

Chapter 10

Towards An Actual Gödel Machine Implementation: A Lesson in Self-Reflective Systems¹

Bas R. Steunebrink and Jürgen Schmidhuber
IDSIA & University of Lugano, Switzerland
{bas,juergen}@idsia.ch

Recently, interest has been revived in self-reflective systems in the context of Artificial General Intelligence (AGI). An AGI system should be intelligent enough to be able to reason about its own program code, and make modifications where it sees fit, improving on the initial code written by human programmers. A pertinent example is the Gödel Machine, which employs a proof searcher—in parallel to its regular problem solves duties—to find a self-rewrite of which it can prove that it will be beneficial. Obviously there are technical challenges involved in attaining such a level of self-reflection in an AGI system, but many of them are not widely known or properly appreciated. In this chapter we go back to the theoretical foundations of self-reflection and examine the (often subtle) issues encountered when embarking on actually implementing a self-reflective AGI system in general and a Gödel Machine in particular.

10.1 Introduction

An Artificial General Intelligence (AGI) system is likely to require the ability of self-reflection; that is, to inspect and reason about its own program code and to perform comprehensive modifications to it, while the system itself is running. This is because it seems unlikely that a human programmer can come up with a completely predetermined program that satisfies sufficient conditions for general intelligence, without requiring adaptation. Of course self-modifications can take on different forms, ranging from simple adaptation of a few parameters through a machine learning technique, to the system having complete read and write access to its own currently running program code [13]. In this chapter we consider the technical implications of approaching the latter extreme, by building towards a programming language plus interpreter that allows for complete inspection and manipulation of its own internals in a safe and easily understandable way.

The ability to inspect and manipulate one's own program code is not novel; in fact, it was standard practice in the old days when computer memory was very limited and expensive. In recent times, however, programmers are discouraged of using self-modifying code because it is very hard for human programmers to grasp all consequences (especially over longer timespans) of self-modifications and thus this practice is considered error-prone. Consequently, many modern

¹Published in the book *Theoretical Foundations of Artificial General Intelligence*, edited by Pei Wang and Ben Goertzel, pages 173–196, Springer, 2012. The current document has a different style but the text is identical.

(high-level) programming languages severely restrict access to internals (such as the call stack) or hide them altogether. There are two reasons, however, why we should not outlaw writing of self-modifying code forever. First, it may yet be possible to come up with a programming language that allows for writing self-modifying code in a safe and easy-to-understand way. Second, now that automated reasoning systems are becoming more mature, it is worthwhile investigating the possibility of letting the machine—instead of human programmers—do all the self-modifications based on automated reasoning about its own programming.

As an example of a system embodying the second motivation above we will consider the Gödel Machine [17, 15, 14, 16], in order to put our technical discussion of self-reflection in context. The fully self-referential Gödel Machine is a universal artificial intelligence that is theoretically optimal in a certain sense. It may interact with some initially unknown, partially observable environment to solve arbitrary user-defined computational tasks by maximizing expected cumulative future utility. Its initial algorithm is not hardwired; it can completely rewrite itself without essential limits apart from the limits of computability, provided a proof searcher embedded within the initial algorithm can first prove that the rewrite is useful, according to its formalized utility function taking into account the limited computational resources. Self-rewrites due to this approach can be shown to be *globally optimal* with respect to the initial utility function (e.g., a Reinforcement Learner’s reward function), relative to Gödel’s well-known fundamental restrictions of provability [7].

In the next section we provide an outline of the specification of the Gödel Machine concept, which then provides the context for a subsequent discussion of self-reflective systems. As alluded to above, a Gödel Machine implementation calls for a system with the ability to make arbitrary changes to its currently running program. But what does that mean? What is a change, how arbitrary can a change be, and what does a running program actually look like? We will see that these are nontrivial questions, involving several subtle but important issues. In this chapter we provide an overview of the issues involved and ways to overcome them.

