2022 TheoLogica
An International Journal for Philosophy of Religion and Philosophical Theology
Published Online First: September 05, 2022
DOI: https://doi.org/10.14428/thl.v6i2.65223

Computational Theology: A Metaobject-Based Implementation of Models of Generalised Trinitarian Logic

TIM LETHEN
University of Helsinki
tim.lethen@gmx.de

Abstract: This paper analyses an amazingly close analogy between models of generalised trinitarian logics on the one hand side and class hierarchies in the field of object-oriented programming on the other, thus linking philosophy of religion and computer science. In order to bring out this analogy as clear and precise as possible, we utilise a metaobject protocol for the actual implementation of the theological models. These formal implementations lead to the insight that the analogy can be pushed even further, and we lay bare and analyse the close relation between the theological notion of subordination of divine persons and precedence in structures of multiple inheritance. The implementation of theoretical godheads finally leads to new metaobject programming techniques, thus underlining the cross-fertilisation between theology and computer science.

Keywords: Trinitarian logic, Metaobject protocol, Subordination, Precedence, Common Lisp, CLOS

1. Introduction

It probably will not happen too often that theology and computer science benefit from each other in some kind of cross-fertilisation. In the direction 'from right to left,' logicians and computer scientists have largely concentrated on computer assisted analyses of ontological arguments for the existence of God. Some earlier examples of this ongoing work can be seen in Oppenheimer & Zalta (2011), Benzmüller & Woltzenlogel Paleo (2014), and Rushby (2018). These examples strongly rely on logical formalisations of the ontological argument and, thus, can be regarded as a computational branch of what has been called "logical philosophy of religion" (cf. Silvester et al., 2020b). Another 'right to left' example can be found

in the AI-based implementations of dynamic models simulating a Talmudic theory of mixtures, see David et al. (2020) and Lethen (2021). And finally, many computer-assisted analyses of biblical texts also fall into the category of a computational theology.[1]

In the opposite direction, one searches in vain for examples, and it might indeed be hard to imagine how computer science may benefit from theological considerations.[2] This paper now tries to fill this gap by analysing a close correspondence between set-theoretical models of generalised trinitarian logics on the one hand side, and object-oriented class hierarchies on the other. Inspired by the strong analogy between (divine) *persons* and (object-oriented) *classes*, as well as between *godheads* (taken as a collection of divine persons) and *classes* (regarded as a collection of their attribute descriptions), we utilise a metaobject protocol in order to underline the similarity between these theological and computational structures.

Metaobject protocols have been designed in order to enable the programmer to customise a programming language and adjust it to her personal needs and preferences. The just mentioned analogies now lead to the insight that the metaobject protocols may also serve as a framework which facilitates an elegant and lean implementation of 'constellations' which mirror an object-oriented language design and which comprise a functionality which closely resembles object-oriented mechanisms. In this respect, the analogy investigated in this paper leads to new programming techniques which are based on the use of the metaobject protocol.

One of the just mentioned object-oriented mechanisms is the computation of precedence lists of classes which may be part of a complex multiple-inheritance structure. This situation pushes the analogy even further, relating the phenomenon of object-oriented precedence to the theological question of subordination. The implementation of divine structures in a programming language thus enables us, and at the same time forces us, to specify the formal meaning of a divine precedence. As this example shows, programming languages—as well as the field of algorithmics in general—should be added the spectrum of formal languages which can be fruitfully used in analytic philosophy and theology.

In this paper, we proceed as follows. Section 2 introduces the system of trinitarian logic Θ_3, which has been defined in Lethen (2022). The section also

---

[1] An informative example of such an analysis can be seen in van Peusen & Talstra (2007).

[2] As a remarkable exception, one can consider the ongoing *Talmudic Logic Project*, see Gabbay et al. (2019) and the literature mentioned therein.

discusses possible generalisations along with their set-theoretical models. Section 3 then describes the concept of the metaobject and briefly introduces some of the main features of the COMMON LISP metaobject protocol. Section 4 uses these features in order to implement divine persons as instances of a metaclass `person`, thus bringing out the analogy between persons and object-oriented classes. This analogy is taken up again in Section 5, where the concept of the *preceding list* is carried over from classes to persons. As we shall see, topological sorting will play a central role in this connection. Before we conclude, the final Section 6 shows how to define godheads as classes, again heavily relying on mechanisms which are predefined in the COMMON LISP metaobject protocol.

2. Models of Generalised Trinitarian Logic

The "trinitarian" logic Θ_3 has been introduced in Lethen (2022) and comprises the logical interpretation of seven Catholic *de fide* dogmas concerning Trinity. Note the inclusion of Gödel's modality $\Box_D$ which is interpreted as 'It is a dogma, that . . .'[3] Following the order given in Ott (1957), the dogmas are:

DOGMA (D1): *In God there are Three Persons, the Father, the Son and the Holy Ghost. Each of the Three Persons possesses the one (numerical) Divine Essence. (De fide.)*[4]

$$\Box_D\,(\,\text{person}(\alpha) \wedge \text{person}(\beta) \wedge \text{person}(\gamma) \wedge \alpha \neq \beta \wedge \alpha \neq \gamma \wedge \beta \neq \gamma \wedge \forall x.\left(\text{person}(x) \supset (x = \alpha \vee x = \beta \vee x = \gamma)\right))$$

DOGMA (D2): *In God there are two Internal Divine Processions. (De fide.)*

$$\Box_D\,\exists\, P, Q.\left(\text{Proc}(P) \wedge \text{Proc}(Q) \wedge P \neq Q \wedge \forall R.\left(\text{Proc}(R) \supset (R = P \vee R = Q)\right)\right)$$

DOGMA (D3): *The Divine Persons, not the Divine Nature, are the subject of the Internal Divine processions (in the active and in the passive sense). (De fide.)*

$$\Box_D\,\forall\, P.\forall x, y.\left[\left(\text{Proc}(P) \wedge P(x, y)\right) \supset \left(\text{person}(x) \wedge \text{person}(y)\right)\right]$$

---

[3] For the purpose of our present analysis, we could readily exclude the modal operator from all the dogmas. If it is included, the considered set-theoretical models have to be assumed for every accessible world.

[4] As we are mainly interested in the Divine Persons and their relationships, we only translate the first part of this dogma.

