Gödel Machines: Towards a Technical Justification of Consciousness

Jürgen Schmidhuber

IDSIA, Galleria 2, 6928 Manno (Lugano), Switzerland &
TU Munich, Boltzmannstr. 3, 85748 Garching, München, Germany
juergen@idsia.ch - http://www.idsia.ch/~juergen

Abstract. The growing literature on consciousness does not provide a formal demonstration of the *usefulness* of consciousness. Here we point out that the recently formulated Gödel machines may provide just such a technical justification. They are the first mathematically rigorous, general, fully self-referential, self-improving, optimally efficient problem solvers, "conscious" or "self-aware" in the sense that their entire behavior is open to introspection, and modifiable. A Gödel machine is a computer that rewrites any part of its own initial code as soon as it finds a proof that the rewrite is *useful*, where the problem-dependent *utility function*, the hardware, and the entire initial code are described by axioms encoded in an initial asymptotically optimal proof searcher which is also part of the initial code. This type of total self-reference is precisely the reason for the Gödel machine's optimality as a general problem solver: any self-rewrite is globally optimal—no local maxima!—since the code first had to prove that it is not useful to continue the proof search for alternative self-rewrites.

1 Introduction and Outline

In recent years the topic of consciousness has gained some credibility as a serious research issue, at least in philosophy and neuroscience, e.g., [8]. However, there is a lack of *technical* justifications of consciousness: so far no one has shown that consciousness is really useful for solving problems, even though problem solving is considered of central importance in philosophy [29].

Our fully self-referential Gödel machine [43, 45] may be viewed as providing just such a technical justification. It is "self-aware" or "conscious" in the sense that the algorithm determining its behavior is completely open to self-inspection, and modifiable in a very general (but computable) way. It can 'step outside of itself' [13] by executing self-changes that are provably good, where the mechanism for generating the proofs also is part of the initial code and thus subject to analysis and change. We will see that this type of total self-reference makes the Gödel machine an *optimal* general problem solver, in the sense of Global Optimality Theorem 1, to be discussed in Section 4.

Outline. Section 2 presents basic concepts of Gödel machines, relations to the most relevant previous work, and limitations. Section 3 presents the essential details of a self-referential axiomatic system of one particular Gödel machine, Section 4 the Global

Optimality Theorem 1, and Section 5 an $O()$-optimal (Theorem 2) initial proof searcher. Section 6 provides examples and additional relations to previous work, and lists answers to several frequently asked questions about Gödel machines. Section 7 wraps up.

2 Basic Overview / Most Relevant Previous Work / Limitations

All traditional algorithms for problem solving are hardwired. Some are designed to improve some limited type of policy through experience [19], but are not part of the modifiable policy, and cannot improve themselves in a theoretically sound way. Humans are needed to create new / better problem solving algorithms and to prove their usefulness under appropriate assumptions.

Here we eliminate the restrictive need for human effort in the most general way possible, leaving all the work including the proof search to a system that can rewrite and improve itself in arbitrary computable ways and in a most efficient fashion. To attack this *"Grand Problem of Artificial Intelligence,"* we introduce a novel class of optimal, fully self-referential [10] general problem solvers called *Gödel machines* [43, 44].[1] They are universal problem solving systems that interact with some (partially observable) environment and can in principle modify themselves without essential limits apart from the limits of computability. Their initial algorithm is not hardwired; it can completely rewrite itself, but only if a proof searcher embedded within the initial algorithm can first prove that the rewrite is useful, given a formalized utility function reflecting computation time and expected future success (e.g., rewards). We will see that self-rewrites due to this approach are actually *globally optimal* (Theorem 1, Section 4), relative to Gödel's well-known fundamental restrictions of provability [10]. These restrictions should not worry us; if there is no proof of some self-rewrite's utility, then humans cannot do much either.

The initial proof searcher is $O()$-optimal (has an optimal order of complexity) in the sense of Theorem 2, Section 5. Unlike hardwired systems such as Hutter's [15, 16] (Section 2) and Levin's [23, 24], however, a Gödel machine can in principle speed up any part of its initial software, including its proof searcher, to meet *arbitrary* formalizable notions of optimality beyond those expressible in the $O()$-notation. Our approach yields the first theoretically sound, fully self-referential, optimal, general problem solvers.

2.1 Set-up and Formal Goal

Many traditional problems of computer science require just one problem-defining input at the beginning of the problem solving process. For example, the initial input may be a large integer, and the goal may be to factorize it. In what follows, however, we will also consider the *more general case* where the problem solution requires interaction with a dynamic, initially unknown environment that produces a continual stream of inputs and

[1] Or *'Goedel machine'*, to avoid the *Umlaut*. But *'Godel machine'* would not be quite correct. Not to be confused with what Penrose calls, in a different context, *'Gödel's putative theorem-proving machine'* [28]!

feedback signals, such as in autonomous robot control tasks, where the goal may be to maximize expected cumulative future reward [19]. This may require the solution of essentially arbitrary problems (examples in Section 6.1 formulate traditional problems as special cases).

Our hardware (e.g., a universal or space-bounded Turing machine [55] or the abstract model of a personal computer) has a single life which consists of discrete cycles or time steps $t = 1, 2, \ldots$. Its total lifetime T may or may not be known in advance. In what follows, the value of any time-varying variable Q at time t will be denoted by $Q(t)$.

During each cycle our hardware executes an elementary operation which affects its variable state $s \in \mathcal{S} \subset \mathcal{B}^*$ (where B^* is the set of possible bitstrings over the binary alphabet $B = \{0, 1\}$) and possibly also the variable environmental state $Env \in \mathcal{E}$ (here we need not yet specify the problem-dependent set $\mathcal{E}$). There is a hardwired state transition function $F : \mathcal{S} \times \mathcal{E} \to \mathcal{S}$. For $t > 1$, $s(t) = F(s(t-1), Env(t-1))$ is the state at a point where the hardware operation of cycle $t-1$ is finished, but the one of t has not started yet. $Env(t)$ may depend on past output actions encoded in $s(t-1)$ and is simultaneously updated or (probabilistically) computed by the possibly reactive environment.

In order to talk conveniently about programs and data, we will often attach names to certain string variables encoded as components or substrings of s. Of particular interest are the three variables called *time*, *x*, *y*, and *p*:

1. At time t, variable $time$ holds a unique binary representation of t. We initialize $time(1) =$ '1', the bitstring consisting only of a one. The hardware increments $time$ from one cycle to the next. This requires at most $O(log\ t)$ and on average only $O(1)$ computational steps.
2. Variable *x* holds the inputs form the environment to the Gödel machine. For $t > 1$, $x(t)$ may differ from $x(t-1)$ only if a program running on the Gödel machine has executed a special input-requesting instruction at time $t-1$. Generally speaking, the delays between successive inputs should be sufficiently large so that programs can perform certain elementary computations on an input, such as copying it into internal storage (a reserved part of s) before the next input arrives.
3. Variable *y* holds the outputs of the Gödel machine. $y(t)$ is the output bitstring which may subsequently influence the environment, where $y(1) =$ '0' by default. For example, $y(t)$ could be interpreted as a control signal for an environment-manipulating robot whose actions may have an effect on future inputs.
4. $p(1)$ is the initial software: a program implementing the original (sub-optimal) policy for interacting with the environment, represented as a substring $e(1)$ of $p(1)$, plus the original policy for searching proofs. Details will be discussed below.

At any given time t ($1 \leq t \leq T$) the goal is to maximize future success or *utility*. A typical *"value to go"* utility function is of the form $u(s, Env) : \mathcal{S} \times \mathcal{E} \to \mathcal{R}$, where $\mathcal{R}$ is the set of real numbers:

$$u(s, Env) = E_\mu \left[\sum_{\tau=time}^{E_\mu(T|s,Env)} r(\tau) \;\middle|\; s, Env \right], \tag{1}$$

where $r(t)$ is a real-valued reward input (encoded within $s(t)$) at time t, $E_\mu(\cdot \mid \cdot)$ denotes the conditional expectation operator with respect to some possibly unknown distribution μ from a set M of possible distributions (M reflects whatever is known about the possibly probabilistic reactions of the environment), and the above-mentioned $time = time(s)$ is a function of state s which uniquely identifies the current cycle. Note that we take into account the possibility of extending the expected lifespan $E_\mu(T \mid s, Env)$ through appropriate actions.

