*TR IDSIA-19-03, Version 3*
*11 December 2003, arXiv:cs.LO/0309048 v3*
*(v1: 25 September 2003; v2: 28 October 2003)*

# Gödel Machines: Self-Referential Universal Problem Solvers Making Provably Optimal Self-Improvements

Jürgen Schmidhuber
*IDSIA, Galleria 2, 6928 Manno-Lugano, Switzerland*
`juergen@idsia.ch - http://www.idsia.ch/~juergen`

**Abstract**

An old dream of computer scientists is to build an optimally efficient universal problem solver. We show how to solve arbitrary computational problems in an optimal fashion inspired by Kurt Gödel's celebrated self-referential formulas (1931). Our Gödel machine's initial software includes an axiomatic description of: the Gödel machine's hardware, the problem-specific utility function (such as the expected future reward of a robot), known aspects of the environment, costs of actions and computations, and the initial software itself (this is possible without introducing circularity). It also includes a typically sub-optimal initial problem-solving policy and an *asymptotically* optimal proof searcher searching the space of computable proof techniques—that is, programs whose outputs are proofs. Unlike previous approaches, the self-referential Gödel machine will rewrite *any* part of its software, including axioms and proof searcher, as soon as it has found a proof that this will improve its future performance, given its typically limited computational resources. We show that self-rewrites are globally optimal—no local minima!—since provably none of all the alternative rewrites and proofs (those that could be found by continuing the proof search) are worth waiting for.

# Contents

# 1 Introduction and Overview

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.

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 feedback signals, such as in autonomous robot control tasks, where the goal may be to maximize expected cumulative future reward [18]. This may require the solution of essentially arbitrary problems (examples in Section 3.2 formulate traditional problems as special cases).

## 1.1 Previous Work: Best General Methods Need Proof Searchers!

Neither Levin's *universal search* [22] nor its incremental extension, the *Optimal Ordered Problem Solver* [38, 40], nor Solomonoff's recent ideas [50] are 'universal enough' for such general setups, and our earlier self-modifying online learning systems [29, 32, 46, 45, 47] are not necessarily optimal. Hutter's recent AIXI model [15] does execute optimal actions in very general environments evolving according to arbitrary, unknown, yet computable probabilistic laws, but only under the unrealistic assumption of *un*limited computation time. AIXI's asymptotically optimal, space/time-bounded cousin AIXI*(t,l)* [15] may be the system conceptually closest to the one pesented here. 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 [52], AIXI*(t,l)* needs an initial offline setup phase (prior to interaction with the environment) to examine all proofs of length at most $l_P$, 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 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 $O(l_P 2^{l_P})$, and the online computation time per cycle is $O(t2^l)$.

AIXI*(t,l)* is related to Hutter's 'fastest' algorithm for all well-defined problems (HSEARCH [16]) which also uses a general proof searcher, and also is asymptotically optimal in a certain sense. Assume discrete input/output domains $X/Y$, a formal problem specification $f : X \rightarrow Y$ (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]. That is, HSEARCH and AIXI*(t,l)* boast an optimal *order* of complexity. This somewhat limited notion of optimality, however, can be misleading despite its wide use in theoretical computer science, as it hides the possibly huge but problem-independent constants which could make AIXI*(t,l)* and HSEARCH practically infeasible.

## 1.2 Basic Principles of Gödel Machines

Our novel Gödel machine[1] derives its name and its power from exploiting provably useful changes of *any* part of its own code in self-referential fashion. Its theoretical advantages over the previous approaches can be traced back to the fact that its notion of optimality is less restricted and that it has *no* unmodifiable software *at all.*

Its bootstrap mechanism is based on a simple idea. Its initial software or program $p$ includes an axiomatic description of (possibly stochastic) environmental properties and of the Gödel machine's goals and means. The latter not only include some initial $p$-encoded problem soving strategy but also a $p$-encoded systematic proof searcher seeking some program *switchprog* for modifying the current $p$ together with a formal proof that the execution of *switchprog* will improve $p$, according to some $p$-encoded utility function or optimality criterion represented as part of the $p$-encoded goals. *Any* utility function (such as some robot's expected or worst-case future reward in its remaining lifetime) can be plugged in as an axiom stored in the initial $p$. In particular, utility may take into account expected computational costs of proof searching and other actions, to be derived from the $p$-encoded axioms. During runtime, $p$ systematically makes pairs *(switchprog, proof)* until it finds a *proof* of: *'the immediate rewrite of* p *through current program* switchprog *implies higher utility than leaving* p *as is'.* Then it executes *switchprog*, which may completely overwrite $p$. The **Global Optimality Theorem** (Theorem 2.1) shows that 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-improvements, relative to the given resource limitations and the initial proof search strategy.

Unlike HSEARCH, the Gödel machine does not waste time on finding programs that provably compute $f(z)$ for *all* $z \in X$ when the *current* $x \in X$ is the only object of interest. It is neither limited to AIXI*(t,l)*'s assumption of an enumerable environmental distribution nor to its particular *expected reward* utility function, since we may plug in *worst-case* or other types of utility functions as well.

[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'* [27]!

## 1.3 Fast Initial Proof Searcher

There are many ways of initializing the proof searcher. But since proof verification is a fast business (e.g., [9]), we may construct an *asymptotically optimal* initialization called *Bias-Optimal Proof Search* (BIOPS)—see Section 2.3. BIOPS uses variants of *Universal Search* [22] and the *Optimal Ordered Problem Solver* OOPS [38, 40] to efficiently search the space of *online proof techniques:* proof-generating programs that may read the Gödel machine's current state.[2] If some unknown 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 by the Gödel machine), then BIOPS will need at most $O(f(k))$ steps (Theorem 2.2).

That is, while the hardwired brute force theorem provers of AIXI*(t,l)* and HSEARCH systematically search in *raw* proof space—they can hide the proof search cost (exponential in proof size) in their asymptotic optimality notation [16, 15]—the BIOPS of the *initial*, not yet self-improved Gödel machine already produces many proofs much faster. Unlike AIXI*(t,l)*, a BIOPS-initialized Gödel machine can also profit from *online* proof search which may exploit information obtained through interaction with the environment.

The basic ideas of BIOPS, however, should be conceptually separated from those of the Gödel machine, whose proof searcher could be initialized in other ways as well. Unlike AIXI*(t,l)* and HSEARCH, the Gödel machine can improve the proof searcher itself, even if the latter is BIOPS, since the *asymptotic* optimality of BIOPS does not necessarily imply optimality with respect to the Gödel machine's utility function.

## 1.4 Outline

Section 2 will formally describe details of a particular Gödel machine, focusing on its novel aspects, skipping over well-known standard issues treated by any proof theory textbook. Section 2.2.1 will introduce the essential instructions invoked by proof techniques to compute axioms and theorems and to achieve goals, thus relating syntax to semantics. Section 2.2.2 will elaborate on the above-mentioned Global Optimality Theorem 2.1. Section 2.3 will describe the conceptually separate BIOPS initialization and its asymptotic optimality properties. Section 3 will discuss the Gödel machine's limitations, possible types of self-improvements, and additional differences from previous work. Section 4 will summarize.

