

VARIAC: an Autogenous Cognitive Architecture

J Storrs Hall

^a *Storrmont: Laporte, PA 18626, USA*

Abstract. Learning theory and programs to date are inductively bounded: they can be described as “wind-up toys” which can only learn the kinds of things that their designers envisioned. We conjecture [1] that general intelligence involves an unbounded learning ability. VARIAC is an experimental cognitive architecture designed to learn by modifying and extending itself, including its ability to learn, so that it can learn to become a better learner.

Keywords. autogeny, learning, cognitive architectures, automatic programming

Rationale

0.1. Automatic Programming

The approach to AI described herein is informed by the point of view that learning is programming: learning a category is designing an algorithm to distinguish members from non-members; learning a skill is writing a program to perform it; learning general knowledge about the world is building a model that can predict the consequences of situations or actions.

Thus any general learning program is a program that writes programs. It must invent both algorithms and representations. For human programmers, as programs become more complex it is advantageous to develop new notations to handle abstractions efficiently. This will hold true for AI systems as well. An unbounded learning system will be one which invents new programming languages.

Search, as exemplified by Solomonoff induction, is in theory capable of producing arbitrarily complex programs; but in practice the limitation of finite computational resources represents a formidable obstacle to the approach.

Automatic programming, a robust field from the beginnings of AI until the early 1980s, experienced a significant decline thereafter. The Stanford PSI project [2], which constructed LISP programs per a natural language dialogue with the user, appears to have been the high point of classical automatic programming. Lou Steinberg, a PSI principal, has conjectured that automatic programming is “AI complete” in the sense that it requires general knowledge and competence to understand what program the user wants without requiring him to specify it in the same detail he would have used in a programming language.¹

¹Louis Steinberg, personal communication with the author, April 2006

For a situated, experiential robot, however, automatic programming can obtain traction from another source: the system can attempt to write programs that model (and predict) phenomena, and ones that drive the robot in imitation of observed actions. If the robot already has programs that perform actions or model phenomena that are similar to the new ones, the problem is reduced to modifying an existing program – a considerably simpler task, and a technique universally used by both human programmers and evolution.

0.2. Self-improvement and “wind-up toys”

An AI capable of unbounded self-improvement must be able to invent new representations in which to think. At the base, any robot or organism has the representation of the world given by the raw signals from its sensors. It imposes an ontology on the world by interpreting these as more abstract concepts, in representations that make use of implicit notions of 3-dimensional space or rigid objects, for example.

In most robotic and AI systems to date, such an ontology is prepared for the system, in its entirety, by its human programmers. A few notable exceptions, such as AM and Eurisko, to the contrary notwithstanding, concept formation has been one of the most poorly developed of AI subfields. We believe a large part of this is due to overly simplistic notions of concept (such as predicates in FOPC). A more complete notion, including the requirement to implement the abilities to recognize, predict, and plan with the phenomenon involved, reveals the full scope of the problem and the challenge.

A robot which had induced (or been given) notions of 3D space and objects from sensor data might proceed to formulate kinematics, then dynamics with the concepts of force and acceleration, and ultimately laws of universal invariants such as the conservation of energy. Although fancifully ambitious, this progression demonstrates that an unbounded learning process must form concepts not only from patterns of sense data but of previously learned concepts.

1. Background

1.1. Computational Learning Theory

It is common in computational learning theory to represent situations or actions to be categorized by numeric vectors in some high-dimensional space. In the simplest case, the category can then be described by a hyperplane separating instances from non-instances. In the general case, however, the description of a category can be as complex as the description of an arbitrary subset of the set of possible descriptions of cases.

Consider for example a category of numeric vectors which includes just those having a prime number of 1 bits in their floating-point representations. It is difficult to imagine a geometric definition of this category, but one based on a program is straightforward. We conjecture that many real-world categories have structures as complex, if not as capricious, as this example: fractal by virtue of recursive description. (In CLT, the use of kernel functions, for example, represents a step in this direction.) In other words, we conjecture that many of the categories an AGI must learn will be best described by programs.

1.2. AI and Automatic Programming

Historically, AI has been the driving force behind many if not most of the advances in programming languages. LISP introduced abstract syntax trees (ASTs) as a basic datatype, and automatic storage management. It was also arguably the first functional programming language. PLANNER, PROLOG, and various theorem provers introduced automatic inference, now widely used as the mechanism behind the type systems of most modern languages. SNOBOL (designed for natural language processing) introduced pattern-matching. The “object-oriented” semantics of many modern languages can be traced back to “frame-based” AI systems of the 1970s.