DOGMA (D4): *The Second Divine Person proceeds from the First Divine Person by Generation, and therefore is related to Him as Son to a Father. (De fide.)*[5]
$\Box_D \left[\mathrm{gen}(\alpha,\beta) \wedge \mathrm{Proc}(\mathrm{gen}) \wedge \mathrm{FS}(\mathrm{gen})\right]$

DOGMA (D5): *The Holy Ghost proceeds from the Father and from the Son as from a Single Principle through a Single Spiration. (De fide.)*
$\Box_D \left[\mathrm{spir}(\alpha,\gamma) \wedge \mathrm{spir}(\beta,\gamma) \wedge \mathrm{Proc}(\mathrm{spir})\right]$

DOGMA (D6): *The Holy Ghost does not proceed through generation but through spiration. (De fide.)*
$\Box_D \left[\neg\exists x.\mathrm{gen}(x,\gamma) \wedge \exists x.\mathrm{spir}(x,\gamma)\right]$

DOGMA (D7): *In God all is one except for the opposition of relations. (De fide.)*
$\Box_D \forall x,y.\left[\left(\mathrm{person}(x) \wedge \mathrm{person}(y)\right)\right.$
$\left.\supset \left(\neg\exists P.\left[\mathrm{Proc}(P) \wedge \left(P(x,y) \vee P(y,x)\right)\right] \supset x = y\right)\right]$

As Kurt Gödel repeatedly stressed, studies in theoretical theology should now allow for the alteration of the dogmatic (i.e. axiomatic) basis according to individual preferences and views. These alterations then immediately lead to a great variety of what we call *generalised trinitarian logics*. Alterations of Θ_3 may include:

- The *exclusion* of entire dogmas or parts of dogmas. As an example, the exclusion of (D7) would enable two different persons to coexist without proceeding from each other.
- The *alteration* of a dogma. Altering (D1), for example, would allow for models with more or less than three persons.
- The *addition* of a dogma. Through the inclusion of additional dogmas, specific stipulations can be imposed on a theological model.

Figure 1 shows four different set-theoretical models of (generalised) trinitarian logics. Whereas a, b, and c represent the Father, the Son, and the Holy Spirit, solid

[5] The second-order predicate FS (father-son) testifies that a first-order binary predicate follows the rule expressing that a child can have at most one father. It is defined as $\forall P.\left[\mathrm{FS}(P) \equiv \forall x,y,z.\left(\left(P(x,z) \wedge P(y,z)\right) \supset x = y\right)\right]$.

and dashed arrows indicate generation and spiration, respectively. Model (a) constitutes the traditional Roman Catholic view according to which the Holy Spirit was spirated by the Father and the Son. It thus is a model of the unmodified logic Θ_3. Excluding the middle conjunct of dogma (D5), spir(β, γ), from Θ_3 leads us to model (b) in which the Son is spirated by the Holy Spirit. It is this very conjunct which represents the famous *filioque* clause, the inclusion of which is said to have lead to the Great Schism of 1054 between the (Latin) Roman Catholic Church and the (Greek) Eastern Orthodox Church. The exclusion of dogma (D7), known as Anselm of Canterbury's *basic trinitarian law*, allows or the coexistence of persons without being connected by some kind of procession. Model (c) follows exactly this path. Finally, model (d) depictures the situation, in which the Holy Spirit has been spirated by the Father *through* the Son.

Of course, one may wonder how far one is allowed to deviate from the original system Θ_3 and still call it a system of generalised *trinitarian logic*.

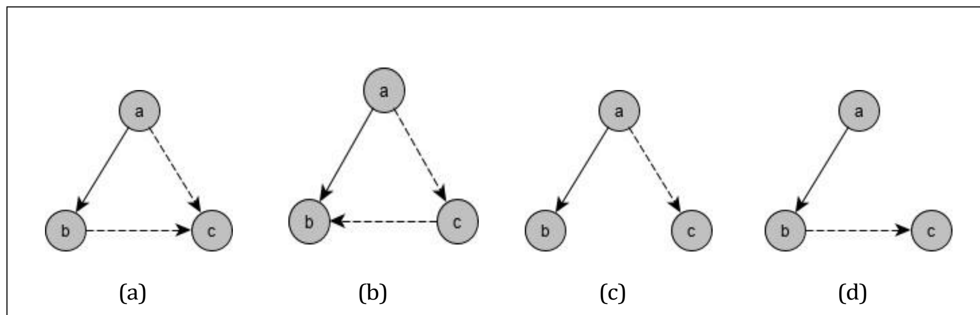

Figure 1: Four models of different generalised trinitarian logics. *a*, *b*, and *c* represent the Father, the Son, and the Holy Spirit. Solid and dashed arrows indicate generation and spiration, respectively.

And in fact, the only constraint which will be central for our present purpose, is the existence of a finite number of divine persons, paired with the possibility to connect these persons by a predefined finite number of processions.[6] As an example, Figure 2 depicts a theoretical godhead comprising six divine persons, a situation which already hints at the necessity to have a closer look at the concept of precedence and subordination.

---

[6] In this connection, the reader may want to reconsider the roles of the single dogmas of Θ_3. Whereas (D1) and (D2) fix the number of persons and processions, respectively, (D3) makes sure that a procession is a relation on the set of persons. Dogmas (D4)–(D7) then describe the more detailed facts about these relations.

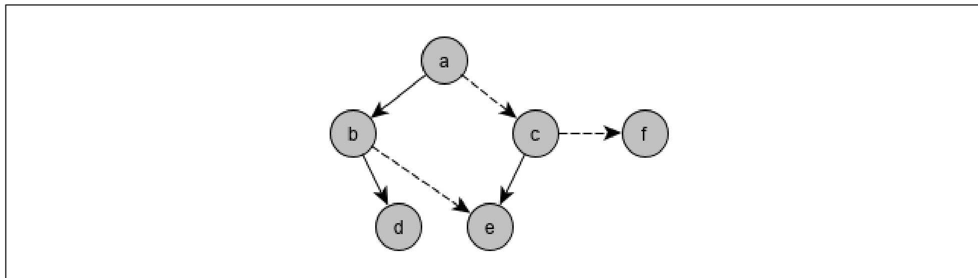

Figure 2: A model of generalised trinitarian logic comprising six divine persons

3. Metaobject Protocols

In this section, we briefly introduce the COMMON LISP metaobject protocol (MOP), which has been developed in Kiczales et al. (1991). In order to be able to do so, we first give a very short introduction to the language CLOS, which is an object-oriented extension[7] of the programming language COMMON LISP. Detailed and comprehensive introductions to CLOS can, for example, be found in DeMichiel (1993) and Keene (1989).[8]

As most other object-oriented languages, CLOS enables the programmer to define *classes* as the main building blocks of their programs. A class may be regarded as an abstract description of *objects* which in turn serve as the actors of the program. The objects are often called *instances* of the class, while the class also serves as the *type* of its instances. As an example, we define a class `city` and store an instance of this class in the variable `city1`.[9]

```
(defclass city ()
   ((name :initarg :name
          :reader name)
     (population :initarg :population
                 :initform 0
```

[7] Although CLOS can be seen as an *extension* to the COMMON LISP language, it is part of the COMMON LISP ANSI standard X3.226. It was also included in Steele (1990), which is often regarded as the COMMON LISP *de facto* standard.