# 2 Formal Details of a Particular Gödel Machine

Unless stated otherwise or obvious, throughout the paper newly introduced variables and functions are assumed to cover the range implicit in the context. $B$ denotes the binary

[2]The concept of online proof techniques raises several unconventional issues concerning the connection between syntax and semantics. Proofs are just symbol strings produced from other symbol strings according to certain syntactic rules. Such a symbol string, however, may be interpreted in online fashion as a statement about the computational costs of the program that computes it (a semantic issue), and may suggest a Gödel machine-modifying algorithm whose execution would be semantically useful *right now* as the proof is being created. BIOPS must deal with the fact that the utility of certain self-modifications may depend on the remaining lifetime, and with the problem of producing the right proof at the right time.

alphabet $\{0, 1\}$, $B^*$ the set of possible bitstrings over $B$, $l(q)$ denotes the number of bits in a bitstring $q$; $q_n$ the $n$-th bit of $q$; $\lambda$ 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 $q_0 := q_{0:0} := \lambda$). Occasionally it may be convenient to consult Figure 1 at the end of this paper.

## 2.1 Overview of Hardware and Initial Software / Proof Searcher

The Gödel machine's life consists of discrete cycles or time steps $t = 1, 2, \dots$. 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 $Q(t)$; its initial value by $Q(1)$.

During each cycle the Gödel machine hardware (which may be emulated as unchangeable software on a traditional computer) executes an elementary operation which affects its variable state $s \in \mathcal{S} \subset \mathcal{B}^*$ (without loss of generality we assume $s$ is encoded as a bitstring) and the variable environmental state $Env \in \mathcal{E}$, where the possibly uncountable set $\mathcal{E}$ may be unknown or only partially known.

Instead of insisting on a linear relationship between cycles and physical time, we use a hardware-oriented, causal model of time embodied by a hardwired state transition function $F : \mathcal{S} \times \mathcal{E} \rightarrow \mathcal{S}$. 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 $s(t-1)$ and is simultaneously updated or (probabilistically) computed by the possibly reactive environment, whose computational resources may vastly exceed the ones of the Gödel machine.

For example, if the hardware is a Turing machine[3] (TM) [52], then $s(t)$ is a bitstring that encodes the current contents of all tapes of the TM, the positions 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.

In order to conveniently talk about programs and data, we will often attach names to certain string variables encoded as components or substrings of $s$. Of particular interest are 7 variables called *time*, *x*, *y*, *p*, *proof*, *switchbit*, *switchprog*, to be explained below. All these variables except for the hardware-written *time* and $x$ are in principle writable by programs stored within $s$, although at least the initial software or policy $p(1)$ will be set up such that $p$ initially can modify only certain limited parts of $s$, as long as it cannot prove the usefulness of modifying other parts as well.

[3]Turing reformulated Gödel's unprovability results in terms of Turing machines (TMs) [52] 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.

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 environmental inputs. 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. $y(t)$ is an 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 policy for interacting with the environment and for proof searching. Certain parts of $s$ may be used by the policy for computing temporary results. Using some sort of process interleaving technique, the initial $p(1)$ is written such that it will allocate a fixed fraction of the time to a subpolicy $e$, which somehow computes outputs $y$ in response to inputs $x$. We may initialize $e$ by any known problem solving algorithm, to exploit from the start whatever initial prior knowledge is available.

   In the remaining time $p$ will execute a proof-searching subroutine that systematically tests proof-generating programs called *proof techniques* written in a universal programming language $\mathcal{L}$ implemented as part of the initial software $p(1)$ running on the Gödel machine's hardware. A special role is played by three global variables (initially) writable *only* by proof techniques:

   (a) *switchprog* holds a program whose execution may directly modify any part of the Gödel machine except for variables that can be set directly only by the hardware or by the environment, such as *time* and $x$ (or, in case a TM is used as hardware, the representation of the current internal state of the TM's finite state automaton).

   (b) *switchbit* is a binary variable initialized by '0'. Setting it to '1' starts the unstoppable process of transferring control to the current *switchprog*.

   (c) *proof* holds a proof incrementally computed by the currently tested proof technique. The proof searcher's goal is to find a proof stating that *switchbit* should be set, which implies that the current *switchprog* will be executed. So the decision of whether the Gödel machine should be changed in a particular way (out of many possible ways) will be reduced to the binary decision of whether *switchbit* should adopt a particular value. The next subsections will describe details; the final Section 4 will summarize everything.

## 2.2 How Online Proof Techniques Connect Syntax and Semantics

Theorem proving requires an axiom scheme yielding an enumerable set of axioms of a formal logic system whose formulas and theorems are symbol strings over some finite alphabet that may include traditional symbols of logic, such as $\rightarrow, \wedge, =, (, ), \forall, \exists, \ldots, c_1, c_2, \ldots, f_1, f_2, \ldots$.

A proof is a sequence of theorems. Each theorem is either an axiom or inferred from previous theorems by applying one of several given inference rules such as *modus ponens* combined with *unification* [9]. It is obvious and well-known that theorems can be uniquely encoded as bitstrings, and that one can write a program that systematically enumerates all possible proofs—compare Gödel's original paper [10] and any textbook on logic or proof theory, e.g., [9]. In what follows, we do not have to specify all details of a particular axiomatic encoding—it suffices to recall that concepts such as *elementary hardware operations*, *computational costs*, *utility functions*, *probability*, *provability*, etc. can be formalized.

First consider a brute force proof searcher systematically generating all proofs in order of their sizes. To produce a certain proof, this approach takes time exponential in proof size.

Instead our $p(1)$ will produce many proofs with low algorithmic complexity [48, 20, 25] much more quickly. It runs and evaluates *proof techniques* composed from instructions of the $p(1)$-encoded language $\mathcal{L}$. For example, $\mathcal{L}$ may be a variant of PROLOG [7] or the universal FORTH[26]-inspired programming language used in recent experiments with OOPS [38, 40]. $\mathcal{L}$ includes instructions that allow any part of $s$ to be read, such as inputs $x$, or the code of $p(1)$. Other instructions in $\mathcal{L}$ may write on $s^p$, a part of $s$ reserved for temporary results of proof techniques. The output of any tested proof technique written in $\mathcal{L}$ is 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). Certain *long* proofs can be produced by *short* proof techniques.

What is the best way of systematically testing proof techniques? Section 2.3 will later present a general, *bias-optimal* [38, 40] proof searcher initialization $p(1)$ of $p$. This initialization, however, is not essential for understanding the basic principles of the Gödel machine, as there are many alternative initializations.

For now it is more important to specify the precise goals of the proof search, details of the proofs that can be derived by proof techniques, and ways of verifying proofs and translating their results into online changes of the Gödel machine software. This is most conveniently done by describing the essential instructions for generating/checking axioms/theorems and for transferring control to provably good Gödel machine-changing programs.

### 2.2.1 Instructions/Subroutines for Making & Verifying Axioms & Theorems and for Initiating Online Self-Improvements

Here we describe axioms and goals and effects of the proof search. We use an 'operational' or procedure-oriented point of view, specifying those instructions that proof techniques may invoke to compute axioms and theorems.