LISP and PROLOG are prime languages for writing programs about programs for two main reasons. First is that they do have ASTs as primitive datatypes, together with a collection of operations on them of which needed functionality can be easily composed. Second is that their semantics are already reasonably abstract, so that neither programs written in them, nor programs written by those programs, need be concerned with details such as register usage, linkage conventions, or storage allocation.

Properly designed and implemented, high-level programming abstractions can reduce the size of programs by an order of magnitude as compared with the same programs in lower-level languages. But the effect is compounded in automatic programming programs (APPs): not only is the APP simplified compared to a low-level program doing the same task, but the task itself is simpler, since the object program is also simplified. For higher-order APPs, e.g. ones that write APPs themselves, the effect is exponential in order. We will refer to this phenomenon as the *recursive simplification* of the automatic programming problem.

1.3. Automatic Design

In the 1990s, the author and colleagues at Rutgers developed an architecture for the automatic design of microprocessors. It was based on recursive search within a hierarchy of abstraction levels, guided by a utility-based evaluation function. The utility function used different simulators at each abstraction level (ranging from RTL at the highest level to SPICE at the lowest) to evaluate the expected performance of a candidate design. This was augmented at any given level with an improved estimate recursively backed up from lower levels in the search tree by taking standard statistical measures of the population there. The evaluation function further estimated expected cost of search, and expected utility gain from further searching, by similar statistical means.

This system was able to produce pipelined, single cycle-per-instruction RISC microprocessor designs from high-level descriptions of the desired instruction set. It is the intent of the VARIAC project to adapt this architecture to produce programs from high-level specifications in essentially the same way.

1.4. Procedural Embedding of Knowledge

Classic AI programs such as SHRDLU embedded much of their knowledge in programs. SHRDLU, for example [3], had a language called PROGRAMMAR for expressing grammatical knowledge, and used MICRO-PLANNER for the semantics of its blocks-world. Follow-on knowledge-representation languages such as KRL-0 [4] attempted to retain

this stance, but typically lost Turing completeness in the face of the pressure towards declarative semantics: the more closely the representation resembled logic or natural language text, the more likely the semantics was to resemble syntactic inference mechanisms. This trend continued throughout the “expert systems” era.

The necessities of robotics, among other considerations, have forced a return to tightly-coded semantics in languages that are more directly adapted to real-time control than is the raw predicate calculus. These programs often make use of trained recognizers for predetermined concepts, but they do not originate their own concepts. No significant automatic programming is done, either by the robotic control programs or in the process of writing them. What is more, such low-level programs typically give up the generality of expressiveness that was the intent of declarative representations.

The main objections to procedural embedding are that procedural programs are opaque to inference, resistant to composition, and brittle under modification. Thus it seems desirable to design a programming language that is transparent, composable, and robust, while retaining the ability to specify semantics simply and directly.

2. Key Concepts in VARIAC

The design of VARIAC² is shaped by two main pressures. The first is the state of the art in programming language and compiling technology, which defines just how expressive a language we can implement with a usable degree of efficiency. The present section describes the language from this engineering point of view. The other pressure is the goal of building a system that can escape the hold of the so-called bootstrap fallacy and be capable of open-ended self-improvement. The following sections describe how such a system can be built given the capabilities described in this one.

There are many techniques, from AI and other fields, that are well understood and can be put to good use in the implementation of a programming language. Somewhere between the difficulty of a conventional optimizing compiler and that of a full-fledged automatic programming system with natural language input, the problem of compiling an extremely abstract yet formally specified language is a good match for much of the current inventory of symbolic AI techniques. Most of the following language features are implemented in one or more existing programming languages; no language has all of them. Each is an abstraction, in the sense that it specifies a set of concerns the programmer need not worry about (as garbage collection relieves the concern for storage allocation).

2.1. Higher-order Functional and Relational

Programming language theory addresses the procedural/declarative dichotomy by way of functional programming. A *pure* functional language describes a function as a declarative combination of primitive functions; no sequential or procedural semantics are implicit. In a major step in the direction of autogeny, higher-order functional languages compute functions directly as the values of expressions. This is a major source of abstraction and expressive power; see *e.g.* [5].

