Mathematical Society, Providence, RI.

[Turing, 1936] Turing, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 43, pages 230–265.

[von Neumann, 1945] von Neumann, J. (1945). First draft of a report on the edvac. In Taub, A., editor, *John von Neumann: Collected Works*. Pergamon Press, New York.

[Wilkes et al., 1951] Wilkes, M., Wheeler, D., and Gill, S. (1951). *The Preparation of Programs for an Electronic Digital Computer, with special reference to the EDSAC and the use of a library of subroutines*. Addison-Wesley Press, Cambridge, MA.

[Woodward, 2003] Woodward, J. (2003). *Making Things Happen: A Theory of Causal Explanation*. Oxford University Press, Oxford.