10.2 The Gödel Machine Concept

One can view a Gödel Machine as a program consisting of two parts. One part, which we will call the *solver*, can be any problem-solving program. For clarity of presentation, we will pretend the *solver* is a Reinforcement Learning (RL) [10] program interacting with some external environment. This will provide us with a convenient way of determining *utility* (using the RL program’s reward function), which will be an important topic later on. But in general, no constraints are placed on the *solver*. The second part of the Gödel Machine, which we will call the *searcher*, is a program that tries to improve the entire Gödel Machine (including the *searcher*) in a provably optimal way. This *searcher* will be the main topic of this section.

A Gödel Machine’s *searcher* is said to hold a function computing the machine’s expected cumulative future utility. This utility function is then used to construct a *target theorem*; that is, the Gödel Machine is *only* allowed to perform a self-modification if it is provably useful to do so. As we will see below, the *target theorem* specifies precisely when a rewrite is considered “useful.”

Basically, the function for determining the expected cumulative future utility, shown as u below, sums all rewards for all future time steps. Here “time steps” actually means not clock ticks, but execution of elementary instructions. Indeed, each instruction takes time to execute, so if we can find a way to explicitly represent the instructions that are going to be executed in the future, we automatically have a window into a future time. An obvious choice of such a representation is the *continuation*, which is a well-studied concept in light of λ -calculus-based programming languages (e.g., Lisp, Scheme) [12]. Intuitively, a continuation can be seen as the opposite of a call stack; instead of showing “where we came from,” a continuation explicitly shows “what is going to happen next.” Note that in all but the simplest cases, a continuation will only be partially expanded. For example, suppose the current continuation is `{ A(); if B() then C() else D() }`; this continuation specifies that the next thing to be done is expanding A and executing its body, and then the conditional statement will be executed, which means that first B will be expanded and depending on its result, either C or D will be expanded. Note that before executing B, it is not

clear yet whether C or D will be executed in the future; so it makes no sense to expand either of them before we know the result of B.

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

$$u_{\bar{s}}(s, c) = \mathbb{E}_{\mu_s, M_s}[u'] \quad \text{with } u'(env) = r_{\bar{s}}(s, env) + \mathbb{E}_{\kappa_c, K_c}[u_{\bar{s}} \mid env] \quad (10.1)$$

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

It is *crucial* to note that u and r are taken from the evaluating state \bar{s} , *not* from the state s under evaluation. Doing the latter would break the global optimality [16] of the Gödel Machine, because then it would be capable of rewriting its utility function in arbitrary, even detrimental ways. Therefore, the original, unchanged utility function of \bar{s} must be used to first show that a rewrite is useful. In contrast, μ and M *are* taken from s , because for any interesting Gödel Machine, c will both contain instructions that affect the external environment (e.g., output to actuators) and instructions that update the internal model of the environment (encoded in s) accordingly. To make use of these updated models, u must extract them from s . Likewise, κ and K are taken from c , because after each instruction the continuation necessarily changes, resulting in new κ and K .

It is also important to notice that the function u is recursive. In formula (10.1), a Gödel Machine’s life ends when the continuation sample space K_c is empty. In that case, the term $\mathbb{E}_{\kappa_c, K_c}[u_{\bar{s}} \mid env]$ is zero and recursion on u stops. In the most common case, however, an instruction will be deterministic, e.g., $K_c = \{(s', c')\}$. Then formula (10.1) reduces to $u_{\bar{s}}(s, c) = \mathbb{E}_{\mu_s, M_s}[r_{\bar{s}}(s, \cdot)] + u_{\bar{s}}(s', c')$. So for chains of deterministic state transitions, the expected cumulative future utility reduces to a summation of expected rewards in the successive states. But in e.g. RL settings, even those expected rewards are usually zero, because most instructions are not involved with receiving information from the environment (which in RL is usually the source of rewards).