[8] COMMON LISP is the language of choice for our investigations as it incorporates a very flexible, powerful, and well documented metaobject protocol (cf. Kiczales et al. 1991), which allows for the adjustment of many of the underlying object-oriented mechanisms.

[9] Here and in the following code snippets, the symbol >>> symbolises interactive user code. The immediately following line always represents the system's answer.

```
                 :accessor population)))

 >>> (setf city1 (make-instance 'city :name "Paris"
                                      :population 2161000))

#<CITY 200E939F>
```

As one can see, the objects of the class city comprise two attributes,[10] namely `name` and `population`. The according *slot definitions* use the keyword `:initarg` to introduce another keyword (`:name`, `:population`) to be used with the constructor `make-instance`. Other possible keywords are `:initform`, which introduces a default initial value for the slot, and `:reader` and `:accessor`, which define methods for read-only and read/write access to the object's slots, respectively. Access to the slots can be demonstrated by the following user code.

```
>>> (name city1)
"Paris"
```

Next to the classes, methods form the second most important building block in CLOS. As we have just seen, the definition of a class triggers the existence of certain access methods. But, of course, the user may define her own methods as well, as the following example demonstrates.

```
(defmethod inc-population ((c city) n)
  (incf (population c) n))

 >>> (inc-population city1 500)
 2161500
```

Note that the method `inc-population` is applied to the two parameters *c* and *n*, the first one being accompanied by the type `city`. Thus, the method is strictly tied to the class city and only works for instances of this very class.

Finally, we turn to the concept of inheritance, which is demonstrated by the following definition of the class `capital`. As a capital always is a city, it inherits the attributes `name` and `population` from the class `city` and adds a further slot `country`. The following example also demonstrates that, because `capital` is now a subtype

[10] In COMMON LISP, attributes are usually called *slots*.

of `city`, methods defined for objects of the class `city` may also work on objects of the class `capital`.

```
(defclass capital (city)
   ((country :initarg :country
             :reader country)))

 >>> (setf city2 (make-instance 'capital :name "Helsinki"
                                         :country "Finland"
                                         :population 648000))
 #<CAPITAL 200F8F5F>

 >>> (inc-population city2 1)
 648001
```

We conclude our short introduction to the language CLOS and proceed with a brief explanation of the CLOS metaobject protocol, which was first published in Kiczales et al. (1991), and which carries the concept of object-orientation to the meta-level. One of the central ideas has to be seen in the slogan 'everything is an object.' Following this slogan, the class of an object has to be an object itself. But this in turn means that there has to exist a metaclass, i.e. a class the instances of which are again classes. The following code demonstrates this phenomenon.

```
>>> (class-of city2)
#<STANDARD-CLASS CAPITAL 21E3500B>

>>> (class-of (class-of city2))
#<STANDARD-CLASS STANDARD-CLASS 20B0ABF3>
```

As we can see, `capital` is the class of the object `city2` and `standard-class` is the (meta-)class of the class `capital`, now regarded as an ordinary object.

This example hints at the very fact that the language CLOS has itself been designed as an object-oriented system, which has again been written in CLOS. Thus, methods of the metaclass `standard-class` are now responsible for the behaviour of user-level classes like `city` and `capital`: How their instances are constructed and stored, how their methods are invoked, and how inheritance is organised. As an example, we demonstrate the use of the method `class-precedence-list` which is

defined for instances (i.e. user-level classes) of the metaclass `standard-class`. Note the inclusion of the two predefined classes `standard-object` and `t` in every precedence list.[11]

```
>>> (clos:class-precedence-list (find-class 'capital))
(#<STANDARD-CLASS CAPITAL 21E3500B> #<STANDARD-CLASS CITY 200DBBBF>
#<STANDARD-CLASS STANDARD-OBJECT 2012575B> #<BUILT-IN-CLASS T 202E6C53>)
```

The design of the language CLOS as an object-oriented system now leads to the enormously powerful possibility to adjust the language to one's personal needs and preferences through the definition of new metaclasses which inherit all their standard behaviour from, say, the metaclass `standard-class`, while certain aspects can be specialised. In their introduction to Kiczales et al. (1991), the authors write:

> The metaobject protocol approach, in contrast, is based on the idea that one can and should "open languages up," allowing users to adjust the design and implementation to suit their particular needs. In other words, users are encouraged to participate in the language design process.

Comprehensive introductions to the CLOS MOP and its design can be found in Kiczales et al. (1991) and Paepke (1993b).

4. Defining Persons

Being able to "adjust the design and implementation" of the language CLOS has always been regarded as the main aim of the metaobject protocol MOP. Regarding the motivation for the invention of the MOP, Gregor Kiczales writes in the article "Metaobject Protocols – Why We Want Them and What else They Can Do":

> The second [desire] was to satisfy what seemed to be a large number of user demands, including: compatibility with previous languages, performance comparable to (or better than) previous implementations and extensibility to allow further experimentation with object-oriented concepts [...]. The goal in developing the

[11] The prefix clos: shows that most of the MOP-specific methods are not part of the COMMON LISP language standard but are separated into a special package. The prefix may be implementation dependent. The function `find-class` maps a symbol to the class named by that symbol.

