On the Need for a General Language for

General Intelligence

Sudharsan Iyengar and Theron Rabe

siyengar@winona.edu

Computer Science Department

Winona State University

*Abstract*—General Intelligence can be assessed based on the accuracy, speed, and process with which one arrives at decisions. This is apparent in decision making processes where decision making is accomplished through deductive, inductive, and/or abductive reasoning. We first define the notion of a self evolving general language L , a superset of all languages. Additionally, we define and develop a process of reduction R, for improving the accuracy and speed of decision-making in L . Similar to natural languages, L is an incremental and self evolving language. Similar to intelligent processing, R accommodates all possible inputs. Finally, we present the limitations of λ-calculus with respect to L and propose remedies that provide us an implementation platform for L and R.

Keywords—Artificial General Intelligence; General Language; Reduction; lambda calculus;

# I. Introduction

The original goal of artificial general intelligence is to simulate intelligent behavior at or above human level [1]. This goal necessitates a means of comparison between intelligent agents. One such comparison method is to measure the speed, efficiency, and process with which the agent arrives at proper decisions [2]. In other words, when an agent is given some problem, its intelligence with respect to that problem is a combination of the amount of time it takes to find a solution, and how accurate that solution is. From this, we can argue that general intelligence should demonstrate speed and accuracy in solving not only pre-defined problems, but in general, an evolving scope of problems.

Humans use intelligence by reasoning over observations [2]. This reasoning falls under various forms of deductive, inductive, or abductive reasoning. If a person's repeated observations indicate that *A* must always imply *B* (i.e. *if A, then B* is learned), then upon subsequent observation of *A*, *definitely B* can be decided through deductive reasoning. Alternatively, if a person has at some time observed and learned that *A* possibly implies *B*, then upon subsequent observation of *A*, *maybe B* can be decided through inductive reasoning. Finally, if a person has learned through observation that *A* may imply *B*, then upon observing *B*, *maybe* *A* is decided using abductive reasoning. Since inductive and abductive reasoning are inherently uncertain, they are of particular interest for solving problems suited for general intelligence, which usually involve a high degree of uncertainty. Such problems include machine learning, pattern classification, natural language processing, statistical analysis, computer vision, and data compression [3,6].

Natural languages, the primary visible forum for intelligence, are incremental, self-evolving, self-mutating, and the evaluation of a statement using a language generates another statement in the same language. All decisions are based on previous experiences. Previously unknown expressions are either deemed irrelevant (and hence discarded) or considered acceptable new knowledge that is integrated into the language. Inductive and abductive reasoning requires earlier experience and decisions that help similar current decisions. For example, when a person (infers) induces that they will be hungry later because they did not eat breakfast, they must have previously experienced that not eating breakfast could cause hunger. The same can be said for a person who (feels) observes hunger and abductively attributes it to not having eaten breakfast. At the time of originally deciding that “not eating breakfast causes hunger”, a minimum of two critical observations should have occurred. First, the person must have observed that they did not eat breakfast. Second, they must have later observed hunger. Any number of other intermediate observations (like “drank coffee” or “watched TV”) may have occurred between these critical observations. Initially, all the relevant observations are included, considered, and processed in arriving at “not eating breakfast causes hunger”. The number of observations included in such intermediate decision making can be called the critical observations’ distance. Upon such repeated processing one tends to shorten the distance between observations in decision making. Thus, the act of correlating hunger to not eating breakfast is an act of shortening the distance between them (and pruning other insignificant observations).

When observations are later reasoned over, the observations that compose their distance may be ignored. For instance, if a person had “gone walking” after “not eating breakfast” and before becoming “hungry”, the observations made on “gone walking” may not contribute to making the intelligent decision that “not eating breakfast” causes “hunger”. The association of these two observations are utilized in inductive (forward or anticipatory) and abductive (backward or causal) reasoning. This process of learning, and subsequent shortening of the distance between cause and effect observations, is elementary in demonstrative intelligence.