²VARIAC is an acronym for Vectors, Abstraction, and Recursion Integrated for Autogenous Cognition. Note also that in electrical engineering, a variac is an auto-transformer.

The sequence- and side-effects-free nature of functional languages encourages a “value semantics,” essentially a mathematician’s view of the data instead of an assembly language programmer’s. This is a substantial abstraction (although challenging to implement efficiently). Similarly, functional languages almost universally have automatic storage allocation. Research in functional languages has made substantial progress in the efficient compilation of programs using these abstractions.

Conceptually, a function is a table wherein the result to be returned is stored at an index specified by the argument(s). In a language with a level of abstraction above the concerns of data-storage in computer memory and time-sequencing of instructions, there is no need to make a distinction between a function represented as callable code, an array stored in coordinate memory, or a database relation whose access is based on whatever indexing scheme is appropriate. Once the view of data as a table is adopted, it is no longer necessary to assign a preferred direction from arguments to values; relational languages such as Prolog and Kanren take this view. In other words, the same predicate/relation can be used for, e.g., addition and subtraction: `plus(2,2,X)` or `plus(2,Y,4)`.

Human memory is associative; recall is based on parallel pattern-matching. There is no clear distinction between memory and program in the human mind. VARIAC’s view of program and data is similar: any object can be function or array. For example, the function `Sin` takes a number and returns the trigonometric sine, while `Sin` is a one-dimensional array with an unbounded numerical index. The distinction (based on capitalization) is syntactic only; both refer to the same object. One can assign an array value to `FOO` and then use `foo` as a function without further definition. (Indeed, the expression $\text{Sin}^2 + \text{Cos}^2$ evaluates to an object very similar to the scalar 1, the technical difference being its more limited domain when it in turn is used as a function!)

2.2. *Scientific / Numerical*

Any representations and algorithms must be couched in terms of primitive objects and operations. We assume that any possible representations and algorithms can ultimately be couched in terms of bit strings and Turing machines; universality in this sense is well-understood and need not concern us further. For reasons of practicality, however, VARIAC provides more developed representations, of which bit strings and state machines are a special case. These address two major areas of concern: descriptions of the physical world, and descriptions of language and algorithm constructs.

It is only reasonable, when describing the physical world, to avail ourselves of the centuries of work that have been done in the physical sciences to develop formal representations thereof. The language of physical science is numbers, vectors, matrices, and equations, simple and differential. Furthermore, this language is also used to describe statistics and probability, which are fast becoming an integral part of the AI practitioner’s toolkit.

Symbolic methods for manipulating programs and logical formulae, necessary for optimization of programs, are essentially of the same kind as those for manipulating equations and mathematical formulae. It is therefore surprising that there remains a gap between numerical languages (e.g. Fortran) and symbolic ones (Prolog) in practice. VARIAC attempts to close the gap and be equally facile at symbolic and numerical manipulation.

2.3. *Discrete and Continuous*

Arrays may be discrete or continuous in space and signals may be discrete or continuous in time. Pictures, sonar depth fields, maps, and so forth are among the many things that benefit from being treated as continuous, at a level of abstraction above the details of discretization.

For example, the occupancy field which is the basis of Hans Moravec's breakthrough Bayesian sensor fusion algorithm for mobile robots [6] is just such a surface, as is the sensor model which must be added to it. Suppose WO is an initial estimate of the occupancy field, $Bearing$ and Pos status signals for the robot, $Dist$ the distance reading from the sensor, and $Model$ the learned sensor model, a 3D array indexed by the reading and 2 spatial dimensions. Then

$WO += Bearing \text{ rotate } Pos \text{ translate } [Dist; ;] Model$

is the entire code for the world model. Its value, as well as those of its inputs, are reactive, i.e. signals that are functions of time. There are no loops or other control structure. The accumulate ($+=$) function adds successive values of a discrete signal or integrates a continuous one.

2.4. *Frame-arrays and Tensor Fields*

Minsky's concept of a "frame" was widely adopted by the AI community in the 1970s and 1980s; but Minsky himself has opined that the more important aspect of his idea, the notion of the frame-array, was largely ignored. The frame itself was an organizing principle for the data representing a situation as it might be observed and interpreted by a robot. The frame-array was a collection of such frames, indexed or generated in such a way that the results of typical actions or events were automatically predicted by the presentation of an appropriate new frame.