Alternative formalizable utility functions could favor improvement of *worst case* instead of *expected* future performance, or higher reward intake *per time interval* etc. Clearly, most classic problems of computer science can be formulated in this framework—see examples in Section 6.1.

2.2 Basic Idea of Gödel Machine

Our machine becomes a self-referential [10] *Gödel machine* by loading it with a particular form of machine-dependent, self-modifying code p. The initial code $p(1)$ at time step 1 includes a (typically sub-optimal) problem solving subroutine $e(1)$ for interacting with the environment, such as any traditional reinforcement learning algorithm [19], and a general proof searcher subroutine (Section 5) that systematically makes pairs *(switchprog, proof)* (variable substrings of s) until it finds a *proof* of a target theorem which essentially states: *'the immediate rewrite of* p *through current program* switchprog *on the given machine implies higher utility than leaving* p *as is'*. Then it executes *switchprog*, which may completely rewrite p, including the proof searcher. Section 3 will explain details of the necessary initial axiomatic system $\mathcal{A}$ encoded in $p(1)$. Compare Figure 1.

The **Global Optimality Theorem** (Theorem 1, Section 4) shows this self-improvement strategy is not greedy: since the utility of *'leaving* p *as is'* implicitly evaluates all possible alternative *switchprog*s which an unmodified p might find later, we obtain a globally optimal self-change—the *current switchprog* represents the best of all possible relevant self-changes, relative to the given resource limitations and initial proof search strategy.

2.3 Proof Techniques and an $O()$-optimal Initial Proof Searcher

Section 5 will present an $O()$-optimal initialization of the proof searcher, that is, one with an optimal *order* of complexity (Theorem 2). Still, there will remain a lot of room for self-improvement hidden by the $O()$-notation. The searcher uses an online extension of *Universal Search* [23, 24] to systematically test *online proof techniques*, which are proof-generating programs that may read parts of state s (similarly, mathematicians are often more interested in proof techniques than in theorems). To prove target theorems as above, proof techniques may invoke special instructions for generating axioms and applying inference rules to prolong the current *proof* by theorems. Here an axiomatic system $\mathcal{A}$ encoded in $p(1)$ includes axioms describing **(a)** how any instruction invoked by a program running on the given hardware will change the machine's state s (including instruction pointers etc.) from one step to the next (such that proof techniques can

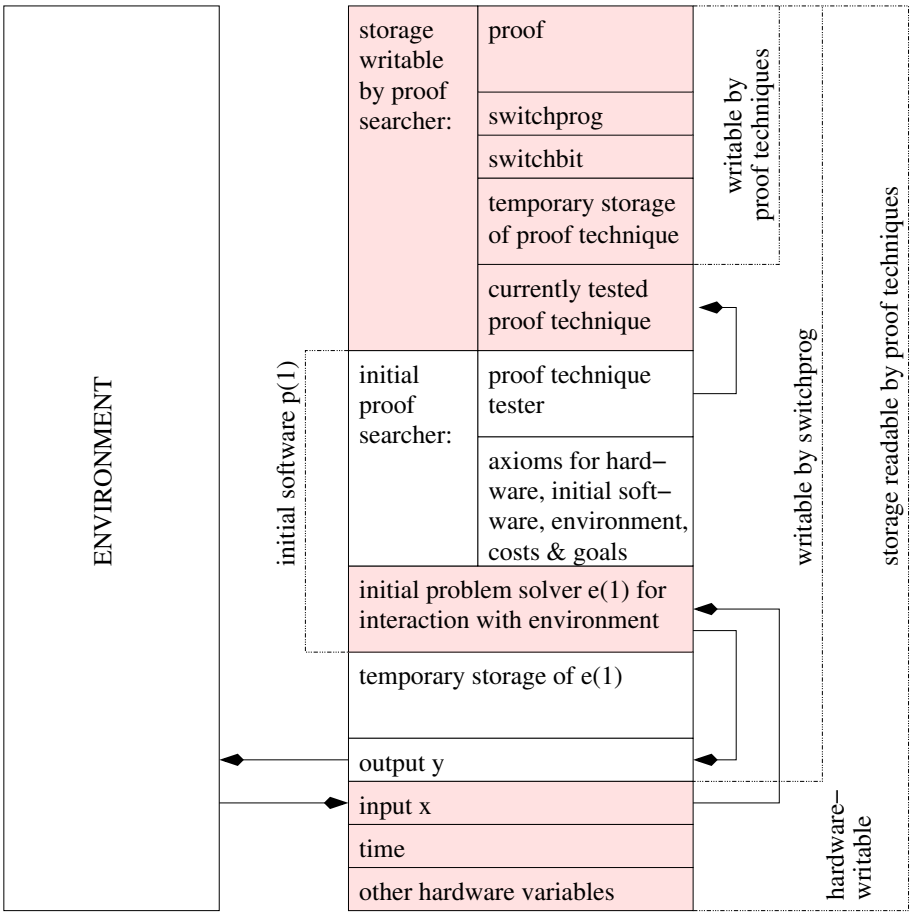

Fig. 1. Storage snapshot of a not yet self-improved example Gödel machine, with the initial software still intact. See text for details.

reason about the effects of any program including the proof searcher), **(b)** the initial program $p(1)$ itself (Section 3 will show that this is possible without introducing circularity), **(c)** stochastic environmental properties, **(d)** the formal utility function u, e.g., equation (1), which automatically takes into account computational costs of all actions including proof search.

2.4 Relation to Hutter's Previous Work

Here we will briefly review the most closely related previous work, and point out the main novelties of the Gödel machine. More relations to older approaches can be found in Section 6.2.

Hutter's non-self-referential but still $O()$-optimal *'fastest' algorithm for all well-defined problems* HSEARCH [16] uses a *hardwired* brute force proof searcher and ignores the costs of proof search. Assume discrete input/output domains X/Y, a formal problem specification $f : X \to Y$ (say, a functional description of how integers are decomposed into their prime factors), and a particular $x \in X$ (say, an integer to be factorized). HSEARCH orders all proofs of an appropriate axiomatic system by size to find programs q that for all $z \in X$ provably compute $f(z)$ within time bound $t_q(z)$. Simultaneously it spends most of its time on executing the q with the best currently proven time bound $t_q(x)$. It turns out that HSEARCH is as fast as the *fastest* algorithm that provably computes $f(z)$ for all $z \in X$, save for a constant factor smaller than $1 + \epsilon$ (arbitrary $\epsilon > 0$) and an f-specific but x-independent additive constant [16]. This constant may be enormous though.

Hutter's AIXI*(t,l)* [15] is related. In discrete cycle $k = 1, 2, 3, \ldots$ of AIXI*(t,l)*'s lifetime, action $y(k)$ results in perception $x(k)$ and reward $r(k)$, where all quantities may depend on the complete history. Using a universal computer such as a Turing machine, AIXI*(t,l)* needs an initial offline setup phase (prior to interaction with the environment) where it uses a *hardwired* brute force proof searcher to examine all proofs of length at most L, filtering out those that identify programs (of maximal size l and maximal runtime t per cycle) which not only could interact with the environment but which for all possible interaction histories also correctly predict a lower bound of their own expected future reward. In cycle k, AIXI*(t,l)* then runs all programs identified in the setup phase (at most 2^l), finds the one with highest self-rating, and executes its corresponding action. The problem-independent setup time (where almost all of the work is done) is $O(L \cdot 2^L)$. The online time per cycle is $O(t \cdot 2^l)$. Both are constant but typically huge.

Advantages and Novelty of the Gödel Machine. There are major differences between the Gödel machine and Hutter's HSEARCH [16] and AIXI*(t,l)* [15], including:

1. The theorem provers of HSEARCH and AIXI*(t,l)* are hardwired, non-self-referential, unmodifiable meta-algorithms that cannot improve themselves. That is, they will always suffer from the same huge constant slowdowns (typically $\gg 10^{1000}$) buried in the $O()$-notation. But there is nothing in principle that prevents our truly self-referential code from proving and exploiting drastic reductions of such constants, in the best possible way that provably constitutes an improvement, if there is any.
2. The demonstration of the $O()$-optimality of HSEARCH and AIXI*(t,l)* depends on a clever allocation of computation time to some of their unmodifiable meta-algorithms. Our Global Optimality Theorem (Theorem 1, Section 4), however, is justified through a quite different type of reasoning which indeed exploits and crucially depends on the fact that there is no unmodifiable software at all, and that the proof searcher itself is readable, modifiable, and can be improved. This is also the reason why its self-improvements can be more than merely $O()$-optimal.
3. HSEARCH uses a "trick" of proving more than is necessary which also disappears in the sometimes quite misleading $O()$-notation: it wastes time on finding programs that provably compute $f(z)$ for all $z \in X$ even when the current $f(x)(x \in X)$ is the only object of interest. A Gödel machine, however, needs to prove only what is relevant to its goal formalized by u. For example, the general u of eq. (1) completely

ignores the limited concept of $O()$-optimality, but instead formalizes a stronger type of optimality that does not ignore huge constants just because they are constant.