We are now in a position to introduce the *target theorem*, which is based on the utility function u . Let s be an expected future state *before* some time t . Furthermore, let $c^* = \{\text{wait_until}(t); \text{switchprog}()\}$ denote the continuation that waits until time t (e.g., by continually calling a primitive clock function until the returned time tests as greater or equal to t) and then executes *switchprog*, which may inspect and modify any part of s . Now $u_{\bar{s}}(s, c^*)$ tells us what is the expected cumulative future utility of running *switchprog* on s at time t . If this utility compares positively against the expected cumulative future utility of *not* switching, then the Gödel Machine must perform the switch at time t . Thus, the *searcher* is to find a state s , a time t , and a nullary procedure *switchprog* such that $\text{tt}_{\bar{s}}(s, t, \text{switchprog})$ holds, where:

$$\text{tt}_{\bar{s}}(s, t, \text{switchprog}) = [u_{\bar{s}}(s, \{\text{wait_until}_s(t); \text{switchprog}()\}) > u_{\bar{s}}(s, \{\text{scheduler}_s()\})] \quad (10.2)$$

where *scheduler* is the procedure that interleaves the Gödel Machine’s *solver* and *searcher*. Thus $\{\text{scheduler}_s()\}$ denotes the continuation that “proceeds as usual” from s (including searching for

new *switchprogs*). Note that, since *switchprog* may overwrite the scheduler, it must as its last instruction invoke a new scheduler (if needed), or simply invoke the old scheduler if it has been left intact.

There are two ways in which this target theorem can be used to find provably useful self-rewrites: (1) generate a candidate *switchprog* (e.g., a program performing optimization techniques also applied by state-of-the-art compilers), generate a future switch time t at least as far ahead as the time the evaluation of the utility of running *switchprog* is expected to take, and make s a copy of the current state \bar{s} ; then run both continuations specified by tt on s (this is safe because s acts like a sandbox) until the inequality can be shown to be true or false, or until the current time exceeds t ; if the inequality is false or time has run out, generate a new *switchprog* and start over; else, swap the current state for s , wait until time t and execute *switchprog*. (2) Start with an axiomatic system with axiom and rules, and generate theorems (or generate programs that calculate theorems [16]), testing them for equivalence with tt . The first method can be seen as a top-down approach to proving tt and the latter as bottom-up. Hybrid approaches are also possible, so this specification of the target theorem allows for a *family* of possible Gödel Machine implementations [19].

To make sure the Gödel Machine is at least *asymptotically* optimal even before the first self-rewrite, it may be initialized with Hutter’s non-self-referential but *asymptotically fastest algorithm for all well-defined problems* HSEARCH [8], which uses a hardwired brute force proof searcher and ignores the costs of proof search. Given some problem, the Gödel Machine may decide to replace HSEARCH by a faster method suffering less from large constant overhead, but even if it does not, its performance will not be less than asymptotically optimal.

Having explained an example of a system needing self-reflection, we will now move on to explaining the technical aspects of attaining self-reflection.

10.3 The Theoretical Foundations of Self-Reflective Systems

In the mid-eighties, there was a fashion for reflective interpreters, a fad that gave rise to a remarkable term: “reflective towers.” Just imagine a marsh shrouded in mist and a rising tower with its summit lost in gray and cloudy skies—pure Rackham! (...) Well, who hasn’t dreamed about inventing (or at least having available) a language where anything could be redefined, where our imagination could gallop unbridled, where we could play around in complete programming liberty without trammel nor hindrance? [12]

The reflective tower that Queinnec is so poetically referring to, is a visualization of what happens when performing self-reflection.² A program running on the n th floor of the tower is the effect of an evaluator running on the $(n - 1)$ th floor. When a program running on the n th floor performs a reflective instruction, this means it gains access to the state of the program running at the $(n - 1)$ th floor. But the program running on the n th floor can also invoke the evaluator function, which causes a program to be run on the $(n + 1)$ th floor. If an evaluator evaluates an evaluator evaluating an evaluator etc. etc., we get the image of “a rising tower with its summit lost in gray and cloudy skies.” If a program reflects on a reflection of a reflection etc. etc., we get the image of the base of the tower “shrouded in mist.” What happens were we to stumble upon a ground floor? In the original vision of the reflective tower, there is none, because it extends infinitely in both directions (up and down). Of course in practice such infinities will have to be relaxed, but the point is that it will be of great interest to see exactly when, where, and how problems will surface, which is precisely the point of this section.