Apart from standard arithmetic and function-defining instructions [38, 40] that modify $s^p$, the $p(1)$-encoded programming language $\mathcal{L}$ includes special instructions for prolonging the current *proof* by correct theorems, for deleting previously proven theorems from *proof* to free storage, for setting *switchprog*, and for checking whether a provably optimal Gödel machine-modifying program was found and should be executed now. Below we list all six special instructions for modifying *proof, switchbit, switchprog* (there are no others). Two of them form a non-traditional interface between syntax and semantics (Items 4 and 5; marked by '♠'), relating the search in the space of abstract symbol strings representing proofs to the actual, continually changing Gödel machine state and goal. The nature of the six *proof*-modifying instructions below makes it impossible to insert an incorrect theorem into *proof*, thus trivializing proof verification:

1. **get-axiom(n)** takes as argument an integer $k$ (computed by a prefix of the currently tested proof technique with the help of arithmetic instructions such as those used in previous work [38, 40]). Then it generates the $k$-th axiom (if it exists, according to the axiom scheme outlined below) and appends the axiom as a theorem to the current theorem sequence in *proof*. The initial axiom scheme encodes:

   (a) **Hardware axioms:** A compact axiomatic description of the finite computer hardware (or the unchangeable software in case the Gödel machine hardware is emulated by software), formally specifying how certain components of $s$ (other than the environmental inputs $x$) may change from one cycle to the next. For example, the following axiom could describe how some 64-bit hardware's instruction pointer stored in $s_{1:64}$ is continually increased by 64 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:

   $$(\forall t \forall n : [(n < 2^{58}) \wedge (n > 0) \wedge (t > 1) \wedge (t < T) \wedge (string2num(s_{1:64}(t)) = n)$$

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

   Here the semantics of used symbols such as '(' (open parenthesis) and '>' (greater than) and '$\rightarrow$' (implies) are the traditional and obvious ones, while '*string2num*' symbolizes a function translating bitstrings into numbers.

   (b) **Reward axioms:** Axioms defining the computational costs of any hardware instruction, and physical costs of output actions (if any, e.g., when the $y(t)$ are interpreted as robot control signals). Related axioms assign values to certain input events $x$ which may encode rewards for desired behavior, 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. Example axiom (unexplained symbols carry the obvious meaning):

   $$(\forall t_1 \forall t_2 : [(t_1 < t_2) \wedge (t_1 \geq 1) \wedge (t_2 \leq T)] \rightarrow [R(t_1, t_2) = r(t_1) + R(t_1 + 1, t_2)]),$$

   where $r(t)$ is interpreted as the real-valued reward at time $t$, and $R(t_1, t_2)$ as the cumulative reward between times $t_1$ and $t_2$.

   (c) **Environment axioms:** Axioms embodying limited knowledge about the environment, in particular, axioms restricting the way the environment will produce new inputs $x$ in reaction to sequences of outputs $y$. For example, it may be known in advance that the environment is sampled from an unknown probability distribution that is computable, given the previous history [48, 49, 15], or at least limit-computable [36, 37]. Or, more restrictively, the environment may be some unknown but deterministic computer program [53, 34] sampled from the Speed Prior [39] which assigns low probability to environments that are hard to compute by any method. Or the interface to the environment is Markovian [30], that is, the current input always uniquely identifies the environmental state—a lot of work has been done on this special case [28, 2, 51]. Even more restrictively, the environment may evolve in completely predictable fashion known in advance. All such prior assumptions are perfectly formalizable.

(d) **Uncertainty axioms:** Standard axioms for arithmetics and calculus and probability theory [19] and statistics and string manipulation that (in conjunction with the environment axioms) allow for constructing proofs concerning (possibly uncertain) properties of $s$ as well as bounds on expected remaining lifetime / costs / rewards, given some time $\tau$ and certain hypothetical values for components of $s(\tau)$ etc. An example theorem saying something about expected properties of future inputs might look like this:

$$(\forall t_1 : [(1 \leq t_1) \wedge (t_1 + 15597 < T) \wedge (s_{5:9}(t_1) = \text{`01011'}) \wedge (x_{40:44}(t_1) = \text{`00000'})]$$
$$\rightarrow (\exists t : [(t_1 < t < t_1 + 15597) \wedge (P_\mu(x_{17:22}(t) = \text{`011011'} \mid s(t_1)) > \frac{998}{1000})])),$$

where $P_\mu(. \mid .)$ represents a conditional probability with respect to an axiomatized prior distribution $\mu$ encoded in the environment axioms (Item 1c).

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

Since many theorems about the Gödel machine's behavior will typically be provable even in the absence of precise initialization information, however, we do not necessarily insist that the initial state is fully axiomatized, just as we do not insist that the rules governing the environment's behavior are precisely known in advance.

Item 4 will describe an instruction that permits the online creation of theorems closely related to the initialization axioms above, through subroutines that can read parts of the current Gödel machine state and subsequently transform the observations into theorems.

(f) **Utility axioms:** An axiomatic description of the Gödel machine's overall goal in the form of a utility function. A typical *'value to go'* utility function (to be maximized) is of the form $u(s, Env) : \mathcal{S} \times \mathcal{E} \rightarrow \mathcal{R}$, where $\mathcal{R}$ the set of real numbers:

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

where $E_\mu(. \mid .)$ denotes the conditional expectation operator with respect to some axiomatized prior distribution $\mu$ encoded in the environment axioms (Item 1c). Note that *time* is a function of $s$ and uniquely identifies the current cycle, and that we take into account the possibility of extending the expected lifespan $E_\mu(T \mid s, Env)$ through appropriate actions leading to 'good' values of $s$.

Alternative utility functions would favor improvement of *worst case* instead of *expected* future performance, or higher reward intake *per time interval* etc. See also examples in Section 3.2.

2. **apply-rule(k, m, n)** takes as arguments the index $k$ (if it exists) of an inference rule (e.g., [9]) 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. **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.

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

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

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). That is, we are willing to 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.*[4] *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 $(s_{6:9}(7775555) =$ '1001'), 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.

5. ♠ **check()** translates syntax into semantics. It 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 bit-encoded symbol strings of the (decoded) form

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

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 the true value of $switchbit(t_1)$, which is part of $s(t_1)$, by $b \in \{0, 1\}$. This should 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 deterministic 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

[4]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 deducable 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.

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 *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 $p$.

*With the typical example u of formula (1) the utility of switching to* switchprog *at a certain time depends on the remaining expected lifetime. So the set of possible target theorems (2) and the expected utility of self-changes may vary over time as more and more of the lifetime is consumed. Given the possibly limited axiomatized knowledge about how the environment will evolve, proof techniques may reason about the code of* check() *described above. They can prove a goal theorem of the form (2) from the* current, unmodified *axioms only if the potential upcoming transfer of control to the* current switchprog *provably yields higher expected cumulative reward within the resulting expected remaining lifetime than ignoring* switchprog *and continuing the proof search (thus eventually creating and evaluating many alternative* switchprogs*). Of course, this fully takes into account the time needed to complete the switch to* switchprog *and to execute* switchprog*, which will consume part of the remaining life. One way a proof technique could start to infer target theorem (2) would be to first prove a prediction about parts of s at some time $t_1$ in the near future (that is, later than the current* time*), such as*

$$((switchprog(\mathbf{t_1}) = \text{‘011010’}) \wedge (p(\mathbf{t_1}) = p(1)) \wedge \ldots), \tag{3}$$

*without predicting the value of* switchbit($t_1$) *yet, and a related theorem about effects of alternative values of* switchbit($t_1$)*, such as*

$$([(switchprog(\mathbf{t_1}) = \text{‘011010’}) \wedge (p(\mathbf{t_1}) = p(1)) \wedge \ldots] \rightarrow (\textbf{target theorem (2)})) \tag{4}$$

*Now (2) can be derived from (3) and (4) in the obvious way.*

Clearly, the axiomatic system used by the machine must be strong enough to permit proofs of target theorems. In particular, the theory of uncertainty axioms (Item 1d) must be sufficiently rich.