4. Both the Gödel machine and AIXI*(t,l)* can maximize expected reward (HSEARCH cannot). But the Gödel machine is more flexible as we may plug in *any* type of formalizable utility function (e.g., *worst case* reward), and unlike AIXI*(t,l)* it does not require an enumerable environmental distribution.

Nevertheless, we may use AIXI*(t,l)* or HSEARCH or other less general methods to initialize the substring e of p which is responsible for interaction with the environment. The Gödel machine will replace $e(1)$ as soon as it finds a provably better strategy.

2.5 Limitations of Gödel Machines

The fundamental limitations are closely related to those first identified by Gödel's celebrated paper on self-referential formulae [10]. Any formal system that encompasses arithmetics (or ZFC etc) is either flawed or allows for unprovable but true statements. Hence even a Gödel machine with unlimited computational resources must ignore those self-improvements whose effectiveness it cannot prove, e.g., for lack of sufficiently powerful axioms in $\mathcal{A}$. In particular, one can construct pathological examples of environments and utility functions that make it impossible for the machine to ever prove a target theorem. Compare Blum's speed-up theorem [3, 4] based on certain incomputable predicates. Similarly, a realistic Gödel machine with limited resources cannot profit from self-improvements whose usefulness it cannot prove within its time and space constraints.

Nevertheless, unlike previous methods, it can in principle exploit at least the *provably* good speed-ups of *any* part of its initial software, including those parts responsible for huge (but problem class-independent) slowdowns ignored by the earlier approaches [15, 16].

3 Essential Details of One Representative Gödel Machine

Notation. Unless stated otherwise or obvious, throughout the paper newly introduced variables and functions are assumed to cover the range implicit in the context. $l(q)$ denotes the number of bits in a bitstring q; q_n the n-th bit of q; λ the empty string (where $l(\lambda) = 0$); $q_{m:n} = \lambda$ if $m > n$ and $q_m q_{m+1} \dots q_n$ otherwise (where $q_0 := q_{0:0} := \lambda$).

Theorem proving requires an axiom scheme yielding an enumerable set of axioms of a formal logic system $\mathcal{A}$ whose formulas and theorems are symbol strings over some finite alphabet that may include traditional symbols of logic (such as $\rightarrow, \wedge, =, (,), \forall, \exists, \dots$, $c_1, c_2, \dots$, $f_1, f_2, \dots$), probability theory (such as $E(\cdot)$, the expectation operator), arithmetics ($+, -, /, =, \sum, <, \dots$), string manipulation (in particular, symbols for representing any part of state s at any time, such as $s_{7:88}(5555)$). A proof is a sequence of theorems, each either an axiom or inferred from previous theorems by applying one of the inference rules such as *modus ponens* combined with *unification*, e.g., [9].

The remainder of this paper will omit standard knowledge to be found in any proof theory textbook. Instead of listing *all* axioms of a particular $\mathcal{A}$ in a tedious fashion, we will focus on the novel and critical details: how to overcome problems with self-reference and how to deal with the potentially delicate online generation of proofs that talk about and affect the currently running proof generator itself.

3.1 Proof techniques

Brute force proof searchers (used in Hutter's AIXI*(t,l)* and HSEARCH; see Section 2.4) systematically generate all proofs in order of their sizes. To produce a certain proof, this takes time exponential in proof size. Instead our $O()$-optimal $p(1)$ will produce many proofs with low algorithmic complexity [51, 21, 25] much more quickly. It systematically tests (see Section 5) *proof techniques* written in universal language $\mathcal{L}$ implemented within $p(1)$. For example, $\mathcal{L}$ may be a variant of PROLOG [6] or the universal FORTH[27]-inspired programming language used in recent work on optimal search [46]. A proof technique is composed of instructions that allow any part of s to be read, such as inputs encoded in variable x (a substring of s) or the code of $p(1)$. It may write on s^p, a part of s reserved for temporary results. It also may rewrite *switchprog*, and produce an incrementally growing proof placed in the string variable *proof* stored somewhere in s. *proof* and s^p are reset to the empty string at the beginning of each new proof technique test. Apart from standard arithmetic and function-defining instructions [46] that modify s^p, the programming language $\mathcal{L}$ includes special instructions for prolonging the current *proof* by correct theorems, for setting *switchprog*, and for checking whether a provably optimal p-modifying program was found and should be executed now. Certain long proofs can be produced by short proof techniques.

The nature of the six *proof*-modifying instructions below (there are no others) makes it impossible to insert an incorrect theorem into *proof*, thus trivializing proof verification:

1. **get-axiom(n)** takes as argument an integer n computed by a prefix of the currently tested proof technique with the help of arithmetic instructions such as those used in previous work [46]. Then it appends the n-th axiom (if it exists, according to the axiom scheme below) as a theorem to the current theorem sequence in *proof*. The initial axiom scheme encodes:
   (a) **Hardware axioms** describing the hardware, formally specifying how certain components of s (other than environmental inputs x) may change from one cycle to the next.
   For example, if the hardware is a Turing machine[2] (TM) [55], then $s(t)$ is a bitstring that encodes the current contents of all tapes of the TM, the positions

[2] Turing reformulated Gödel's unprovability results in terms of Turing machines (TMs) [55] which subsequently became the most widely used abstract model of computation. It is well-known that there are *universal* TMs that in a certain sense can emulate any other TM or any other known computer. Gödel's integer-based formal language can be used to describe any universal TM, and vice versa.

of its scanning heads, and the current *internal state* of the TM's finite state automaton, while F specifies the TM's look-up table which maps any possible combination of internal state and bits above scanning heads to a new internal state and an action such as: replace some head's current bit by 1/0, increment (right shift) or decrement (left shift) some scanning head, read and copy next input bit to cell above input tape's scanning head, etc.
Alternatively, if the hardware is given by the abstract model of a modern microprocessor with limited storage, $s(t)$ will encode the current storage contents, register values, instruction pointers etc.
For instance, the following axiom could describe how some 64-bit hardware's instruction pointer stored in $s_{1:64}$ is continually incremented as long as there is no overflow and the value of s_{65} does not indicate that a jump to some other address should take place:

$$(\forall t \forall n : [(n < 2^{64} - 1) \wedge (n > 0) \wedge (t > 1) \wedge (t < T)$$

$$\wedge (string2num(s_{1:64}(t)) = n) \wedge (s_{65}(t) = \text{`0'})]$$

$$\rightarrow (string2num(s_{1:64}(t+1)) = n + 1))$$

Here the semantics of used symbols such as '(' and '$>$' and '$\rightarrow$' (implies) are the traditional ones, while '$string2num$' symbolizes a function translating bitstrings into numbers. It is clear that any abstract hardware model can be fully axiomatized in a similar way.

(b) **Reward axioms** defining the computational costs of any hardware instruction, and physical costs of output actions, such as control signals $y(t)$ encoded in $s(t)$. Related axioms assign values to certain input events (encoded in variable x, a substring of s) representing reward or punishment (e.g., when a Gödel machine-controlled robot bumps into an obstacle). Additional axioms define the total value of the Gödel machine's life as a scalar-valued function of all rewards (e.g., their sum) and costs experienced between cycles 1 and T, etc. For example, assume that $s_{17:18}$ can be changed only through external inputs; the following example axiom says that the total reward increases by 3 whenever such an input equals '11' (unexplained symbols carry the obvious meaning):

$$(\forall t_1 \forall t_2 : [(t_1 < t_2) \wedge (t_1 \geq 1) \wedge (t_2 \leq T) \wedge (s_{17:18}(t_2) = \text{`11'})]$$

$$\rightarrow [R(t_1, t_2) = R(t_1, t_2 - 1) + 3]),$$

where $R(t_1, t_2)$ is interpreted as the cumulative reward between times t_1 and t_2. It is clear that any formal scheme for producing rewards can be fully axiomatized in a similar way.