According to Queinnec, there are two things that a reflective interpreter must allow for. First:

²*Reflection* can have different meanings in different contexts, but here we maintain the meaning defined in the introduction: the ability to inspect and modify one’s own currently running program.

Reflective interpreters should support introspection, so they must offer the programmer a means of grabbing the computational context at any time. By “computational context,” we mean the lexical environment and the continuation. [12]

The *lexical environment* is the set of bindings of variables to values, for all variables in the lexical scope³ of the currently running program. Note that the lexical environment has to do only with the internal state of a program; it has nothing to do with the *external environment* with which a program may be interacting through actuators. The *continuation* is a representation of future computations, as we have seen in the previous section. And second:

A reflective interpreter must also provide means to modify itself (a real thrill, no doubt), so (...) that functions implementing the interpreter are accessible to interpreted programs. [12]

In this section we will explore what these two requirements for self-reflection mean technically, and to what extent they are realizable theoretically and practically.

10.3.1 Basic λ -calculus

Let us first focus on representing a computational context; that is, a lexical environment plus a continuation. The most obvious and well-studied way of elucidating these is in λ -calculus, which is a very simple (theoretical) programming language. The notation of expressions in λ -calculus is a bit different from usual mathematical notation; for example, parentheses in function application are written on the outside, so that we have $(f x)$ instead of the usual $f(x)$. Note also the rounded parentheses in $(f x)$; they are part of the syntax and always indicate function application, i.e., they are not allowed freely. Functions are made using *lambda abstraction*; for example, the identity function is written as $\lambda x.x$. So we write a λ symbol, then a variable, then a period, and finally an expression that is the *body* of the function. Variables can be used to name expression; for example, applying the identity function to y can be written as “ $g(y)$ where $g(x) = x$,” which is written in λ -calculus as $(\lambda g.(g y) \lambda x.x)$.

Formally, the language (syntax) Λ_1 of basic λ -calculus is specified using the following recursive grammar.

$$\Lambda_1 ::= v \mid \lambda v.\Lambda_1 \mid (\Lambda_1 \Lambda_1) \quad (10.3)$$

This expresses succinctly that (1) if x is a variable then x is an expression, (2) if x is a variable and M is an expression then $\lambda x.M$ is an expression (*lambda abstraction*), and (3) if M and N are expressions then $(M N)$ is an expression (*application*).

In pure λ -calculus, the only “operation” that we can perform that is somewhat akin to evaluation, is β -reduction. β -reduction can be applied to a Λ_1 (sub)expression if that (sub)expression is an application with a lambda abstraction in the operator position. Formally:

$$(\lambda x.M N) \xrightarrow{\beta} M[x \leftarrow N] \quad (10.4)$$

So the “value” of supplying an expression N to a lambda abstraction $\lambda x.M$ is M with all occurrences of x replaced by N . Of course we can get into all sorts of subtle issues if several nested lambda abstractions use the same variable names, but let’s not go into that here, and assume a unique variable name is used in each lambda abstraction.

At this point one might wonder how to *evaluate* an arbitrary Λ_1 expression. For example, a variable should evaluate to its binding, but how do we keep track of the bindings introduced by lambda abstractions? For this we need to introduce a *lexical environment*, which contains the bindings of all the variables in scope. A lexical environment is historically denoted using the letter ρ and is represented here as a function taking a variable and returning the value bound to it. We can then specify our first evaluator \mathcal{E}_1 as a function taking an expression and a lexical

³For readers unfamiliar with the different possible scoping methods, it suffices to know that lexical (also called static) scoping is the intuitive, “normal” method found in most modern programming languages.

environment and returning the value of the expression. In principle this evaluator can be specified in any formalism, but if we specify it in λ -calculus, we can easily build towards the infinite reflective tower, because then the computational context will have the same format for both the evaluator and evaluated expression.