Note, however, that 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's the precise utility of the switch.

6. **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*.

### 2.2.2 Global Optimality Theorem: Self-Improvement Strategy is not Greedy

Intuitively, at any given time the Gödel machine should execute some self-modification algorithm only if it is the 'best' of all possible self-modifications, given the optimality criterion, 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 2.1 (Global Optimality of Self-Changes)** *Given any formalizable utility function u (Item 1f), and assuming consistency of the underlying formal system, any Gödel machine self-change 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.** The 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*, or (2) keep searching for *proof*s and *switchprog*s 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 axiomatic consistency. □

That is, only if it is provable that the current proof searcher cannot be expected to find sufficiently quickly an even better *switchprog*, will it also be provable that the *current switchprog* should be executed. So the trick is to let the formalized improvement criterion take into account the (expected) performance of the original search method, which automatically enforces globally optimal self-changes, relative to the provability restrictions.

Will the Gödel machine ever prove a target theorem? This obviously depends on the nature of environment and utility function. See Section 3.1 for intuitive examples of very simple, limited self-rewrites (affecting only near-future rewards) whose benefits should be easily provable, given appropriate axioms, and compare Section 3.4 on fundamental limitations of the Gödel machine.

## 2.3 Bias-Optimal Proof Search (Biops)

Given time, the Gödel machine's proof searcher should systematically generate and test all potentially relevant proof techniques, to compute alternative *proof*s and *switchprog*s. There are many ways of initializing the proof searcher. Therefore the following details of a particularly fast example initialization are not really essential for understanding the basic principles of the Gödel machine.

Let us construct an initial $p(1)$ whose proof searcher is optimal in a certain limited sense to be described below, but *not necessarily optimal* in the potentially different and more powerful Gödel machine sense embodied by the given utility function $u$. We introduce *Bias-Optimal Proof Search* (Biops) which essentially is an application of the recent *Optimal Ordered Problem Solver* Oops [38, 40] to a sequence of proof search tasks.

As long as $p = p(1)$ (which is true at least as long as no target theorem has been found), $p$ searches a space of *self-delimiting* [23, 6, 25, 38] programs written in $\mathcal{L}$. The reader is asked to consult previous work for details [38, 40]; here we just outline the basic procedure: any currently running proof technique $w$ is an instruction sequence (whose bitstring-encoded representation starts somewhere in $s^p$) which is read incrementally, instruction by instruction. Each instruction is immediately executed while being read; this may modify $s^p$, and may transfer control back to a previously read instruction (e.g., a loop). To read and execute a previously unread instruction right after the end of the proof technique prefix read so far, $p$ first waits until the execution of the prefix so far *explicitly requests* such a prolongation, by setting an appropriate signal in the internal state. Prefixes may cease to request any further instructions—that's why they are called self-delimiting. So the proof techniques form a prefix code, e.g., [25]: no proof technique that at some point ceases to request a prolongation of its code can be prefix of another proof technique.

Now to our limited notion of initial optimality. What is a good way of systematically testing proof techniques? Biops starts with a probability distribution $P$ on the proof techniques $w$ written in language $\mathcal{L}$. $P$ represents the searcher's initial bias. $P(w)$ could be based on program length, e.g., $P(w) = 2^{-l(w)}$ for binary programs $w$ [23, 25], or on a probabilistic version of the syntax diagram of $\mathcal{L}$, where the diagram's edges are labeled with transition probabilities [38, 40]. Biops is *near-bias-optimal* in the sense that it will not spend more time on any proof technique than it deserves, according to the probabilistic bias, namely, not more than its probability times the total search time:

**Definition 2.1 (Bias-Optimal Searchers [38, 40])** 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 $n = 1$.*

The $k$-th proof search task of Biops ($k = 1, 2, \ldots$) will be to find a proof of a target theorem applicable to the state of the Gödel machine shortly after the point where the theorem is found. The first such proof may actually provoke a complete Gödel machine rewrite that abolishes Biops. So Subsection 2.3.1 will focus just on the very first proof search task. Subsection 2.3.3 will then address the case where the proof searcher has survived the most recent task's solution and wants to profit from earlier solved tasks when possible.

### 2.3.1 How the Initial Proof Searcher May Use Biops to Solve the First Proof Search Task

Biops first invokes a variant of Levin's universal search [22] (Levin attributes similar ideas to Adleman [24]). Universal search is a simple, asymptotically optimal [22, 24, 25, 16, 50], near-bias-optimal [38, 40] way of solving a broad class of problems whose solutions can be quickly verified. It was originally described for universal Turing machines with unlimited storage [22]. In realistic settings, however, we have to introduce a recursive procedure [38, 40] for time-optimal backtracking in program space to perform efficient storage management on *realistic, limited computers.*

Previous practical variants and extensions of universal search have been applied [33, 35, 47, 38, 40] to *offline* program search tasks where the program inputs are fixed such that the same program always produces the same results.

This is not the case in the present *online* setting: the same proof technique started at different times may yield different proofs, as it may read parts of $s$ (such as the inputs) that change as the Gödel machine's life proceeds. Nevertheless, we may apply a variant of universal search to the space of proof-generating programs as follows.

For convenience, let us first rename the storage writable by proof techniques: we place *switchprog*, *proof*, and all other proof technique-writable storage cells in a common address space called *temp* encoded somewhere in $s$, with $k$-th component $temp_k$. To efficiently undo *temp* changes, we introduce global Boolean variables $mark_k$ (initially FALSE) for all $temp_k$. In the method below we notationally suppress the task dependency seen in Def. 2.1:

**Method 2.1** In the $i$-th phase ($i = 1, 2, 3, \ldots$) DO:

Make an empty stack called *Stack*. FOR all self-delimiting 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. Whenever the execution of $w$ changes some $temp_k$ whose $mark_k =$ FALSE, set $mark_k :=$ TRUE and save the previous value $\overline{temp}_k$ by pushing the pair $(k, \overline{temp}_k)$ onto *Stack*.
2. Pop off all elements of *Stack* and use the information contained therein to undo the effects of $w$ on *temp* and to reset all $mark_k$ to FALSE. This does not cost significantly more time than executing $w$.[5]

Method 2.1 is conceptually very simple: it essentially just time-shares all program tests such that each program gets at most a constant fraction of the total search time. *Note that certain long proofs producible by short programs (with repetitive loops etc.) are tested early by this method.* This is one major difference between BIOPS and more traditional brute force proof searchers that systematically order proofs by their sizes, instead of the sizes (or probabilities) of their proof-computing proof techniques.