Consider a robot head with two degrees of freedom, such that its pose can be specified by a 2-vector. In a fixed environment, a camera in the head will return a specific pixel array for any given pose. A two-dimensional array, indexed by pose, of two-dimensional arrays of pixels is thus a very simplistic form of a frame-array for this robot. In physics and engineering, such a space mapping each point to a structured value is called a "tensor field." It is a device of enormously flexible representational power. Unlike physics, and more like Minsky, we allow any structured value, not only numeric arrays, to be "mapped into" each point of a field.

2.5. *Reactive: the Garbage Collection of Time*

Typically functional languages have addressed the declarative/procedural gap by being "impure," that is by mixing sequential semantics with the functional as does LISP, or by the use of "IO monads" as in HASKELL [7], an explicitly-computed trace of the interactions the program involves.

Another option, and the approach taken with VARIAC, is that the "program" is a declarative description of an equation, circuit, or machine that has a well-defined dynamic behavior in time. The equations of physics, as well as standard signal systems in control theory, have this semantics, explicitly referring to time derivatives and integrals of quantities. Simulation languages such as LABVIEW also take this approach, and it has much in common with stream- and dataflow-based languages.

Simply put, the semantics of a reactive³ language are that instead of a flow of control which sequentially activates the statements of the program, all the program elements are active all the time, as if they were components of a circuit. Such a program can be compiled (not without some difficulty!) into one which both takes advantage of such parallelism as is available, and uses scheduling and interrupts to emulate the “always-on” effect. Reactivity is thus to sequencing and coordination as garbage collection is to storage allocation: it provides a model of unbounded processor time just as CONS provides a model of unbounded storage. In both cases the resulting program is simplified from one which confronts the details directly.

The result of combining relational and reactive semantics is that a VARIAC program is the equivalent of a set of simultaneous equations, a circuit, or a constraint network. Given this view, it is straightforward to create structures such as semantic networks with parallel spreading activation behavior (e.g. [9]). Another paradigm that maps neatly into this framework is Minsky’s “Society of Mind” agencies (hereinafter “Minsky SOM”), with nodes activating each other in hierarchical cascades. These can be enhanced by virtue of the fact that VARIAC allows the passing of arbitrary values, including functions, from node to node.

2.6. *Reflective: First-class Types*

Modern programming language theory relies heavily on the theory of types. Following the Curry-Howard Isomorphism, any program using algebraic constructive types can be mapped onto a proof that the result has in fact the desired type. Besides being a boon to compiler writers (and thus to automatic programming), types are a major step in the direction of a language to describe data.

Another language to describe data is the string patterns in SNOBOL and its successors. Optimized specializations of these as language grammars are found in parser generators such as YACC.

Finally, symbolic mathematics programs such as MACSYMA have been developed, again originally an offshoot of AI (as in SAINT), which implement a considerable ability to describe patterns of numbers and other mathematical concepts.

VARIAC’s types are essentially a unification of these forms of specifying values and patterns. More importantly, they are an integral part of the language. Typically, existing programming languages either have static typing, meaning that the compiler reasons about types before emitting object code in which the types are implicit in the representations, or dynamic, meaning that types are explicitly represented in datastructures. Only in a few unusual languages (e.g. BERTRAND [10] and Q⁴) is type used as an integral part of the specification of the computation.

Types in VARIAC are values which can be combined and manipulated as easily as numbers or symbolic expressions. This is a key capability for general knowledge representation as well as for self-describing and self-extending programs. The type description ability is based on David McAllester’s ONTIC Language [11] which was in use as the description language for some automatic theorem provers in the 1990s but never expanded into a full programming language. It has much in common with both the patterns

³Not a particularly descriptive name, but the one that has come to be accepted in the programming language theory community. See e.g. [8].

⁴<http://www.musikwissenschaft.uni-mainz.de/~ag/q/qdoc.pdf>

and grammars mentioned above and with classic AI knowledge-representation languages such as KRL-0 and MDS [12].

Since programs and objects are unified in VARIAC (as in any higher-order functional language), types represent abstractions of programs and form a key ingredient of our automatic programming methodology. Rather than merely searching a syntactic space of programs, as is done in genetic programming for example, we can narrow the semantic space with abstractions of increasing specificity. (For example, when attempting the formulation of the inverse kinematics of a robot arm, an abstraction might include domain and range specifications, and the fact that the desired function is an inverse of a known forward kinematic function.)