There are three syntactic cases in language Λ_1 and \mathcal{E}_1 splits them using double bracket notation. Again, we have to be careful not to confuse the specification language and the language being specified. Here they are both λ -calculus in order to show the reflective tower at work, but we should not confuse the different floors of the tower! So in the specification below, if the Λ_1 expression between the double brackets is floor n , then the right-hand side of the equal sign is a Λ_1 expression on floor $n - 1$. \mathcal{E}_1 is then specified as follows.

$$\mathcal{E}_1[x] = \lambda\rho.(\rho x) \quad (10.5)$$

$$\mathcal{E}_1[\lambda x.M] = \lambda\rho.\lambda\varepsilon.(\mathcal{E}_1[M] \rho[x \leftarrow \varepsilon]) \quad (10.6)$$

$$\mathcal{E}_1[(M N)] = \lambda\rho.((\mathcal{E}_1[M] \rho) (\mathcal{E}_1[N] \rho)) \quad (10.7)$$

It should first be noted that all expression on the right-hand side are lambda abstractions expecting a lexical environment ρ , which is needed to look up the values of variables. To *start* evaluating an expression, an empty environment can be provided. According to the first case, the value of a variable x is its binding in ρ . According to the second case, the value of a lambda abstraction is itself a function, waiting for a value ε ; when received, M is evaluated in ρ extended with a binding of x to ε . According to the third case, the value of an application is the value of the operator M applied to the value of the operand N , assuming the value of M is indeed a function. Both operator and operand are evaluated in the same lexical environment.

For notational convenience, we will abbreviate nested applications and lambda abstractions from now on. So $((f x) y)$ will be written as $(f x y)$ and $\lambda x.\lambda y.M$ as $\lambda xy.M$.

We have now introduced an explicit lexical environment, but for a complete computational context, we also need a representation of the continuation. To that end, we rewrite the evaluator \mathcal{E}_1 in *continuation-passing style* (CPS). This means that a continuation, which is a function historically denoted using the letter κ , is extended whenever further evaluation is required, or invoked when a value has been obtained. That way, κ always expresses the future of computations—although, as described before, it is usually only partially expanded. The new evaluator \mathcal{E}'_1 works explicitly with the full computational context by taking both the lexical environment ρ and the continuation κ as arguments. Formally:

$$\mathcal{E}'_1[x] = \lambda\rho\kappa.(\kappa (\rho x)) \quad (10.8)$$

$$\mathcal{E}'_1[\lambda x.M] = \lambda\rho\kappa.(\kappa \lambda\varepsilon\kappa'.(\mathcal{E}'_1[M] \rho[x \leftarrow \varepsilon] \kappa')) \quad (10.9)$$

$$\mathcal{E}'_1[(M N)] = \lambda\rho\kappa.(\mathcal{E}'_1[M] \rho \lambda f.(\mathcal{E}'_1[N] \rho \lambda x.(f x \kappa))) \quad (10.10)$$

In the first and second case, the continuation is immediately invoked, which means that the future of computations is reduced. In these cases this is appropriate, because there is nothing *extra* to be evaluated. In the third case, however, two things need to be done: the operator and the operand need to be evaluated. In other words, while the operator (M) is being evaluated, the evaluation of the operand (N) is a computation that lies in the future. Therefore the continuation, which represents this future, must be extended. This fact is now precisely represented by supplying to the evaluator of the operator $\mathcal{E}'_1[M]$ a new continuation which is an extension of the supplied continuation κ : it is a function waiting for the value f of the operator. As soon as f has been received, this extended continuation invokes the evaluator for the operand $\mathcal{E}'_1[N]$, but *again* with a new continuation: $\lambda x.(f x \kappa)$. This is because, while the operand N is being evaluated, there is still a computation lying in the future; namely, the actual invocation of the value (f) of the operator on the value (x) of the operand. At the moment of this invocation, the future of computations is exactly the same as the future before evaluating both M and N ; therefore, the continuation κ of that moment must be passed on to the function invocation. Indeed, in (10.9) we see what will be done with this continuation: a lambda abstraction evaluates to a binary function, where ε is the value of its operand and κ' is the continuation at the time when the function is actually invoked.

This is then also the continuation that has to be supplied to the evaluator of the function’s body ($\mathcal{E}'_1[M]$).