Thus general intelligence possesses a primitive decision-making process that shortens the distance between critical observations. This process lowers the complexity of the decision-making process by removing unnecessary intermediate steps. In other words, an intelligent agent must be able to take an observation series *A→B→C*, and simplify it to decide that observation *A* may directly indicate *C* without regard to the presence of observation *B*, where applicable. We call this decision-making process reduction.

Section II defines and describes the properties of the reduction process. Section III defines and describes the general language. Section IV presents the limitations of λ-calculus with respect to a general language and presents our approach to modifying λ-calculus so as to implement the features described in this paper.

# II. Reduction - a way to process phrases in a language

Reduction is manifested at multiple layers of abstraction within intelligent thought. People use language as a means of abstracting their thought process. An observation, when abstracted by language, becomes a phrase. A phrase is either a symbol, or a sequence of symbols within a language. For example, “rain” is a phrase composed of a single symbol that represents the observation of water falling from the sky, in English. In the same way reduction permits simplification of non-critical observations, it permits simplification of their abstractions. When a complex phrase is interpreted by an intelligent agent, reduction can be applied to the phrase to shorten the relation between its sub-phrases or observations, thus simplifying the task of reasoning over their semantical correlations.

For example, let us take the sentence: “Enough humidity has gathered in the air as to generate clouds of an unmaintainable density” which could be interpreted to the phrase “It is raining”.

This sentence has multiple subphrases (observations) viz. enough humidity, gathered in the air, generate clouds, and unmaintainable density.

Upon reasoning, the phrase becomes simpler but interprets the same. By utilizing a ‘shortened’ version of the original phrase, one is able to simplify the semantic interpretation of the original phrase. In other words, the reduced version is faster to interpret. With respect to language, reduction is the translation of phrases to semantically-equivalent (or -approximately equivalent), but syntactically-minimal previously learned abstract phrases.

We now present the properties of such a reduction process. Correlation between phrases and semantics, when indicated, are presumed. The establishment and verification of the semantics to phrases are beyond the scope of this paper.

*Definition* 1: A language L is a tuple (T, N, G, S) where T is a set of terminal phrases, N is a set of non-terminal phrases, G is a grammar, and S is a semantics.

## Definition 2: Given a language L , a phrase P in L is a sequence of symbols of the form {s1, s2, .., sn} such that 0<i<n, ∀si ∈ P (si ∈ (T ⋃ N)). All members of the power set P (T ⋃ N) meet the definition of a phrase.

## Definition 3: Given a language L , its grammar G is a set of production rules, each of the form A→B, where A and B are phrases in L .

*Definition 4*:

*N* = {*A* | (*A*→*B*) ∈ *G*}

*T* = {*t* | (*t* ∉ *N*) ∧ (∃(*A→B*) ∈ *G* : (*t* ⊆ *B*) ᐯ (*t* ⊂ *A*))}

## Definition 5: Given a language L , its semantics S is a set of tuples (t, b) where t ∈ T, and b is an observation. An observation is some mechanical or logical effect on an L interpreter.

It is important to note that a phrase contains terminal and non-terminal symbols, but the semantics of the phrase is expressed by way of terminals only.

## Definition 6: Given a phrase P, the set of symbols used in P is denoted {P}. The distance of P is the cardinality of the set (P ⋂ N), denoted Pc.

## Definition 7: Given a language L , the evaluation of a phrase P in L , denoted P(), is a function such that:

*P*() = {*b* | ∀*t* ∈ (*P* ⋂ *T*) : (*t*, *b*) ∈ *S*} ⋃

{*P*’() | ∀*n* ∈ (*P* ⋂ *N*) : (*n→P*’) ∈ *G*}

Where *P*' is some partial evaluation of *P*.

An evaluation function correlates a phrase to its abstracted observations, thus causing a series of mechanical or logical effects on an interpreter. We argue that the evaluation of a phrase is dependent on the distance of the phrase. Terminals need no further reduction as they carry semantics.