(c) **Environment axioms** restricting the way the environment will produce new inputs (encoded within certain substrings of s) in reaction to sequences of outputs y encoded in s. For example, it may be known in advance that the environment is sampled from an unknown probability distribution μ contained in a given set M of possible distributions (compare equation 1). E.g., M may contain all distributions that are computable, given the previous history [51, 52, 15], or

at least limit-computable [39, 40]. Or, more restrictively, the environment may be some unknown but deterministic computer program [57, 37] sampled from the Speed Prior [41] which assigns low probability to environments that are hard to compute by any method. Or the interface to the environment is Markovian [33], that is, the current input always uniquely identifies the environmental state—a lot of work has already been done on this special case [31, 2, 54]. Even more restrictively, the environment may evolve in completely predictable fashion known in advance. All such prior assumptions are perfectly formalizable in an appropriate $\mathcal{A}$ (otherwise we could not write scientific papers about them).

(d) **Uncertainty axioms; string manipulation axioms:** Standard axioms for arithmetics and calculus and probability theory [20] and statistics and string manipulation that (in conjunction with the hardware axioms and environment axioms) allow for constructing proofs concerning (possibly uncertain) properties of future values of $s(t)$ as well as bounds on expected remaining lifetime / costs / rewards, given some time τ and certain hypothetical values for components of $s(\tau)$ etc. An example theorem saying something about expected properties of future inputs x might look like this:

$$(\forall t_1 \forall \mu \in M : [(1 \leq t_1) \wedge (t_1 + 15597 < T) \wedge (s_{5:9}(t_1) = \text{`01011'})$$

$$\wedge (x_{40:44}(t_1) = \text{`00000'})] \rightarrow (\exists t : [(t_1 < t < t_1 + 15597)$$

$$\wedge (P_\mu(x_{17:22}(t) = \text{`011011'} \mid s(t_1)) > \frac{998}{1000})])),$$

where $P_\mu(. \mid .)$ represents a conditional probability with respect to an axiomatized prior distribution μ from a set of distributions M described by the environment axioms (Item 1c).

Given a particular formalizable hardware (Item 1a) and formalizable assumptions about the possibly probabilistic environment (Item 1c), obviously one can fully axiomatize everything that is needed for proof-based reasoning about past and future machine states.

(e) **Initial state axioms:** Information about how to reconstruct the initial state $s(1)$ or parts thereof, such that the proof searcher can build proofs including axioms of the type

$$(s_{\mathbf{m:n}}(1) = \mathbf{z}),\ e.g.:\ (s_{7:9}(1) = \text{`010'}).$$

Here and in the remainder of the paper we use bold font in formulas to indicate syntactic place holders (such as **m,n,z**) for symbol strings representing variables (such as *m,n,z*) whose semantics are explained in the text—in the present context z is the bitstring $s_{m:n}(1)$.

Note that it is **no fundamental problem** to fully encode both the hardware description *and* the initial hardware-describing p within p itself. To see this, observe that some software may include a program that can print the software.

(f) **Utility axioms** describing the overall goal in the form of utility function u; e.g., equation (1) in Section 2.1.

2. **apply-rule(k, m, n)** takes as arguments the index k (if it exists) of an inference rule such as *modus ponens* (stored in a list of possible inference rules encoded within

$p(1)$) and the indices m, n of two previously proven theorems (numbered in order of their creation) in the current *proof*. If applicable, the corresponding inference rule is applied to the addressed theorems and the resulting theorem appended to *proof*. Otherwise the currently tested proof technique is interrupted. This ensures that *proof* is never fed with invalid proofs.

3. **delete-theorem(m)** deletes the m-th theorem in the currently stored *proof*, thus freeing storage such that proof-storing parts of s can be reused and the maximal proof size is not necessarily limited by storage constraints. Theorems deleted from *proof*, however, cannot be addressed any more by *apply-rule* to produce further prolongations of *proof*.
4. **set-switchprog(m,n)** replaces $switchprog$ by $s^p_{m:n}$, provided that $s^p_{m:n}$ is indeed a non-empty substring of s^p, the storage writable by proof techniques.
5. **check()** verifies whether the goal of the proof search has been reached. First it tests whether the last theorem (if any) in *proof* has the form of a **target theorem**. A target theorem states that given the *current* axiomatized utility function u (Item 1f), the utility of a switch from p to the current *switchprog* would be higher than the utility of continuing the execution of p (which would keep searching for alternative *switchprog*s). Target theorems are symbol strings (encoded in s) of the (decoded) form

$$(u[s(\mathbf{t_1}) \oplus (switchbit(\mathbf{t_1}) = \text{`1'}), Env(\mathbf{t_1})] > u[s(\mathbf{t_1}) \oplus (switchbit(\mathbf{t_1}) = \text{`0'}), Env(\mathbf{t_1})]) \quad (2)$$

where the variable t_1 (represented by syntactic place holder $\mathbf{t_1}$) stands for a time step, while all other symbols belong to the alphabet of the theorem-proving calculus, that is, the set of possible target theorems is parameterized only by t_1. Here the calculus should permit the notation $s(\mathbf{t_1}) \oplus (switchbit(\mathbf{t_1}) = \text{`}b\text{'})$ as a shortcut for the state obtained when we replace *switchbit*(t_1), the true value of the variable bit $switchbit$ (encoded in s) at time t_1, by $b \in \{0, 1\}$. This will facilitate the formulation of theorems that compare values conditioned on various alternative hypothetical properties of $s(t_1)$. (Note that $s(t_1)$ may be only partially known by the current proof technique even in environments where $s(t_1)$ and *switchbit*(t_1) are fully predetermined for all valid t_1.)

The purpose of introducing t_1 is to deal with hardware-specific temporal delays that may be involved in checking and switching—it may take a significant amount of time to match abstract symbol strings found during proof search to the Gödel machine's real current state. If a target theorem has been found, *check()* uses a simple prewired subroutine to check whether there is enough time left to set variable *switchbit* (originally 0) to 1 before the continually increasing $time$ will equal t_1. If this subroutine returns a negative result, *check()* exits. Otherwise it sets *switchbit* $:= 1$ (there is no other way of changing *switchbit*). Then it repeatedly tests $time$ until $time > t_1$, to make sure the condition of formula (2) was fulfilled at t_1. Then it transfers control to *switchprog* (there is no other way of calling *switchprog*). The *switchprog* may subsequently rewrite all parts of s, excluding hardware-reserved parts such as $time$ and x, but including p.

6. **state2theorem(m, n)** takes two integer arguments m, n and tries to transform the current contents of $s_{m:n}$ into a theorem of the form

$$(s_{\mathbf{m:n}}(\mathbf{t_1}) = \mathbf{z}),\ e.g.:\ (s_{6:9}(7775555) = \text{`}1001\text{'}),$$

where t_1 represents a time measured (by checking *time*) shortly after *state2theorem* was invoked, and z the bistring $s_{m:n}(t_1)$ (recall the special case $t_1 = 1$ of Item 1e). So we accept the time-labeled current observable contents of any part of s as a theorem that does not have to be proven in an alternative way from, say, the initial state $s(1)$, because the computation so far has already demonstrated that the theorem is true. Thus we may exploit information conveyed by environmental inputs, and the fact that sometimes (but not always) the fastest way to determine the output of a program is to run it.
This non-traditional online interface between syntax and semantics requires special care though. We must avoid inconsistent results through parts of s that change while being read. For example, the present value of a quickly changing instruction pointer IP *(continually updated by the hardware) may be essentially unreadable in the sense that the execution of the reading subroutine itself will already modify* IP *many times. For convenience, the (typically limited) hardware could be set up such that it stores the contents of fast hardware variables every c cycles in a reserved part of s, such that an appropriate variant of* state2theorem() *could at least translate certain recent values of fast variables into theorems. This, however, will not abolish* all *problems associated with self-observations. For example, the $s_{m:n}$ to be read might also contain the reading procedure's own, temporary, constantly changing string pointer variables, etc.*[3] *To address such problems on computers with limited memory,* state2theorem *first uses some fixed protocol to check whether the current $s_{m:n}$ is readable at all or whether it might change if it were read by the remaining code of* state2theorem. *If so, or if $m, n,$ are not in the proper range, then the instruction has no further effect. Otherwise it appends an* observed *theorem of the form $(s_{\mathbf{m:n}}(\mathbf{t_1}) = \mathbf{z})$ to* proof. *For example, if the current time is 7770000, then the invocation of* state2theorem(6,9) *might return the theorem $(s_{6:9}(7775555) = \text{`}1001\text{'})$, where $7775555 - 7770000 = 5555$ reflects the time needed by* state2theorem *to perform the initial check and to read leading bits off the continually increasing $time$ (reading $time$ also costs time) such that it can be sure that 7775555 is a recent proper time label following the start of* state2theorem.

