Talk About Types

Isaac Wilhelm

(please ask before citing/circulating)

Abstract

Many metaphysical theories—of identity, existence, and so on—are formulated using higher-order languages like the simply typed lambda calculus. But as I argue, for the purposes of metaphysical theorizing, a different language would be better: the calculus of constructions. This language allows for quantification over types. And it has many other expressive advantages over the languages currently being used in the philosophical literature. So for the purposes of metaphysical theorizing, it is the better language.

1 Introduction

Metaphysical theories are often formulated using higher-order languages (Bacon & Russell, 2019; Caie et al., 2020; Dorr, 2016; Goodman, 2017; Rayo, 2006; Williamson, 2003). For instance, when theorizing about identity and when theorizing about existence, many philosophers prefer to use a language called the 'simply typed lambda calculus'. Languages like that contain variables which belong to many different grammatical categories: singular term variables, predicate variables, sentence variables, and more. Each of those variables can be used to express a corresponding class of generalizations: singular term variables can be used to generalize over entities, predicate variables can be used to generalize over properties of entities, sentence variables can be used to generalize over propositions, and so on. So these higher-order languages, like the simply typed lambda calculus, are extremely expressive: they can be used to generalize in many different ways.

But as I argue here, these particular higher-order languages are not always expressive enough. They do not contain variables whose grammatical categories themselves can vary. In particular, these languages do not contain variables which can be used to generalize over entities *and also* properties of entities *and also* propositions, all at once. So these languages cannot express patterns that run across the grammatical hierarchy: patterns that obtain among entities, properties of entities, propositions, and so on.

Fortunately, some languages—familiar to computer scientists, but not to philosophers—do better. These languages allow for quantification over all 'types', where types are certain grammatical categories. A particularly simple example of such a language, on which I focus in this paper, is called the 'calculus of constructions'. It contains the linguistic resources to quantify into type position. It contains many other linguistic resources too. And those resources allow it to capture patterns that run through the type-theoretic hierarchy.

For example, consider the following truth about the hierarchy of types.

Everything, of every type, is self-identical. (Self)

For reasons discussed later, the simply typed lambda calculus—which lacks the vocabulary needed to quantify over types—has trouble expressing (Self). The calculus of constructions, however, has no such trouble. It can be used to formulate the sentence below.

$$\mathbb{\forall}\alpha\forall_\alpha x(x \equiv_\alpha x)$$

Later on, I discuss sentences like this in detail. But for now, simply note the following: '$\mathbb{\forall}\alpha$' says 'For all types α', '$\forall_\alpha x$' says 'For all x of type α', and '$x \equiv_\alpha x$' says 'x is self-identical'. So the above sentence, of the calculus of constructions, expresses (Self).

In order to formulate appropriately general claims about identity, existence, grounding, modality, essence, and many other tools in the metaphysician's toolkit, we need to quantify over types. We need a language which allows us to talk about all the types that there are.

Many higher-order languages, upon which philosophers have focused, do not allow for that. The calculus of constructions, however, does. So as I argue in this paper, for the purposes of metaphysical theorizing, the calculus of constructions is the better language.

My argument does not proceed by (i) proposing necessary and sufficient conditions for one language to be metaphysically better than another, and then (ii) showing that according to those conditions, the calculus of constructions is better than other languages in the literature. My argument is a bit looser, as arguments in meta-philosophy generally tend to be. Basically, I argue that when it comes to formulating metaphysical theories, the calculus of constructions does a better job—than other languages in the literature—of capturing what those theories are intended to capture. The calculus of constructions allows metaphysicians to say what they really want to say. So it is, generally, the better language.

To make this argument, I focus on two case studies; that is, two areas of metaphysical theorizing. First, I discuss theories of identity. Second, I discuss theories of what exists. In both cases, I argue, the calculus of constructions provides metaphysicians with the linguistic resources they need, in order to properly formulate their theories. And in both cases, I argue, other higher-order languages—like the simply typed lambda calculus—fall short.

In Section 2, I present the simply typed lambda calculus. In Section 3, I present the calculus of constructions. In Section 4, I argue that for the purposes of theorizing about identity, the calculus of constructions is better than the simply typed lambda calculus. In Section 5, I argue that for the purposes of theorizing about existence—in particular, theorizing about absolutely everything—the calculus of constructions is the better language too. In Section 6, I discuss a potential shortcoming of the calculus of constructions.

2 The Simply Typed Lambda Calculus

In what follows, I present an informal characterization of the simply typed lambda calculus; see Appendix 1 for more details. I also compare the simply typed lambda calculus

to first-order logic. In particular, I explain why the simply typed lambda calculus is more expressive.

Before doing that, however, it is worth explaining why I have chosen to compare my preferred language for metaphysical theorizing—the calculus of constructions—to the simply typed lambda calculus in particular. In this paper, I focus on the simply typed lambda calculus—rather than propositional logic, first-order logic, or some other such language—for three reasons. First, the simply typed lambda calculus has been used to formulate extremely sophisticated theories of identity, existence, and so on. It is quite prominent—and it is only becoming more so—in the contemporary metaphysics literature. Second, of all the languages which are that prominent, the simply typed lambda calculus is among the most expressive. So by focusing on the simply typed lambda calculus, I make my task—of identifying expressive shortcomings of prominent languages—quite hard. Third, the simply typed lambda calculus is like a simplified version of the calculus of constructions. So the simply typed lambda calculus serves as a convenient stepping stone, on the way to understanding that more complicated language.

Like first-order logic, the simply typed lambda calculus contains constants, predicates, and variables. Unlike first-order logic, however, it contains many different types of variables, beyond the specific type of variable used in first-order logic. For instance, the simply typed lambda calculus has variables with the same grammatical categories as predicates, variables with the same grammatical categories as sentences, and more.

Because of all these linguistic resources, the simply typed lambda calculus has a rather complicated grammar. Lots of machinery is required, in order to define the grammatical strings of the simply typed lambda calculus. Hence the use of what are called 'types'. In the simply typed lambda calculus, types are used to (i) classify the different bits of vocabulary, and (ii) describe which bits of vocabulary can be combined with which other bits of vocabulary, in order to create grammatical strings of symbols. So in the simply typed lambda calculus, types are grammatical categories: each grammatical string of symbols is

assigned a unique type, and that type represents the grammar of that string.

Here is a rough characterization of types. First, there is a type of propositions: this is the grammatical category of sentences. In what follows, I often represent the type of propositions by 't'. Second, there is the type of entities: this is the grammatical category of singular terms, like the variable 'x' or constant symbol 'c' of first-order logic. In what follows, I often represent the type of entities by 'e'. Third, all other types are constructed from t and e. The construction operation—call it a 'constructor'—takes any types x and y, and produces a type. In what follows, I express this constructor using the English phrase '...-to-...', where types are substituted for the ellipses.

For instance, there is a type—the 'entity-to-proposition' type, or the 'e-to-t' type—which represents the grammatical category of expressions that can be combined with one expression of entity type to form an expression of proposition type. So the entity-to-proposition type represents the grammatical category of monadic first-order predicates. For instance, if 'P' is a predicate which can be combined with a singular term like 'c' to form a sentence 'Pc', then 'P' has type entity-to-proposition.