## Definition 8: The complexity of an evaluation, denoted O(P()), is given as follows:

*O*(*P*()) = 1 if ∀*s* ∈ *P* : (*s* ∈ *T*)

*O*(*P*()) = *f*(*Pc*) if ∃*s* ∈ *P* : (*s* ∈ *N*)

where f is some mathematical function

## Definition 9: Given a language L , the reduction of a phrase P with respect to L , denoted R (P, L ), is a function such that R (P, L ) = p, where p is a phrase in L , and

R(*P*, L) = R(*p*, L)

*P*() = *p*()

*O*(*p*()) ≤ *O*(*P*())

First, that the reduction of phrase *P* is equivalent to the reduction of its reduction, *p*. That is, the reduction function is final. Second, that the evaluation of the phrase *P* will be equivalent to the evaluation of its reduction, *p*. In other words, reduction does not change the semantics of a phrase. Third, the complexity of evaluating the reduced phase is less than or equal to that of the original.

An input string is reduced in formal languages by iteratively applying the rewrite rules specified in the language's formal grammar, on an input string, until it cannot be further reduced. Since natural languages have no exact formal grammar, their reduction is more difficult to achieve. Reduction of a natural language depends on an accumulated familiarity with the phrases that constitute the language. The correlations and equivalences amongst these accumulated phrases behave as the language's grammar. Because reduction of a natural language depends on phrases having been learned and subsequently used in an meaningful way, natural language reduction appears indicative of intelligence.

Thus, to replicate this act of intelligence using artificial systems, the reduction process must be achievable in a language that is being prescribed through free use of previously unknown phrases that could become part of the language. Thus our proposal for a framework for a general language as opposed to a specific natural language. Since general intelligence processes must be applicable in broad domains we define a general language next.

# III. General Language

We note that the behavior of intelligence is dependent on what is known, understood, and utilized. Contrast this with an artificial system that can process phrases in the French language. This system is demonstratively limited in what it can accomplish because it is programmed as such, and it does not accommodate and/or learn other phrases. Humans on the other hand possess the ability to behave on what is assimilated, but additionally also accept and ingest new information, and thus evolve or grow. In fact, this is modus-operandi of human behavior. (Ironically, we consider this intelligent behavior and not the ability to process teraflops in milliseconds.) Importantly, note the language of a person is but that which has been assimilated and unrestricted, in contrast to what might be prescribed to be English, French, or the sign-language.

For the purposes of developing an intelligent machine we describe the notion of an unrestricted general language. This general language must satisfy the following three criteria:

* *General language must accept all possible phrases*
* *General language must be Turing-complete*
* *General language must be interpretable in-order*

Primarily, all potential phrases must be acceptable in the general language. This requirement implies that a general language has no predefined syntax rules. This is important as the order of the phrases is immaterial as long as the sentence is interpretable. Arguably, capability of interpretation without strict limitations on the order of the phrases, captures elementary intelligence. An example of this would be interpreting poetry as opposed to prose. Additionally, the general language must accept new previously unencountered phrases - as legitimate phrases. The interpretation of such phrases is subject to the intent of observations associated with the phrase and other considerations.

Secondly, the general language must have Turing-complete semantics, so as to enable inference of a type 0 grammar [7]. Given this feature, we can automate the grammar application of this language, giving us the possibility of developing an AGI system.

Thirdly, we note that intelligent behavior generally interprets observations as they are input - without the need for a pre-requisite forward (anticipatory) reference. As such, the general language must accomplish interpretation without a requirement of forward reference. This requirement is further explained.

Since, the general language lacks definite syntax rules, it must accommodate an infinite alphabet. An infinite set of symbols cannot be enumerated, as required for a formal grammar, but the set of contextually pertinent symbols can be. Consequently, during forward interpretation when a new symbol is encountered, the interpretation process must treat that symbol as a valid member of the language's alphabet in order to accept possible phrases with the new symbol.

## Remedy 1: Represent infinite alphabet through its encountered subset.

This simplification permits an interpreter to reason a partial formal grammar over an alphabet. Note that as a consequence, the interpreter must posses the ability to maintain a dynamic alphabet and grammar rules. As a general language interpreter is used, it will encounter an increasingly large set of phrases. As such, it must maintain a repository of phrases encountered so far, and utilize this repository in its future interpretations.