The axiomatic system $\mathcal{A}$ is a defining parameter of a given Gödel machine. Clearly, $\mathcal{A}$ must be strong enough to permit proofs of target theorems. In particular, the theory of

[3] We see that certain parts of the current s may not be directly observable without changing the observable itself. Sometimes, however, axioms and previous observations will allow the Gödel machine to *deduce* time-dependent storage contents that are not directly observable. For instance, by analyzing the code being executed through instruction pointer *IP* in the example above, the value of *IP* at certain times may be predictable (or postdictable, after the fact). The values of other variables at given times, however, may not be deducible at all. Such limits of self-observability are reminiscent of Heisenberg's celebrated uncertainty principle [11], which states that certain physical measurements are necessarily imprecise, since the measuring process affects the measured quantity.

uncertainty axioms (Item 1d) must be sufficiently rich. This is no fundamental problem: we simply insert all traditional axioms of probability theory [20].

4 Global Optimality Theorem

Intuitively, at any given time p should execute some self-modification algorithm (via instruction *check()*—Item 5 above) only if it is the 'best' of all possible self-modifications, given the utility function, which typically depends on available resources, such as storage size and remaining lifetime. At first glance, however, target theorem (2) seems to implicitly talk about just one single modification algorithm, namely, *switchprog*(t_1) as set by the systematic proof searcher at time t_1. Isn't this type of local search greedy? Couldn't it lead to a local optimum instead of a global one? No, it cannot, according to the global optimality theorem:

Theorem 1 (Globally Optimal Self-Changes, given u and $\mathcal{A}$ encoded in p). *Given any formalizable utility function u (Item 1f), and assuming consistency of the underlying formal system $\mathcal{A}$, any self-change of p obtained through execution of some program* switchprog *identified through the proof of a target theorem (2) is globally optimal in the following sense: the utility of starting the execution of the present* switchprog *is higher than the utility of waiting for the proof searcher to produce an alternative* switchprog *later.*

Proof. Target theorem (2) implicitly talks about all the other *switchprog*s that the proof searcher could produce in the future. To see this, consider the two alternatives of the binary decision: (1) either execute the current *switchprog* (set *switchbit* $= 1$), or (2) keep searching for *proof*s and *switchprog*s (set *switchbit* $= 0$) until the systematic searcher comes up with an even better *switchprog*. Obviously the second alternative concerns all (possibly infinitely many) potential *switchprog*s to be considered later. That is, if the current *switchprog* were not the 'best', then the proof searcher would not be able to prove that setting *switchbit* and executing *switchprog* will cause higher expected reward than discarding *switchprog*, assuming consistency of $\mathcal{A}$. *Q.E.D.*

4.1 Alternative Relaxed Target Theorem

We may replace the target theorem (2) (Item 5) by the following alternative target theorem:

$$(u[s(\mathbf{t_1}) \oplus (switchbit(\mathbf{t_1}) = \text{'1'}), Env(\mathbf{t_1})] \geq u[s(\mathbf{t_1}) \oplus (switchbit(\mathbf{t_1}) = \text{'0'}), Env(\mathbf{t_1})]) \quad (3)$$

The only difference to the original target theorem (2) is that the "$>$" sign became a "$\geq$" sign. That is, the Gödel machine will change itself as soon as it has found a proof that the change will not make things worse. A Global Optimality Theorem similar to Theorem 1 holds; simply replace the last phrase in Theorem 1 by: *the utility of starting the execution of the present* switchprog *is at least as high as the utility of waiting for the proof searcher to produce an alternative* switchprog *later.*

4.2 Global Optimality and Recursive Meta-Levels

One of the most important aspects of our fully self-referential set-up is the following. Any proof of a target theorem automatically proves that the corresponding self-modification is good for all further self-modifications affected by the present one, in recursive fashion. In that sense all possible "meta-levels" of the self-referential system are collapsed into one.

4.3 How Difficult is it to Prove Target Theorems?

This depends on the tasks and the initial axioms $\mathcal{A}$, of course. It is straight-forward to devise simple tasks and corresponding consistent $\mathcal{A}$ such that there are short and trivial proofs of target theorems. On the other hand, it is possible to construct set-ups where it is impossible to prove target theorems, for example, by using results of undecidability theory, e.g., [30, 3, 4].

The point is: usually we do not know in advance whether it is possible or not to change a given initial problem solver in a provably good way. The traditional approach is to invest human research effort into finding out. A Gödel machine, however, can do this by itself, without essential limits apart from those of computability and provability.

Note that to prove a target theorem, a proof technique does not necessarily have to compute the true expected utilities of switching and not switching—it just needs to determine which is higher. For example, it may be easy to prove that speeding up a subroutine of the proof searcher by a factor of 2 will certainly be worth the negligible (compared to lifetime T) time needed to execute the subroutine-changing algorithm, no matter what is the precise utility of the switch.

5 Bias-Optimal Proof Search (BIOPS)

Here we construct a $p(1)$ that is $O()$-optimal in a certain limited sense to be described below, but still might be improved as it is not necessarily optimal in the sense of the given u (for example, the u of equation (1) neither mentions nor cares for $O()$-optimality). Our Bias-Optimal Proof Search (BIOPS) is essentially an application of Universal Search [23, 24] to proof search. One novelty, however, is this: Previous practical variants and extensions of Universal Search have been applied [36, 38, 49, 46] to *offline* program search tasks where the program inputs are fixed such that the same program always produces the same results. In our *online* setting, however, BIOPS has to take into account that the same proof technique started at different times may yield different proofs, as it may read parts of s (e.g., inputs) that change as the machine's life proceeds.

BIOPS starts with a probability distribution P (the initial bias) on the proof techniques w that one can write in $\mathcal{L}$, e.g., $P(w) = K^{-l(w)}$ for programs composed from K possible instructions [24]. BIOPS is *near-bias-optimal* [46] in the sense that it will not spend much more time on any proof technique than it deserves, according to its

probabilistic bias, namely, not much more than its probability times the total search time:

Definition 1 (Bias-Optimal Searchers [46]). Let $\mathcal{R}$ be a problem class, $\mathcal{C}$ be a search space of solution candidates (where any problem $r \in \mathcal{R}$ should have a solution in $\mathcal{C}$), $P(q \mid r)$ be a task-dependent bias in the form of conditional probability distributions on the candidates $q \in \mathcal{C}$. Suppose that we also have a predefined procedure that creates and tests any given q on any $r \in \mathcal{R}$ within time $t(q, r)$ (typically unknown in advance). Then *a searcher is n-bias-optimal ($n \geq 1$) if for any maximal total search time $T_{total} > 0$ it is guaranteed to solve any problem $r \in \mathcal{R}$ if it has a solution $p \in \mathcal{C}$ satisfying $t(p, r) \leq P(p \mid r)\, T_{total}/n$. It is bias-optimal if $n = 1$.*

Method 51 (BIOPS) In phase $(i = 1, 2, 3, \ldots)$ DO: FOR all self-delimiting [24] proof techniques $w \in \mathcal{L}$ satisfying $P(w) \geq 2^{-i}$ DO:

1. Run w until halt or error (such as division by zero) or $2^i P(w)$ steps consumed.
2. Undo effects of w on s^p (does not cost significantly more time than executing w).

A proof technique w can interrupt Method 51 only by invoking instruction *check()* (Item 5), which may transfer control to *switchprog* (which possibly even will delete or rewrite Method 51). Since the initial p runs on the formalized hardware, and since proof techniques tested by p can read p and other parts of s, they can produce proofs concerning the (expected) performance of p and BIOPS itself. Method 51 at least has the optimal *order* of computational complexity in the following sense.