3. From Programs to Cognition

A growing program will need to be both scientist and engineer in its own world – to construct explanatory and predictive models, and also to construct competent control programs as it learns skills. It must parse the world, as well as language it hears, into coherent structures of useful pieces. It must clothe frameworks of words with the fabric of imitated actions, cut and stitched to fit.

3.1. *Sigmas*

The simplest way to learn and predict is to copy a trace of the experiential stream into a recorded trajectory in some appropriate representation. In the simplest forms, this can be a series of points in some n -dimensional space. Conceptually straightforward, although computationally expensive, nearest-neighbor methods can then map any given situation to previously experienced ones (or to averages of clusters thereof) and predict the evolution of the present state by a simple geometric quadrature. Learning is simple – it consists of adding new experiential tracks to the memory.

In practice this is far too abstract a view. If the experiential stream were simply recorded without compression, it would imply a human long-term memory at least a billion times the size of current best estimates, and processing power orders of magnitude beyond that. In practice the “firehose of experience” is compressed into a few bits per second of memorable information. More than compressed, however, it is re-represented into the form of useful abstractions. Re-representation is well understood and is used extensively in most AI and robotics architectures, simple examples being the formation of maps and 3D models from rangefinder and camera datastreams.

In order for this CBR-like learning method to work, it is only necessary for the re-representation to map into a space with a useful metric susceptible to quadrature, and for the space to be abstract and compressed enough for the operations to be computationally tractable.⁵ As far as we know, this is not possible for general experience as a whole, but it is possible for many specific instances and categories. We claim that it is reasonable to identify a “concept” with a particular abstraction space (with a metric that supports

⁵Note that estimates of brain processing power give 1 or 2 orders of magnitude more processing power per bit of memory than is typical with von Neumann computer architectures, an argument that simple, brute-force hardware associative memory may be a more appropriate model than ingenious hashing and indexing schemes on serial processors.

CBR-like learning) and the functions that perform re-representation into that space from whatever other representations are available.

We refer to such a space, functions, the associative memory for trajectories, and the quadrature mechanism as a “sigma,” for situation-goal-memory-action. Such a sigma, in a reactive implementation, can be used as either a controller or prediction machine (in a manner similar to that of Minsky [13]) by re-routings of its addressing and output connections.

There is a straightforward translation from any reactive functional program with well-defined types to a network of appropriately connected sigmas (with appropriately initialized traces). Note that in this translation, not only are the units that capture the experiential memory of the system (if any) represented as functions implemented as interpolating associative memories, but so too are the functions that map between the different representation spaces, and thus define the concepts.

It is possible, if not trivial, to produce figures of merit for any such re-representation structure at the highest levels, which transforms the problem of experiential learning and concept formation to an abstract programming problem in a form amenable to our rational utility-based program construction system.

3.2. *Active Production Networks*

Networks of reactive functions map directly onto circuit models at any level from digital gate logic to the connected modules of digital signal processing, so it is essentially trivial to implement any of the many standard sensory and control architectures in the literature in this form. Somewhat less obvious is how to implement the complex interpretation capability that is conventionally done with search-based algorithms, such as natural-language parsing.

A method that shows promise in this regard is the Active Production Network developed by Mark Jones of (the then) Bell Telephone Laboratories in the 1980s [14]. It is a parallel message-passing semantic network model, with a structure that reflects the grammar it accepts. The network accepts words sequentially, and its state in terms of active messages and accumulated values at the nodes reflects its interpretation of each initial substring. This formulation of a grammar matches very well with spreading-activation semantic and agent networks. (Note that at a level of abstraction including both the pure functional and the reactive aspects, APNs and recursive descent parsers unify.⁶)

Experimental APNs have shown not only the basic ability to parse, but remarkable robustness in the face of misformed or noisy input. In addition, they interface very naturally to semantic models expressed as reactive functional programs, in such a way that semantic constraints are automatically incorporated into the parsing process. It is suggestive to note that some theories of evolutionary neuroscience [15] place language and fine manipulation ability together. Minsky SOM agencies for complex motor control look remarkably like APNs; we conjecture that a unified formulation can be used for act interpretation as well, and for language understanding and generation.⁷

⁶i.e. the essential distinctions are differing treatments of concerns that the abstraction has already automated.

⁷Unpublished experiments by Jones using APNs for text generation were promising although preliminary. Private communication with the author, 1988.