## Definition 10: A set of encountered phrases {p0..pn}, represents an interpreter’s history **P**.

Due to general language’s need to be interpreted in-order, a function defined within phrase *pi* must be expressed in terms relative to phrases *p0..i*. In other words, the semantics of some future phrase is determined by its relation to past encountered phrases. Therefore, ***P*** represents a learned subset of the general language, as expressed in terms of ***P***. This makes ***P*** an evolving construct analogous to a human’s understanding and use of natural language. For example, a person might equate the phrase “rain” to “water that falls from the sky”, but “water that falls from the sky” is just another phrase that can only be defined in terms of other learned phrases.

## Definition 11: ∀pi ∈ **P** (pi() = f(p0..pi))

where f is some computable function

Since a general language interpretation machine must be Turing-complete, it must support a means of defining and applying functions that support arbitrary recursion and abstraction. [4]

## Definition 12: ∀pi ∈ **P** (∃A ∈ **P** ∧ ∃B ∈ **P** : pi(A) = B)

for any decidable *pi*(*A*)

Therefore, a general language function is a means of rewriting arbitrary phrases into other arbitrary phrases, as derived exclusively from a set of encountered phrases. Because all formal grammars can be expressed as a set of phrase rewriting rules [7], all formal grammars can be directly derived from general language expressions. For this reason, deriving general recursive phrase→phrase rewrite functions by reasoning over ***P*** is equivalent to deriving a formal grammar for a language that contains all the same phrases as ***P***.

A machine that correctly interprets a general language, regardless of the semantics of that general language, will learn both the phrases and the grammar that constitute a subset of the general language. Since all languages are subsets of the general language, a general language interpreter can learn natural languages by interpreting an input that causes it to construct a ***P*** that is approximate to some desired natural language in both phrase content and grammar. Because reduction is a computable function for any language with a formal grammar and all computable functions may be contained in ***P***, approximation of a natural language via restriction of the general language permits reduction of that natural language with as much accuracy as permitted by the grammar defined in ***P***.

If semantics are defined for a general language, reduction of natural languages can be approximated. Reduction of a natural language is an act of intelligence that improves the speed and accuracy with which decisions can be made for problems with uncertain solutions.

We call for the need of a formal semantics for a general language. Given formal semantics for a general language, an abstract machine can be designed for evaluation of general language strings. A machine that evaluates general language has an inherent ability to learn, due to general language's requirement of an extensible alphabet. Furthermore, since the interpretation machine must be Turing-complete, it has the ability to derive and perform any computable function over its learned alphabet. Provided with the correct input string, an abstract machine that evaluates general language can learn both the phrases that constitute a natural language, as well as the functions that correlate those phrases within its language. Thus, a general language interpreter is capable of improving its intelligence with respect to any language, and therefore, any problem domain, through experience.

##### IV. A look at λ-calculus and its limitations

To address the semantics for the general language, and exemplify the ambiguities that arise in doing so, we start with a Turing-complete language, and progressively remove all syntax rules. We use λ-calculus [5] as the starting language.

To exemplify the ambiguities that arise from removing syntax rules from λ-calculus, we will examine three syntactically invalid λ-expressions:

1. λ*xyz.a*
2. λλ*x.F.a*
3. λλ*x.xy.a*

Expressions (1), (2), and (3) each define a function whose body is composed of the symbol a and whose abstraction declaration contains syntax errors. Thus, in order for λ-calculus to meet the requirements of the general language, its semantics must be altered in such a way that each of these expressions is syntactically valid and unambiguously outputs the symbol *a*.

Expression (1)'s abstraction declaration contains three symbols (*x*, *y*, *z*) where only one is allowed by λ-calculus' formal grammar. To make this syntax valid, we suggest modifying λ-calculus such that a function with multiple symbols between λ and '.' is semantically equivalent to its fully curried version.

## Remedy 2: λS.a = λs1.λs2. … λsn-1.λsn.a

for any sequence *S* of symbols *s1..sn*

With this modification, Expression (1) becomes syntactically valid. And given any three inputs, Expression 1 retains unambiguous output of symbol *a*.