There is also a type—the '(entity-to-proposition)-to-proposition' type—which represents the grammatical category of expressions that can be combined with one expression of entity-to-proposition type to form an expression of proposition type. So the (entity-to-proposition)-to-proposition type represents the grammatical category of monadic second-order predicates. For instance, if 'F' is a predicate which can be combined with the predicate 'P'—which has entity-to-proposition type—to form the sentence 'FP', then 'F' has type (entity-to-proposition)-to-proposition.

Now for the vocabulary of the simply typed lambda calculus. That vocabulary consists of, among other things, the following symbols. Note that in what follows, I use letters from the middle of the greek alphabet—'σ', 'τ', and so on—to represent types.

(1) There is a constant symbol '$\rightarrow$'.

(2) For each type σ, there are infinitely many constants 'a_σ', 'b_σ', and so on, and infinitely

many variables 'x_σ', 'y_σ', and so on.

(3) For each type σ, there is a quantifier symbol '$\forall_\sigma$' and an identity symbol '$\equiv_\sigma$'.

Other vocabulary—such as '$\neg$', '$\wedge$', '$\vee$', '$\leftrightarrow$', and '$\exists_\sigma$'—are defined in terms of these symbols.

Each symbol, in the above list, is called a 'term'. Roughly put, in the simply typed lambda calculus, a 'term' is a grammatical string of symbols. The above symbols are the basic terms, from which more complicated terms are constructed.

An aside: in what follows, I often refer to the 'semantic values' of particular expressions. My use of the English phrase 'semantic value' follows the usage in (Krämer, 2014, pp. 715-716) and (Linnebo, 2006, p. 154): the semantic value of an expression in a sentence is, roughly, the contribution which that expression makes to the truth value of that sentence. But just to be clear: the notion of a semantic value should not be understood in first-order terms. In particular, many semantic values are not entities, where entities are the semantic values of first-order variables. So do not be fooled by the fact that the English expression 'semantic value' is a predicate which attaches to names and definite descriptions. The semantic values of many formal expressions are not the semantic values of names or definite descriptions – that is, not entities. The candidate semantic values of predicates, for instance, are properties.

In the simply typed lambda calculus, the symbols which represent the types of terms are *not* meaningful expressions; only the terms themselves are meaningful. For in the simply typed lambda calculus, only terms—not type symbols—have semantic values. For instance, take the type symbol 'e' and the term 'a_e'. In the simply typed lambda calculus, 'a_e' is a name: so its candidate semantic values are entities. And so 'a_e' is a meaningful expression. The subscript '$_e$', in contrast, is not a term; so it is not a meaningful expression. Rather, '$_e$' is a bookkeeping device. It is used by us, in the metalanguage, to describe a certain linguistic feature of the term 'a_e': namely, that 'a_e' has the type of entities, and so 'a_e' has entities as its candidate semantic values. The subscript '$_e$' is no more meaningful than, for example, the subscript '$_2$' on the variable 'x_2' in first-order logic.

Here is another way to put the same basic point. The simply typed lambda calculus

contains terms and types. The terms are part of the object language: they are used to describe the world. The types are part of the metalanguage: they are used to describe terms in the object language. And these two languages are distinct: the symbols for types are used in the metalanguage, but not in the object language.

The simply typed lambda calculus is extremely expressive: it has infinitely many quantifier symbols and infinitely many identity symbols. To see the expressiveness which this yields, start by considering the quantifiers. Each '$\forall_\sigma$' of the simply typed lambda calculus resembles the universal quantifier '$\forall$' of first-order logic. For each '$\forall_\sigma$' can be used to generalize over the semantic values of all terms of type σ. Roughly put, '$\forall_\sigma$' can be used to say things like "Everything of type σ is such-and-such."[1] And because of that, when taken together, the '$\forall_\sigma$' of the simply typed lambda calculus are far more expressive than the '$\forall$' of first-order logic. The '$\forall$' of first-order logic can be used to make claims about all—but also *only*—entities: '$\forall$' can only be used to make claims about the semantic values of the variables 'x', 'y', and so on. But in the simply typed lambda calculus, for each σ, there is a quantifier '$\forall_\sigma$' which can be used to make claims about the semantic values of the variables 'x_σ', 'y_σ', and so on. One such quantifier—namely, '$\forall_e$'—can only be used to make claims about all entities. But another quantifier—namely, '$\forall_{e-to-t}$'—can be used to make claims about all properties of entities. And similarly for all the other quantifiers: each one can be used to generalize in a unique way.

An aside: following orthodoxy, I assume that in languages like the simply typed lambda calculus, quantification over non-entities—like properties of entities, or propositions—cannot be understood in terms of quantification over entities.[2] The contents expressed by sentences which contain quantifiers like '$\forall_{e-to-t}$', for instance, cannot be expressed by sentences con-

[1]For the sake of brevity, throughout this paper, I occasionally write in a way which attributes types to the semantic values of terms. Those passages are just convenient shorthands for more complicated passages which attribute types to terms whose semantic values are such-and-such. For instance, "Everything of type σ is such-and-such" should be read as shorthand for "Every semantic value of every term of type σ is such-and-such."

[2]For endorsements of this, see (Boolos, 1984; Krämer, 2014; Rayo, 2006; Rayo & Yablo, 2001; Williamson, 2003).

taining only the quantifier '$\forall_e$'. For whereas the candidate semantic values of variables like 'x_e' are entities, the candidate semantic values of variables like 'x_{e-to-t}' are properties. And properties are not just special sorts of entities; they are something else.[3]

Finally, take the identity symbols of the simply typed lambda calculus. Roughly put, each '$\equiv_\sigma$' can be used to say things like "This semantic value, of type σ, is identical to this semantic value of type σ." And because of that, when taken together, the '$\equiv_\sigma$' of the simply typed lambda calculus are far more expressive than the '=' of first-order logic. For '=' can only be used to make claims about the identities of entities. But in the simply typed lambda calculus, for each σ, the symbol '$\equiv_\sigma$' can be used to make claims about the identities of the semantic values of any terms of type σ. One such symbol—namely, '$\equiv_e$'—can only be used to make claims about the identities of entities. But another such symbol—namely, '$\equiv_{e-to-t}$'—can be used to make claims about the identities of properties of entities. And similarly for all the other identity symbols: each one can be used to make a unique class of identity claims.

It is worth clarifying a piece of informal terminology which I have been using: a 'claim' is any term with grammatical category t. So roughly put, claims are sentences. For instance, when I wrote that "The simply typed lambda calculus can be used to make claims about identity," I just meant that in the simply typed lambda calculus, there are terms—with grammatical category t—which feature an identity symbol like '$\equiv_\sigma$'.

3 The Calculus of Constructions

In this section, I present the calculus of constructions. To start, I explain some key structural differences between the calculus of constructions and the simply typed lambda calculus. Then I briefly summarize some grammatical categories that the calculus of con-

[3]So when I write things like "The symbol '$\forall_{e-to-t}$' can be used to make claims about all properties," I write loosely. Strictly speaking, vocabulary like '$\forall_{e-to-t}$' cannot be understood in terms of English expressions like 'for all properties': the grammatical features of those English expressions are more similar to the grammatical features of '$\forall_e$' than to the grammatical features of '$\forall_{e-to-t}$'.

structions features. After that, I describe some of the most important vocabulary in the calculus of constructions; for more details, see Appendix 2. Finally, I explain why the calculus of constructions is more expressive than the simply typed lambda calculus.