3.3. *An Economy of Mind*

It is becoming common for computational neuroscientists to speak of the dopamine reward-prediction error signal mechanism as the “currency” of the brain. It clearly performs a central role in the motivation and decision-making function. Less obviously, however, it turns out to play a crucial role in learning and concept formation as well.

This agrees well with the experience of the author and his colleagues at Rutgers in the field of agoric and utility-based design, if learning is equated with designing a complex structure such as a program. In such a design process, it is necessary to value components and sub-assemblies in order to choose between design alternatives, and ultimately to produce a valuation surface for an abstract space of partial designs. In the brain, the dopamine system “backs up” the reward signal to perform a kind of credit assignment. In agoric systems going back to Holland’s “bucket brigade,” a market model performs the same function.

Utility-based design performs a best-first search in a tree-structured space generated by an abstraction hierarchy over the object structure. As such it is easily controlled and is appropriate for use in the base VARIAC language implementation. A more complete, parallel, and open-ended market model, such as CHARLES SMITH [16], seems more appropriate to an architecture of general intelligence, but comes with many more open questions.

4. Towards an Autogenous Architecture

The field of AGI, as distinct from more general application-oriented AI, has as yet little general agreement on the specifics of the architecture of a general cognitive agent. One methodological element that does seem much more prevalent in AGI than in AI at large, however, is the notion of building a “child machine,” as proposed by Turing, and educating it to adult competence rather than building a mature system *in toto* [17].

Given this position, our experimental program for AGI development consists of a blocks-world situation reminiscent of SHRDLU. It will employ a physical robot (currently under construction) with arm(s) and binocular vision. It is immobile, allowing for relatively powerful processing hardware; the current setup has ten general-purpose and approximately 300 vector processing units. This is expected to increase over the course of the project.

The software architecture of the robot controller is also similar to that of SHRDLU, except that the world simulator is replaced by a robot controller and vision interpretation stack. The most innovative element of the SHRDLU architecture, the crosstalk between the parser and the world model which was used for disambiguation [18], is retained and extended. The grammar, model, controller, and higher-level vision are all implemented as reactive function networks capable of self-extension or modification; the form and extent of self-modification possible or desirable is the key object of the research and as yet a very open question.

4.1. *Learning*

We conjecture that most learning is done by imitation and/or analogy. Early experiments will thus be of the form *I touch the block; you touch the block*. Key early issues are the

ability to parse the sensory stream in space into objects and in time into actions. Then comes the ability to match objects and actions with words, and finally the ability to put those together into novel structures.

We further conjecture that learning by verbal instruction is similar to learning by imitation. If the language of thought is formulated properly and integrated with semantic cognition, parsing sentences should result in structures not unlike the ones from parsing the experiential stream of watching an exemplar activity. Indeed, if the language is learned by experience, these will be the only structures available to assign as meanings to new words and constructs!

The emphasis in the 80s on applications displaced blocks-world experiments from the forefront of AI, but we feel that this was a mistake. A deskbound robot could, after it became proficient with blocks, proceed to learn with more complex toys, say Lego sets. It could learn to assemble or repair complex machinery. It could play chess, Monopoly, or marbles; read, write, and draw pictures. It could be a laboratory chemist or a hibachi chef. If we assume – and this is the scientific hypothesis to be determined by experiment – that a desktop world is a sufficient semantic domain to be a basis for understanding general language, then a desktop robot could grow to be a completely open-ended, general intelligence.

4.2. Implementation

The kernel of VARIAC is a unification-augmented term-graph rewriting engine with reflexive types. That is, each expression has a type, which can itself be any expression. (This continues recursively until grounded in a primitive type.) This generates a language, if used raw, of an expressive power similar to PROLOG (because it has unification and backtracking), but completely declarative. For example, a symbolic algebra package can be written in about a page of code.

The status of VARIAC as of even date represents about a year of mostly exploratory and definitional work. The current interpreter is third in a series of implementations adding successively more of the listed abstractions to the semantic base. In a longer retrospective, VARIAC is a development of the machine definition language Caslor [19] and the methods of program construction derive from the AI-in-design work of [16].

