Univalent Foundations of AGI are (not) All You Need

Alexey Potapov(✉) and Vitaly Bogdanov

SingularityNET Foundation, Amsterdam, The Netherlands
{alexey, vitaly}@singularitynet.io

Abstract. We consider homotopy type theory (HoTT) as a possible basis for Artificial General Intelligence (AGI) and study how it will frame the traditional problems of symbolic Artificial Intelligence (AI), which are not avoided, but can be addressed in a constructive way. We conclude that HoTT is suitable for building a language of a cognitive architecture, but it is not sufficient by itself to build an AGI system, which should contain grounded types and operation, including those that alter already defined types in a not strictly provable (within available types themselves) way.

Keywords: AGI · HoTT · Symbol grounding · Subsymbolic

1 Introduction

Artificial Intelligence (AI) in general and Artificial General Intelligence (AGI) in particular have deep connections with foundations of mathematics and computer science (models of computation, formal languages, etc.). Traditional foundations of mathematics based on set theory and predicate logic heavily influenced the mainstream of XX century AI with Prolog as a prominent example of simultaneous intersection between foundations of mathematics, AI, and programming languages. More broadly, Good Old-Fashioned AI (GOFAI) is referred to as symbolic due to the physical symbol system hypothesis (PSSH) [1]. For instance, Lisp relies not on a predicate calculus, but on lambda calculus, which is in essence a formalism for manipulating symbols.

One of the main branches in the field of AGI is that of 'cognitive architectures' (CAs). Development of CAs typically leads to (or even starts with) choosing or inventing a computational formalism (which can turn into a full-fledged yet domain-specific programming language), a sort of 'language of thought'. This opens up ample possibilities for bringing fundamental mathematics and computer science to AGI.

However, symbolic AI is considered to be fragile, prone to the frame and symbol grounding problems, and is opposed to subsymbolic AI, which includes most notably deep neural networks (DNNs) that have begun to dominate the AI field – in particular for machine learning, computer vision and natural language processing. The share of DNNs in AGI is also increasing. But do they reject (symbolic) foundations of mathematics and build on something different and novel? Apparently, they rely on areas of traditional mathematics with set-theoretic and predicate logic foundations. One may

B. Goertzel et al. (Eds.): AGI 2021, LNAI 13154, pp. 184–195, 2022.
https://doi.org/10.1007/978-3-030-93758-4_19

also note that both PSSH and its criticism consider computers as such to be physical symbol systems (and even DNNs are split into logical operations at the bottom).

Meanwhile, there are new candidates to the foundations of modern mathematics. For example, the author of [2] cites Bertrand Russell: "Modern logic, as I hope is now evident, has the effect of enlarging our abstract imagination, and providing an infinite number of possible hypotheses to be applied in the analysis of any complex fact. In this respect it is the exact opposite of the logic practised by the classical tradition," and proposes homotopy type theory (HoTT, [3]) as "philosophy's new new logic". If "new logic" was so fruitful for philosophy, AI, and computing, couldn't "new new logic" contribute to AGI?

Indeed, category theory and type theories (which can serve as alternatives to set theory as a foundation of mathematics) already contributed a lot into computer science and the design of programming languages. They are also far from being ignored in AGI (e.g. [4, 5]), but they are no close to being mainstream as well. Apparently, since the existing incarnations of dependent and homotopy type theories as programming languages such as Agda, Idris, Coq have the most straightforward use as proof assistants, they may seem too GOFAI-ish in the context of AI.

At the same time, the necessity for neural-symbolic integration has been recently recognized even in the deep learning community. "A sound reasoning layer", "certain manipulation of symbols" is necessary even for applied AI systems (see, for example, [6]), so symbolic AI is regaining attention.