The calculus of constructions contains all the grammatical categories of the simply typed lambda calculus, and more besides. It contains both the type of propositions t and the type of entities e. But it also contains other grammatical categories. Many of these are grammatical categories *of grammatical categories*. For instance, in the calculus of constructions, there is a grammatical category of all types: t and e have that grammatical category, and all other types do too.

Because of this, in the calculus of constructions, many grammatical categories are *terms*. The type t is a term, for instance, as is the type e. They are terms because in the calculus of constructions—as in the simply typed lambda calculus—a term is just a string of symbols with some grammatical category or other. And in the calculus of constructions, many grammatical categories—like types, for instance—have grammatical categories of their own. So many grammatical categories are terms, in the calculus of constructions.

This is, perhaps, the most significant difference between the calculus of constructions and the simply typed lambda calculus. The simply typed lambda calculus does not contain vocabulary which can be used to assign grammatical categories to grammatical categories. The calculus of constructions, however, does.

Here are some details. In the calculus of constructions, two new grammatical categories—called 'sorts'—play an especially important role. One sort, represented by the symbol '$*$', is the grammatical category of all types. So for instance, the type of entities e has grammatical category $*$, as does the type of propositions t. The other sort is represented by the symbol '$\Box$': it is a grammatical category which, very roughly, corresponds to operations over types. For instance, recall the constructor '...-to-...' from the simply typed lambda calculus. The calculus of constructions contains something analogous to this constructor. And that something can be assigned a grammatical category, by using the '$\Box$' symbol.

The analog of '. . .-to-. . .', in the calculus of constructions, is pretty complex. In particular, the calculus of constructions contains a fairly complicated procedure for building grammatical categories. Basically, the procedure constructs more complex grammatical categories by using variables which range over simpler grammatical categories. Here is an example.

$$\Pi x{:}e.t$$

This is a grammatical category. As mentioned above, e is the type of entities and t is the type of propositions. The '.', in the calculus of constructions, acts like the '. . .-to-. . .' constructor of the simply typed lambda calculus. So $\Pi x{:}e.t$ is the grammatical category of expressions which combine with an expression of grammatical category e to produce an expression of grammatical category t. And so the grammatical category $\Pi x{:}e.t$, in the calculus of constructions, is the analog of the simply typed lambda calculus's e-to-t.

The symbols 'x', ':', and 'Π' play several roles. The 'x' is a variable; so like all variables, 'x' is a term. And so 'x' has a grammatical category: its particular grammatical category, in this example, is e. The string '$x{:}e$'—which is a basic expression in the calculus of constructions—encodes all that information: it says, among other things, that the grammatical category of 'x' is the type of entities. The symbol 'Π' is used to bind the 'x'. In this particular example, there is little need for binding: 'x' is basically a dummy variable in the expression '$\Pi x{:}e.t$'. But generally, 'x' must be bound because, roughly put, the procedure for building grammatical categories can generate expressions in which 'x' occurs freely on the right side of the '.'. And certain terms can be substituted for those free occurrences of 'x'. The symbol 'Π' helps capture that.

In general, for almost any grammatical categories A and B, and for any variable 'x' with grammatical category A, the following is a grammatical category in the calculus of constructions.

$$\Pi x{:}A.B$$

The string 'x:A' says that 'x' has grammatical category A. The '.' acts, as before, like the '. . .-to-. . .' of the simply typed lambda calculus. So intuitively, '$\Pi x{:}A.B$' is the grammatical category of expressions which combine with an expression of grammatical category A to produce an expression of grammatical category B.

Using these linguistic resources, the calculus of constructions blurs the line between object language and metalanguage. The basic expressions, in the calculus of constructions, take the form 'M:N' where 'M' is a term, 'N' is a grammatical category, and the grammatical category of M is N. And in two different ways, expressions of the form 'M:N' act like object language expressions and also metalanguage expressions. Let us see how, using the expression 'x:e' as an example.

First, 'x:e' expresses both object language content and metalanguage content at once. For it expresses (i) whatever 'x' expresses, and (ii) that the symbol 'x' has 'e' as its grammatical category. Note that (i) is the sort of content that object language expressions are typically used to express, while (ii) is the sort of content that metalanguage expressions are typically used to express. So in the string 'x:e', both that object language content and that metalanguage content get expressed 'at once'.

Second, in some cases, a grammatical category can appear on both the right side *and* the left side of a colon. In 'x:e', for instance, the grammatical category represented by 'e' appears on the right. But the calculus of constructions allows for other expressions, in which 'e' appears on the left. For instance, 'e:$*$' is a well-formed expression of the calculus of constructions. It expresses the fact, mentioned earlier, that e has grammatical category $*$. In 'x:e', 'e' helps to express a metalanguage fact about 'x': the fact that 'x' is a term of grammatical category 'e'. In 'e:$*$', however, 'e' helps to express something more akin to an object language fact: for in 'e:$*$', 'e' appears as a term, and so 'e' helps to express a content that object language expressions are typically used to express.

With that as background, here is some vocabulary of the calculus of constructions—some new, and some already introduced—that will be used throughout this paper.

(1) There are constant symbols '$*$' and '$\Box$' such that $*$ has grammatical category $\Box$.

(2) There are constant symbols 'e' and 't', both of which have grammatical category $*$.

(3) For all strings of symbols A, if there is a sort s such that A has grammatical category s, then there are infinitely many constant symbols 'c', 'd', and so on, each of which has grammatical category A.

(4) For all strings of symbols A, if there is a sort s such that A has grammatical category s, then there are infinitely many variables 'x', 'y', and so on, each of which has grammatical category A.

(5) For all strings of symbols A and B, if there are sorts s and s' such that A has grammatical category s and B has grammatical category s', then for all variables 'x' such that x has grammatical category A, the string '$\Pi x{:}A.B$' has grammatical category s'.

(6) There are constants '$\rightarrow$', '$\approx$', '$\equiv$', '$\mathbb{V}$', '$\mathbb{I}$', and '$\forall$'. For the grammatical categories of these constants, see Appendix 2.

Each symbol in the list above, with the exception of '$\Box$', is called a 'term'. Roughly put, in the calculus of constructions, a term is a string of symbols which has some grammatical category or other. The above symbols are the basic terms, from which more complicated terms can be built.

Before continuing, some more terminology. If A and B are terms such that A has grammatical category B and B has grammatical category $\Box$, then A is called a 'constructor' and B is called a 'kind'. So $*$ is a kind, since $\Box$ is its grammatical category. If A and B are terms such that A has grammatical category B and B has grammatical category $*$, then A is called an 's-term' and B is called a 'type'. So e is a type, and any term whose grammatical category is e counts as an s-term.

The s-terms, in the calculus of constructions, form a copy of the simply typed lambda calculus. In particular, in a precise and natural way, the simply typed lambda calculus properly embeds within the calculus of constructions (Barendregt, 1992). Each term of the former corresponds to an s-term of the latter, and each type of the former corresponds to a

type of the latter. So the calculus of constructions contains the simply typed lambda calculus within it.