For the reader previously unfamiliar with CPS it will be very instructive to carefully compare (10.5)–(10.7) with (10.8)–(10.10), especially the application (third) case.

Let’s return now to the whole point of explicitly representing the lexical environment and the continuation. Together they constitute the computational context of a program under evaluation; for this program to have self-reflective capabilities, it must be able to “grab” both of them. This is easily achieved now, by adding two special constructs to our language, **grab-r** and **grab-k**, which evaluate to the current lexical environment and current continuation, respectively.

$$\mathcal{E}'_1[\mathbf{grab-r}] = \lambda\rho\kappa.(\kappa \rho) \quad (10.11)$$

$$\mathcal{E}'_1[\mathbf{grab-k}] = \lambda\rho\kappa.(\kappa \kappa) \quad (10.12)$$

These specifications look deceptively simple, and indeed they are. Because although we now have a means to grab the computational context at any time, there is little we can do with it. Specifically, there is no way of inspecting or modifying the lexical environment and continuation after obtaining them. So they are as good as black boxes.

Unfortunately there are several more subtle issues with the evaluator \mathcal{E}'_1 , which are easily overlooked. Suppose we have a program π in language Λ_1 and we want to evaluate it. For this we would have to determine the value of $(\mathcal{E}'_1[\pi] \rho_0 \lambda x.x)$, supplying the evaluator with an initial environment and an initial continuation. The initial continuation is easy: it is just the identity function. The initial environment ρ_0 and the extension of an environment (used in (10.9) but not specified yet) can be specified as follows.

$$\rho_0 = \lambda x.\perp \quad (10.13)$$

$$\rho[x \leftarrow \varepsilon] = \lambda y.\mathbf{if} (= x y) \varepsilon (\rho y) \quad (10.14)$$

So the initial environment is a function failing on every binding lookup, whereas an extended environment is a function that tests for the new binding, returning either the bound value (ε) when the variables match ($(= x y)$), or the binding according to the unextended environment ((ρy)). But here we see more problems of our limited language Λ_1 surfacing: there are no conditional statements (**if**), no primitive functions like $=$, and no constants like \perp , so we are not allowed to write the initial and extended lexical environment as above. Also the language does not contain numbers to do arithmetics (which has also caused our examples of Λ_1 expressions to be rather abstract). Admittedly, conditionals, booleans, and numbers *can* be represented in pure λ -calculus, but that is more of an academic exercise than a practical approach. Here we are interested in building towards practically feasible self-reflection, so let’s see how far we can get by extending our specification language to look more like a “real” programming language.

10.3.2 Constants, Conditionals, Side-effects, and Quoting

Let’s extend our very austere language Λ_1 and add constructs commonly found in programming languages, and in Scheme [11] in particular. Scheme is a dialect of Lisp, is very close to λ -calculus, and is often used to study reflection in programming [1, 12, 9]. The Scheme-like language Λ_2 is specified using the following recursive grammar.

$$\Lambda_2 ::= v \mid \lambda v.\Lambda_2 \mid (\Lambda_2 \Lambda_2) \mid c \mid \mathbf{if} \Lambda_2 \Lambda_2 \Lambda_2 \mid \mathbf{set!} v \Lambda_2 \mid \mathbf{quote} \Lambda_2 \quad (10.15)$$

where constants (c) include booleans, numbers, and, notably, primitive functions. These primitive functions are supposed to include operators for doing arithmetics and for inspecting and modifying data structures (including environments and continuations!), as well as IO interactions. The **if** construct introduces the familiar *if-then-else* expression, **set!** introduces assignment (and thereby side-effects), and **quote** introduces quoting (which means treating programs as data).