We are rethinking the design of the OpenCog cognitive architecture (e.g., [7]), which contains essentially symbolic components like the Atomspace knowledge base, the metagraph Pattern Matcher, the Unified Rule Engine, but which was also successfully used to build neural-symbolic systems (e.g. [8]). In this paper, we explore the possible role of HoTT in AGI and discuss symbolic/subsymbolic dichotomy as it relates to this example.

2 Symbolic Systems and Mathematics

Any calculus is typically considered as a bunch of rules for manipulating symbols syntactically without relying on their meaning (semantics). But what do we really imply by this? For example, lambda calculus has a few rules for manipulating symbolic expressions of form

```
<expr> ::= <constant> | <variable> | <expr> <expr> | λ<variable>.<expr> | (<expr>)
```

This description of syntax already involves some meta-symbols. If we try to describe syntactically the lambda calculus rules such as β-reduction, we will have some difficulties. Typically, it is written as $((\lambda x.M)\ E) \rightarrow (M[x := E])$. But as a purely symbolic expression it means nothing and does nothing. We can try to describe the content of this rule in more detail, but we will inevitably end up with some symbols, for which we suppose some meaning. While calculi deal with symbols syntactically, they themselves rely on semantics. Even if we describe a calculus formally as a collection of syntactically defined rules, this description will rely on symbols of meta-language with

semantic grounding. This grounding can reside in the minds of humans, who understand what these symbolic descriptions of, say, lambda calculus means, or it can take the form of a piece of code running on a certain computer (or another physical device).

It is impossible to completely get rid of grounded symbols in mathematics, but mathematicians have tried to reduce their usage as much as possible (and represent them in the form of alternative foundations of mathematics). The bare minimum is a set of instructions for a certain computer or non-reducible operations of any equivalent definition of the algorithm. Whether this minimum is really enough is disputable. Mapping theorem proofs into, say, Turing machines is not convenient in practice. Working mathematicians don't always use this level of strictness. Also, classical foundations of mathematics are not constructive. This doesn't necessarily mean that the mentioned minimum is not enough, since we hope to implement AGI as a computer program. It may also mean that the semantics even of foundational mathematical concepts are not clear enough and not fully understood by humans themselves.

Indeed, both the notion of sets in set theory and the notion of truth in predicate logic (in the sense of both their computational and semantic groundings) are problematic, which was shown by numerous paradoxes (especially, for naïve set theory). These paradoxes in turn, served to motivate constructive mathematics. However, constructive mathematics was too restrictive, and didn't allow for many proofs that looked natural in traditional mathematics.

HoTT doesn't completely reject useful but non-constructive axioms such as the law of excluded middle or the axiom of choice, but determines circumstances in which they hold constructively. HoTT attempts to formalize mathematics with computer proof assistants readily providing a computational framework for representing mathematical statements and manipulating with them. Thus, it is an interesting candidate for novel foundations of at least symbolic AI. A few attempts have already been made to adopt dependent and HoTT for knowledge representation (e.g. [9, 10]), but no work has considered the implications of HoTT for symbolic AI and cognitive architectures in the AGI context.

3 HoTT and Symbol Grounding

HoTT Primitives

In HoTT, we cannot introduce a symbol (variable), without indicating its type. $a:A$ is a judgement in HoTT, while $a \in A$ is a proposition in set theory. Of course, $a:A$ is not just a meaningless symbolic expression. It has a predefined interpretation in HoTT (both computational and semantic). Definitional (judgmental) equality is another primitive in HoTT. The symbol $\equiv$ for defining equality has also a predefined special meaning.

There is also propositional equality written as $x =_A y$, which implies that we can chain definitional equalities (within type A) together and transform x to y. An interesting aspect is that propositional equality is itself a type, whose elements are proofs of equality of x and y (if this type is inhabited).

However, in order to get non-trivial equalities, some computation rules are necessary. For example, in order to infer that $(\lambda x.x + x)(2) \equiv 2 + 2$ we need to have

β-reduction, which can be treated as a definitional equality $(\lambda x.\ \Phi)(a) \equiv \Phi'$, but with Φ' represented not symbolically, but computed by a grounded function, which replaces all occurrences of x in Φ by a avoiding name collisions.