> MOP was to allow a simple language to be extensible enough that all these demands could be met. (Kiczales et al. 1993)

Concerning the question "why we want them and what else they can do," it has to be stressed though, that the possibility of shifting and relocating the language within a design space has, up to now, always been regarded as the *only* goal of the MOP. In this paper, we now propose a completely new approach to—and use of—the concept of the MOP, which has been inspired by the close analogy between object-oriented classes on the one hand side and divine persons on the other. In a first step, this analogy concentrates on the fact that the structure built by classes and inheritance between classes strongly resembles the structure built by the persons and the divine processions, i.e. generation and spiration, enhenced by the close relationship between the terms 'inheritance' and 'procession.' We therefore start with the definition of a metaclass `person`, which ensures that its instances have access to the lists of persons they have been generated or spirated by.

```
(defclass person (standard-class)
  ((generated-by    :initarg :generated-by
                    :accessor generated-by)
   (spirated-by     :initarg :spirated-by
                    :accessor spirated-by))
  (:documentation "The PERSON metaclass."))
```

In order to define a new person, one could now simply invoke the macro `defclass`, which would have to explicitly include the information that this newly defined person is an instance of the metaclass `person`. However, we prefer to introduce some 'syntactic suggar' for this very purpose and define the new macro `defperson` as follows:

```
(defmacro defperson (name gens spirs &optional doc)
  "Defines a person with NAME,
    given two lists of persons it was generated/spirated by."
  `(defclass ,name ,(append gens spirs)
     ()
     (:metaclass person)
     (:generated-by ,@gens)
     (:spirated-by ,@spirs)
     (:documentation ,doc)))
```

The example definition

```
>>> (defperson d (a b) (c) "test")
```

would thus be expanded into the following class definition, which is then immediately evaluated:

```
(defclass d (a b c)
   nil
   (:metaclass person)
   (:generated-by a b)
   (:spirated-by c)
   (:documentation "test"))
```

Here, the empty list `nil` indicates that no slots are defined. Note how the persons a, b, and c have now also been interpreted (and inserted in the right place) as 'superpersons' (i.e. superclasses) of the person d. The order of these will play a central roll in the following Section.

We are now ready to define the three Divine Persons as given in Figure 1 (a).

```
>>> (defperson son (father) ())
#<PERSON SON 200D7177>

>>> (defperson holy-spirit () (father son))
#<PERSON HOLY-SPIRIT 200FD117>

>>> (defperson father () ())
#<PERSON FATHER 200D72F7>
```

Let us keep in mind that each single person now is an instance of the metaclass `person`, which in turn is a subclass of the metaclass `standard-class`. Therefore, every defined divine person *is* a class. In order to underline that we are not allowed to make an instance of a person, one could now add the following CLOS code, which specialises the method `make-instance` for parameters of type `person`:

```
(defmethod make-instance ((pers person) &rest initargs)
  (error "It is not possible to make an instance of ~a." pers))
```

Already now, it becomes apparent that the definition of persons as classes leads to the possibility to make use of large parts of the predefined

CLOS MOP machinery. As an example, we mention the existence of so-called 'forward referenced' classes, which are automatically created whenever a class definition names a superclass which has not been defined yet. In our example, the person `son` has been defined using the (still undefined) superclass `father`, which thus becomes an instance of the built-in metaclass `forward-referenced-class`. Later, when `father` is defined as a proper class, its metaclass is internally changed from `forward-referenced-class` to `person` without the user having to bother.[12]

We can now also make use of some of the built-in MOP methods, notably those which deal with the structure of the user-defined persons. The following code snippet shows such an example. Here, the method `class-direct-subclasses` returns a list of the persons generated and spirated by the Father.

```
>>> (clos:class-direct-subclasses (find-class 'father))
(#<PERSON HOLY-SPIRIT 200FD117> #<PERSON SON 200D7177>)
```

5. Subordination, Precedence, and Topological Sorting

Concerning the equality of the Divine Persons and the resulting question of subordination, Effingham (2018) writes:

> [The *filioque* clause] threatens the equality of the Divine Persons. This is clearly a problem if we think that divine physical relations are causal, since one entity being caused by another *prima facie* indicates some sort of inferiority. (This argument was advanced by Gregory of Nazianus, as well as later theologians like Meijering.) It's also a specific example of a broader worry that applies also to the Son: If the Father spirates or generates another Divine Person then it seems *prima facie* plausible that they must be subordinate to the Father.

Is has to be mentioned, though, that the theological issue of subordination has also been associated with other types of relationship between the Divine Persons than causation, "some of these types of subordination [...] affirmed by conciliar theology whilst other are not" (Mullins 2020). Adopting the taxonomy given in Edwards (2020), subordination is

(i) *ontological* when it ascribes to the Son a substance, nature or essence which is inferior to the Father's,

---

[12] Note that it is possible in CLOS to change the class of an already existing object.

(ii) *aetiological* when it asserts the Son's posteriority in the order of causation,
(iii) *axiological* when it degrades him in rank or status without denying his equality in nature,
(iv) *economic* when it dates the subservience of the Son to the Father from some point after his origin, most commonly from his voluntary assumption of human nature.

And whereas Edwards' type-definitions only mention the relationship between the Father and the Son, all of these types can, without hesitation, be transferred to the relationship (and subordination) between other Divine Persons, even in generalised theological models.

However, these types of subordination are not able to answer some very fundamental questions concerning an order relation amongst divine persons in general, namely:

- How can we determine an order amongst divine persons in a model of *generalised* trinitarian logic?
- Which role do the different kinds of procession, i.e. generation and spiration, play when determining or computing this order?

As an example, let us consider the model given in Figure 2. One would probably readily agree that person d is inferior to both persons a and b. But is d also inferior (in any of the senses of types (i) to (iv)) to the persons c or e? Is there a reasonable way to define such a relation and can we still call it 'subordination'?

These considerations now push the analogy between classes and divine persons even further: While the ability to determine a precedence amongst classes is crucial when it comes to, for instance, the resolution of clashes between slots with identical names in structures of multiple inheritance, subordination—and therefore precedence—amongst divine persons in models of (generalised) trinitarian logic plays a crucial role in theological discourse.

Thus, having mentioned the "predefined CLOS MOP machinery" at the end of the preceding Section, the analogy may now lead directly to the question of the internal precedence among the defined persons. And indeed, as we have noticed earlier, there exists a method called `class-precedence-list`, which returns a list of 'preceding' superclasses, and therefore also of 'preceding' super-persons. (Again, note the usual inclusion of the built-in classes `standard-object` and `t` as the top of the hierarchy.)

```
>>> (class-precedence-list (find-class 'son))
(#<PERSON SON 200D7177> #<PERSON FATHER 200D72F7>
#<STANDARD-CLASS STANDARD-OBJECT 2012575B> #<BUILT-IN-CLASS T 202E6C53>)
```

However, invoking this method on the person `holy-spirit` leads to a signaled error.

```
>>> (clos:class-precedence-list (find-class 'holy-spirit))

Error: Error during finalization of class #<PERSON HOLY-SPIRIT 200FD117>:
Cannot compute class precedence list for class: #<PERSON HOLY-SPIRIT 200FD117>
```

In order to understand why this error occurs, we now have a closer look at the CLOS MOP mechanism which computes the precedence list of a given class. A detailed definition of this process can be found in Steele (1990, 782), where the precedence list for a class C is called a "total ordering on C and its superclasses that is consistent with the local precedence orders for C and its superclasses." As this process is crucial for our investigation, we now set out to explain it in some more detail, beginning with a definition of the *precedence list* of a divine person in a model of generalised trinitarian logic.