In order to appropriately model side-effects, we need to introduce the storage, which is historically denoted using the letter σ . From now on the lexical environment (ρ) does not bind variables

to values, but to addresses. The storage, then, binds addresses to values. This setup allows `set!` to change a binding by changing the storage but not the lexical environment. The new evaluator \mathcal{E}_2 is then specified as follows.

$$\mathcal{E}_2[x] = \lambda\rho\sigma\kappa.(\kappa\ \sigma\ (\sigma\ (\rho\ x))) \quad (10.16)$$

$$\mathcal{E}_2[\lambda x.M] = \lambda\rho\sigma\kappa.(\kappa\ \sigma\ \lambda\varepsilon\sigma'\kappa'.(\mathcal{E}_2[M]\ \rho[x \leftarrow \alpha]\ \sigma'[\alpha \leftarrow \varepsilon]\ \kappa')) \quad (10.17)$$

$$\mathcal{E}_2[(M\ N)] = \lambda\rho\sigma\kappa.(\mathcal{E}_2[M]\ \rho\ \sigma\ \lambda\sigma'f.(\mathcal{E}_2[N]\ \rho\ \sigma'\ \lambda\sigma''x.(f\ x\ \sigma''\ \kappa))) \quad (10.18)$$

where α is a fresh address. Again, it will be very instructive to carefully compare the familiar (10.8)–(10.10) with (10.16)–(10.18). The main difference is the storage which is being passed around, invoked (10.16), and extended (10.17). The new cases are:

$$\mathcal{E}_2[c] = \lambda\rho\sigma\kappa.(\kappa\ \sigma\ c) \quad (10.19)$$

$$\mathcal{E}_2[\text{if } C\ T\ F] = \lambda\rho\sigma\kappa.(\mathcal{E}_2[C]\ \rho\ \sigma\ \lambda\sigma'c.(\text{if } c\ \mathcal{E}_2[T]\ \mathcal{E}_2[F]\ \rho\ \sigma'\ \kappa)) \quad (10.20)$$

$$\mathcal{E}_2[\text{set! } x\ M] = \lambda\rho\sigma\kappa.(\mathcal{E}_2[M]\ \rho\ \sigma\ \lambda\sigma'\varepsilon.(\kappa\ \sigma'[(\rho\ x) \leftarrow \varepsilon]\ \varepsilon)) \quad (10.21)$$

$$\mathcal{E}_2[\text{quote } M] = \lambda\rho\sigma\kappa.(\kappa\ \sigma\ M) \quad (10.22)$$

Constants simply evaluate to themselves. Similarly, quoting an expression M returns M unevaluated. Conditional statements force a choice between evaluating the *then*-body ($\mathcal{E}_2[T]$) and the *else*-body ($\mathcal{E}_2[F]$). Note that in (10.21) only the storage (σ') is changed, such that variable x retains the address that is associated with it in the lexical environment, causing future look-ups (see (10.16)) to return the new value (ε) for x .

As an example of a primitive function, the binary addition operator can be specified as follows.

$$+ = \lambda x\sigma\kappa.(\kappa\ \sigma\ \lambda y\sigma'\kappa'.(\kappa'\ \sigma'\ (+\ x\ y))) \quad (10.23)$$

Note the recursion; $+$ is specified in terms of the $+$ operator used one floor lower in the reflective tower. The same holds for `if` in (10.20). Where does it bottom out? The marsh is still very much shrouded in mist.

The reason that we cannot specify primitive procedures and constructs nonrecursively at this moment is because the specifications so far say nothing about the data structures used to represent the language constructs. Theoretically this is irrelevant, because the ground floor of the reflective tower is infinitely far away. But the inability to inspect and modify data structures makes it hard to comply with Queinnek’s second condition for self-reflection, namely that everything—including the evaluator—should be modifiable. There are now (at least) two ways to proceed: (1) set up a recursive loop of evaluators with shared state to make the reflective tower “float” without a ground floor, or (2) let go of the circular language specification and retire to a reflective bungalow where everything happens at the ground floor. Both options will be discussed in the next two sections, respectively.