Expression (2) contains two consecutive λ symbols, so it can be referenced in parts. Call part “λ*x.F* ” the inner function, and everything else the outer function. Let *F* to be some oracle function that returns either symbol *a* or symbols *xy*. The output of *F* becomes the output of the inner function, which by way of Remedy (1) becomes the abstractions used by the outer function. Should *F* return symbol *a*, the outer function no longer outputs symbol *a*, and instead behaves as the identity function. Although the behavior of Expression (2) may arbitrarily change, it remains unambiguous in either definition it is dynamically given. We suggest the acceptance of semi-decidable function definitions by means of evaluating all definitions. Since definition is a prerequisite of application, any definition must be evaluated before its function can be applied. Because a function could potentially be applied immediately after definition, the expression containing its definition must be evaluated in-order.

## Remedy 3: ∀pi ∈ P (pi() = p1(), p2(), …, pi-2(), pi-1())

Where f is some computable function, and *pi* is a sub-phrase of phrase *P*

Strings must be evaluable in-order.

Expression (3) also appears to have an inner and outer function. Ambiguously, the inner function may consist of either λ*x.xy* or λ*x.x*, depending on which function (inner or outer) owns symbol *y*. Should the inner function be provided another function for input *x*, that function *x* may be applied to one of two input sources, and in one of two orders. A function abstracted by *x* may be applied to *y*, or to whatever expression follows that which provided *x*. Additionally, that application may occur either before or after *y* has been provided with an expression to abstract. Which of these evaluation patterns is taken affects Expression 3’s ability to output symbol *a*. To correct this ambiguity, we suggest marking both the start and end of both function definitions and function inputs with dedicated symbols.

## Remedy 4: λx.y z → (λx.y) [z]

By using these symbols purposefully and without restriction we can preserve the general language's first requirement (lack of syntax rules) and prevent ambiguity. This language is Turing complete and thus implementable on a computer.

##### V. Current Work & The Eesk Programming Language

We have designed and implemented a high-level programming language Eesk that attempts to be a general language. The Eesk system behaves as a lambda calculus interpreter that has, for the most part, remedied the ambiguities related to the double-lambda problem described above. With a few exceptions, this language meets all the three criteria of the general language.

The Eesk runtime environment has shown equivalent to an abstract machine that performs reduction on arbitrary learned languages for all halting inputs that have been tested. We intend to continue developing this system to use as a framework for further investigating the use of general language reduction as an approach to improving both the speed and accuracy of artificial general intelligence.

As with any correct implementation of the general language, Eesk’s syntax is arbitrary. Valid Eesk is defined as any sequence of symbols. Conceptually, any symbol is either of the terminal or non-terminal type. Operators may be treated as terminal symbols. Operators that may be applied to an operand of one type may equally be applied to any operand of the other type. Thus, the language is weakly and dynamically typed. Since the typing is implicit, automatic, and prone to change, it does not necessarily concern an Eesk programmer.

Similar to other homoiconic functional languages like Scheme and Racket [8,9], Eesk is lexically scoped and full funarg [10] capable. The availability of symbols to their sub- and super-scope can be explicitly decided using “public” and “private” modifiers. Declaration of new symbols is done implicitly upon first encounter, defaulting to accessibility for all sub-scopes, but not the super-scope.

Due to general language’s third requirement, Eesk may be parsed by a means as simple as LL(1) [11]. Each symbol encountered by such a naive left-to-right parser could be translated directly into machine code without respect to what symbols come next. The current implementation however, uses a recursive descent approach instead. Each descent may be implicitly escaped by encountering the end of a symbol stream. This solution permits much of the computational expense associated with determining scope to be handled at compile time.

To accommodate the remedies prescribed in this paper, Eesk employs a runtime architecture composed of three stacks, separating it from the list-processing approaches taken by philosophically similar languages [8, 9, 14]. The first of these stacks is used to store intermediate computed symbols, and the second to store function arguments. The Eesk calling convention causes these first two stacks to exchange responsibilities. This stack rotation method allows Eesk functions to both accept and produce syntactically arbitrary Eesk expressions without causing stack corruption. Furthermore, stack rotation permits the elements belonging to many sequential dynamic data structures to be accessed in constant time.