5. Summary

No matter what formalisms and datastructures it manipulates, an AI must, in effect, write programs. Understanding is in essence the ability to simulate and thus to predict. Learning is the ability to understand things one did not before, and thus involves implementing new simulations – creating new representations and new algorithms. *Every AI system implemented to date* has been a “wind-up toy,” built with unextendable basic representations designed entirely by human programmers.⁸

We can cut the problem of AGI in half by designing a programming language at such a high level that it significantly reduces not only the work we must do to implement the system initially, but the work the system must do to reprogram, and thus to improve, itself. The first half of the problem, implementing the language thus defined, avails us

⁸With the possible exception of some AGI systems under development!

of many of the narrow-AI tools that have been developed in the past half-century: Expert systems, planning and design algorithms, theorem provers, utility-based search, and many other standard AI techniques are applicable to the task of compiling a very-high-level but formally-specified programming language.

More important than the specific functions and datastructures of the language are its abstractions, the concerns it lets the programmer (including the automatic programming system) ignore. All of the abstractions we have chosen have been implemented in some existing programming language, with the exception of the reflexive type system. The integration of these abstractions into a single coherent language is certainly non-trivial but appears to be a straightforward development project.

References

- [1] J. Storrs Hall, *Self-Improving AI: an Analysis*, AI@50, Dartmouth College, June, 2006
- [2] Cordell C. Green, Richard J. Waldinger, David R. Barstow, Robert A. Elschlager, Douglas B. Lenat, Brian P. McCune, David E. Shaw, and Louis I. Steinberg, *Progress report on program-understanding systems.*, Stanford University, Stanford, CA, 1974
- [3] Terry Winograd, *Procedures as a Representation for Data in a Computer Program for Understanding Natural Language*. MIT AI Technical Report 235, February 1971
- [4] Bobrow, D. G., and T. Winograd. (1977). An overview of KRL-0, a knowledge representation language. *Cognitive Science* **1**(1):3-46.
- [5] J. Hughes, *Why Functional Programming Matters*, *Computer Journal* **32**:2 pp98-107, 1989.
- [6] Hans P. Moravec. *Sensor fusion in certainty grids for mobile robots*. *AI Magazine*, **9**(2):61-74, 1988.
- [7] Philip Wadler. *Comprehending Monads*. Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice. 1990.
- [8] Zhanyong Wan, *Functional Reactive Programming for Real-Time Reactive Systems*, Ph.D. Dissertation, Yale University, 2002.
- [9] Ian Horswill. "Cerebus: A Higher-Order Behavior-Based System." *AI Magazine*, 2001.
- [10] Wm Leler. *Constraint Programming Languages: Their Specification and Generation*. Addison-Wesley Publishing Co., Reading, Massachusetts, 1988.
- [11] David McAllester. *Ontic proof verification system*. <ftp://ftp.ai.mit.edu/pub/ontic/ontic.tar.Z>.
- [12] Chitoor V. Srinivasan, *Model Space of the Meta Description System*, Report SOSAP-TR-19, Department of Computer Science, Rutgers University. 1976. MDS was explicitly self-referential (hence the "Meta").
- [13] Minsky, Marvin. *The Emotion Machine*. New York, Simon and Schuster, 2006, pp 159-160.
- [14] Mark A. Jones and Alan S. Driscoll, *Movement In Active Production Networks*, *Proc Assoc. Comp. Ling.* 1985, pp. 161-166; citeseer.ist.psu.edu/580365.html
- [15] Ornella Castelli and Carlo Peretto, *The Phylogenesis of Language: The Grammar of Gestures and the Manipulation of Words*, *Human Evolution* **21**(1), March 2006
- [16] J Storrs Hall, Louis Steinberg, and Brian D. Davison (1998) *Combining agoric and genetic methods in stochastic design*, *Nanotechnology* **9**(3) (September 1998) 274-284; Steinberg, Hall, and Davison, (1998): *Highest Utility First Search Across Multiple Levels of Stochastic Design*, pp. 477-484. Proceedings of the Fifteenth National Conference on AI, Madison, 1998.
- [17] David Vernon, Giorgio Metta, and Giulio Sandini, A Survey of Artificial Cognitive Systems: Implications for the Autonomous Development of Mental Capabilities in Computational Agents, *IEEE Transactions on Evolutionary Computation*, Special Issue on Autonomous Mental Development, **11**(2), 2007
- [18] Terry Winograd, *Understanding Natural Language*, New York: Academic Press, 1972, p. 5.
- [19] J. Storrs Hall, (1999): *Towards a Hardware Description Language for Molecular Machinery*, Seventh Foresight Conference on Nanotechnology, Santa Clara, CA