In the AGI context, it is natural to ask where these judgements and definitions come from. Application of HoTT to Automated Theorem Proving (ATP) supposes that they are provided by mathematicians. They pick judgements interesting to them as true a priori, by human's definition. HoTT can be used to define arbitrary theories and deduce consequences from them. However, the question of adequacy of such theories to reality is not considered. While it is not a (notable) problem in ATP, it becomes very important if we consider HoTT for knowledge representation in AGI.

Constructable = Existent

The book [2] discusses the possibility of using (modal) HoTT as a new logic for philosophy, and natural language especially, pointing out the controversy regarding whether each common noun should denote a type or whether they should be formed by predication on some master type Entity. Consider a very simple sentence "John is a man". It is natural to consider Man as a type and John as its instance defining `John: Man`. However, it will not be a proposition in such a form and cannot be negated or appear in conditionals. Representing this expression as a dependent sum `(John, r):` $\sum_{\texttt{x:Entity}}$ `Man(x)`, where `r: Man(John)` doesn't have this problem.

The author of [2] extends this idea by considering such intermediate dependent types as `AnimatedObject`, which provide additional context for propositions like "John is a man" and prohibit reference to John as, say, a meteorological event. However, even flexible use of dependent types doesn't eliminate the presence of judgements and doesn't answer the question of where these judgements come from. How are `r: Man(John)` and `John: AnimatedObject` different from just `John: Man` in terms of necessity for introducing new members of types in runtime? Let us consider this question on the fully constructive level of program code (using Idris-like syntax).

Consider the classical syllogism as a slightly more complex example: All humans are mortal. Socrates is a human. Therefore, Socrates is mortal. What are the types in this case? Should Socrates be a type, a type constructor, a variable, a function?

If we write `Socrates: Human`, which seems natural, it implies that `Socrates` is a variable of type `Human`, and we need to construct its value. We could have one constructor `data Human = HumanC` and write `Socrates = HumanC`. However, any two such variables will be equal (due to `refl` x`:` $x =_A x$). In fact, there will be only one member of `Human` and this isn't what we want. Rather, we would prefer to have `Socrates` as a type constructor. This can be a question of syntax, but in dependently typed languages we cannot write `Socrates: Human`separately. Instead, we should collect all type constructors together:

```
data Human : Type where
     Socrates : Human
     Plato : Human
```

or, with syntactic sugar, `data Human = Socrates | Plato | …`, which will be the simplest sum type. Then, we can write `x: Human, x = Socrates`, and two variables with values `Socrates` and `Plato` will not be equal.

But what should be done in a situation where a previously unknown person appears? We will have to extend the existing type with a novel constructor so that we will be able to distinguish this person from other persons. Distinctions can be considered as one of very basic ontological notions [11], which has quite an interesting connection to equality in HoTT, although we will not explore this connection here. The issue we consider here is that type theories don't provide computational operations for altering types. But do we need these operations with dependent types?

Even if `Human`is defined as a dependent type, it still needs to have constructors:

```
data Human : Entity -> Type where
     SocratesIsHuman : Human Socrates
     PlatoIsHuman : Human Plato
```

together with `data Entity = Socrates | Plato`. Any new entity or witness will require new constructors.

We can try avoiding enumeration of all members of `Entity` or `Human` types by building them on top of some infinite type, whose members will serve as features of entities (e.g. name strings). Will it work?

We can either have a parameterized constructor or a dependent type.

```
data Human : Type where
     HumanName : String -> Human
socrates : Human
socrates = HumanName "Socrates"
```

or

```
data Human : String -> Type where
     HumanName : Human s
socrates : Human "Socrates"
socrates = HumanName
```

In the first case, we can construct any number of members of `Human`, while in the second case, we can construct any number of types `Human x` for some concrete `x`.