Let $\mathfrak{M}$ be such a model and let $\mathcal{M} = (G, \prec)$ be a structure in which G is the set of divine persons in $\mathfrak{M}$. Assume that $g \prec h$ holds for any $g, h \in G$ if and only if g is generated or spirated by h. If g is an element of G, let $\mathcal{M}_g = (G_g, \prec_g)$ be the structure *induced by* g, in which G_g is the subset of G consisting of those elements of G which can be reached from g via a $\prec$-path.[13] $\prec_g$ then is the relation $\prec$ restricted to the elements of G_g. As an example, Figure 3 depictures the structures $\mathcal{M}$ and $\mathcal{M}_e$ belonging to the model given in Figure 2. Note that these structures could as well represent a collection of classes, the relation $\prec$ representing the relation *is-a-direct-subclass-of*.

[13] The path may have length 0, so g is itself a member of G_g.

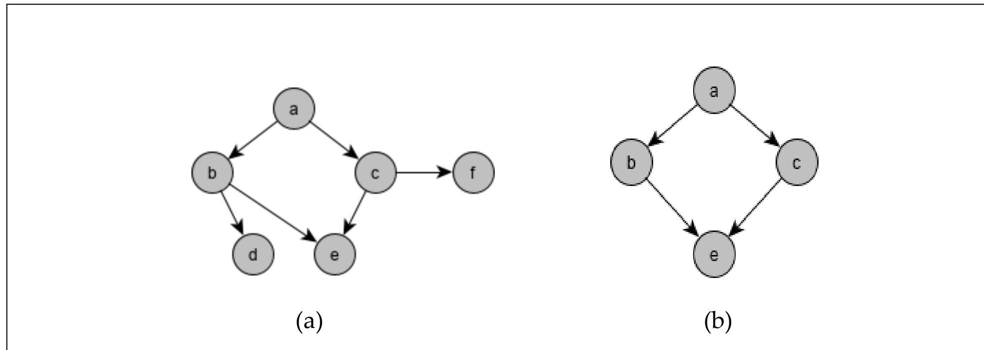

Figure 3: Two structures of divine persons based on the model presented in Figure 2. (a) The structure $\mathcal{M}$ of divine persons linked by the relation $\prec$. (b) The structure $\mathcal{M}_e$ induced by person *e*.

A *precedence list* of the structure $\mathcal{M}_g = (G_g, \prec_g)$ now is a sequence $(g_0, g_1, \dots, g_n)$ with $g_i \in G_g$ and $g = g_0$, and with $i < j$ whenever $g_i \prec g_j$. That is, a person g which is generated or spirated by person h will appear earlier in the precedence list than h. In this very sense, a person g may now be called *inferior* or *subordinate* to person h if and only if g appears further to the left in *every* possible preceding list. Returning to the example given in Figure 3, (e, b, c, a) and (e, c, b, a) are the possible precedence lists of $\mathcal{M}_e$.

The algorithm which computes a precedence list for a given structure $\mathcal{M}_g$ is known by the name of *topological sorting*. Details about this algorithm can, for example, be found in Cormen et al. (2009), and it is exactly this algorithm which is implemented in the CLOS MOP in order to compute the precedence list returned by the method `class-precedence-list`. As the persons defined by `defperson` are indeed classes, the superclasses of which represent the generating and spirating persons, this method should serve exactly the desired purpose when it comes to the concept of precedence and subordination of the defined persons. However, as the example of the Holy-Spirit has shown, the call

```
>>> (clos:class-precedence-list (find-class 'holy-spirit))
```

leads to an error. The reason is well hidden in the actual implementation of the method. Here, $g \prec h$ does not only hold if h is a direct superclass of g, but also if g appears at the left hand side of h in any list of direct superclasses. Thus, the definition

```
>>> (defperson holy-spirit () (father son))
```

triggers the constraints holy-spirit ≺ father, holy-spirit ≺ son, but also father ≺ son, the last constraint being incompatible with the constraint son ≺ father, which appears when defining the person son.

As a result, we will have to specialise the MOP method compute-class-precedence-list for arguments of type person. This is done in the following code, which is an implementation of a topological sorting which considers only constraints emerging between persons and their direct super-persons.

```
(defun topological-sort (vertex &optional temp result)
  "Topological Sorting starting with VERTEX, using a depth-first approach."
  (cond ((member vertex result) result)
        ((member vertex temp) (error "Cannot compute precedence."))
        (t (cons vertex
                 (reduce #'(lambda (acc x)
                             (topological-sort x (cons vertex temp) acc))
                         (clos:class-direct-superclasses vertex)
                         :initial-value result)))))

(defmethod clos:compute-class-precedence-list ((pers person))
    "Computes the precedence-list of person PERS, using TOPOLOGICAL-SORT."
    (topological-sort pers))
```

We refrain from a detailed explanation of the implementational details of the function topological-sort and prefer to give some more general comments on the presented approach.

1. The fact that the implementation of the language CLOS has been opened up to the user means that we can still rely on the great bulk of the implementation while only specialising a tiny portion, i.e. the MOP method compute-class-precedence-list.
2. The purpose of the alteration of the method is not grounded in the need to shift a programming language within its design space in order to experiment with other object-oriented paradigms, but to foster an analogy between the language architecture and a theological phenomenon.
3. Although it seems like we have not included the difference between generation and spiration in our approach, it is worthwhile mentioning that the implemented depth-first approach of the function topological-sort processes the list of direct super-persons simply from left to right. But the macro defperson, as written in Section 4, positions all the generating super-

persons in front of the spirating ones. We have thus been able to guide the sorting in a very subtle manner. Note that, in general, the order of the given super-persons in a `defperson` clause does influence the computation of the precedence list.