None of the proof techniques can produce an incorrect *proof*, due to the nature of the theorem-generating instructions from Section 2.2.1. A proof technique $w$ can interrupt

[5]Even faster search is possible for proof techniques that do not read changing parts of $s$ in phase $i$: we do not have to recompute results of their own, shorter, previously executed prefixes. The precise, recursive, backtracking-based procedure for this case is more complex than Method 2.1 and has been described elsewhere [38, 40].

Method 2.1 only by invoking the instruction *check()* which may transfer control to *switchprog* (which possibly will delete or rewrite Method 2.1).

Clearly, since the initial $p$ runs on the formalized hardware of the Gödel machine, and since proof techniques tested by $p$ can read $p$ and other parts of $s$, they can produce proofs concerning the (expected or worst-case) performance of $p$ itself.

### 2.3.2 Asymptotic Optimality Despite Online Proof Generation

*Online* proof techniques whose outputs depend on ephemeral events such as inputs should be viewed as a potentially extremely useful asset, not as an unwanted burden. Their advantages become particularly obvious when the original axioms state that parts of certain appropriately marked inputs may be taken as teacher-given theorems without further proof. Such external online help may be dangerous but can greatly speed up the proof search, since there will be short/probable (and therefore frequently tested) programs exploiting the helpful information by simply appending copies of teacher-given theorems to *proof*. Similarly, the outcome of online experiments may severely restrict the possible futures of the environment. This in turn could be used by the Gödel machine to greatly reduce the search space of relevant proof techniques and of candidates for useful behavior.

As already mentioned above, however, online proof techniques will in general produce different outputs at different times. Is this worrisome when it comes to ensuring the property of bias-optimality? Not really: since proof techniques are general programs, some of them may compensate 'just in the right way' for online Gödel machine state changes that are caused by time-varying inputs and various processes running on the Gödel machine. In particular, some proof techniques may produce appropriate target theorems no matter what's the precise time at which they are started, e.g., by appropriately computing useful values for *switchprog* and $t_1$ in formula (2) in response to the current *time*, or by waiting for a certain type of repetitive, informative input until it reappears, etc. No matter how proof techniques compute proofs, Method 2.1 has the optimal order of computational complexity in the following sense.

**Theorem 2.2 (Asymptotically optimal online proof search)** *If independently of variable* time *some unknown 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 by the Gödel machine), then Method 2.1 will need at most $O(f(k))$ steps.*

**Proof.** It is easy to see that Method 2.1 will need at most $O(f(k)/P(w)) = O(f(k))$ steps—the constant factor $1/P(w)$ does not depend on $k$. $\square$

So the method is asymptotically as fast as the fastest (unknown) such proof technique, assuming that the Gödel machine's lifetime is sufficient to absorb the constant factor. In general, however, *no* method can generate more than a small fraction of the possible proofs, not even Method 2.1, despite its limited optimality properties. For example, each single cycle in principle could already give rise to a whole variety of distinct theorems producible through instruction *state2theorem* (Item 4 in Section 2.2.1). Most of them will never be generated though, since the instruction itself will typically already consume *several* cycles.

We observe that BIOPS searches for a provably good *switchprog* in a near-bias-optimal and asymptotically optimal way, exploiting the fact that proof verification is a simple and fast business. But here the expression *'provably good'* refers to a *typically different and more powerful* sense of optimality depending on the Gödel machine's utility function! Compare Section 3.5.2 and note that the total utility of a Gödel machine may be hard to verify—the evaluation may consume its entire lifetime.

### 2.3.3 How a Surviving Proof Searcher May Use BIOPS to Solve Remaining Proof Search Tasks

In what follows we assume that the execution of the *switchprog* corresponding to the first found target theorem has not rewritten the code of $p$ itself—the current $p$ is still equal to $p(1)$—and has reset *switchbit* and returned control to $p$ such that it can continue where it was interrupted. In that case the BIOPS subroutine of $p(1)$ can use OOPS [38, 40] to accelerate the search for the $n$-th target theorem ($n > 1$) by reusing proof techniques for earlier found target theorems where possible. The basic ideas are as follows (details: [38, 40]).

Whenever a target theorem has been proven, $p(1)$ *freezes* the corresponding proof technique: its code becomes non-writable by proof techniques to be tested in later proof search tasks. But it remains readable, such that it can be copy-edited and/or invoked as a subprogram by future proof techniques. We also allow prefixes of proof techniques to temporarily rewrite the probability distribution on their suffixes [40, 38], thus essentially rewriting the probability-based search procedure (an incremental extension of Method 2.1) based on previous experience. As a side-effect we metasearch for faster search procedures, which can greatly accelerate the learning of new tasks [40, 38].

Given a new proof search task, BIOPS performs OOPS [40, 38] by spending half the total search time on a variant of Method 2.1 that searches only among self-delimiting [23, 6] proof techniques starting with the most recently frozen proof technique. The rest of the time is spent on fresh proof techniques with arbitrary prefixes (which may reuse previously frozen proof techniques though) [38, 40]. (We could also search for a *generalizing* proof technique solving all proof search tasks so far. In the first half of the search we would not have to test proof techniques on tasks other than the most recent one, since we already know that their prefixes solve the previous tasks [38, 40].)

It can be shown that OOPS is essentially *8-bias-optimal* (see Def. 2.1), given either the initial bias or intermediate biases due to frozen solutions to previous tasks [38, 40]. This result immediately carries over to BIOPS. To summarize, BIOPS essentially allocates part of the total search time for a new task to proof techniques that exploit previous successful proof techniques in computable ways. If the new task can be solved faster by copy-editing / invoking previously frozen proof techniques than by solving the new proof search task from scratch, then BIOPS will discover this and profit thereof. If not, then at least it will not be significantly slowed down by the previous solutions—BIOPS will remain 8-bias-optimal.

Recall, however, that BIOPS is not the only possible way of initializing the Gödel machine's proof searcher.

# 3 Discussion

Here we list a few examples of possible types of self-improvements, Gödel machine applicability to various tasks defined by various utility functions and environments, probabilistic hardware, relations to previous work, and fundamental limitations of Gödel machines.

## 3.1 Possible Types of Gödel Machine Self-Improvements

Which provably useful self-modifications are possible? There are few limits to what a Gödel machine might do.

In one of the simplest cases it might leave its basic proof searcher intact and just change the ratio of time-sharing between the proof searching subroutine and the subpolicy $e$—those parts of $p$ responsible for interaction with the environment.

Or the Gödel machine might modify $e$ only. For example, the initial $e$ may regularly store limited memories of past events somewhere in $s$; this might allow $p$ to derive that it would be useful to modify $e$ such that $e$ will conduct certain experiments to increase the knowledge about the environment, and use the resulting information to increase reward intake. In this sense the Gödel machine embodies a principled way of dealing with the exploration vs exploitation problem [18]. Note that the *expected* utility of conducting some experiment may exceed the one of not conducting it, even when the experimental outcome later suggests to keep acting in line with the previous $e$.

