Project to Build Programs that Understand

Eric B. Baum

Baum Research Enterprises
41 Allison Road
Princeton NJ 08540
ebaum@fastmail.fm

Abstract

This extended abstract outlines a project to build computer programs that understand. Understanding a domain is defined as the ability to rapidly produce computer programs to deal with new problems as they arise. This is achieved by building a CAD tool that collaborates with human designers who guide the system to construct code having certain properties. The code respects Occam's Razor, interacts with a domain simulation, and is informed by a number of mechanisms learned from introspection, the coding employed in biology, and analysis.

Introduction

This extended abstract outlines a project to build computer programs that understand.
Definition 1: Understanding a domain is defined as the ability to rapidly produce programs to deal with new problems as they arise in the domain.

This definition is proposed to model human understanding (which I hold to be of the domain of the natural world and extensions we have made into related domains such as mathematics), and also accurately describes biological evolution, which thus may be said to *understand* what it is doing.

A program that can rapidly produce programs for new problems is rather unlike the standard kinds of programs that we usually see. They deal with contingencies that had been previously planned for. We will describe both a new programming method, and new style of program, in order to accomplish our task.
Hypothesis 1: The property of understanding in thought and evolution arises through Occam's Razor. (Baum, 2004, 2007)

By finding a concise genome that solves a vast number problems, evolution built a program comprising a hierarchic collection of modules that generalize-- that know how to rapidly produce programs to solve new problems. That genetic program also encodes inductive biases that construct a similar hierarchic collection of modules within your mind that rapidly produce programs to solve new problems.

As a result of conciseness and the structure of the programs it requires, random mutations in the genome lead to meaningful programs a sizable fraction of the time. For example, a point mutation of a fly's genome may create a fly with an extra pair of wings instead of halteres, or a fly with an extra pair of legs instead of antennae (Kirschner and Gerhart 2005). These are recognizably *meaningful* in that they are functional in an interesting way. Evolution thus rapidly searches over meaningful outcomes looking for one that solves new problems[1]. That is, according to our definition, evolution understands.

Another example within evolution is the coding of limbs (Kirschner and Gerhart 2005). If asked to code up a limb, human software engineers might attempt to separately specify the structure of the bones, the muscles, the nerves, and the blood vessels. If done that way, the program would be huge, like standard engineering specs for machines such as a jet aircraft, and it couldn't adapt to new uses, and it couldn't evolve. A favorable mutation in the bone structure would have to be matched by one in each of the other systems to survive. Proponents of Intelligent Design would have a point.

Instead, biology is much more concisely coded. The bones grow out in a concisely and hierarchically specified way. There is no separate detailed specification of exactly how the muscles, nerves, or blood vessels grow. The muscle cells perform a search, and attach themselves to the bones, and grow so as to be functional in context. A muscle cell that is being useful expands (which is why rowers have big hearts). The nerves seek out muscles, and learn how to be functional. The vascular system grows out in a search toward cells that scream for oxygen. As a result of this extremely concise encoding, if you take up new exercises, your muscles adapt and grow in appropriate ways, and if a mutation changes the genetic coding of the

[1] Language facilitates thinking in similar fashion. You can come up with incredibly concise formulations of big new ideas, just a sentence or several in natural language, that grow out into detailed executables much like concise specifications in the genome result in interacting searches that robustly construct a limb, or like RBP finds a high level plan and refines it through a series of searches.(Baum, 2008b)

bone structure, everything else adapts to make a functional system, so mutations can easily explore meaningful possibilities.

The same kind of conciseness is present in the genetic coding for your mind. Enough initial structure is provided, also called inductive bias, that a series of search programs can build a hierarchic series of modules to solve new problems.

Previous attempts to produce understanding programs have mostly followed one of two paths. One path has been purely automatic methods, such as direct attempts to simulate evolution. This is hopeless because, while we may already have or may soon have computational resources comparable to those of the brain, we will never be able to compete with evolution-- which ran through some 10^{44} creatures (Smith, 2006, footnote 95) (furthermore each creature individually interacting with the world and thus expensive to simulate). To evaluate other automatic attempts, ask yourself what is enforcing Occam nature and keeping the solution constrained enough to generalize to new problems. An approach that can too easily add new knowledge without adequate constraint can (and will) simply memorize examples that it sees, rather than generalizing to new problems. In my view, this is a common flaw in AI/AGI approaches.