Finally, a 'type variable' is a variable whose grammatical category is $*$. I call these 'type variables' because, roughly put, they range over types. Throughout this paper, I represent type variables using letters from the beginning of the Greek alphabet: 'α', 'β', and so on.

Now for the constant symbols in Condition (6). The symbol '$\rightarrow$' should be familiar. It is basically just the material conditional which appears in the simply typed lambda calculus.

The symbol '$\approx$' is unlike anything in the simply typed lambda calculus, however. It combines with two types to produce a sentence. Roughly put, '$\approx$' can be used to say things like "This type is identical to this type." For instance, if σ is a type, then '$\sigma \approx \sigma$' says that σ is identical to itself.[4]

The symbols '$\mathbb{V}$' and '$\mathbb{E}$' are quantifiers which range over types. Roughly put, '$\mathbb{V}$' can be used to say things like "All types are thus-and-so." Similarly, '$\mathbb{E}$' can be used to say things like "Some types are thus-and-so." For instance, if 'α' is a type variable and σ is a specific type, '$\mathbb{V}\alpha(\alpha \approx \alpha)$' says that every type is self-identical and '$\mathbb{E}\beta(\beta \approx \sigma)$' says that some type is identical to σ.

The constant '$\equiv$', like '$\approx$', can be used to make identity claims: '$\equiv$' combines with a type, a type, a term of the former type, and a term of the latter type, to produce a sentence. Roughly put, '$\equiv$' can be used to say things like "The semantic value of this term, of this type, is identical to the semantic value of this term of this type." For instance, if σ is a type, τ is a type, A is a term of type σ, and B is a term of type τ, then '$A \equiv_{\sigma,\tau} B$' says that A is identical to B.

Finally, consider '$\forall$'. This constant can be used to make quantificational claims about the semantic values of terms whose grammatical categories are types. It combines with a type, a variable of that type, and a term, to produce a sentence. Roughly put, '$\forall$' can be

[4]Strictly speaking, in the calculus of constructions, '$\sigma \approx \sigma$' is not a complete string. To be complete, it would have to be followed by a colon, which would have to be followed by that string's grammatical category: in particular, it should be written '$(\sigma \approx \sigma)$:t'. But for brevity, throughout this paper, I often drop the colon and the grammatical category from the strings which I write.

used to say things like "Everything of thus-and-so type is self-identical." For instance, if σ is a type and 'x' is a variable whose grammatical category is σ, then '$\forall_\sigma x(x \equiv_{\sigma,\sigma} x)$' says that everything of type σ is self-identical.

The symbols '$\equiv$' and '$\forall$' are extremely expressive pieces of vocabulary – more expressive than any symbols in the simply typed lambda calculus. The symbol '$\equiv$' can be used to make identity claims using any terms of any types. In order to do similarly, the simply typed lambda calculus requires infinitely many identity symbols: one symbol '$\equiv_\sigma$' for each type σ. Similarly, the symbol '$\forall$' can be used to generalize over all semantic values of all terms of all types. In order to do similarly, the simply typed lambda calculus requires infinitely many quantifier symbols: one symbol '$\forall_\sigma$' for each type σ. So '$\equiv$' and '$\forall$' combine the expressive powers of infinitely many identity symbols and infinitely many quantifier symbols of the simply typed lambda calculus.

There are obvious notational similarities between instances of '$\forall_\sigma$' in the calculus of constructions and instances of '$\forall_\sigma$' in the simply typed lambda calculus. But they are not the same symbol, because their semantic contents are radically different. In the simply typed lambda calculus, the subscript '$_\sigma$' in '$\forall_\sigma$' is a semantically insignificant part of the expression. That subscript is a mere bookkeeping device; like the subscript '2' on the variable 'x_2' in first-order logic. In the calculus of constructions, however, the subscript '$_\sigma$' in '$\forall_\sigma$' has semantic content. The symbol '$\forall$' is a meaningful expression on its own, the symbol 'σ' is a meaningful expression on its own, and when the two are combined—as described in Appendix 2—the resulting expression '$\forall_\sigma$' is meaningful as well. Because of this, the calculus of constructions allows for sentences which *quantify into* the subscript spot of '$\forall_\sigma$'; I give examples of this in Section 4. And so the '$\forall_\sigma$' in the calculus of constructions is a much more expressive bit of vocabulary than the '$\forall_\sigma$' in the simply typed lambda calculus.

It will be convenient to introduce a few notational shorthands. As in the simply typed lambda calculus, the calculus of constructions can be used to define the logical constants '$\neg$',

'$\wedge$', '$\vee$', '$\leftrightarrow$', and '$\exists$'. In addition, for all types σ, let '$\equiv_\sigma$' be shorthand for '$\equiv_{\sigma,\sigma}$'.[5]

4 Identity

In this section, I argue that when it comes to expressing claims about identity, the calculus of constructions is better than the simply typed lambda calculus. For there are many patterns concerning identity which the calculus of constructions can be used to describe. These patterns resist being described by the simply typed lambda calculus, however. Hence the expressive superiority of the calculus of constructions.

In particular, when it comes to expressing these identity patterns, the calculus of constructions has three advantages—call them 'expressive advantages'—over the simply typed lambda calculus. First, in order to capture these patterns, the calculus of constructions often requires just one sentence; the simply typed lambda calculus, in contrast, requires infinitely many. Second, arguably, the simply typed lambda calculus cannot fully capture these patterns at all: some patterns have features which infinite lists of sentences simply cannot describe. Third, the calculus of constructions can be used to capture patterns among the types themselves; the simply typed lambda calculus, however, cannot.

To illustrate the first two expressive advantages, consider the following 'self-identity' pattern: everything of every type is identical to itself. The calculus of constructions can be used to express the self-identity pattern with just one sentence: namely, the sentence from Section 1, reproduced below.

$$\mathbb{\forall}\alpha\forall_\alpha x(x \equiv_\alpha x) \tag{Ref}$$

No one sentence of the simply typed lambda calculus, however, expresses the self-identity pattern.[6] Any attempt to express this pattern must use infinitely many sentences: a formal

[5] I also assume the classical introduction and elimination rules for '$\forall$', '$\mathbb{\forall}$', and '$\mathbb{\exists}$'.

[6] Throughout this section, I occasionally claim that certain sentences in the simply typed lambda calculus—or certain collections of sentences—do not say the same things as certain sentences in the calculus of constructions. The relevant notion of 'saying the same thing', for my purposes here, is informal and intuitive: I just

analog of "Everything of type e is self-identical," a formal analog of "Everything of type e-to-t is self-identical," and so on.