Such representations are convenient for databases. However, the question is how we interpret these definitions. Do we assume that all possible instances of `Human` with all possible names really exist, or do they exist in possible worlds?

Let us consider a proposition "All humans are mortal". Such propositions are expressed via functional types. For our simple types, it should look like

```
f : Human -> Mortal
```

but it will be trivially true for inhabited types independent of connections between `Human` and `Mortal`. If we directly map this proposition to a type, it should look like

```
f : Human x -> Mortal x
```

The proposition expressed in this type is true if we can provide an implementation of a total function, which will work for any `x`. Let us proceed with the representation in which `Human x` is a dependent type also.

If `Mortal x` is defined similarly to `Human x`, then such function trivially exists:

```
f : (Human x) -> (Mortal x)
f HumanName = MortalName
```

Then, we can write

```
socrates : Human "Socrates"
socrates = HumanName
```

and we can prove, by declaring and providing an instance of type `Mortal` "`Socrates`":

```
y : Mortal "Socrates"
y = f socrates
```

It follows from our definitions that "all humans are mortal" is a mathematical rather than empirical truth. If we consider only created instances as existent and creatable instances as potentially existent, then we should not consider the possibility of constructing a proof as a proof. Rather, we should consider something like `f HumanName = MortalName` not as a proof, but as a possible knowledge base entry, which we shouldn't arbitrarily add to our knowledge base, but should prove it empirically before adding. This doesn't correspond to semantics of dependent types.

Thus, an attempt to declare types in such ways that we can create their instances, which factual existence is not provided a priori, forecloses the possibility of considering types as propositions, whose inhabitants are their proofs.

Let us note that modal HoTT doesn't avoid the necessity of alteration of types, because propositions can change, say, from possible to necessary with new information.

Syllogism Example

If we define `Human` by enumerating all known humans, then implementation of `f` will be total only if mortality of all these humans is already known. It can be done via enumerating all facts of all mortal entities as constructors of `Mortal`. However, we can have a general definition of human mortality as prior knowledge.

In this context, it doesn't matter too much if `Human` is defined as a plain sum type, or dependent sum over `Entity` or `String`. In all cases we should enumerate all humans known to the system, e.g.

```
data Human : String -> Type where
     SocratesIsHuman : Human “Socrates”
     PlatoIsHuman : Human “Plato”
```

It looks conceptually better to have an `Entity` type with distinct entities as members, and `Name` as one of the dependent types describing their properties. It might be a minor point, but let us stick to this option.

```
data Entity = Socrates | Plato
data Human : Entity -> Type where
     SocratesIsHuman : Human Socrates
     PlatoIsHuman : Human Plato
data EntityName : Entity -> String -> Type where
     SocratesIsSocrates : EntityName Socrates “Socrates”
     PlatoIsPlate : EntityName Plato “Plato”
```

Let us note that we could define a more restrictive `HumanName` type, and `EntityName` could have a constructor that utilizes `HumanName` in a general way. More interesting are the constructors that the type `Mortal` should have. They can be defined as

```
data Mortal : Entity -> Type where
     SocratesIsMortal : Mortal Socrates
     PlatoIsMortal : Mortal Plato
```

Then, the proposition that all humans are mortal can be proven by listing all mortal humans, because there is no other way to construct `Mortal x` other than by providing a concrete constructor:

```
f : Human x -> Mortal x
f SocratesIsHuman = SocratesIsMortal
f PlatoIsHuman = PlatoIsMortal
```

A dependently typed language compiler will check that the function is total and that we didn’t miss any case. Alternatively, we can have the rule that all humans are mortal as a constructor

```
data Mortal : Entity -> Type where
     HumanIsMortal : Human x -> Mortal x
```

Therefore, the proof of the corresponding proposition is trivial (and redundant)

```
f : Human x -> Mortal x
f = HumanIsMortal
y : Mortal Socrates
y = f SocratesIsHuman
```

Thus, we can prove that `Mortal Socrates` is populated given the evidence (minor premise) `Human Socrates` and the major premise `Human x -> Mortal x`. Everything is represented by type constructors.