The Gödel machine might also modify its very axioms to speed things up. For example, it might find a proof that the original axioms should be replaced or augmented by theorems derivable from the original axioms.

The Gödel machine might even change its own utility function and target theorem, but can do so only if their *new* values are provably better according to the *old* ones.

In many cases we do not expect the Gödel machine to replace its proof searcher by code that completely abandons the search for proofs. Instead we expect that only certain subroutines of the proof searcher will be sped up—compare the example at the end of Item 5 in Section 2.2.1—or that perhaps just the order of generated proofs will be modified in problem-specific fashion. This could be done by modifying the probability distribution on the proof techniques of the initial bias-optimal proof searcher from Section 2.3.

Generally speaking, the utility of limited rewrites may often be easier to prove than the one of total rewrites. For example, suppose it is 8.00pm and our Gödel machine-controlled agent's permanent goal is to maximize future expected reward. Part thereof is to avoid hunger. There is nothing in its fridge, and shops close down at 8.30pm. It does not have time to optimize its way to the supermarket in every little detail, but if it does not get going right now it will stay hungry tonight (in principle such near-future consequences of actions should be easily provable, possibly even in a way related to how humans prove advantages of potential actions to themselves). That is, if the agent's previous policy did not already include, say, an automatic daily evening trip to the supermarket, the policy provably should be rewritten at least in a very limited and simple way right now, while there is still time, such that the agent will surely get some food tonight, without affecting less urgent future behavior that can be optimized / decided later, such as details of the route to the food, or of tomorrow's actions.

In certain uninteresting environments reward is maximized by becoming dumb. For example, a given task may require to repeatedly and forever execute the same pleasure center-activating action, as quickly as possible. In such cases the Gödel machine may delete most of its more time-consuming initial software including the proof searcher.

Note that there is no reason why a Gödel machine should not augment its own hardware. Suppose its lifetime is known to be 100 years. Given a hard problem and axioms restricting the possible behaviors of the environment, the Gödel machine might find a proof that its expected cumulative reward will increase if it invests 10 years into building faster computational hardware, by exploiting the physical resources of its environment.

## 3.2 Example Applications

**Example 3.1 (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 [39], according to which hard-to-compute environmental reactions are unlikely. This permits a computable strategy for making near-optimal predictions [39]. One by-product of maximizing expected reward is to maximize expected lifetime.*

Less general, more traditional examples that do not involve significant interaction with an environment are also easily dealt with in the reward-based framework:

**Example 3.2 (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 3.3 (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.*

**Example 3.4 (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 2.3 run in parallel.*

## 3.3 Probabilistic Gödel Machine Hardware

Above we have focused on an example deterministic machine. It is straight-forward to extend this to computers whose actions are computed in probabilistic fashion, given the current state. Then the expectation calculus used for probabilistic aspects of the environment simply has to be extended to the hardware itself, and the mechanism for verifying proofs has to take into account that there is no such thing as a certain theorem—at best there are formal statements which are true with such and such probability. In fact, this may be the most realistic approach as any physical hardware is error-prone, which should be taken into account by realistic probabilistic Gödel machines.

Probabilistic settings also automatically avoid certain issues of axiomatic consistency. For example, predictions proven to come true with probability less than 1.0 do not necessarily cause contradictions even when they do not match the observations.

## 3.4 Limitations of the Gödel machine

The fundamental limitations are closely related to those first identified by Gödel [10]: Any formal system that encompasses arithmetics 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 particular, one can construct examples of environments and utility functions that make it impossible for the Gödel 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.

## 3.5 More Relations to Previous Work on 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.

### 3.5.1 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) [45]. 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 [32, 46, 45, 47]. 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 [45].

The Gödel machine's training algorithm is theoretically 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 [21], our Meta-Genetic Programming [29] extending Cramer's Genetic Programming [8, 1], our metalearning economies [29] extending Holland's machine learning economies [14], and gradient-based metalearners for continuous program spaces of differentiable recurrent neural networks [31, 12]. All these methods, however, could be used to seed $p(1)$ with an initial policy.

### 3.5.2 Gödel Machine vs Oops and Oops-rl

The Optimal Ordered Problem Solver Oops [38, 40] (used by Biops in Section 2.3) is a bias-optimal (see Def. 2.1) way of searching for a program that solves each problem in an ordered sequence of problems of a reasonably general type, continually organizing and managing and reusing earlier acquired knowledge. Solomonoff recently also proposed related ideas for a *scientist's assistant* [50] that modifies the probability distribution of universal search [22] based on experience.

As pointed out earlier [38] (section on Oops limitations), however, Oops-like methods are not directly applicable to general lifelong reinforcement learning (RL) tasks [18] such as those for which Aixi [15] was designed. The simple and natural but limited optimality notion of Oops is *bias-optimality* (Def. 2.1): 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 [38]: 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. It is possible to use two Oops -modules as components of a rather general reinforcement learner (Oops-rl), one module learning a predictive model of the environment, the other one using this *world model* to search for an action sequence maximizing expected reward [38, 42]. Despite the bias-optimality properties of Oops for certain ordered task sequences, however, Oops-rl is not necessarily the best way of spending limited computation time in general RL situations.

A provably optimal RL machine must somehow *prove* properties of otherwise un-testable 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 in principle 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.

### 3.5.3 Gödel Machine vs Aixi etc.

Unlike Gödel machines, Hutter's recent Aixi *model* [15] generally needs *unlimited* computational resources per input update. It combines Solomonoff's universal prediction scheme [48, 49] 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 $\mu$. Aixi defines a mixture distribution $\xi$ as a weighted sum of distributions $\nu \in \mathcal{M}$, where $\mathcal{M}$ is any class of distributions that includes the true environment $\mu$. For example, $\mathcal{M}$ may be a sum of all computable distributions [48, 49], 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 $\xi$-predicted reward up to some given horizon. Recent work [17] demonstrated Aixi 's optimal use of observations as follows. The Bayes-optimal policy $p^\xi$ based on the mixture $\xi$ 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 $p^\mu$ which knows $\mu$ in advance. The necessary condition that $\mathcal{M}$ admits self-optimizing policies is also sufficient. Furthermore, $p^\xi$ is Pareto-optimal in the sense that there is no other policy 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 the introduction. Both methods could be used to seed the Gödel machine with an initial policy. Unlike Aixi*(t,l)* and Hsearch, however, the Gödel machine is not only *asymptotically* optimal but optimal relative to any given formal self-improvement criterion (which does not have to ignore constant slowdowns just because they are constant). It is defined this way. Unlike Aixi*(t,l)* it does not have to make the unrealistic assumption that all events of the entire life are memorizable (which is actually a minor issue considering the other costs of Aixi*(t,l)*). Moreover, the Gödel machine neither requires Aixi*(t,l)*'s assumption of an enumerable environmental distribution nor Aixi*(t,l)*'s particular utility function—we may plug in other types of utility functions as well.