Moreover, even this infinite list is not enough. The simply typed lambda calculus has no way of saying that the types e, e-to-t, and so on, are all the types that there are. So the simply typed lambda calculus cannot be used to express all the relevant features of the self-identity pattern. For the self-identity pattern amounts to *more* than merely that the semantic values of all terms of types e, e-to-t, and so on, are self-identical. The self-identity pattern *also* concerns the fact that these self-identities are exhaustive: once the self-identities for the semantic values of all terms of types e, e-to-t, and so on, have been specified, no more self-identities remain. And the simply typed lambda calculus cannot be used to say that.[7]

Another example illustrates a related, but distinct, expressive limitation of the simply typed lambda calculus. Consider the following 'instantiation' pattern: everything instantiates something of some type or other. The calculus of constructions can be used to express the instantiation pattern with the sentence below.

$$\mathbb{V}\alpha\forall_{\alpha}x\mathbb{I}\beta\exists_{\beta}y\big(y(x)\big) \tag{Ins}$$

When it comes to the instantiation pattern, the simply typed lambda calculus faces the two problems from before. First, any attempt to express this pattern must use infinitely many sentences: a formal analog of "Everything of type e instantiates something of type e-to-t," a formal analog of "Everything of type e-to-t instantiates something of type (e-to-t)-to-t," and so on. Second, the simply typed lambda calculus has no way of saying that the types e,

mean that in some informal and intuitive sense, the relevant sentences in the simply typed lambda calculus are not expressively equivalent to the relevant sentences in the calculus of constructions. This intuitive notion of expressive inequivalence is supported by certain model-theoretic results for both languages. For lack of space, I do not discuss that here.

[7]Here is another way to put the point: if there were more types than just e, e-to-t, and so on, then (Ref) would still capture the self-identity pattern but the infinite list would not. So (Ref), but not the infinite list, captures the self-identity pattern in worlds with more types than just e, e-to-t, and so on. And in fact, it is reasonable to think that possibly, e, e-to-t, and so on, do not exhaust the types: expressions like 'α' count as types in the calculus of constructions—because their grammatical category is $*$—but they do not correspond to any type in the simply typed lambda calculus.

e-to-t, and so on, are all the types that there are; so the infinite list does not say enough.

But there is a third problem: the infinite list of sentences says *too much*. To see why, it helps to compare the instantiation pattern to the following 'specific instantiation' pattern: for every type σ, everything of that type instantiates something *of the specific type* σ*-to-*t. Note that these two patterns are different. The specific instantiation pattern says that for every type σ, everything of type σ instantiates something of some particular type that is defined using σ: namely, the type σ-to-t. The instantiation pattern, in contrast, says something weaker: for every type σ, everything of type σ instantiates something of some type or other. The instantiation pattern does not say exactly what that other type is.

So here is why the infinite list of sentences, in the simply typed lambda calculus, says too much: at best, that list captures the specific instantiation pattern rather than the instantiation pattern. Each axiom in the list has the following form: everything of some fixed type σ instantiates something of the specific type σ-to-t. But that is the specific instantiation pattern, not the instantiation pattern. So the infinite list of sentences says too much. It captures the stronger, specific instantiation pattern, rather than the weaker instantiation pattern.

The calculus of constructions can be used to express more complicated views as well. For example, take the 'completeness' view: everything is metaphysically definable in terms of (i) the fundamental, and (ii) purely logical notions (Bacon, 2020; Sider, 2011). To see how the calculus of constructions can be used to say this, let 'P' be a predicate which—following Bacon (2020, pp. 565-566)—represents the property of being purely logical. Let 'F' be a predicate which represents the property of being fundamental. Then take the principle below.

$$\forall\alpha\forall_\alpha x\exists\!\!\!\exists\beta_1\cdots\exists\!\!\!\exists\beta_n\exists_\gamma y\exists_{\beta_1}y_1\cdots\exists_{\beta_n}y_n\big(P(y)\wedge F(y_1)\wedge\cdots\wedge F(y_n)\wedge x\equiv_\alpha y(y_1,\ldots,y_n)\big) \quad \text{(Fun)}$$

(Fun) says that for every x, there are fundamental $y_1,\ldots,y_n$ of some types or other, and

there is a purely logical operation y, such that x is identical to $y_1, \ldots, y_n$ standing in y.[8] In other words, everything is defined in terms of purely logical operations on fundamentalia.

(Fun) does a better job of capturing the intuitive content of the completeness view than any sentence of the simply typed lambda calculus. To see why, note that the simply typed lambda calculus can only be used to formulate principles like this: *for the specific types* $\sigma_1, \ldots, \sigma_n$, and for every x, there are fundamental $y_1, \ldots, y_n$—of types $\sigma_1, \ldots, \sigma_n$ respectively—and there is a purely logical operation y, such that x is identical to $y_1, \ldots, y_n$ standing in y (Bacon, 2020, pp. 566-567). This principle does not allow the types $\sigma_1, \ldots, \sigma_n$ to change from one choice of x to the next. And because of that, this principle is committed to the following: only finitely many levels, in the type-theoretic hierarchy, contain fundamentalia. In other words, the fundamentalia used to define any given x, and the fundamentalia used to define any given y, always have exactly the same types: namely, σ_1, …, σ_n.

(Fun) is not committed to that. For in (Fun), the terms 'y_1', …, 'y_n' have type variables as their grammatical categories. Those type variables range over the entire hierarchy of types. So (Fun) allows for the possibility that infinitely many levels, in the type-theoretic hierarchy, contain fundamentalia. In other words, the fundamentalia used to define any given x, and the fundamentalia used to define any given y, may have different types.

Now for the third expressive advantage of the calculus of constructions over the simply typed lambda calculus: the former, but not the latter, has the resources to express patterns among types themselves. For instance, consider the 'type identity' pattern: all types are self-identical. The calculus of constructions can be used to express this pattern, as follows.

$$\mathbb{V}\alpha(\alpha \approx \alpha) \qquad \text{(Ref}\approx\text{)}$$

The simply typed lambda calculus, however, cannot express the type identity pattern. For the simply typed lambda calculus cannot be used to formulate claims about types at all.

[8]Note that '$y(y_1, \ldots, y_n)$' is shorthand for '$\Big(\big((yy_1)y_2\big)\cdots\Big)y_n$', and '$\gamma$' is shorthand for '$\Pi z_1{:}\beta_1\Big(\Pi z_2{:}\beta_2.\big(\cdots(\Pi z_n{:}\beta_n.\alpha)\cdots\big)\Big)$'.

To summarize: because it allows for quantification over types, the calculus of constructions is significantly more expressive than the simply typed lambda calculus. Type variables can be used to capture many, many patterns among terms across the type-theoretic hierarchy. Those patterns cannot be captured using the limited resources of the simply typed lambda calculus.

So when it comes to theorizing about identity, the calculus of constructions is the better language. There are many features of identity which the simply typed lambda calculus cannot be used to express. The calculus of constructions, however, can be used to express those features. So it is better.

5 Existence

In this section, I argue that the calculus of constructions is particularly well-equipped for theorizing about absolutely everything. Vocabulary like '$\mathbb{\forall}$', '$\mathbb{\exists}$', and 'Π'—call these 'type manipulators'—are better for theorizing about absolutely everything than the vocabulary of other higher-order languages.[9] For those other languages cannot be used to achieve the kind of generality that type manipulators can.

What is there? "Everything," says Quine (1948, p. 21), and in so answering, he leaves nothing out; or at least, so say proponents of absolutely unrestricted quantification. According to the intuitive version of their view, there are quantifiers whose variables have the following feature: absolutely everything is a candidate semantic value of those variables. Call this intuitive version 'Absolute Generality'; and call those variables, whose candidate semantic values include absolutely everything, the 'absolutely general' variables. According to a standard precisification of Absolute Generality, the absolutely general variables are first-order: they are variables, that is, of type e (Cartwright, 1994; Linnebo, 2006; McGee, 2006; Williamson, 2003). Call this precisification 'Absolute Generality$_e$'.