Our representation of the syllogism looks natural, but it implies that meeting any new person (or acquiring new information) as well as generalizing particular facts will require altering types. This situation is not essentially different from the problem of closed worlds in other logical systems. However, a conceptually interesting question arises – if there is a mathematically sound way of introducing new types or their constructors, shouldn't it be expressible in HoTT and implementable in a corresponding programming language?

Distinction and Identification

Many dependently typed languages inherit Haskell syntax in declaring types that obscures their nature. Why do we need special syntax and `data` keyword? We would like just to declare `Entity: Type`, and then to use definitional equality, e.g., `Entity = Socrates | Plato`. This looks natural because we define one type using such operations over types as a sum or product (which can also be dependent sums and products). But sum operands are types. `Socrates` and `Plato` are not types. In category-theoretic interpretations, they are (names of) functors `() -> Entity`, which correspond to certain elements of `Entity`. `Entity` is a sum of unit types, which is mathematically sensible, but not adequately expressive for knowledge representation.

In Coq, which is used concretely for HoTT, type definitions look like

```
Inductive nat : Type :=
  | O : nat
  | S : nat → nat.
```

The above looks more like a definitional equality. However, type `nat` is not defined via other members of `Type` and expressions over them. Symbols `O` and `S` are directly introduced as just having types `nat` and `nat → nat` correspondingly.

`nat: Type` is not definitionally equal to anything else, although it could be defined as a sum of `()` and `nat → nat`, which would indeed reduce to primitive operations on types. An inductive definition constructively (finitely) defines an infinite type, for all members of which we can prove common properties. However, it will obscure the fact that we want to construct expressions like `O` or `(S (S O))` as having the type `nat`. Let us forget mathematical interpretations for a moment and look at these as symbols. We introduce symbols `O: nat` and `S: nat → nat`, for which there are no definitional equalities. If there were an interpreter of symbolic expressions, it could interpret these symbols only to themselves.

We can compose various expressions from symbols. We have this capability prior to types and their constructors. Expression typing imposes restrictions on valid expressions. For example, `nat: Type` allows `nat` to appear on right-hand sides of colon expressions (and some other places, where `Type` members are valid, e.g. as an argument of a symbol, which type is `Type -> _`).

What is the difference between `S: nat → nat` as «a constructor» from a function with the same signature? `S` doesn't have a body. There is no definitional equality for it. It is not further reducible.

Functional languages with type systems use constructors also for checking totality of functions. For example, if we write

```
f : nat → nat
f 0 = (S 0)
```

we can conclude that `f` is not total. We could make no distinction between a type constructor and a non-total function. Such a function does not reduce further in such cases, e.g. `f (S 0)` would be well-typed but non-reducible, similarly to `S (S 0)`. In the context of AGI, this may make sense. A toddler can know that `2 + 2 = 4`, but can have no idea about `4 + 4`, although knowing that `4` is also a number and can be added, and knowing that 4 apples and 4 apples are `4 + 4` apples, so + will act both as a function and as a constructor. However, separating a complete inductive definition of a type from variables and functions is very convenient for constructing proofs by induction, which is one of the main features of HoTT.

Nevertheless, in an open system, what is initially a constructor can get a definitional equality. For example, the system is told about some Jonny, and it introduces new entity `Jonny: Entity`, `JonnyName: EntityName Jonny` "`Jonny`". Then, it appears that Jonny is John, whom it already knew. It just needs to identify them `Jonny = John`. This definitional equality can be propagated to various propositional equalities. Thus, two basic operations are distinguishing and identification of symbols and expressions.