While both Hsearch and Aixi*(t,l)* feature hardwired 'meta-algorithms' that carefully allocate time to various subroutines in an unmodifiable way, the Gödel machine is *self-*

*referential* and does not have any unchangeable software. It can replace the proof searcher itself by a faster, perhaps more focused or more domain-specific proof searcher in an online fashion, once the initial axioms together with experiences collected so far yield a proof there is one. It is precisely these *self-referential* aspects of the Gödel machine that relieve us of much of the burden of careful algorithm design—they make the Gödel machine both conceptually simpler *and* more general than AIXI*(t,l)* and HSEARCH.

## 3.6 Practical Issues

We believe that Gödel machines should be built. Some people may be sceptical about the practicality of systematic proof technique testers as components of general problem solvers. But until a short while ago some people were also sceptical about the feasibility of universal program search methods. Recent work [38, 40], however, already has shown that certain general methods can indeed offer certain practical advantages over traditional planning systems and reinforcement learners.

The well-developed field of automated theorem proving provides a rich set of alternatives for the initial Gödel machine set-up. Practical issues include:

1. Identify a particularly convenient hardware, presumably not based on Turing machines, but perhaps on recurrent neural network-like parallel systems, possibly emulated by software for a traditional personal computer.

2. Identify a particularly convenient formal language for describing the initial axioms and the set of possible proofs. For efficiency reasons this language should include a calculus for shortcuts and variables like the one informally used by mathematicians, who rarely use truly formal argumentation. Their papers not only abound with sentences like *'It is easy to see ...'*; they almost always also use informal sentences such as *'Let $f$ denote a function ...'* before they reuse the symbol $f$ in more formal statements. To facilitate a mimicry of elegant human proof techniques, the theorem-proving calculus should fully formalize temporary bindings of symbols traditionally introduced through text to symbols occurring in theorem sequences.

3. Identify a natural set of instructions for the programming language used by the initial proof searcher to construct proof techniques that calculate proofs—compare Section 2.2.1.

4. Identify potentially useful high-level theorems about the initial proof searcher's behavior and performance, to be added to the initial axioms, such that the proof searcher's initial task is facilitated because it does not have to prove these theorems from scratch.

## 3.7 Are Humans Probabilistic Gödel Machines?

We do not know. We think they better be. Their initial underlying formal system for dealing with uncertainty seems to differ substantially from those of traditional expectation calculus and logic though—compare Items 1c and 1d in Section 2.2.1 as well as the supermarket example in Section 3.1.

## 4 Summary / Conclusion

Given is an arbitrary formalizable problem whose solution may require interaction with a possibly reactive environment. For example, the goal may be to maximize the future expected (or worst-case) reward of a robot. While executing its initial problem solving strategy, the Gödel machine simultaneously runs a proof searcher which systematically and repeatedly tests proof techniques. Proof techniques are programs that may read any part of the Gödel machine's state, and write on a reserved part which may be reset for each new proof technique test. In our example Gödel machine this writable storage includes the variables *proof* and *switchprog*, where *switchprog* holds a potentially unrestricted program whose execution could completely rewrite any part of the Gödel machine's current software. Normally the current *switchprog* is not executed. However, proof techniques may invoke a special subroutine *check()* which tests whether *proof* currently holds a proof showing that the utility of stopping the systematic proof searcher and transferring control to the current *switchprog* at a precisely defined point in the near future exceeds the utility of continuing the search until some alternative *switchprog* is found. Such proofs are derivable from the proof searcher's axiom scheme which formally describes the utility function to be maximized (typically the expected future reward in the expected remaining lifetime of the Gödel machine), the computational costs of hardware instructions (from which all programs are composed), and the effects of hardware instructions on the Gödel machine's state. The axiom scheme also formalizes known probabilistic properties of the possibly reactive environment, and also the *initial* Gödel machine state and software, which includes the axiom scheme itself (no circular argument here). Thus proof techniques can reason about expected costs and results of all programs including the proof searcher.

Once *check()* has identified a provably good *switchprog*, the latter is executed (some care has to be taken here because the proof verification itself and the transfer of control to *switchprog* also consume part of the typically limited lifetime). The discovered *switchprog* represents a *globally* optimal self-change in the following sense: provably *none* of all the alternative *switchprog*s and *proof*s (that could be found in the future by continuing the proof search) is worth waiting for.

There are many ways of initializing the proof searcher. Although identical proof techniques may yield different proofs depending on the time of their invocation (due to the continually changing Gödel machine state), there are bias-optimal and asymptotically optimal proof searcher initializations based on variants of universal search [22] and the *Optimal Ordered Problem Solver* [38, 40] (Section 2.3).

Our Gödel machine will never get worse than its initial problem solving strategy, and has a chance of getting much better, provided the nature of the given problem allows for a provably useful rewrite of the initial strategy, or of the proof searcher. The Gödel machine may be viewed as a self-referential universal problem solver that can formally talk about itself, in particular about its performance. It may 'step outside of itself' [13] by rewriting its axioms and utility function or augmenting its hardware, provided this is provably useful. Its conceptual simplicity notwithstanding, the Gödel machine explicitly addresses the *'Grand Problem of Artificial Intelligence'* [44] by optimally dealing with limited computational resources in general environments, and with the possibly huge (but constant) slowdowns buried by previous approaches [16, 15] in the widely used but often misleading *O()*-notation of theoretical computer science.

No complex proofs were necessary in this paper as the Gödel machine is designed precisely such that we may leave all complex proofs to the Gödel machine itself. Its main limitation is that it cannot profit from self-improvements whose usefulness it cannot prove in time.

## 5 Acknowledgments

Thanks to Alexey Chernov, Marcus Hutter, Jan Poland, Sepp Hochreiter, Ray Solomonoff, Leonid Levin, Shane Legg, Alex Graves, Matteo Gagliolo, Viktor Zhumatiy, and several other researchers, for useful comments on drafts or summaries or earlier versions [41].

## 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] G.J. Chaitin. A theory of program size formally identical to information theory. *Journal of the ACM*, 22:329–340, 1975.

[7] W.F. Clocksin and C.S. Mellish. *Programming in Prolog (3rd ed.).* Springer-Verlag, 1987.

[8] 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.

[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] L.P. Kaelbling, M.L. Littman, and A.W. Moore. Reinforcement learning: a survey. *Journal of AI research*, 4:237–285, 1996.

[19] A. N. Kolmogorov. *Grundbegriffe der Wahrscheinlichkeitsrechnung*. Springer, Berlin, 1933.

[20] A.N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems of Information Transmission*, 1:1–11, 1965.

[21] D. Lenat. Theory formation by heuristic search. *Machine Learning*, 21, 1983.

[22] L. A. Levin. Universal sequential search problems. *Problems of Information Transmission*, 9(3):265–266, 1973.

[23] L. A. Levin. Laws of information (nongrowth) and aspects of the foundation of probability theory. *Problems of Information Transmission*, 10(3):206–210, 1974.

[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] C. H. Moore and G. C. Leach. FORTH - a language for interactive computing, 1970. http://www.ultratechnology.com.

[27] R. Penrose. *Shadows of the mind*. Oxford University Press, 1994.

[28] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, 3:210–229, 1959.

[29] J. Schmidhuber. Evolutionary principles in self-referential learning. Diploma thesis, Institut für Informatik, Technische Universität München, 1987.

[30] 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*, pages 500–506. Morgan Kaufmann, 1991.