[9] I call these 'type manipulators' because they can be used to manipulate variables in type position.

Proponents of Absolute Generality$_e$ often supplement their view with related views about specific higher-order languages. For those higher-order languages can be used to solve a number of problems which might otherwise arise for Absolute Generality$_e$. Williamson (2003) and Rayo (2006), for instance, argue that proponents of Absolute Generality$_e$ ought to countenance an infinite hierarchy of languages—of higher and higher orders—to deal with various linguistic problems.

Certain variables—associated with certain languages in these hierarchies—are special. In particular, take the 'lowest-level quantifiers': in the hierarchies discussed by Williamson (2003) and Rayo (2006), these would be the quantifiers of the lowest language in the hierarchy; in the version of the simply typed lambda calculus discussed in (Williamson, 2013), these would be the quantifiers '$\forall_e$' and '$\exists_e$' associated with variables of type e. Let the 'lowest-level variables' be the variables associated with these lowest-level quantifiers. According to proponents of Absolute Generality$_e$, those lowest-level variables are absolutely general. Nothing whatsoever is ineligible to be those variables' semantic values. Their candidate semantic values are absolutely unrestricted.

It is true, of course, that other variables in these language hierarchies—associated with other quantifiers—have different candidate semantic values.[10] And those candidate semantic values are not among the candidate semantic values of the lowest-level variables. But according to proponents of Absolute Generality$_e$, this does not undermine the claim that the lowest-level variables are absolutely general. Those lowest-level variables still range over absolutely everything.

Because of this, I think that Absolute Generality$_e$ is not the most satisfying way of capturing the intuitive idea of absolutely general quantification. There is a gap, I think, between Absolute Generality and Absolute Generality$_e$. And the calculus of constructions can be used to shrink that gap: it contains quantificational expressions which, intuitively, do

[10]For instance, in Williamson's hierarchy (2013, pp. 237-238), the candidate semantic values of variables with different types are different. Similarly, the candidate semantic values of the variables in Rayo's hierarchy of plural languages are different from one another too (2006, p. 227).

a better job than Absolute Generality$_e$ of capturing Absolute Generality.

Because of limitations in the English language, the gap between Absolute Generality and Absolute Generality$_e$ is extremely difficult to describe. The clearest descriptions of that gap, unfortunately, beg the question against proponents of Absolute Generality$_e$. As a result, Absolute Generality$_e$ is hard to criticize.

Nevertheless, I think it is worth presenting that question-begging description of the gap – doing so helps illuminate, at least roughly and approximately, why Absolute Generality$_e$ feels a bit unsatisfying. So here is that description: the lowest-level variables of Absolute Generality$_e$, which are said to be absolutely general, fail to range over some perfectly good semantic values. In particular, those lowest-level variables fail to range over the candidate semantic values of the other variables. So in some question-begging yet also intuitive sense, those lowest-level variables are not as general as one might have hoped.[11]

The calculus of constructions can be used to articulate the gap between Absolute Generality and Absolute Generality$_e$. For in the calculus of constructions, the following sentence is true.

$$\exists\!\!\!\exists\alpha\exists_\alpha x\forall_e y\neg(x \equiv_{\alpha,e} y) \tag{Abs}$$

Very roughly put, (Abs) says that something, expressed by a term of some type, is not an entity. The candidate semantic values of 'y' do *not* include absolutely everything. But for proponents of Absolute Generality$_e$, 'y' is a lowest-level variable: it is the sort of variable associated with the lowest-level quantifier '$\forall_e$'. So contrary to Absolute Generality$_e$, in some question-begging but intuitive sense, the lowest-level variables do not range over absolutely everything. Those variables range over all entities, but there is more to reality than that. And that is what (Abs) says.

Proponents of Absolute Generality$_e$ might object as follows. The type manipulators in (Abs), they might claim, are incoherent, illegitimate, and incomprehensible. If this is right, then of course, such vocabulary should not be used for formulating philosophical views about

[11]For a more technical version of the complaint which I am making, see (Krämer, 2017, pp. 512-516).

absolutely everything.

But this claim is wrong. Type manipulators are coherent, legitimate, comprehensible tools for philosophical theorizing. To see why, just note that they can be used to formulate good theories. In Section 4, I used type manipulators to formulate attractive metaphysical theories of identity: (Ref), (Ins), (Fun), and (Ref≈). In addition, type manipulators can be used to formulate attractive linguistic theories too. For instance, take conjunction. For each type σ, there is a constant term '$\wedge_\sigma$' which combines with two terms 'M' and 'N'—both of type σ—to produce the term '$M \wedge_\sigma N$'.[12] Because each '$M \wedge_\sigma N$' obeys introduction and elimination rules which are characteristic of the first-order conjunctive symbol '$\wedge$', each '$\wedge_\sigma$' is like a generalized conjunction symbol. Type manipulators can be used to theorize about all the generalized conjunctions '$\wedge_e$', '$\wedge_t$', '$\wedge_{e-to-t}$', and so on, at once. And so those type manipulators can be used to describe the common inferential roles—given by the relevant introduction and elimination rules—that those conjunction symbols share.[13]

Note that the calculus of constructions contains variables which are reasonably good candidates for being absolutely general: the variable 'x' of grammatical category α, for instance, where 'α' is a type variable. And the calculus of constructions contains complex expressions which play the role of absolutely general quantifiers too: for instance, the expression '$\forall\alpha\forall_\alpha x$'. The candidate semantic values of 'x' include every candidate semantic value for every term of every type whatsoever: entities, properties of entities, propositions, and so on.[14] So intuitively, 'x' is absolutely general.

And so for the purposes of theorizing about absolutely everything, the calculus of constructions is better than other languages discussed in the philosophical literature. For the calculus of constructions does a better job of respecting the intuitive idea of absolute generality than other higher-order languages. The variables whose grammatical categories

[12]Each '$\wedge_\sigma$' can be taken as primitive, or defined in terms of other constants.

[13]Type manipulators can be used to formulate other theories as well. For instance, they are useful in computer science: they feature in theorem-provers like Coq, which were used to establish the four-color theorem (Gonthier, 2008).

[14]The calculus of constructions manages to allow for this while also avoiding paradoxes analogous to Russell's (Girard, 1986; Mitchell, 1996).

are type variables, in particular, are better candidates for being absolutely general than any variables in the simply typed lambda calculus. And as a bonus, the calculus of constructions can be used to articulate a gap—between Absolute Generality and Absolute Generality$_e$—that is quite hard to articulate.

6 Limitations of the Calculus of Constructions

The calculus of constructions, though more expressive than many other languages, faces some important limitations of its own. After explaining these limitations, I address some concerns which one might have about them. As I ultimately argue, the limitations are linguistic, not worldly. So for the purposes of metaphysical theorizing, the limitations are not all that problematic.

The limitations basically amount to this: the calculus of constructions cannot be used to formulate generalizations across grammatical categories like kinds and sorts. For example, the calculus of constructions cannot be used to say that all terms of all kinds—that is, all terms which have a kind as their grammatical category—are self-identical. For in the calculus of constructions, no variable ranges over the semantic values of all terms of all kinds.