A second approach has been to craft such a program by hand. I believe this is also hopeless for a variety of reasons. First, a wealth of experience in computational learning theory indicates that finding Occam representations in any interesting hypothesis space is NP-hard. If finding Occam software is NP-hard, it is no more likely to be susceptible to hand solution than other huge NP-hard problems. Second, a wealth of experience with solving AGI by hand indicates it is hard. Winograd's SHRDLU, for example, seemed like a particularly well crafted project, yet a few years later he threw up his hands and wrote a book explaining why crafting an understanding program is impossible (Winograd and Flores 1987). Third, when its structure is examined (say by introspection and/or attempting to program it) the program of mind (and also the program of biological development) seems to contain a large number of ingeniously crafted modules. At the very least, figuring out each of these is a PhD dissertation level project, and some of them may be much harder. Evolution, which understands what it is doing, applied massive computational resources in designing them. And the Occam codings evolution found may inherently be hard to understand (Baum, 2004 chap 6). Finally, the problems are hard both at the inner level (figuring out ingenious, concise, fast modules to solve subproblems) and at outer levels of organizing the whole program. Humans are simply not competent to write computer programs to deal with hyper-complex domains in a robust way.

Hand crafting has at times sought to benefit from introspection. A problem with this is that people do not have introspective access to the internals of the meaningful modules, which are hidden from introspection by information hiding. Consider, for example, when you invoke Microsoft Word. You know why you are invoking it, and what you expect it to do, but you do not have much idea of its internal code. The same is true of meaningful internal modules within your mental program. However, we appeal (based on observation) to:

Hypothesis 2: People do have introspective access to a meaning-level (Baum 2007, 2004 chap 14), at which they call meaningful modules by name or other pointer.

I use the term *concept* to convey a name (word) or other mental construction that you might think that has some meaning, i.e. that corresponds to a recognizable function in the world, and *module* to convey the computational procedure that would be summoned to compute a concept. Thus we may think of concepts as names that invoke modules.

You can state in words why you call a given module, and you can give examples of concepts (for example, inputs and outputs of the associated module). This is incredibly powerful and detailed information, that we will use to get a huge jump on evolution.

Note that the problem of finding an appropriate module is distinct from that of finding an appropriate concept, so this dichotomy factors the problem of producing code for new problems. For example, executing a certain module might create in a simulation model a corresponding goal condition (namely, realize a concept). *Agents* are simply modules that recognize patterns and then may take other computational actions, as in, for example (Baum and Durdanovic, 2000, 2002) . In the present work, patterns are typically recognized in a simulation domain or annotated simulation domain or model, that is seeing particular annotations may be critical to the recognition of the pattern. The recognition will frequently involve taking a series of simulated actions on the model and then checking for a pattern or condition (as in (Baum and Durdanovic, 2000). Agents when activated will frequently post annotations on a model, which is how much of perception proceeds.

The solution we propose, by which to construct understanding programs, is thus based on the following objectives and principles.

(1) Humans can not write the detailed code. As much as possible it must be automated, but, the system must also support as much guidance as possible from humans.

(2) The goal is to produce a program that will rapidly assemble programs to solve new problems. To that end, we have to ask what kinds of modules can be provided that can be flexibly assembled. We also have to ask what kinds of modules can be provided for guiding the assembly process, for example by finding high level plans that are then refined. We propose mechanisms for these tasks. Assembling programs for new tasks from primitive level instructions without a detailed plan is prohibitively expensive. But if a program can be divided into half a dozen tasks, and a program for each them assembled by combining a handful of meaningful modules, the search becomes manageable. Thus we need building systems and

building blocks and means of adapting to new contexts.

(3) The reason understanding is possible is that the natural world (and relatedly, mathematics) have a very concise underlying structure that can be exploited to do useful computations. We embody this by providing domain simulations. The domain simulations typically live in 3 (or 2) Euclidean dimensions, plus a time dimension, and provide direct access to causal structure. All modules and agents interact with the domain simulation (in fact, with multiple copies, each agent may explore its own copy). Thus all thought may be model/image based in a way much more powerful than other systems of which we are aware. This grounds all our computations, that is to say, will allow us to choose modules that perform functions that are meaningful in the real world.

All agent or instruction calls are with respect to a particular domain simulation position. Everything is thus context dependent, and agents can communicate by taking actions on the internal model and by perceiving the model. A caching mechanism detects when a module recursively calls the same module in the identical position, and returns a default value at the inner call, and thus enables concise recursive coding without looping.

(4) Economics simulations as in Hayek (Baum and Durdanovic 2000, 2002; Baum 2004 chap 10) are used at multiple levels to implement the invisible hand and assign credit so that components of the program all have to contribute and survive in a competitive environment. This greatly contributes to conciseness. Such economies also have the property of efficiently caching the execution of modules at each level of abstraction, greatly speeding computation.