[31] J. Schmidhuber. A self-referential weight matrix. In *Proceedings of the International Conference on Artificial Neural Networks, Amsterdam*, pages 446–451. Springer, 1993.

[32] J. Schmidhuber. On learning how to learn learning strategies. Technical Report FKI-198-94, Fakultät für Informatik, Technische Universität München, 1994. See [47, 45].

[33] 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.

[34] 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.

[35] J. Schmidhuber. Discovering neural nets with low Kolmogorov complexity and high generalization capability. *Neural Networks*, 10(5):857–873, 1997.

[36] J. Schmidhuber. Algorithmic theories of everything. Technical Report IDSIA-20-00, quant-ph/0011122, IDSIA, Manno (Lugano), Switzerland, 2000. Sections 1-5: see [37]; Section 6: see [39].

[37] 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.

[38] J. Schmidhuber. Optimal ordered problem solver. Technical Report IDSIA-12-02, IDSIA, Manno-Lugano, Switzerland, 2002. Available at arXiv:cs.AI/0207097 or http://www.idsia.ch/~juergen/oops.html. *Machine Learning Journal*, Kluwer, 2003, accepted.

[39] 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.

[40] J. Schmidhuber. Bias-optimal incremental problem solving. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems 15*, pages 1571–1578, Cambridge, MA, 2003. MIT Press.

[41] 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, v1: Sep, v2: Oct, v3: Dec 2003.

[42] J. Schmidhuber. The new AI: General & sound & relevant for physics. In B. Goertzel and C. Pennachin, editors, *Real AI: New Approaches to Artificial General Intelligence.* 2003. Accepted. Also available as TR IDSIA-04-03, cs.AI/0302012.

[43] J. Schmidhuber. The new AI: General & sound & relevant for physics. Technical Report TR IDSIA-04-03, Version 1.0, cs.AI/0302012 v1, February 2003.

[44] J. Schmidhuber. Towards solving the grand problem of AI. In P. Quaresma, A. Dourado, E. Costa, and J. F. Costa, editors, *Soft Computing and complex systems*, pages 77–97. Centro Internacional de Mathematica, Coimbra, Portugal, 2003. Based on [43].

[45] 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.

[46] J. Schmidhuber, J. Zhao, and M. Wiering. Simple principles of metalearning. Technical Report IDSIA-69-96, IDSIA, 1996. See [47, 45].

[47] 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.

[48] R.J. Solomonoff. A formal theory of inductive inference. Part I. *Information and Control*, 7:1–22, 1964.

[49] R.J. Solomonoff. Complexity-based induction systems. *IEEE Transactions on Information Theory*, IT-24(5):422–432, 1978.

[50] 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.

[51] R. Sutton and A. Barto. *Reinforcement learning: An introduction.* Cambridge, MA, MIT Press, 1998.

[52] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2*, 41:230–267, 1936.

[53] 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.

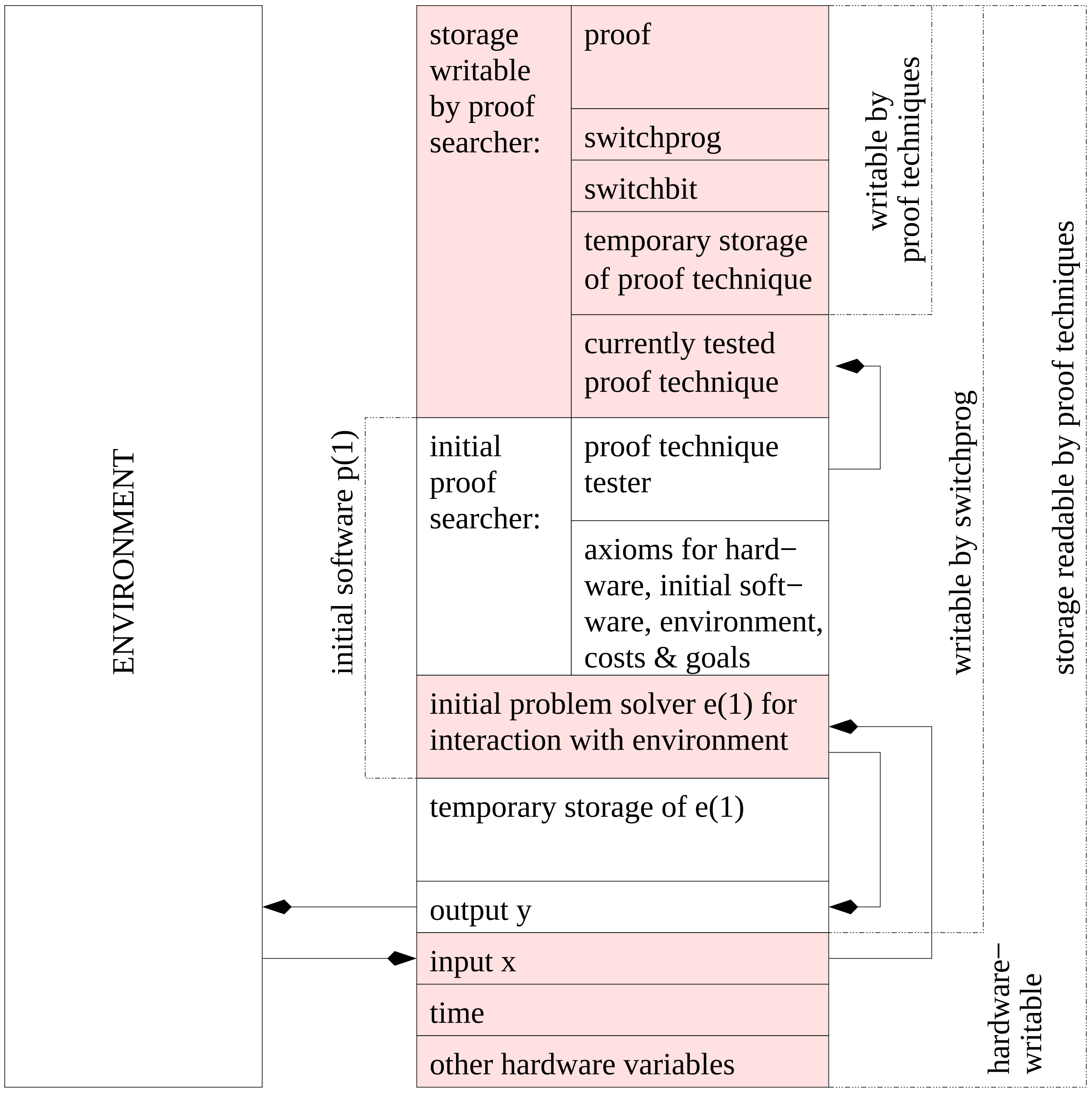
ENVIRONMENT
storage writable by proof searcher:
proof
switchprog
switchbit
temporary storage of proof technique
currently tested proof technique
writable by proof techniques
initial software p(1)
initial proof searcher:
proof technique tester
axioms for hard–ware, initial soft–ware, environment, costs & goals
writable by switchprog
storage readable by proof techniques
initial problem solver e(1) for interaction with environment
temporary storage of e(1)
output y
input x
time
other hardware variables
hardware–writable

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