Grounded Types and Non-Provability

Information doesn't come from nowhere. Rather, it is communicated to the system via certain interfaces, which might be good to formalize. Functional languages typically do so by using `IO Monad`. However, it separates pure code from side effects rather than answers the question of how pure code emerges from external information.

Instead of talking about arbitrary program I/O, let us consider an agent with some sensors. We may try supposing that we know the type of data coming from these sensors. Can we indeed know? Suppose that we have a video sensor. An observation at moment `t` will be $x_t: \mathbb{R}^{N\times M}$. However, as we articulated, everything constructible is not just possible, but existent. It is incorrect to say that any member of $\mathbb{R}^{N\times M}$ of exists as sensory input, so the type of sensory input cannot be equal to $\mathbb{R}^{N\times M}$. It differs from the set-theoretic representation, in which $x_t \in \mathbb{R}^{N\times M}$ would be a valid proposition.

Thus, it seems more correct to have a type `Observation`, whose constructors are not known to the agent, and instances of this type are constructed outside the agent. We refer to such types as grounded. Receiving a new observation can be thought of as

adding a new constructor or member to such a type, which can be generalized. Either the type system of our agent will contain no types corresponding to real-world entities besides prior knowledge (and a dependently typed language would just serve for interpreting another language to work with runtime information), which will suffer from the symbol grounding problem, or the agent must be capable of the act of forming a new type and populating it with new members and functorial constructors. What is a proposition at some level of reasoning can become a judgment at another level.

Impossibility of an absolute proof of inductive generalizations based on a finite number of arguments was realized already by Bacon. We can construct absolute proofs within consistent definitions of types, but they will be model assumptions, whose correspondence to reality will never be perfect. Such is the case for the scientific methodology in general, in which theories are constructed on the basis of available information and then experimentally verified together with their consequences. Interestingly, the criterion of falsifiability says that scientific theories should admit the possibility to be proven to be false in principle by new evidence. But shouldn't it be true about the scientific method itself? It cannot be formally proven, but it is supported by a huge corpus of knowledge and predictions resulting from its usage meaning its high degree of adequacy to reality.

Of course, we would like to have justified methods for machine learning. All such methods (including deep learning) are expressed mathematically and/or as program code. For some of them we have proofs of optimality, although under strong assumptions, making these methods narrowly applicable and inadequate to reality in general case. There are proofs of optimality for universal methods like Solomonoff induction, but these are, unfortunately, not constructive. There are also proofs that universal computable predictors cannot exist [12], though interpretation of such theorems in the context of building AGI is also debatable. Nevertheless, neither a universal and practically applicable induction method nor a constructive method of synthesis of efficient specialized algorithms is currently known.

Generally, AGI will unavoidably have a heuristic or non-axiomatic (in sense of [13]) element. This implies not only that it is not provably reducible to some deductive system, but also that it should contain some collection of algorithms (or, rather, subprograms, which are not necessarily provably terminating), which can be justified only partially, with the choice between them ultimately being empirical. Both the design of a language for CAs with the choice of base formalism (such as HoTT) and the amount of prior information (including built-in algorithms) are heuristic in nature.

There is some content of intelligence that cannot be derived a priori by 'pure reason'. This is obviously the case for concrete knowledge about the world, but it may not be as obvious that the methods of reasoning and learning also cannot strictly be provably optimal. While any prior judgement or axiom built into an AGI system should be falsifiable in principle, that can be considered as a possibility of this system to rewrite any piece of its code as in the Gödel machine [14], we believe that most nontrivial acts of self-improvement are not strictly provable (in terms of increase in expected future rewards). Rather, such improvements should be put forward and tested in practice, similarly to scientific hypotheses. It should be underlined that total functions, which type-check as members of types, which, in turn, serve as provable propositions, are not Turing-complete.

4 Conclusion

Homotopy type theory is attractive, because instead of using sets and meta-language of logic separately, it uses only types, which encompass both objects and propositions, which are also objects of higher types. This is convenient for AGI in terms of uniformly and constructively representing and reasoning about knowledge both of the external world and reasoning itself. Besides, type theories in the form of programming languages provide the convenient tool of pattern matching, which, combined with knowledge retrieval queries, could provide a basis for "language of thought".