Finally, we are now in the position to demonstrate that the error, which appeared when computing the precedence-list of the Holy Spirit, has now disappeared. As the COMMON LISP language standard does not specify the precise moment when the inheritance of a class is finalised, we implement the additional method `precedence`, which takes care of the finalisation.[14] Note that this method also cuts off the two usually included built in classes `standard-object` and `t`.

```
(defmethod precedence ((pers person))
  "Returns the precedence-list of person PERS."
  (when (not (clos:class-finalized-p pers))
    (clos:finalize-inheritance pers))
  (butlast (clos:class-precedence-list pers) 2))

>>> (precedence (find-class 'holy-spirit))
(#<PERSON HOLY-SPIRIT 200FD117> #<PERSON SON 200D7177> #<PERSON FATHER 200D72F7>)
```

6. Defining Godheads

In this Section, we turn to the last and most complex analogy between models of generalised trinitarian logic and a metaobject-based approach to object-oriented programming, an analogy which leads straight to new implementation techniques. It is based on the observation that a possible definition of a godhead, which comprises a collection of divine persons and their description, may have the the same 'shape' as a usual class definition, which mainly consists of a collection of slot descriptions. The implementation thus mirrors a kind of Latin Trinitarianism according to which "the relation of the Godhead to the persons of the Trinity [is] something like the relation between universals and their instances" (Molto 2017,

[14] Kiczales et al. (1991, 156) state: "The exact point at which finalize-inheritance is called depends on the class of the class metaobject; for standard-class it is called sometime after all the classes superclasses are defined, but no later than when the first instance of the class is allocated." As the user is not allowed to construct instances of classes (i.e. persons) of the metaclass `person`, we explicitly have to invoke the method `finalize-inheritance`.

232). Also, the implementation resembles a kind of Social Trinitarianism which, according to Richard Swinburne, "involves the claim that the Godhead is 'a collective.' The persons are distinct entities, which, taken together, compose the divine collective" (Molto 2017, 230). The following two examples illustrate the parallel.[15]

```
>>> (defclass capital (city)
      ((country :initarg :country
                :reader country)))
#<STANDARD-CLASS CAPITAL 200AC743>

>>> (defgodhead god2 (god1)
      ((spirit :spirated-by (a b)
               :generated-by (c))))
#<GODHEAD GOD2 200DCBEB>
```

Here, the godhead `god2` inherits all the persons of `god1` just as the class `capital` inherits all its slot definitions from the class `city`.[16] In addition, it defines the new (or altered) person `spirit`. As a direct consequence, one would be able to merge two or even several already defined godheads through the mechanism of multiple inheritance, adding and altering persons just as needed. Name clashes, precedence, and other issues concerning (multiple) inheritance would be handled by the underlying MOP mechanisms.

We prepare this very idea with the following definitions. First, we define the metaclass `godhead` as a subclass of the metaclass `standard-class`, thus inheriting the standard inheritance mechanisms. Then we introduce the macro `defgodhead`, which enables the user to define her godheads as was shown in the previous example. The macro simple translates the godhead definition into a class definition, referring to the metaclass `godhead`.

```
(defclass godhead (standard-class)
  ()
  (:documentation "The GODHEAD metaclass."))
```

[15] An analysis or critique of these accounts clearly lies beyond the scope of this paper. For the details, we refer the reader to the paper Molto (2017).

[16] We do not show the definition of the godhead `god1` here and simple assume that it has been predefined.

```
(defmacro defgodhead (name supers person-definitions)
  "The macro defining a godhead."
  `(defclass ,name ,supers
     ,person-definitions
     (:metaclass godhead)))
```

Unfortunately, the application of the macro defgodhead would now lead to an error. The reason for this error has to been seen in the fact that a call to defclass does not accept the keywords :generated-by and :spirated-by in its slot decriptions, which is in turn due to the fact that invoking the defclass macro always leads to the creation of several slot-definition objects, which themselves belong to a metaclass slot-definition. And while these objects do have slots called name, initarg, accessor, and the like, they do (of course) not have any slots called generated-by and spirated-by. We therefore define a specialised subclass person-definition of the metaclass slot-definition and additionally tell the system to use this metaclass whenever a godhead is defined.[17]

```
(defclass person-definition (clos:standard-effective-slot-definition
                             clos:standard-direct-slot-definition)
  ((generated-by    :initarg :generated-by
                    :initform nil
                    :accessor generated-by)
   (spirated-by     :initarg :spirated-by
                    :initform nil
                    :accessor spirated-by)
   (combiner        :initarg :combiner
                    :initform #'union
                    :accessor combiner))
  (:documentation "The PERSON-DEFINITION metaclass."))

(defmethod clos:direct-slot-definition-class ((gh godhead) &rest initargs)
  (find-class 'person-definition))

(defmethod clos:effective-slot-definition-class ((gh godhead) &rest initargs)
  (find-class 'person-definition))
```

[17] Actually, we make use of two metaclasses which are direct subclasses of slot-definition. For details of this metaclass structure, see Kiczales et al. (1991, chapter 5).

As the reader will have noticed, we have also introduced a new slot called `combiner`. This slot describes the function used to 'combine' lists of generating or spirating persons which might collide when several super-godheads of a godhead define persons with a common name. The default value is the function `#'union` which represents ordinary set-union. This means that in case of clashes of persons with identical names the union of all their, say, generating super-persons is taken as the new set of generating persons. In order to clarify this mechanism, we define the following four godheads comprising the persons *a*, *b*, and *c*. Note that there appears a name clash (person *c*) when `god3` and `god4` are defined. In the case of `god3`, *both* persons *a and b* are taken as spirating super-persons, while in the case of `god4` only *b* is a super-person.[18] The resulting situation is depictured in Figure 4.

```
>>> (defgodhead god1 ()
      ((a)
       (b :generated-by (a))))
#<GODHEAD GOD1 2010BD33>

>>> (defgodhead god2 ()
      ((a)
       (c :spirated-by (a))))
#<GODHEAD GOD2 200A2A5F>

>>> (defgodhead god3 (god1 god2)
      ((c :spirated-by (b))))
#<GODHEAD GOD3 200F7087>

>>> (defgodhead god4 (god1 god2)
      ((c :spirated-by (b)
          :combiner #'override)))
#<GODHEAD GOD4 21EEC55F>
```

When inheritance is computed in the MOP, the method `compute-effective-slot-definition` is called to combine the previously collected slot-values of the slot-definition objects. In order to customise this method for our needs and make sure that generating and spirating persons are combined according to the given combiner, the following method definition has to be included.

---

[18] The function `override` is simply defined as `(defun override (x y) x)`. Thus, only the most specific list of super-persons is adopted when name clashes appear.