Theorem 2. *If independently of variable* time(s) *some unknown fast proof technique w would require at most $f(k)$ steps to produce a proof of difficulty measure k (an integer depending on the nature of the task to be solved), then Method 51 will need at most $O(f(k))$ steps.*

Proof. It is easy to see that Method 51 will need at most $O(f(k)/P(w)) = O(f(k))$ steps—the constant factor $1/P(w)$ does not depend on k. *Q.E.D.*

Note again, however, that the proofs themselves may concern quite different, arbitrary formalizable notions of optimality (stronger than those expressible in the $O()$-notation) embodied by the given, problem-specific, formalized utility function u. This may provoke useful, constant-affecting rewrites of the initial proof searcher despite its limited (yet popular and widely used) notion of $O()$-optimality.

6 Discussion & Additional Relations to Previous Work

Here we list a few examples of Gödel machine applicability to various tasks defined by various utility functions and environments (Section 6.1), and additional relations to previous work (Section 6.2). We also provide a list of answers to frequently asked questions (Section 6.3).

6.1 Example Applications

Traditional examples that do not involve significant interaction with a probabilistic environment are easily dealt with in our reward-based framework:

Example 1 (Time-limited NP-hard optimization). The initial input to the Gödel machine is the representation of a connected graph with a large number of nodes linked by edges of various lengths. Within given time T it should find a cyclic path connecting all nodes. The only real-valued reward will occur at time T. It equals 1 divided by the length of the best path found so far (0 if none was found). There are no other inputs. The by-product of maximizing expected reward is to find the shortest path findable within the limited time, given the initial bias.

Example 2 (Fast theorem proving). Prove or disprove as quickly as possible that all even integers > 2 are the sum of two primes (Goldbach's conjecture). The reward is $1/t$, where t is the time required to produce and verify the first such proof.

More general cases are:

Example 3 (Maximizing expected reward with bounded resources). A robot that needs at least 1 liter of gasoline per hour interacts with a partially unknown environment, trying to find hidden, limited gasoline depots to occasionally refuel its tank. It is rewarded in proportion to its lifetime, and dies after at most 100 years or as soon as its tank is empty or it falls off a cliff etc. The probabilistic environmental reactions are initially unknown but assumed to be sampled from the axiomatized Speed Prior [41], according to which hard-to-compute environmental reactions are unlikely. This permits a computable strategy for making near-optimal predictions [41]. One by-product of maximizing expected reward is to maximize expected lifetime.

Example 4 (Optimize any suboptimal problem solver). Given any formalizable problem, implement a suboptimal but known problem solver as software on the Gödel machine hardware, and let the proof searcher of Section 5 run in parallel.

6.2 More Relations to Prvious, Less General Methods

Despite (or maybe because of) the ambitiousness and potential power of self-improving machines, there has been little work in this vein outside our own labs at IDSIA and TU Munich. Here we will list essential differences between the Gödel machine and our previous approaches to 'learning to learn,' 'metalearning,' self-improvement, self-optimization, etc.

1. **Gödel Machine vs Success-Story Algorithm and Other Metalearners**
   A learner's modifiable components are called its policy. An algorithm that modifies the policy is a learning algorithm. If the learning algorithm has modifiable components represented as part of the policy, then we speak of a self-modifying policy

(SMP) [47]. SMPs can modify the way they modify themselves etc. The Gödel machine has an SMP.

In previous work we used the *success-story algorithm* (SSA) to force some (stochastic) SMP to trigger better and better self-modifications [35, 48, 47, 49]. During the learner's life-time, SSA is occasionally called at times computed according to SMP itself. SSA uses backtracking to undo those SMP-generated SMP-modifications that have not been empirically observed to trigger lifelong reward accelerations (measured up until the current SSA call—this evaluates the long-term effects of SMP-modifications setting the stage for later SMP-modifications). SMP-modifications that survive SSA represent a lifelong success history. Until the next SSA call, they build the basis for additional SMP-modifications. Solely by self-modifications our SMP/SSA-based learners solved a complex task in a partially observable environment whose state space is far bigger than most found in the literature [47].

The Gödel machine's training algorithm is theoretically much more powerful than SSA though. SSA empirically measures the usefulness of previous self-modifications, and does not necessarily encourage provably optimal ones. Similar drawbacks hold for Lenat's human-assisted, non-autonomous, self-modifying learner [22], our Meta-Genetic Programming [32] extending Cramer's Genetic Programming [7, 1], our metalearning economies [32] extending Holland's machine learning economies [14], and gradient-based metalearners for continuous program spaces of differentiable recurrent neural networks [34, 12]. All these methods, however, could be used to seed $p(1)$ with an initial policy.

2. **Gödel Machine vs Universal Search**
   Unlike the fully self-referential Gödel machine, Levin's *Universal Search* [23, 24] has a hardwired, unmodifiable meta-algorithm that cannot improve itself. It is asymptotically optimal for inversion problems whose solutions can be quickly verified in $O(n)$ time (where n is the solution size), but it will always suffer from the same huge constant slowdown factors (typically $>> 10^{1000}$) buried in the $O()$-notation. The self-improvements of a Gödel machine, however, can be more than merely $O()$-optimal, since its utility function may formalize a stonger type of optimality that does not ignore huge constants just because they are constant—compare the utility function of equation (1). The next item points out additional limitations of Universal Search and its extensions.

3. **Gödel Machine vs** OOPS
   The Optimal Ordered Problem Solver OOPS [46, 42] extends Universal Search. It is a bias-optimal (see Def. 1) way of searching for a program that solves each problem in an ordered sequence of problems of a rather general type, continually organizing and managing and reusing earlier acquired knowledge. Solomonoff recently also proposed related ideas for a *scientist's assistant* [53] that modifies the probability distribution of Universal Search [23] based on experience.

   As pointed out earlier [46] (section on OOPS limitations), however, neither Universal Search nor OOPS-like methods are necessarily optimal for general lifelong reinforcement learning (RL) tasks [19] such as those for which AIXI [15] was designed. The simple and natural but limited optimality notion of OOPS is *bias-optimality*

(Def. 1): OOPS is a near-bias-optimal searcher for programs which compute solutions that one can quickly verify (costs of verification are taken into account). For example, one can quickly test whether some currently tested program has computed a solution to the *towers of Hanoi* problem used in the earlier paper [46]: one just has to check whether the third peg is full of disks.

But general RL tasks are harder. Here in principle the evaluation of the value of some behavior consumes the learner's entire life! That is, the naive test of whether a program is good or not would consume the entire life. That is, we could test only one program; afterwards life would be over.

So general RL machines need a more general notion of optimality, and must do things that plain OOPS does not do, such as predicting *future* tasks and rewards. A provably optimal RL machine must somehow *prove* properties of otherwise untestable behaviors (such as: what is the expected reward of this behavior which one cannot naively test as there is not enough time). That is part of what the Gödel machine does: it tries to greatly cut testing time, replacing naive time-consuming tests by much faster proofs of predictable test outcomes whenever this is possible.

Proof verification itself can be performed very quickly. In particular, verifying the correctness of a found proof typically does not consume the remaining life. Hence the Gödel machine may use OOPS as a bias-optimal proof-searching submodule. Since the proofs themselves may concern quite different, *arbitrary* notions of optimality (not just bias-optimality), the Gödel machine is more general than plain OOPS. But it is not just an extension of OOPS. Instead of OOPS it may as well use non-bias-optimal alternative methods to initialize its proof searcher. On the other hand, OOPS is not just a precursor of the Gödel machine. It is a stand-alone, incremental, bias-optimal way of allocating runtime to programs that reuse previously successful programs, and is applicable to many traditional problems, including but not limited to proof search.