However, such languages themselves cannot get rid of a few meta-language symbols for primitive grounded operations that cannot be described declaratively within languages themselves (besides directly referring to these symbols). For a real-world AGI, which doesn't work in a purely abstract domain, there should be additional grounded types, which stand for interactions with the environment. Definitions of such types are unknown a priori, and since constructability implies existence, constructors of grounded types should be added through observations. This implies that more abstract concepts as derived types can change as well. Any collection of types should be considered as a model that can be more or less applicable to reality rather than an absolute truth about it.

Methods for altering types can utilize available knowledge presented in already defined types, and can be described in the same language, but will unavoidably refer to some grounded symbols, which will make their use not strictly provable. Such grounded operations can be regarded as subsymbolic, although their granularity and quantity are research questions.

Acknowledgments. The authors are grateful to Ben Goertzel for useful references and ideas, which stimulated the study performed in the present paper. Thanks to Janet Adams and James Boyd for proofreading.

References

1. Newell, A., Simon, H.A.: Computer science as empirical inquiry: symbols and search. Commun. ACM **19**(3), 113–126 (1976). https://doi.org/10.1145/360018.360022
2. Corfield, D.: Modal Homotopy Type Theory: The Prospect of a New Logic for Philosophy, p. 191. Oxford University Press, Oxford (2020)
3. Homotopy Type Theory: Univalent Foundations of Mathematics (2013). arXiv preprint, arXiv: 1308.0729
4. Goertzel, B.: A formal model of cognitive synergy. In: Everitt, T., Goertzel, B., Potapov, A. (eds.) AGI 2017. LNCS (LNAI), vol. 10414, pp. 13–22. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63703-7_2
5. Phillips, S.: A general (category theory) principle for general intelligence: duality (adjointness). In: Everitt, T., Goertzel, B., Potapov, A. (eds.) AGI 2017. LNCS (LNAI), vol. 10414, pp. 57–66. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63703-7_6
6. Lamb, L.C., et al.: Graph Neural Networks Meet Neural-Symbolic Computing: A Survey and Perspective (2021). arXiv preprint, arXiv: 2003.00330

7. Goertzel, B., Pennachin, C., Geisweiller, N.: Engineering General Intelligence, Part 1 & 2. Atlantis press, Paris (2014)
8. Potapov, A., Belikov, A., Bogdanov, V., Scherbatiy, A.: Cognitive module networks for grounded reasoning. In: Hammer, P., Agrawal, P., Goertzel, B., Iklé, M. (eds.) AGI 2019. LNCS (LNAI), vol. 11654, pp. 148–158. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-27005-6_15
9. Dapoigny, R., Barlatier, P.: Using a dependently-typed language for expressing ontologies. In: Xiong, H., Lee, W.B. (eds.) KSEM 2011. LNCS (LNAI), vol. 7091, pp. 257–268. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25975-3_23
10. Lai, Z., et al.: Dependently Typed Knowledge Graphs (2020). arXiv preprint, arXiv: 2003.03785
11. Goertzel, B.: Distinction Graphs and Graphtropy: A Formalized Phenomenological Layer Underlying Classical and Quantum Entropy, Observational Semantics and Cognitive Computation (2019). arXiv preprint, arXiv: 1902.00741
12. Legg, Sh.: Machine Super Intelligence. PhD thesis (2008)
13. Wang, P.: On definition of artificial intelligence. J. Artif. Gen. Intell. **19**(2), 1–37 (2019)
14. Schmidhuber, J.: Gödel machines: fully self-referential optimal universal self-improvers. In: Goertzel, B., Pennachin. C. (eds.) Artificial General Intelligence. Cognitive Technologies, pp. 199–226. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-68677-4_7