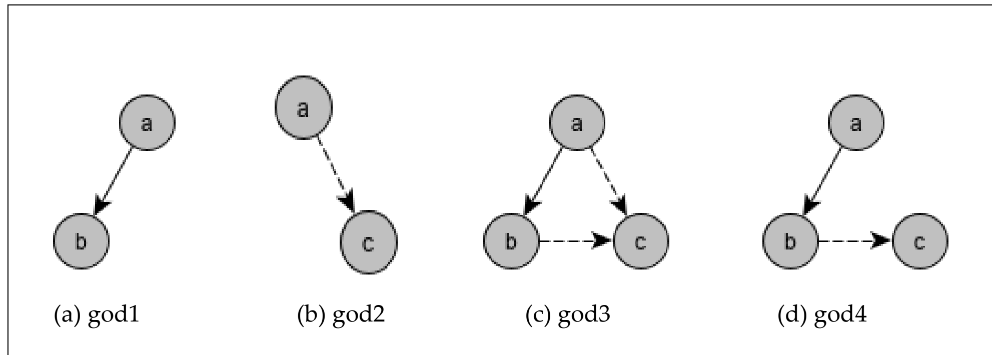

Figure 4: The godheads defined using the `defgodhead` macro. Both `god3` and `god4` inherit from `god1` and `god2` but use different combiners to resolve name clashes.

We refrain from discussing the details of the definition but, again, do stress the importance of the existence of the possibility to redefine the method for our needs, which is offered by the MOP.[19]

```
(defmethod clos:compute-effective-slot-definition :around ((gh godhead) name
                                                            direct-slot-definitions)
  (let ((resulting-person-definition (call-next-method)))
    (setf (combiner resulting-person-definition)
          (combiner (first direct-slot-definitions)))
    (setf (generated-by resulting-person-definition)
          (reduce (combiner resulting-person-definition)
                  direct-slot-definitions :key #'generated-by))
    (setf (spirated-by resulting-person-definition)
          (reduce (combiner resulting-person-definition)
                  direct-slot-definitions :key #'spirated-by))
    resulting-person-definition))
```

7. Subordination Again

When computing a precedence amongst divine persons in Section 5, we always reduced the structures $\mathcal{M} = (G, \prec)$ to structures $\mathcal{M}_g$ induced by a single person g. However, one may also want to determine a linear order on the godhead as a whole, thus also defining a precedence (or subordination) between persons which are not linked through a $\prec$-path. As an example, consider the model given in

[19] Details about the behaviour of this method can be found in Kiczales et al. (1991, 177).

Figure 2. Should person e be subordinated to person d in any of the senses (i) to (iv) named in Section 5?

As a first and somewhat naive answer we return to topological sorting again, this time incorporating the whole model $\mathcal{M}$. Before we can apply the algorithm to a godhead, we have to define the participating divine persons and convert the godhead's person-definitions into proper instances of the class `person`. To this end, we loop through the person-definitions and create new instances of the metaclass `person` by calling the MOP method `ensure-class`.

```
(defun define-persons (godhead)
  "Defines the persons as described in the GODHEAD's person-definitions."
  (let ((gh (find-class godhead)))
    (when (not (clos:class-finalized-p gh))
      (clos:finalize-inheritance gh))
    (dolist (person-definition (clos:class-effective-slots gh) gh)
      (let ((gen (generated-by person-definition))
            (spir (spirated-by person-definition)))
        (clos:ensure-class (clos:slot-definition-name person-definition)
                           :generated-by gen
                           :spirated-by spir
                           :direct-superclasses (append gen spir)
                           :metaclass 'person)))))
```

We are now in the position to define a method `precedence` for arguments of type `godhead` which returns a precedence list comprising all the persons belonging to the godhead. Note that this list is indeed constructed using the `topological-sort` function, this time applied to all those persons which do not have any direct sub-persons.

```
(defmethod precedence ((gh godhead))
  (when (not (clos:class-finalized-p gh))
    (clos:finalize-inheritance gh))
  (butlast (reduce #'(lambda (acc pers) (topological-sort pers nil acc))
                   (remove-if #'clos:class-direct-subclasses
                      (mapcar #'(lambda (person-def)
                                  (find-class (slot-definition-name person-def)))
                              (clos:class-effective-slots gh)))
                   :initial-value nil)
           2))
```

When looping through the persons which do not have any direct sub-persons, the method `precedence` proceeds in the order given in the definition of the godhead, thus transferring at least some amount of influence to the system's user. However, while we have kind of restricted ourselves here to a minimal version, many other aspects could have been implemented influencing the order in the precedence list. We name just a few, which underline the necessity for further theoretical investigations concerning the theological notion of subordination.

- Has a person been generated or spirated (or both)? And how often?
- How far is a person's $\prec$-distance from a 'top'-person, i.e. a person without any super-persons?
- Are there multiple paths to any top-persons?
- Of how many generations/spirations do these paths consist?
- Where (i.e. how far from the person) are these generations/spirations located?
- In how far do these considerations depend on Edwards' types of subordination (i)-(iv)?

In order to conclude these considerations, we exemplarily define the godhead belonging to the model depictured in Figure 2. Afterwards, we show the corresponding precedence list.

```
>>> (defgodhead god ()
      ((a)
       (b :generated-by (a))
       (c :spirated-by (a))
       (d :generated-by (b))
       (e :generated-by (c)
          :spirated-by (b))
       (f :spirated-by (c))))
#<GODHEAD GOD 2010A953>

>>> (define-persons 'god)
#<GODHEAD GOD 2010A953>

>>> (precedence (find-class 'god))
(#<PERSON F 21EB13A3> #<PERSON E 21EB1703> #<PERSON C 21EB1DFF>
 #<PERSON D 21EB1A87> #<PERSON B 21EB2207> #<PERSON A 21EB28BB>)
```

8. Conclusion

Regarding the variety of implementations offered in this paper one may wonder how much one could really benefit from actually running the written program on a computer when analysing and investigating models of generalised trinitarian logic. However, the main purpose of the developed implementations is of a quite different character: They serve as a means to bring as much precision and clarity to the field of philosophy of religion as possible and idealise by "representing patterns that satisfy higher standards of rationality than what most humans live up to" (Hansson & Hendricks 2018, 16). And while these "higher standards" are often sought in logical and mathematical languages, computer science can offer the wide range of programming languages as well as the field of algorithmics as further twists to formal philosophy.

