A Family of Gödel Machine Implementations

Bas R. Steunebrink and Jürgen Schmidhuber

IDSIA & University of Lugano, Switzerland, {bas,juergen}@idsia.ch

Abstract. The Gödel Machine is a universal problem solver encoded as a completely self-referential program capable of rewriting any part of itself, provided it can prove that the rewrite is useful according to some utility function, encoded within itself. Based on experience gained by constructing a virtual machine capable of running the first Gödel Machine implementation written in self-referential code, we discuss several important refinements of the original concept. We also show how different approaches to implementing the proof search leads to a family of possible Gödel Machine implementations.

1 Introduction

The fully self-referential Gödel Machine [8, 7, 9] is a universal AI that is theoretically optimal in a certain sense. It may interact with some initially unknown, partially observable environment to solve arbitrary user-defined computational tasks by maximizing expected cumulative future utility. Its initial algorithm is not hardwired; it can completely rewrite itself without essential limits apart from the limits of computability, provided a proof searcher embedded within the initial algorithm can first prove that the rewrite is useful, according to its formalized utility function taking into account the limited computational resources. Self-rewrites due to this approach can be shown to be *globally optimal* with respect to the initial utility function (e.g., a Reinforcement Learner's reward function), relative to Gödel's well-known fundamental restrictions of provability [2].

The original Gödel Machine description [10] outlines the general concept and provides implementation details only where necessary to address potential doubts about feasibility. To the best of our knowledge, however, no full implementation has existed before. In this paper, we show how inconvenient aspects of the original Gödel Machine specification can be overcome by presenting a new one geared towards implementation, offering clarifications of subtle details. In particular, we specify in novel ways both the utility function framework and the target theorem at the core of the Gödel Machine. Based on experience gained from our ongoing effort to build a full Gödel Machine, we also provide several essential implementation details, including a technique for attaining self-reflectivity, demonstrating that implementing a Gödel Machine is not as hard as it may seem at first sight.

2 Gödel Machine Specification

One can view a Gödel Machine as a program consisting of two parts. One part, which we will call the *solver*, can be any problem-solving program. For clarity of presentation, we will pretend the *solver* is a Reinforcement Learning (RL) program interacting with some environment. This will provide us with a convenient way of determining *utility* (using the RL program's reward function), which will be an important topic later on. But in general, no constraints are placed on the *solver.* The second part of the Gödel Machine, which we will call the *searcher*, is a program that tries to improve the entire Gödel Machine in a provably optimal way. This *searcher* will be the main topic of this paper.

In order to discuss our implementation of a Gödel Machine, we must first critically examine the original specification [10, 9] and show where, how, and why we deviate from it. In [9], a Gödel Machine's *searcher* is said to hold a function computing the machine's expected cumulative future utility as follows.

$$u(s, env) = \mathbb{E}_\mu\left[\sum_{t=time}^{T} r(t) \mid env\right] \tag{ORIG.}$$

So given a state s, environment env, environment distribution μ, lifetime T, current time $time$, and reward function r, the utility of s is determined by summing all expected future rewards up to the expected end of the machine's life (which may be extended through appropriate self-modifications). This utility function u is then used to construct a *target theorem*; that is, the Gödel Machine is *only* allowed to perform a self-modification if a part of memory referred to as *proof* contains a theorem of the following form.

$$u(s(t) \oplus (switchbit(t) = \text{‘1’}), env(t)) > u(s(t) \oplus (switchbit(t) = \text{‘0’}), env(t)) \tag{ORIG.}$$

(The notation $s \oplus (v = x)$ stands for the state obtained when element v of s is set to x; $v(t)$ for (the expected value of) element v at time t.) To understand this theorem, several more details about the *searcher* must be explained. The *searcher* holds a procedure *check* which invokes another procedure referred to as *switchprog* if and only if a bit called *switchbit* is set to 1. The procedure *switchprog*, which is assumed to be created during the proof search, may modify any part of the Gödel Machine's software. But *switchprog* is invoked *only* when the target theorem shown above can be proved to hold. So the target theorem weighs self-modification against proceeding as usual, because *switchprog* cannot be invoked when *switchbit* is set to 0.