This might seem problematic. For it might seem to imply that the calculus of constructions faces versions of the limitations that the simply typed lambda calculus faces. After all, just as the simply typed lambda calculus cannot be used to formulate generalizations across all grammatical categories, the calculus of constructions cannot be used to formulate generalizations across all grammatical categories. Both languages rely on facts about grammatical posits which their terms cannot express.

But there is an important difference between the expressive limitations faced by the simply typed lambda calculus and the expressive limitations faced by the calculus of constructions. In the case of the calculus of constructions, the limitations are linguistic, not worldly. They are limitations on the generalizations which the calculus of constructions can

be used to make about its own linguistic features: in particular, the calculus of constructions cannot express certain claims about a few of its grammatical categories.[15] They are not limitations on the generalizations which the calculus of constructions can be used to make about the non-linguistic world. In the case of the simply typed lambda calculus, however, some of the limitations are worldly rather than linguistic. They are limitations on the generalizations which the simply typed lambda calculus can be used to make about the world: in particular, the simply typed lambda calculus cannot express certain generalizations about properties.[16] They are not merely limitations on the generalizations which the simply typed lambda calculus can be used to make about its own linguistic features.

To see why, consider the following two different clusters of phenomena over which a language might be used to generalize:

(i) phenomena concerning the language's own grammatical categories, linguistic features, and so on, and

(ii) phenomena concerning entities, properties, and other portions of reality.

When it comes to (i), both the calculus of constructions and the simply typed lambda calculus face certain expressive limitations. The calculus of constructions cannot be used to generalize over all of its grammatical categories, since there is no way to quantify into the positions of kinds and sorts. The simply typed lambda calculus cannot be used to generalize over all of its grammatical categories either, since there is no way to quantify into the positions of types.[17] But when it comes to (ii), the calculus of constructions succeeds where the simply typed lambda calculus does not. In the calculus of constructions, portions of reality are represented by s-terms only. And the calculus of constructions has the linguistic resources to generalize over the semantic values of all s-terms whatsoever: entities, properties of entities,

[15] For example, the calculus of constructions cannot be used to express the claim "All kinds, and all sorts, are self-identical."

[16] To give yet another example of this: the simply typed lambda calculus cannot be used to express the claim "All properties, of all adicities, are self-identical."

[17] Even here, however, the calculus of constructions has an advantage over the simply typed lambda calculus. For the former, but not the latter, can be used to generalize over types. So when it comes to phenomena like (i), the calculus of constructions is—once again—more expressive.

and so on. In the simply typed lambda calculus, however, portions of reality are represented by terms. And the simply typed lambda calculus does not have the linguistic resources to generalize over the semantic values of all terms whatsoever.

Therefore, the calculus of constructions really does represent a significant improvement over the simply typed lambda calculus. Whereas some limitations of the simply typed lambda calculus concern its ability to represent worldly phenomena, the limitations of the calculus of constructions only concern its ability to represent linguistic phenomena. So when it comes to representing the non-linguistic world, the calculus of constructions is the better language.

7 Conclusion

For the purposes of doing metaphysics, the calculus of constructions is better than other languages discussed in the philosophical literature, such as the simply typed lambda calculus. It can be used to formulate better theories of identity and better theories of absolute generality. Whether some other language might be even better, for metaphysical theorizing, is an open question. But for now, it is clear that the calculus of constructions is better than the known alternatives.

Appendix 1: The Simply Typed Lambda Calculus

In this appendix, I present a rigorous formulation of the simply typed lambda calculus. To start, I define the types. Then I use those types to define the terms.

Types are the basic grammatical categories in the simply typed lambda calculus. They are defined below.

- Constants 't' and 'e' are types.
- For all types σ and τ, '$\sigma \Rightarrow \tau$' is a type.

- Nothing else is a type.

Now to define terms and their types.

(1) The constant '$\rightarrow$' is a term of type $t \Rightarrow (t \Rightarrow t)$.

(2) For all types σ, there are infinitely many constants 'a_σ', 'b_σ', and so on. Each of these is a term of type σ.

(3) For all types σ, there are infinitely many variables 'x_σ', 'y_σ', and so on. Each of these is a term of type σ.

(4) For all types σ, there is a constant '$\forall_\sigma$'. This constant is a term of type $(\sigma \Rightarrow t) \Rightarrow t$.

(5) For all types σ, there is a constant '$\equiv_\sigma$'. This constant is a term of type $\sigma \Rightarrow (\sigma \Rightarrow t)$.

(6) For all types σ and τ, for all terms M of type $\sigma \Rightarrow \tau$, and for all terms N of type σ, 'MN' is a term of type τ.

(7) For all types σ and τ, for all variables x_σ, and for all terms M of type τ, '$\lambda x_\sigma.M$' is a term of type $\sigma \Rightarrow \tau$.

(8) Nothing else is a term.

Each '$\forall_\sigma$' is a quantifier: for any term 'M' of type $\sigma \Rightarrow t$, '$\forall_\sigma M$' says that everything of type σ has the property expressed by 'M'. Each '$\equiv_\sigma$' is an identity symbol: for any terms 'a_σ' and 'b_σ' of type σ, '$(\equiv_\sigma a_\sigma)b_\sigma$' says that the referent of 'a_σ' is identical to the referent of 'b_e'.

In this paper, I treated logical constants as infix operators. For instance, in place of '$(\rightarrow M)N$', I would write '$M \rightarrow N$'. And I usually dropped the subscripts from terms. For instance, in place of '$P_{e \Rightarrow t} c_e$', I wrote 'Pc'.

Appendix 2: The Calculus of Constructions

In this appendix, I present a rigorous formulation of the calculus of constructions. To start, I define what are called 'pseudo-terms'. Then I formulate the account of which pseudo-terms have which grammatical categories. Finally, I define the terms.

Pseudo-terms are defined below.

- The constants '$\Box$' and '$*$' are pseudo-terms.
- The constants 'e' and 't' are pseudo-terms.
- The constants '$\rightarrow$', '$\equiv$', '$\approx$', '$\forall$', '$\mathbb{V}$', and '$\mathbb{I}$' are pseudo-terms.
- Infinitely many variables 'x', 'y', and so on, are pseudo-terms.
- If A and B are pseudo-terms, then 'AB' is a pseudo-term.
- If A and B are pseudo-terms, and 'x' is a variable, then '$\lambda x{:}A.B$' is a pseudo-term.
- If A and B are pseudo-terms, and 'x' is a variable, then '$\Pi x{:}A.B$' is a pseudo-term.

These are called 'pseudo-terms' (Barendregt, 1991, p. 128), rather than just 'terms', because some pseudo-terms are ungrammatical. As is discussed later, terms are pseudo-terms which have a well-defined grammatical category.

The symbol ':' is used to assign grammatical categories to pseudo-terms. In particular, take any pseudo-term M and any pseudo-term A. The string '$M{:}A$' says, among other things, that the pseudo-term M has grammatical category A.

Now for the account of which pseudo-terms are grammatical. The account is, basically, an account of which strings of the form '$A{:}B$' are well-defined.

(1) The following strings are well-defined: '$*{:}\Box$', '$e{:}*$', and '$t{:}*$'.