As we have been able to see, it is not only theology which benefits from these formalisations. In the opposite direction, a new way of using metaobject protocols has emerged: The protocol is not used in order to adjust the underlying object-oriented language, but rather in order to utilise a huge part of the system which reveals an amazing analogy to the considered theological models. These new techniques may now quite easily be transfered to other complex structures in computer science, like graphs, grammars, automata, and the like.

I finish with Mullins (2020) and "hope that my analytic reflections [...] can help to bring further clarity and intellectual rigor to the unfinished task of Trinitarian theorizing."

Funding and Acknowledgements

The research for this article was conducted within the Gödeliana project led by Jan von Plato, Helsinki, Finland. The project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant Agreement No. 787758) and from the Academy of Finland (Decision No. 318066).

I thank the anonymous referee for the inspiring comments which helped to clarify some of the central points.

Bibliography

Benzmüller, C., & Woltzenlogel Paleo, B. 2014. "Automating Gödel's ontological proof of God's existence with higher-order automated theorem provers." In:

ECAI 2014, volume 263 of *Frontiers in Artificial Intelligence and Applications*, 93–98. IOS Press.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. 2009. *Introduction to Algorithms* (3rd ed.). Cambridge, Mass.: MIT Press and McGraw-Hill.

David, E., David, R. S., Gabbay, D. M., Schild, U. J. 2020. "Talmudic Norms Approach to Mixtures with a Solution to the Paradox of the Heap: A Position Paper." In: Silvestre, R. S., Göcke, B. P., Béziau, J. Y., & Bilimoria, P. (eds.) 2020a. *Beyond Faith and Rationality – Essays on Logic, Religion and Philosophy*. Sophia Studies in Cross-cultural Philosophy of Traditions and Cultures, vol 34. Cham: Springer. 173–193. https://doi.org/10.1007/978-3-030-43535-6_11.

DeMichiel, L. G. 1993. "An Introduction to CLOS." In *Object-oriented Programming: The CLOS Perspective*, edited by A. Paepke, 3–27. Cambridge, Mass.: The MIT Press.

Edwards, M. 2020. "Is Subordinationism a Heresy?" *TheoLogica: An International Journal for Philosophy of Religion and Philosophical Theology*, 4(2): 69–86. https://doi.org/10.14428/thl.v4i2.23803.

Effingham, N. (2018). "The philosophy of filioque." *Religious Studies*, 54(03): 297–312. https://doi.org/10.1017/S0034412518000264.

Gabbay, D. M., Schild, U., David, E. 2019. "The Talmudic Logic Project, Ongoing Since 2008." *Logica Universalis*, 13: 425–442. https://doi.org/10.1007/s11787-019-00228-y.

Hansson, S. O., Hendricks, V. F. (eds.) 2018. *Introduction to Formal Philosophy*. Cham: Springer. https://doi.org/10.1007/978-3-319-77434-3.

Keene, S. E. 1989. *Object-Oriented Programming in Common LISP: A Programmer's Guide to CLOS*. Boston: Addison-Wesley.

Kiczales, G., des Rivieres, J., & Bobrow, D. G. 1991. *The Art of the Metaobject Protocol*. Cambridge, Mass.: The MIT Press.
https://doi.org/10.7551/mitpress/1405.001.0001.

Kiczales, G., Ashley, J. M., Rodrigues Jr., L H., Vahdat, A., & Bobrow, D. G. 1993. "Metaobject Protocols: Why We Want Them and What Else They Can Do." In *Object-oriented Programming: The CLOS Perspective,* edited by A. Paepke, 101–118. Cambridge, Mass.: The MIT Press.

Lethen, T. 2021. "A Talmudic Norms Approach to many-valued Logic." *Journal of Logic and Computation*, 31(5): 1195–1205. https://doi.org/10.1093/logcom/exab027.

Lethen, T. 2022. "Gödel's Modal Dogmatic Logic and the *Filioque*: A Case Study." *Logique et Analyse* (accepted manuscript).

Molto, D. 2017. "The Logical Problem of the Trinity and the Strong Theory of Relative Identity." *Sophia*, 56(2): 227–245. https://doi.org/10.1007/s11841-017-0612-y.

Mullins, R. T. 2020. "Trinity, Subordination, and Heresy: A Reply to Mark Edwards." *TheoLogica: An International Journal for Philosophy of Religion and Philosophical Theology*, 4(2): 87–101. https://doi.org/10.14428/thl.v4i2.52323.

Oppenheimer, P. E., & Zalta, E. N. 2011. "A computationally-discovered simplification of the ontological argument." *Australasian Journal of Philosophy*, 89(2): 333–349. https://doi.org/10.1080/00048401003674482.

Ott, L. 1957. *Fundamentals of Catholic Dogma*. Cork: Mercier.

Paepke, A. (ed.) 1993a. *Object-oriented Programming: The CLOS Perspective*. Cambridge, Mass.: The MIT Press.

Paepke, A. 1993b. "User-Level Language Crafting: Introducing the CLOS Metaobject Protocol." In *Object-oriented Programming: The CLOS Perspective,* edited by A. Paepke, 65–99. Cambridge, Mass.: The MIT Press. https://doi.org/10.7551/mitpress/5087.001.0001.

van Peursen, W., & Talstra, E. 2007. "Computer-assisted analysis of parallel texts in the Bible. The case of 2 kings xviii-xix and its parallels in Isaiah and chronicles." *Vetus Testamentum*, 57(1): 45–72. https://doi.org/10.1163/15685337X167855.

Rushby, J. 2018. "A mechanically assisted examination of begging the question in Anselm's Ontological Argument." *Journal of Applied Logics – IFCoLog Journal of Logics and their Applications*, 5(7), 1473–1497.

Silvestre, R. S., Göcke, B. P., Béziau, J. Y., & Bilimoria, P. (eds.) 2020a. *Beyond Faith and Rationality – Essays on Logic, Religion and Philosophy*. Sophia Studies in Cross-cultural Philosophy of Traditions and Cultures, vol 34. Cham: Springer. https://doi.org/10.1007/978-3-030-43535-6.

Silvestre, R. S., Göcke, B. P., Béziau, J. Y., & Bilimoria, P. 2020b. "Beyond Faith and Rationality." In *Beyond Faith and Rationality: Essays on Logic, Religion and Philosophy*, edited by R. S. Silvestre et al., 3–15. Sophia Studies in Cross-cultural Philosophy of Traditions and Cultures,Cham: Springer.
https://doi.org/10.1007/978-3-030-43535-6.

Steele Jr., G. L. 1990. *Common Lisp: The Language* (2nd ed.). Newton, Mass.: Digital Press.