The concept of only allowing self-modification when it is *provably* profitable is intuitively sound; however, the specification of Gödel Machines as introduced in [10] presents several difficulties when an actual implementation is attempted. We identify three main issues: (1) the time-based view of the future makes the utility function u tricky to implement, glossing over certain subtleties; (2) the target theorem is decoupled from its *proof* and does not explicate the relation between *switchbit* and *switchprog*; and (3) it is not obvious how env, μ, T, and r are encoded in state s. These issues call for more explanation, but as we shall see next, they can actually be solved in a unified way.

The function u for determining the expected cumulative future utility, as shown above, requires us to sum all rewards for all future time steps. Here "time steps" actually means not clock ticks, but execution of elementary instructions. Indeed, each instruction takes time to execute, so if we can find a way to explicitly represent the instructions that are going to be executed in the future, we automatically have a window into a future time. An obvious choice of such a representation is the *continuation*, which is a well-studied concept in light of λ-calculus-based programming languages (e.g., LISP, Scheme) [6]. As we shall see, using continuations will allow us to remove t and T from the utility function while *switchprog* can be explicitly introduced in the target theorem. Intuitively, a continuation can be seen as the opposite of a call stack; instead of showing "where we came from," a continuation explicitly shows "what is going to happen next." Note that in all but the simplest cases, a continuation will only be partially expanded. For example, suppose the current continuation is { `A(); if B() then C() else D() fi` }; this continuation specifies that the next thing to be done is expanding `A` and executing its body, and then the conditional statement will be executed, which means that first `B` will be expanded and depending on its result, either `C` or `D` will be expanded. Note that before executing `B`, it is not clear yet whether `C` or `D` will be executed in the future; so it makes no sense to expand either of them before we know the result of `B`.

In what follows we consistently use subscripts to indicate where some element is encoded. With the use of continuations, u becomes a function of two parameters, $u_{\bar{s}}(s,c)$, which represents the expected cumulative future utility of running continuation c on state s. Here $\bar{s}$ represents the evaluating state (where u is encoded), whereas s is the evaluated state. The reason for this separation will become clear when considering the calculation of u:

$$u_{\bar{s}}(s,c) = \mathbb{E}_{\mu_s,M_s}[\,u'\,] \quad \text{with } u'(\mathit{env}) = r_{\bar{s}}(s,\mathit{env}) + \mathbb{E}_{\kappa_c,K_c}[\,u_{\bar{s}} \mid \mathit{env}\,] \qquad (1)$$

As indicated with subscripts, we have moved the representation M of the (external) environment inside s, because all knowledge a Gödel Machine has must be encoded in s. For clarity, let M be a set of bitstrings, each constituting a representation of the environment held possible by the Gödel Machine. μ is a mapping from M to probabilities, also encoded in s. c encodes not only a (partially expanded) representation of the instructions that are going to be executed in the future, but also a set K of state–continuation pairs representing which possible next states and continuations can result from executing the first instruction in c, and a mapping κ from K to probabilities. So μ and κ are (discrete) probability distributions on sample spaces M and K, respectively. $r_{\bar{s}}(s,\mathit{env})$ determines whether state s is rewarding given environment env. For example, in the case where *solver* (which is part of s) is an RL program, $r_{\bar{s}}(s,\mathit{env})$ will be nonzero only when s represents a state just after performing an input receiving instruction. Finally, the term $\mathbb{E}_{\kappa_c,K_c}[\,u_{\bar{s}} \mid \mathit{env}\,]$ recurses on u with the state and continuation following from executing the next instruction in continuation c.

It is *crucial* to note that u and r are taken from the evaluating state $\bar{s}$, *not* from the state s under evaluation. Doing the latter would break the global optimality [9] of the Gödel Machine, because it would be capable of rewriting its utility function in arbitrary ways. Therefore, the original, unchanged utility function of $\bar{s}$ must be used to first show that a rewrite is useful. In contrast, μ and M *are* taken from s, because for any interesting Gödel Machine, c will both contain instructions that affect the external environment (e.g., output to actuators) and instructions that update the internal model of the environment (encoded in s) accordingly. To make use of these updated models, u must extract them from s. Likewise, κ and K are taken from c, because after each instruction the continuation necessarily changes, resulting in new κ and K.