(5) Other encodings collectively called scaffolds are modeled on those discovered by evolution or perceived by introspection in ways to realize point (2) above. Scaffolds promote conciseness, and provide inductive bias for constructions. For example, scaffolds make great use of search programs modeled on those used in the development of limbs (or in computer chess (Baum 2007)).

We call the system Artificial Genie.

CAD Tool

The first step is to build a CAD tool with which humans collaborate in the construction of code. A number of module constructors are provided, which take as inputs such things as an instruction set out of which to build new programs, a fitness function and/or a definition of a state to be achieved, and examples of a given concept, and return a module computing the concept. This is in principle straightforward to achieve by, for example, genetic programming (or, which we prefer, running a Hayek or other Economic Evolutionary System (EES)).

Once a module is constructed to compute a concept, the CAD tool also creates an instruction invoking the module, and makes that available for inclusion in instruction sets. Note that such instructions can be used in hand-written code, as well as fed to module constructors. So users are enabled to write code in terms of concepts, even though they may be unable to write the modules computing the concepts.

Note that, while above we said simulated evolution was too slow, what is generally done is to try to simulate the whole solution to a problem in one gulp. We are proposing instead to apply EES's or other module constructors to carefully chosen subproblems, and then build hierarchically upward. If the module construction fails, that is the module constructor is not able rapidly enough to construct a satisfactory module to compute the concept, the CAD tool also provides the opportunity to first construct other concepts which may be sub-concepts useful for computing the desired concept, the instructions for which can then be included in instruction sets.

The CAD tool will be used to improve itself until it can also learn from solved examples and natural language descriptions of the solution.(Baum, In prep).

Simulation Model

Our code interacts with a domain simulation that is analogous to mental imagery. This provides numerous critical functions. First, by encoding the underlying causal structure of the domain, the image allows modules to generalize, to encode meaningful function. For example, a simple physics simulation can describe what will happen in any kind of complex system or machine built of physical parts. Such are often easy to construct from ball and spring models(Johnston and Williams, 2008). All generalization ultimately derives from exploiting underlying structure, which is provided by the simulation.

Many AI programs are based on a set of logical axioms and logical reasoning, rather than incorporating a "physics" simulation. If you work that way, however, you can't readily generalize. Say I give you a new object, that you haven't seen before. Since you don't have axioms pertaining to that object, you have no understanding that lets you say anything about it. Logic just treats names (say of objects) as tokens, and if it doesn't recognize a particular token, it knows nothing about it. By contrast, if you have a physics simulation, you can work out what will happen when things are done to the object, when its rotated or dropped, just from its physical structure. This ability to generalize is essential to understanding.

Another problem with the logic approach is the frame problem. With everything in terms of axioms, you have a problem: if something is changed, what else is affected? Logically, there is no way to say, unless you have specific frame axioms, and then you have to do inference, which also may be computationally intractable. This classic problem paralyzed AI for many years. If you are working with a physical simulation, this problem is bypassed. You

just propagate forward-- only things effected by causal chains change.

Interaction with a simulation allows the computational agents to be grounded, that is to say: to perform functions that are meaningful in the real world. Second, the domain simulation provides a medium by which different computational agents can communicate, since agents can both perceive (recognize patterns in) and take actions on the domain simulation. Third, agents can utilize the domain simulation to search to achieve goals. For example, they can achieve subgoals in the context of the current situation and the other agents. Robustness can be achieved by partitioning problems amongst interacting, concisely specified, goal-oriented agents. This is a key element of how biological development achieves robustness. (Development utilizes an analog medium rather than a domain simulation.) The domain simulation naturally biases in the spatial, temporal, causal structure that greatly facilitates dividing problems into parts that can be separately analyzed and then combined. Fourth, the domain simulation provides an intuitive and interactive mechanism for users (program designers) to input training examples. Training examples can be entered as configurations of the domain simulation (for example, small local regions representing some concept) or as worked examples (where, for example the program designer inputs the example as if playing a video game) (which may be accompanied by natural language description of what is being done).

Relevance Based Planning

The power of using domain simulations is demonstrated by our planning system. Our planner finds high level candidate solutions by examining the mental image, where each high level candidate solution is a path that would achieve a goal if a number of sub goals (counter-factuals) can be achieved. The planning system then orchestrates the collaboration of those subprograms that are relevant to achieve the sub goals. Subprograms or agents that are attempting to achieve sub goals perform look-ahead searches on the mental image, and these searches may produce patterns indicating other problems that were not previously anticipated, the perception of which invokes other agents. Because the whole process is grounded by interaction with the mental image, the system explores those subprogram calls that are relevant to achieving the goal. An example of the operation of the planning system is discussed elsewhere in this volume (Baum, 2008a).