(2) For all pseudo-terms A, if there is a sort s such that '$A{:}s$' is well-defined, then there are infinitely many constants 'c', 'd', and so on, such that '$c{:}A$', '$d{:}A$', and so on, are well-defined.

(3) For all pseudo-terms A, if there is a sort s such that '$A{:}s$' is well-defined, then there are infinitely many variables 'x', 'y', and so on, such that '$x{:}A$', '$y{:}A$', and so on, are well-defined.

(4) For all pseudo-terms A and B, if there are sorts s and s' such that '$A{:}s$' and '$B{:}s'$' are well-defined, then for all variables 'x' such that '$x{:}A$' is well-defined, '$(\Pi x{:}A.B){:}s'$' is also well-defined.

(5) For all pseudo-terms A and B, if there are sorts s and s' such that '$A{:}s$' and '$B{:}s'$' are well-defined, then for all variables 'x' such that '$x{:}A$' is well-defined, and for all

pseudo-terms M such that 'M:B' is well-defined, '$(\lambda x{:}A.M){:}(\Pi x{:}A.B)$' is well-defined.

(6) For all pseudo-terms A, B, M, and N such that 'N:A' is well-defined, and for all variables 'x' such that 'x:A' is well-defined, if '$M{:}(\Pi x{:}A.B)$' is well-defined then '$MN{:}B[N/x]$' is well-defined.[18]

(7) The strings below are well-defined.

- '$\rightarrow {:}\big(\Pi x{:}t.(\Pi y{:}t.t)\big)$'
- '$\equiv {:}\Big(\Pi\alpha{:}*.\big(\Pi\beta{:}*.\big(\Pi x{:}\alpha.(\Pi y{:}\beta.t)\big)\big)\Big)$'
- '$\approx {:}\big(\Pi\alpha{:}*.(\Pi\beta{:}*.t)\big)$'
- '$\forall{:}\Big(\Pi\alpha{:}*.\big(\Pi y{:}(\Pi x{:}\alpha.t).t\big)\Big)$'
- '$\mathbb{\forall}{:}\big(\Pi x{:}(\Pi\alpha{:}*.t).t\big)$'
- '$\mathbb{\exists}{:}\big(\Pi x{:}(\Pi\alpha{:}*.t).t\big)$'

(8) Nothing else is well-defined.

Finally, pseudo-term 'A' is a term just in case for some pseudo-term 'B', 'A:B' is well-defined.

In this paper, I adopted the following notational shorthands. For all types σ and τ, '$\equiv_{\sigma,\tau}$' was shorthand for '$(\equiv \sigma)\tau$'; and '$\equiv_{\sigma}$' was shorthand for '$(\equiv \sigma)\sigma$'. For all types σ, all variables 'x' of type σ, and all terms M of type t, '$\forall_{\sigma} xM$' was shorthand for '$(\forall\sigma)(\lambda x{:}\sigma.M)$'. In addition, for all type variables α and all terms M of type t, '$\mathbb{\forall}\alpha M$' was shorthand for '$\mathbb{\forall}(\lambda\alpha{:}*.M)$' and '$\mathbb{\exists}\alpha M$' was shorthand for '$\mathbb{\exists}(\lambda\alpha{:}*.M)$'.

There are semantic theories for the calculus of constructions. For some examples, see (Coquand & Huet, 1988; Huet, 1990; Stefanova & Geuvers, 1995). For semantic theories of simpler languages which, plausibly, can be extended to cover the calculus of constructions, see (Bruce et al., 1990).

Acknowledgements

[removed for anonymous review]

[18]The string '$B[N/x]$' is obtained by replacing each free occurrence of 'x' in 'B' with 'N'.

References

Bacon, A. (2020). Logical Combinatorialism. *The Philosophical Review, 129*(4), 537–589.

Bacon, A., & Russell, J. S. (2019). The Logic of Opacity. *Philosophy and Phenomenological Research, XCIX*(1), 81–114.

Barendregt, H. (1991). Introduction to generalized type systems. *Journal of Functional Programming, 1*(2), 125–154.

Barendregt, H. (1992). Lambda Calculi with Types. In S. Abramsky, D. M. Gabbay, & T. S. E. Maibaum (Eds.), *Handbook of Logic and Computer Science* (Vol. II, pp. 117–309). New York, NY: Oxford University Press.

Boolos, G. (1984). To Be is to be a Value of a Variable. *The Journal of Philosophy, 81*(8), 430–449.

Bruce, K. B., Meyer, A. R., & Mitchell, J. C. (1990). The Semantics of Second-Order Lambda Calculus. *Information and Computation, 85*, 76–134.

Caie, M., Goodman, J., & Lederman, H. (2020). Classical Opacity. *Philosophy and Phenomenological Research, 110*(3), 524–566.

Cartwright, R. L. (1994). Speaking of Everything. *Noûs, 28*(1), 1–20.

Coquand, T., & Huet, G. (1988). The Calculus of Constructions. *Information and Computation, 76*, 95–120.

Dorr, C. (2016). To be F is to be G. *Philosophical Perspectives, 30*, 39–134.

Girard, J. (1986). The System F of Variable Types, Fifteen Years Later. *Theoretical Computer Science, 45*, 159–192.

Gonthier, G. (2008). Formal Proof—The Four-Color Theorem. *Notices of the American Mathematical Society, 55*(11), 1382–1393.

Goodman, J. (2017). Reality is not structured. *Analysis, 77*(1), 43–53.

Huet, G. (Ed.) (1990). *Logical Foundations of Functional Programming.* Addison-Wiley.

Krämer, S. (2014). Semantic values in higher-order semantics. *Philosophical Studies, 168*, 709–724.

Krämer, S. (2017). Everything, and Then Some. *Mind, 126*(502), 499–528.

Linnebo, Ø. (2006). Sets, Properties, and Unrestricted Quantification. In A. Rayo & G. Uzquiano (Eds.), *Absolute Generality* (pp. 149–178). New York, NY: Oxford University Press.

McGee, V. (2006). There's a Rule for Everything. In A. Rayo & G. Uzquiano (Eds.), *Absolute Generality* (pp. 179–202). New York, NY: Oxford University Press.

Mitchell, J. C. (1996). *Foundations for Programming Languages*. Cambridge, MA: MIT Press.

Quine, W. V. (1948). On What There Is. *The Review of Metaphysics, 2*(5), 21–38.

Rayo, A. (2006). Beyond Plurals. In A. Rayo & G. Uzquiano (Eds.), *Absolute Generality* (pp. 220–254). New York, NY: Oxford University Press.

Rayo, A., & Yablo, S. (2001). Nominalism Through De-Nominalization. *Noûs, 35*(1), 74–92.

Sider, T. (2011). *Writing the Book of the World*. New York, NY: Oxford University Press.

Stefanova, M., & Geuvers, H. (1995). A simple model construction for the Calculus of Constructions. In S. Berardi & M. Coppo (Eds.), *Types for Proofs and Programs* (pp. 249–264). Berlin: Springer.

Williamson, T. (2003). Everything. *Philosophical Perspectives, 17*, 415–465.

Williamson, T. (2013). *Modal Logic as Metaphysics*. New York, NY: Oxford University Press.