It is also important to notice that our function u is recursive, whereas the original u is a summation up to the end of the Gödel Machine's lifetime T. In formula (1), a Gödel Machine's life ends when the continuation sample space K_c is empty. In that case, the term $\mathbb{E}_{\kappa_c,K_c}[\,u_{\bar{s}} \mid \mathit{env}\,]$ is zero and recursion on u stops. In the most common case, however, an instruction will be deterministic, e.g., $K_c = \{(s', c')\}$. Then formula (1) reduces to $u_{\bar{s}}(s,c) = \mathbb{E}_{\mu_s,M_s}[\,r_{\bar{s}}(s,\,\cdot\,)\,] + u_{\bar{s}}(s', c')$. So for chains of deterministic state transitions, the expected cumulative future utility reduces to a summation of expected rewards in the successive states. But as observed earlier, even those expected rewards are usually zero, because most instructions are not involved with receiving information from the environment (which in e.g. RL is usually the source of rewards).

We are now in a position to introduce a revised *target theorem*, based on the new utility function. Thanks to our continuation-based perspective, we can explicitly introduce *switchprog* into the target theorem by constructing a continuation out of it. Specifically, let $c^* = \{\mathit{wait\_until}(t); \mathit{switchprog}()\}$ denote the continuation that waits until time t (e.g., by continually calling a primitive clock function until the returned time tests as greater or equal to t) and then executes *switchprog*. Furthermore, let s be an expected future state *before* time t. Now $u_{\bar{s}}(s, c^*)$ tells us what is the expected cumulative future utility of running *switchprog* on s at time t. If this utility compares positively against the expected cumulative future utility of *not* switching, then the Gödel Machine must perform the switch at time t. Thus, the *searcher* is to find a state s, a time t, and a nullary procedure *switchprog* such that $\mathtt{tt}_{\bar{s}}(s, t, \mathit{switchprog})$ holds, where:

$$\mathtt{tt}_{\bar{s}}(s, t, \mathit{switchprog}) = \Big[\, u_{\bar{s}}(s, \{\mathit{wait\_until}_s(t); \mathit{switchprog}()\}) > u_{\bar{s}}(s, \{\mathit{scheduler}_s()\})\,\Big] \quad (2)$$

where *scheduler* is the procedure that interleaves the Gödel Machine's *solver* and *searcher*. Thus $\{\mathit{scheduler}_s()\}$ denotes the continuation that "proceeds as usual" from s (including searching for new *switchprog*s). Note that, since *switchprog* may overwrite the scheduler, it must as its last instruction invoke a new scheduler (if needed), or simply invoke the old scheduler if it has been left intact.

There are two ways in which this target theorem can be used to find provably useful self-rewrites: (1) generate a candidate *switchprog* (e.g., a program performing optimization techniques also applied by state-of-the-art compilers), generate a future switch time t at least as far ahead as the time the evaluation of the utility of running *switchprog* is expected to take, and make s a copy of the current state $\bar{s}$; then run both continuations specified by $\mathtt{tt}$ on s (this is safe because s acts like a sandbox) until the inequality can be shown to be true or false, or until the current time exceeds t; if the inequality is false or time has run out, generate a new *switchprog* and start over; else, swap the current state for s, wait until time t and execute *switchprog*. (2) Start with an axiomatic system with axiom and rules, and generate theorems (or generate programs that calculate theorems [9]), testing them for equivalence with $\mathtt{tt}$. The first method can be seen as a top-down approach to proving $\mathtt{tt}$ and the latter as bottom-up.

Hybrid approaches are also possible, so our specification of the target theorem allows for a *family* of possible Gödel Machine implementations.

To make sure the Gödel Machine is at least *asymptotically* optimal even before the first self-rewrite, it may be initialized with Hutter's non-self-referential but *asymptotically fastest algorithm for all well-defined problems* HSEARCH [3], which uses a hardwired brute force proof searcher and ignores the costs of proof search. Given some problem, the Gödel Machine may decide to replace HSEARCH by a faster method suffering less from large constant overhead, but even if it does not, its performance will not be less than asymptotically optimal.