Like all the modules in Artificial Genie, the planner is supplied as an instruction within the CAD tool that can be incorporated into new modules or agents by automatic module constructors or by program designers. To our knowledge, this also is not a feature of any competing systems. This enables the construction of powerful agents that can invoke planning as part of their computation to achieve goals. Powerful programs can be constructed as compositions of agents by planning on the domain to break goals up into a series of sub goals, and then building agents to deal with the sub goals. An example of a Hayek constructing agents that invoke planning instructions to solve Sokoban problems was exhibited in (Schaul 2005).

Evolutionary Programming, Search, and Scaffolds

Consider a search to find a program to solve a problem. To do this by building the program from primitive instructions will often be prohibitively expensive. One needs appropriate macro instructions so that each discovery step is manageable size: no individual search is vast. Code discovery is only possible in bite sized portions. Scaffolds provide big chunks of the code, and break the full search down into a combination of manageable searches.

As discussed in section 1, this is largely how biological development is coded: as a series of tractable searches (each cell performs a search to produce its cyto-skeleton, mitosis invokes a search to build microtubules for organizing the chromosomes, the nerves perform a series of searches to enervate, the vascular system performs a separate search, and so on. (Kirschner and Gerhart 2005))

Coding in terms of constrained searches is an incredibly concise means of coding. As discussed in (Baum, 2004,2007) this is also the method used in computer chess. A handful of lines of code (encoding alpha-beta + evaluation function + quiescence) creates a search that accurately values a huge number of chess positions. Because the code for this is so concise, Occam's razor applies and it generalizes to almost all chess positions.

One simple form of scaffold supplies much of the search machinery, but requires that an evaluation function or goal condition for the search and/or a set of actions that the search will be over (which may generally be macro actions or agents) be supplied to use the scaffold for a particular problem. For example, the alpha-beta search from chess can be simply modified in this way into programs for other problems (like Othello) cf (Baum 2007). Another use of such search-scaffolds would be to compose them into larger programs in a way analogous to how development builds the body out of a collection of interacting searches. The use of search scaffolds should provide huge inductive bias, greatly speeding the production of programs for new problems. In other words, having such scaffolds will provide understanding of the domain.

Another form of scaffold is used in the evolutionary construction of programs. The scaffold here is basically equivalent to a procedure, with slots (aka arguments) and possibly also annotations respective to some or all of the slots. One can build a program downward by starting with a scaffold, and then constructing subprograms to fit into each of the slots. The annotations may supply guidance as to independent examples to be provided for training the subprograms, or other guidance such as instruction sets to be provided to module constructors to build the

subprograms out of.

The point here is, the naive way to be able to rapidly create a program for a new problem, is to have a set of macros from which the program for the new problem can be rapidly discovered because it is a 5 line program in terms of the macros. Another way, however, is to have a scaffold that builds all or part of the program by starting from the top and building down: discovering programs to fit into slots. The discovery of these subprograms may involve running an EES, or invoking another scaffold. And yet another way, is to have a scaffold like RBP that interacts with the mental image to build a high level plan and then to flesh it out into concrete code.

Evolutionary Economic Systems

The real economy is a powerful system in which billions of different economic actors, who don't even know each other and may not even share a common language, are organized to collaborate. A complex program is like an economy, with large numbers of parts that must work together. The usual process of constructing a complex program is like a command economy, with some human programmers sitting like the Politburo and attempting to fit everything together. As desired programs get more complex, it becomes harder and harder to integrate everything, and pieces are added that are counterproductive, and many opportunities for rationalization are missed.

Artificial Genie supplies an economic framework that motivates all the pieces by simple rules that implement the invisible hand, propagate price signals. Moreover, within the created environment of a simulated economy, major flaws that plague the real economy, such as the tragedy of the commons and theft and the dead-weight loss of taxation, are removed, leaving a perfect computational economy. Since it is made available as a module constructor, the economic framework can readily be applied at each level of the computational hierarchy, both as the simplest subprograms are built from instructions, and as the overall system is integrated, to ensure efficient and harmonious operation. So, a high level program will be composed of economically motivated agents, each of which itself may be a Hayek or other EES, composed of economically motivated agents, rather like the US economy is composed of corporations, each of which may have many subcontractors. This hierarchic construction is straightforward because each satisfactory module (often an EES) that is found, is encapsulated by the CAD tool as an instruction, usable for constructing later agents and programs.