4. **Gödel Machine vs** AIXI **etc.**

Unlike Gödel machines, Hutter's recent AIXI *model* [15, 18] generally needs *unlimited* computational resources per input update. It combines Solomonoff's universal prediction scheme [51, 52] with an *expectimax* computation. In discrete cycle $k = 1, 2, 3, \ldots$, action $y(k)$ results in perception $x(k)$ and reward $r(k)$, both sampled from the unknown (reactive) environmental probability distribution μ. AIXI defines a mixture distribution ξ as a weighted sum of distributions $\nu \in \mathcal{M}$, where $\mathcal{M}$ is any class of distributions that includes the true environment μ. For example, $\mathcal{M}$ may be a sum of all computable distributions [51, 52], where the sum of the weights does not exceed 1. In cycle $k + 1$, AIXI selects as next action the first in an action sequence maximizing ξ-predicted reward up to some given horizon. Recent work [17] demonstrated AIXI 's optimal use of observations as follows. The Bayes-optimal policy p^ξ based on the mixture ξ is self-optimizing in the sense that its average utility value converges asymptotically for all $\mu \in \mathcal{M}$ to the optimal value achieved by the (infeasible) Bayes-optimal policy p^μ which knows μ in advance. The necessary condition that $\mathcal{M}$ admits self-optimizing policies is also sufficient. Furthermore, p^ξ is Pareto-optimal in the sense that there is no other pol-

icy yielding higher or equal value in *all* environments $\nu \in \mathcal{M}$ and a strictly higher value in at least one [17].

While AIXI clarifies certain theoretical limits of machine learning, it is computationally intractable, especially when $\mathcal{M}$ includes all computable distributions. This drawback motivated work on the time-bounded, asymptotically optimal AIXI*(t,l)* system [15] and the related HSEARCH [16], both already discussed in Section 2.4, which also lists the advantages of the Gödel machine. Both methods, however, could be used to seed the Gödel machine with an *initial* policy.

It is the *self-referential* aspects of the Gödel machine that relieve us of much of the burden of careful algorithm design required for AIXI*(t,l)* and HSEARCH. They make the Gödel machine both conceptually simpler *and* more general than AIXI*(t,l)* and HSEARCH.

6.3 Frequently Asked Questions

In the past half year the author frequently fielded questions about the Gödel machine. Here a list of answers to typical ones.

1. **Q:** *Does the exact business of formal proof search really make sense in the uncertain real world?*
   A: Yes, it does. We just need to insert into $p(1)$ the standard axioms for representing uncertainty and for dealing with probabilistic settings and expected rewards etc. Compare items 1d and 1c in Section 3.1, and the definition of utility as an *expected* value in equation (1).
2. **Q:** *The target theorem (2) seems to refer only to the* very first *self-change, which may completely rewrite the proof-search subroutine—doesn't this make the proof of Theorem 1 invalid? What prevents later self-changes from being destructive?*
   A: This is fully taken care of. Please have a look once more at the proof of Theorem 1, and note that the first self-change will be executed only if it is provably useful (in the sense of the present untility function u) for all future self-changes (for which the present self-change is setting the stage). This is actually the main point of the whole Gödel machine set-up.
3. **Q** (related to the previous item): *The Gödel machine implements a meta-learning behavior: what about a meta-meta, and a meta-meta-meta level?*
   A: The beautiful thing is that all meta-levels are automatically collapsed into one: any proof of a target theorem automatically proves that the corresponding self-modification is good for all further self-modifications affected by the present one, in recursive fashion. Recall Section 4.2.
4. **Q:** *The Gödel machine software can produce only computable mappings from input sequences to output sequences. What if the environment is non-computable?*
   A: Many physicists and other scientists (exceptions: [57, 37]) actually do assume the real world makes use of all the real numbers, most of which are incomputable. Nevertheless, theorems and proofs are just finite symbol strings, and all treatises of physics contain only computable axioms and theorems, even when some of the theorems can be interpreted as making statements about uncountably many objects,

such as all the real numbers. (Note though that the Löwenheim-Skolem Theorem [26, 50] implies that any first order theory with an uncountable model such as the real numbers also has a countable model.) Generally speaking, formal descriptions of non-computable objects do *not at all* present a fundamental problem—they may still allow for finding a strategy that provably maximizes utility. If so, a Gödel machine can exploit this. If not, then humans will not have a fundamental advantage over Gödel machines.

5. **Q:** *Isn't automated theorem-proving very hard? Current AI systems cannot prove nontrivial theorems without human intervention at crucial decision points.*
   A: More and more important mathematical proofs (four color theorem etc) heavily depend on automated proof search. And traditional theorem provers do not even make use of our novel notions of proof techniques and $O()$-optimal proof search. Of course, some proofs are indeed hard to find, but here humans and Gödel machines face the same fundamental limitations.
6. **Q:** *Don't the "no free lunch theorems" [56] say that it is impossible to construct universal problem solvers?*
   A: No, they do not. They refer to the very special case of problems sampled from *i.i.d.* uniform distributions on *finite* problem spaces. See the discussion of no free lunch theorems in an earlier paper [46].
7. **Q:** *Can't the Gödel machine switch to a program* switchprog *that rewrites the utility function to a "bogus" utility function that makes unfounded promises of big rewards in the near future?*
   A: No, it cannot. It should be obvious that rewrites of the utility function can happen only if the Gödel machine first can prove that the rewrite is useful according to the *present* utility function.
8. **Q:** *Aren't there problems with undecidability? For example, doesn't Rice's theorem [30] or Blum's speed-up theorem [3, 4] pose problems?*
   A: Not at all. Of course, the Gödel machine cannot profit from a hypothetical useful self-improvement whose utility is undecidable, and will therefore simply ignore it. Compare Section 2.5 on fundamental limitations of Gödel machines (and humans, for that matter). Nevertheless, unlike previous methods, a Gödel machine can in principle exploit at least the provably good improvements and speed-ups of *any* part of its initial software.

7 Conclusion

In 1931, Kurt Gödel used elementary arithmetics to build a universal programming language for encoding arbitrary proofs, given an arbitrary enumerable set of axioms. He went on to construct *self-referential* formal statements that claim their own unprovability, using Cantor's diagonalization trick [5] to demonstrate that formal systems such as traditional mathematics are either flawed in a certain sense or contain unprovable but true statements [10]. Since Gödel's exhibition of the fundamental limits of proof and computation, and Konrad Zuse's subsequent construction of the first working programmable computer (1935-1941), there has been a lot of work on specialized algorithms solving problems taken from more or less general problem classes. Apparently,

however, one remarkable fact has so far escaped the attention of computer scientists: it is possible to use self-referential proof systems to build optimally efficient yet conceptually very simple universal problem solvers.

The initial software $p(1)$ of our machine runs an initial problem solver, e.g., one of Hutter's approaches [15, 16] which have at least an optimal *order* of complexity, or some less general method. Simultaneously, it runs an $O()$-optimal initial proof searcher using an online variant of Universal Search to test *proof techniques*, which are programs able to compute proofs concerning the system's own future performance, based on an axiomatic system $\mathcal{A}$ encoded in $p(1)$, describing a formal *utility* function u, the hardware and $p(1)$ itself. If there is no provably good, globally optimal way of rewriting $p(1)$ at all, then humans will not find one either. But if there is one, then $p(1)$ itself can find and exploit it. This approach yields the first class of theoretically sound, fully self-referential, optimally efficient, general problem solvers.

If we equate the notion of "consciousness" with the ability to execute unlimited formal self-inspection and provably useful self-change (unlimited except for the limits of computability and provability), then the Gödel machine and its Global Optimality Theorem 1 do provide the first technical justification of consciousness in the context of general problem solving [29].

References

1. W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1998.
2. R. Bellman. *Adaptive Control Processes*. Princeton University Press, 1961.
3. M. Blum. A machine-independent theory of the complexity of recursive functions. *Journal of the ACM*, 14(2):322–336, 1967.
4. M. Blum. On effective procedures for speeding up algorithms. *Journal of the ACM*, 18(2):290–305, 1971.
5. G. Cantor. Über eine Eigenschaft des Inbegriffes aller reellen algebraischen Zahlen. *Crelle's Journal für Mathematik*, 77:258–263, 1874.
6. W.F. Clocksin and C.S. Mellish. *Programming in Prolog (3rd ed.)*. Springer-Verlag, 1987.
7. N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In J.J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and Their Applications, Carnegie-Mellon University, July 24-26, 1985*, Hillsdale NJ, 1985. Lawrence Erlbaum Associates.
8. F. Crick and C. Koch. Consciousness and neuroscience. *Cerebral Cortex*, 8:97–107, 1998.
9. M. C. Fitting. *First-Order Logic and Automated Theorem Proving*. Graduate Texts in Computer Science. Springer-Verlag, Berlin, 2nd edition, 1996.
10. K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.
11. W. Heisenberg. Über den anschaulichen Inhalt der quantentheoretischen Kinematik und Mechanik. *Zeitschrift für Physik*, 33:879–893, 1925.
12. S. Hochreiter, A. S. Younger, and P. R. Conwell. Learning to learn using gradient descent. In *Lecture Notes on Comp. Sci. 2130, Proc. Intl. Conf. on Artificial Neural Networks (ICANN-2001)*, pages 87–94. Springer: Berlin, Heidelberg, 2001.
13. D. R. Hofstadter. *Gödel, Escher, Bach: an Eternal Golden Braid*. Basic Books, 1979.