3 Discussion and Conclusion

In this paper we have presented a novel Gödel Machine specification geared towards implementation. Our own approach so far has been to implement a virtual machine capable of running a specially invented programming language with self-referential constructs to attain the self-reflexivity needed for a Gödel Machine. The *solver*, *searcher*, and *scheduler* are then implemented in this language. It should be noted though, that a simpler existing technique can be used to attain self-reflexivity, namely by using *meta-circular evaluators* [1]. A meta-circular evaluator is basically an interpreter for the same programming language as the one in which the interpreter is written. Especially suitable for this technique are homoiconic languages such as Scheme [5], which is very close to λ-calculus and is often used to study meta-circular evaluators and self-reflection in programming in general [1, 6, 4]. So a meta-circular Scheme evaluator is a program written in Scheme which can interpret programs written in Scheme. Using the technique of a global execution environment as insightfully described in [4], complete self-inspection and self-modification can be attained by having a *double nesting* of meta-circular evaluators run the Gödel Machine's *scheduler*. Although this technique is pretty inefficient, there is in principle no need to build a special (virtual) machine. Ultimately, the Gödel Machine should be directly implemented in an assembly language, to make it capable of working in tandem with arbitrary compiled problem solvers, instead of needing access to their source code and translating it into the virtual machine's programming language. This requires an axiomatic encoding of the instruction set of the architecture on which the Gödel Machine is going to run. On the positive side, however, reflexivity comes for free in assembly languages, given von Neumann-like hardware architectures.

It is interesting to note that "gödelizing"[1] an existing problem solver is always harmless. Before the first self-rewrite, the proof searcher of a Gödel Machine will do little but consume a fixed percentage of processing time, say 50%. This loss is easily offset by simply running the entire program on a machine which is twice as fast. The gain is a program that may improve over time in a way that is globally optimal [9] with respect to its initial utility function. We should caution though that this puts a burden on the programmer: a Gödel Machine with a badly chosen utility function is motivated to converge to a "poor" program.

[1] I.e., adding a scheduler to a problem solving program which interleaves that solver with a program searching for provably useful self-rewrites. Thanks to Moshe Looks for suggesting this term to us.

Acknowledgments

This work was partially funded by the Humanobs EU Project (FP7-ICT-231453).

References

1. Abelson, H., Sussman, G.J., Sussman, J.: Structure and Interpretation of Computer Programs. MIT Press, second edn. (1996)
2. Gödel, K.: Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. Monatshefte für Mathematik und Physik 38, 173–198 (1931)
3. Hutter, M.: The fastest and shortest algorithm for all well-defined problems. International Journal of Foundations of Computer Science 13(3), 431–443 (2002)
4. Jefferson, S., Friedman, D.P.: A simple reflective interpreter. LISP and Symbolic Computation 9(2-3), 181–202 (1996)
5. Kelsey, R., Clinger, W., Rees, J., (eds.): Revised^5 report on the algorithmic language Scheme. Higher-Order and Symbolic Computation 11(1) (August 1998)
6. Queinnec, C.: Lisp in Small Pieces. Cambridge University Press (1996)
7. Schmidhuber, J.: Completely self-referential optimal reinforcement learners. In: Duch, W., Kacprzyk, J., Oja, E., Zadrozny, S. (eds.) Artificial Neural Networks: Biological Inspirations - ICANN 2005, LNCS 3697. pp. 223–233. Springer-Verlag Berlin Heidelberg (2005), plenary talk
8. Schmidhuber, J.: Gödel machines: Fully self-referential optimal universal self-improvers. In: Goertzel, B., Pennachin, C. (eds.) Artificial General Intelligence, pp. 199–226. Springer Verlag (2006), variant available as arXiv:cs.LO/0309048
9. Schmidhuber, J.: Ultimate cognition *à la* Gödel. Cognitive Computation 1(2), 177–193 (2009)
10. Schmidhuber, J.: Gödel machines: Self-referential universal problem solvers making provably optimal self-improvements. Tech. Rep. IDSIA-19-03, arXiv:cs.LO/0309048 v2, IDSIA (2003)