Many problems can only be solved after performing a search. For example, deciding on the best move in a chess position typically requires a search; likewise planning problems, optimization problems, even problems of perception such as recognizing some pattern in one's environment or sensory input, almost all require searches. These searches will usually be intractably large if not somehow focused. Extant automatic programming methods do not produce programs that search and hence are limited in their application. Artificial Genie incorporates an economic structure to automatically construct search-programs, programs that when presented with a task to solve, perform a search to find the solution.

Artificial Genie supports the building of perceptual systems consisting of a collection of agents that perceive patterns and post names of recognized concepts, that may enable other perception or action. Such patterns will typically be perceived in the simulation domain, and the perception may involve taking one or more simulated actions on the domain position followed by verifying a condition or checking a pattern. The syntactical analysis of (Hobbs 2004) is a model for such posting of concept labels that enable other conclusions, except that we have added economic structure and interaction with a simulation model. The whole program, from perception through decision, is economically evolved to be concise, efficient, and meaningful, so that such perceptual agents will only survive when they are later relied upon by agents earning reward from the world.

The economics naturally incorporates a smart caching mechanism, by which subprograms that initially require extensive computation, become fast and automatic in most circumstances. Concepts are frequently given a definition in terms of other modules, rather than an explicit algorithm to compute it. Frequently a concept may be defined in terms of a condition is to hold or be achieved (which may in fact involve achieving multiple sub-conditions). You then have to use a module constructor to construct a program that achieves that condition (or recognizes when it holds). The module constructor will typically produce a search-EES. This search-EES caches into agents, methods that work rapidly in certain classes of positions. When the module is called, the search-EES cache will retrieve a solution rapidly, provided one is available. However, if the search-EES fails (ie no agent bids, or no solution is found) the system may return to the definition, and attempt to find a solution by a more extensive search (creating new agents) or other methods. If a solution is then found, the search-EES may be extended with more agents, caching the new solution. This is analogous to how you may have to resort to definitions and a slower, conscious thought process when confronted with a new situation, but as you learn about the new situation, you cache methods for dealing with it rapidly and unconsciously.

First Application

The above framework is to be built in the context of a particular domain. Sokoban has been experimented with to date, but a more practically interesting domain will be next. The tools, once built for a given domain, will be readily adapted to a next domain.

Sketch of Path to Cognition

The tools will then be used to build an inheritance structure corresponding to human cognition. Thought, language, and understanding are possible because we have a collection of meaningful modules, organized in a hierarchical fashion. One thing that is different about our approach from other hierarchic ontologies, however, is that in our view what is inherited are methods, procedures, that interact with a simulation domain, execute on a simulation domain, compose in a robust way using searches that determine meaningful combinations grounded with respect to the simulation domain.

References

Baum, Eric B. 2004. *What is Thought?* . Cambridge, MA: MIT Press.

Baum, Eric B. 2007. *A Working Hypotheses for Artificial General Intelligence.* In *Advances in Artificial General Intelligence: Concepts, Architectures and Algorithms,* eds. Goertzel, B., and Wang, P., IOS Press, pp 55-74.

Baum, Eric B. 2008a. *Relevance Based Planning, a Worked Example.* Submitted (this volume).

Baum, Eric B. 2008b. *A Model of Language as a Cognitive Tool.* In preparation.

Baum, E. B. and I. Durdanovic. 2000. *Evolution of Cooperative Problem-Solving in an Artificial Economy*. Neural Computation 12:2743-2775.

Baum, E. B. and I. Durdanovic. 2002. *An artificial economy of Post production system*s. In *Advances in Learning Classifier Systems*, P.L. Lanzi, W. Stoltzmann and S.M. Wilson (eds.). Berlin: Springer-Verlag, 3-21.

Johnson B. and M-A Williams. 2008. *Comirit: Commonsense Reasoning by Integrating Simulation* and Logic. In. *Artificial General Intelligence 2008, Proceedings of the First AGI Conference.* Eds P. Wang, B. Goertzel and S. Franklin. Washington DC: IOS Press.

Hobbs J. R. 2004. *Discourse and Inference.* Preprint. http://www.isi.edu/~hobbs/disinf-tc.html

Kirschner M. and J. Gerhart. 2005. *The Plausibility of Life*. New Haven: Yale University Press.

Schaul, T. 2005. *Evolution of a compact Sokoban solver.* Master Thesis, École Polytechnique Fédérale de Lausanne. posted on http://whatisthought.com/eric.html.

Smith, W. D. (2006) *Mathematical Definition of `Intelligence' (and Consequences)*, preprint, reference 93 on http://math.temple.edu/~wds/homepage/works.html

Winograd, T. and F. Flores. 1987. *Understanding Computers and Cognition: A New Foundation for Design*. Boston: Addison-Wesley Professional