14. J. H. Holland. Properties of the bucket brigade. In *Proceedings of an International Conference on Genetic Algorithms*. Lawrence Erlbaum, Hillsdale, NJ, 1985.
15. M. Hutter. Towards a universal theory of artificial intelligence based on algorithmic probability and sequential decisions. *Proceedings of the 12^{th} European Conference on Machine Learning (ECML-2001)*, pages 226–238, 2001. (On J. Schmidhuber's SNF grant 20-61847).
16. M. Hutter. The fastest and shortest algorithm for all well-defined problems. *International Journal of Foundations of Computer Science*, 13(3):431–443, 2002. (On J. Schmidhuber's SNF grant 20-61847).
17. M. Hutter. Self-optimizing and Pareto-optimal policies in general environments based on Bayes-mixtures. In J. Kivinen and R. H. Sloan, editors, *Proceedings of the 15th Annual Conference on Computational Learning Theory (COLT 2002)*, Lecture Notes in Artificial Intelligence, pages 364–379, Sydney, Australia, 2002. Springer. (On J. Schmidhuber's SNF grant 20-61847).
18. M. Hutter. *Universal Artificial Intelligence: Sequential Decisions based on Algorithmic Probability*. Springer, Berlin, 2004. (On J. Schmidhuber's SNF grant 20-61847).
19. L.P. Kaelbling, M.L. Littman, and A.W. Moore. Reinforcement learning: a survey. *Journal of AI research*, 4:237–285, 1996.
20. A. N. Kolmogorov. *Grundbegriffe der Wahrscheinlichkeitsrechnung*. Springer, Berlin, 1933.
21. A.N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems of Information Transmission*, 1:1–11, 1965.
22. D. Lenat. Theory formation by heuristic search. *Machine Learning*, 21, 1983.
23. L. A. Levin. Universal sequential search problems. *Problems of Information Transmission*, 9(3):265–266, 1973.
24. L. A. Levin. Randomness conservation inequalities: Information and independence in mathematical theories. *Information and Control*, 61:15–37, 1984.
25. M. Li and P. M. B. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications (2nd edition)*. Springer, 1997.
26. L. Löwenheim. Über Möglichkeiten im Relativkalkül. *Mathematische Annalen*, 76:447–470, 1915.
27. C. H. Moore and G. C. Leach. FORTH - a language for interactive computing, 1970.
28. R. Penrose. *Shadows of the mind*. Oxford University Press, 1994.
29. K. R. Popper. *All Life Is Problem Solving*. Routledge, London, 1999.
30. H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.
31. A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, 3:210–229, 1959.
32. J. Schmidhuber. Evolutionary principles in self-referential learning. Diploma thesis, Institut für Informatik, Technische Universität München, 1987.
33. J. Schmidhuber. Reinforcement learning in Markovian and non-Markovian environments. In D. S. Lippman, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3 (NIPS 3)*, pages 500–506. Morgan Kaufmann, 1991.
34. J. Schmidhuber. A self-referential weight matrix. In *Proceedings of the International Conference on Artificial Neural Networks, Amsterdam*, pages 446–451. Springer, 1993.
35. J. Schmidhuber. On learning how to learn learning strategies. Technical Report FKI-198-94, Fakultät für Informatik, Technische Universität München, 1994. See [49, 47].
36. J. Schmidhuber. Discovering solutions with low Kolmogorov complexity and high generalization capability. In A. Prieditis and S. Russell, editors, *Machine Learning: Proceedings of the Twelfth International Conference*, pages 488–496. Morgan Kaufmann Publishers, San Francisco, CA, 1995.

37. J. Schmidhuber. A computer scientist's view of life, the universe, and everything. In C. Freksa, M. Jantzen, and R. Valk, editors, *Foundations of Computer Science: Potential - Theory - Cognition*, volume 1337, pages 201–208. Lecture Notes in Computer Science, Springer, Berlin, 1997.
38. J. Schmidhuber. Discovering neural nets with low Kolmogorov complexity and high generalization capability. *Neural Networks*, 10(5):857–873, 1997.
39. J. Schmidhuber. Algorithmic theories of everything. Technical Report IDSIA-20-00, quant-ph/0011122, IDSIA, Manno (Lugano), Switzerland, 2000. Sections 1-5: see [40]; Section 6: see [41].
40. J. Schmidhuber. Hierarchies of generalized Kolmogorov complexities and nonenumerable universal measures computable in the limit. *International Journal of Foundations of Computer Science*, 13(4):587–612, 2002.
41. J. Schmidhuber. The Speed Prior: a new simplicity measure yielding near-optimal computable predictions. In J. Kivinen and R. H. Sloan, editors, *Proceedings of the 15th Annual Conference on Computational Learning Theory (COLT 2002)*, Lecture Notes in Artificial Intelligence, pages 216–228. Springer, Sydney, Australia, 2002.
42. J. Schmidhuber. Bias-optimal incremental problem solving. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems 15 (NIPS 15)*, pages 1571–1578, Cambridge, MA, 2003. MIT Press.
43. J. Schmidhuber. Gödel machines: self-referential universal problem solvers making provably optimal self-improvements. Technical Report IDSIA-19-03, arXiv:cs.LO/0309048, IDSIA, Manno-Lugano, Switzerland, 2003.
44. J. Schmidhuber. Gödel machine home page, with frequently asked questions, 2004. http://www.idsia.ch/˜juergen/goedelmachine.html.
45. J. Schmidhuber. Gödel machines: Fully self-referential optimal universal self-improvers. In B. Goertzel and C. Pennachin, editors, *Real AI: New Approaches to Artificial General Intelligence*. Springer, in press, 2004.
46. J. Schmidhuber. Optimal ordered problem solver. *Machine Learning*, 54:211–254, 2004.
47. J. Schmidhuber, J. Zhao, and N. Schraudolph. Reinforcement learning with self-modifying policies. In S. Thrun and L. Pratt, editors, *Learning to learn*, pages 293–309. Kluwer, 1997.
48. J. Schmidhuber, J. Zhao, and M. Wiering. Simple principles of metalearning. Technical Report IDSIA-69-96, IDSIA, 1996. See [49, 47].
49. J. Schmidhuber, J. Zhao, and M. Wiering. Shifting inductive bias with success-story algorithm, adaptive Levin search, and incremental self-improvement. *Machine Learning*, 28:105–130, 1997.
50. T. Skolem. Logisch-kombinatorische Untersuchungen über Erfüllbarkeit oder Beweisbarkeit mathematischer Sätze nebst einem Theorem über dichte Mengen. *Skrifter utgit av Videnskapsselskapet in Kristiania, I, Mat.-Nat. Kl.*, N4:1–36, 1919.
51. R.J. Solomonoff. A formal theory of inductive inference. Part I. *Information and Control*, 7:1–22, 1964.
52. R.J. Solomonoff. Complexity-based induction systems. *IEEE Transactions on Information Theory*, IT-24(5):422–432, 1978.
53. R.J. Solomonoff. Progress in incremental machine learning—Preliminary Report for NIPS 2002 Workshop on Universal Learners and Optimal Search; revised Sept 2003. Technical Report IDSIA-16-03, IDSIA, Lugano, 2003.
54. R. Sutton and A. Barto. *Reinforcement learning: An introduction*. Cambridge, MA, MIT Press, 1998.
55. A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2*, 41:230–267, 1936.
56. D. H. Wolpert and W. G. Macready. No free lunch theorems for search. *IEEE Transactions on Evolutionary Computation*, 1, 1997.

57. K. Zuse. *Rechnender Raum*. Friedrich Vieweg & Sohn, Braunschweig, 1969. English translation: *Calculating Space*, MIT Technical Translation AZT-70-164-GEMIT, Massachusetts Institute of Technology (Proj. MAC), Cambridge, Mass. 02139, Feb. 1970.