Eesk’s third stack maintains control information for the calling convention, and its presence is opaque to an Eesk programmer. The third stack can be modeled using only the first two stacks, but in doing so, the runtime environment loses constant-time lookup of symbols in the super-scope.

Through the remedies provided in this paper, Eesk is a reflective language in which syntax is a first class citizen, and reduction of syntax is the primary mode of evaluation. Eesk expressions can be dynamically generated and evaluated by means of reduction. Beyond the primitive operators suggested for a pure reduction system, Eesk delivers additional predefined (but overridable) operator symbols that permit pattern matching between expressions, similar to use of (quote …) and (match …) in some languages [8,9] of LISP [14] heritage. Also, through intentional placement of function application operators, an Eesk programmer can explicitly denote whether a function is evaluated eagerly or lazily [12]. Additional features provided by the Eesk language framework include first class citizenship of continuations [13] and a foreign function interface.

##### VI. Conclusion

We have defined complementary tools of reduction and general language that characterize general intelligence in language processing. The process of reductions is aimed at simplifying the complexity of decision-making over uncertain problem domains. The beneficial and problematic implications of implementing such a framework is discussed. The use of λ-calculus, and suggestions for modifying its syntactic structure to make it suitable for use as the general language, are presented as well. We are calling on the need for the formulation of formal semantics of the general language as an approach to general intelligence.

##### VII. References

1. Allen Newell and Herbert A. Simon. 1976. Computer science as empirical inquiry: symbols and search. Commun. ACM 19, 3 (March 1976), 113-126. DOI=10.1145/360018.360022 http://doi.acm.org/10.1145/360018.360022
2. Sudharsan, Iyengar. Cognitive Primitives for Automated Learning, Frontiers in Artificial Intelligence and Applications, Vol. 171, AGI 2008, pp. 409-413.
3. Duda, Richard O., David G. Stork. Pattern Classification (Pt.1). (09 November 2000)
4. Turing, A. M. Computability and λ-Definability. The Journal of Symbolic Logic. Vol. 2, No. 4 (Dec., 1937), pp. 153-163
5. Church, Alonzo. An Unsolvable Problem of Elementary Number Theory
6. Murphy, Kevin P. Machine Learning: A Probabilistic Perspective (Adaptive Computation and Machine Learning series) (24 August 2012)
7. Chomsky, Noam. On Certain Formal Properties of Grammars. (1959)
8. Gerald Jay Sussman and Guy L. Steele, Jr.. 1998. Scheme: A Interpreter for Extended Lambda Calculus. Higher Order Symbol. Comput. 11, 4 (December 1998), 405-439. DOI=10.1023/A:1010035624696 <http://dx.doi.org/10.1023/A:1010035624696>
9. Matthew Flatt. 2012. Creating languages in Racket. Commun. ACM 55, 1 (January 2012), 48-56. DOI=10.1145/2063176.2063195 http://doi.acm.org/10.1145/2063176.2063195
10. Joel Moses. 1970. The function of FUNCTION in LISP or why the FUNARG problem should be called the environment problem. SIGSAM Bull. 15 (July 1970), 13-27. DOI=10.1145/1093410.1093411 http://doi.acm.org/10.1145/1093410.1093411
11. D. J. Rosenkrantz and R. E. Stearns. 1969. Properties of deterministic top down grammars. In Proceedings of the first annual ACM symposium on Theory of computing (STOC '69). ACM, New York, NY, USA, 165-180. DOI=10.1145/800169.805431 http://doi.acm.org/10.1145/800169.805431
12. Paul Hudak. 1989. Conception, evolution, and application of functional programming languages. ACM Comput. Surv. 21, 3 (September 1989), 359-411. DOI=10.1145/72551.72554 http://doi.acm.org/10.1145/72551.72554
13. Reynolds, J. C. (1993). The discoveries of continuations. Lisp and symbolic computation, 6(3-4), 233-247.
14. McCarthy, John. Recursive functions of symbolic expressions. Springer Berlin Heidelberg, 1983.