10.4 Nested Meta-Circular Evaluators

Using the well-studied technique of the *meta-circular evaluator* [1], it is possible to attain self-reflectivity in any (Turing-complete) programming language. A meta-circular evaluator is basically an interpreter for the same programming language as the one in which the interpreter is written. Especially suitable for this technique are homoiconic languages such as Lisp and in particular its dialect Scheme [11], which is very close to λ -calculus and is often used to study meta-circular evaluators and self-reflection in programming in general [1, 12, 18, 5, 6, 20, 3, 2, 9]. So a meta-circular Scheme evaluator is a program written in Scheme which can interpret programs written in Scheme. There is no problem with circularity here, because the program running the meta-circular Scheme evaluator itself can be written in any language. For clarity of presentation let us consider how a meta-circular Scheme evaluator can be used to obtain the self-reflectivity needed for e.g. a Gödel Machine.

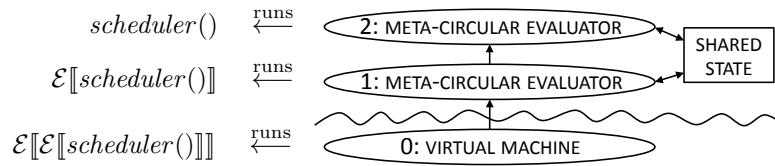


Figure 10.1: The self-inspection and self-modification required for a Gödel Machine implementation can be attained by having a double nesting of meta-circular evaluators run the Gödel Machine’s *scheduler*. Every instruction is grounded in some virtual machine running “underwater,” but the nested meta-circular evaluators can form a loop of self-reflection without ever “getting their feet wet.”

In what follows, let $\mathcal{E}[\pi]$ denote a call to an evaluator with as argument program π . As before, the double brackets mean that program π is to be taken literally (i.e., unevaluated), for it is the task of the evaluator to determine the value of π . Now for a meta-circular evaluator, $\mathcal{E}[\pi]$ will give the same value as $\mathcal{E}[\mathcal{E}[\pi]]$ and $\mathcal{E}[\mathcal{E}[\mathcal{E}[\pi]]]$ and so on. Note that $\mathcal{E}[\mathcal{E}[\pi]]$ can be viewed as the evaluator reading and interpreting its own source code and determining how the program constituting that source code evaluates π . A very clear account of a complete implementation (in Scheme) of a simple reflective interpreter (for Scheme) is provided by Jefferson *et al.* [9], of which we shall highlight one property that is very interesting in light of our goal to obtain a self-reflective system. Namely, in the implementation by Jefferson *et al.*, no matter how deep one would nest the evaluator (as in $\mathcal{E}[\mathcal{E}[\dots\pi\dots]]$), all levels will share the same *global environment*⁴ for retrieving and assigning procedures and data. This implies that self-reflection becomes possible when running a program (in particular, a Gödel Machine’s *scheduler*) in a doubly nested meta-circular interpreter with a shared global environment (see figure 10.1). Note that this setup can be attained regardless of the hardware architecture and without having to invent new techniques.

Consider again the quote at the start of section 10.3. It is interesting to note that this paragraph is immediately followed by equally poetic words of caution: “However, we pay for this dream with exasperatingly slow systems that are almost incompilable and plunge us into a world with few laws, hardly even any gravity.” We can now see more clearly this world without gravity in figure 10.1: the nested meta-circular evaluators are floating above the water, seemingly without ever getting their feet wet.

But this is of course an illusion. Consider again the third quote, stating that in a self-reflective system the functions implementing the interpreter must be modifiable. But what are those “functions implementing the interpreter?” For example, in the shared global environment, there might be a function called `evaluate` and helper functions like `lookup` and `allocate`. These are all modifiable, as desired. But all these functions are composed of *primitive* functions (such as Scheme’s `cons` and `cdr`) and of syntactical compositions like function applications, conditionals, and lambda abstractions. Are those functions and constructs also supposed to be modifiable? All the way down the “functions implementing the interpreter” can be described in terms of machine code, but we cannot change the instruction set of the processor. The regress in self-modifiability has to end *somewhere*.⁵

⁴A global environment can be seen as an extension of the lexical environment. When looking up the value of a variable, first the lexical environment is searched; if no binding is found, the global environment is searched. Similarly for assignment. The global environment is the same in every lexical context. Again, all this has nothing to do with the *external environment*, which lies outside the system.

⁵Unless we consider systems that are capable of changing their own hardware. But then still there are (probably?) physical limits to the modifiability of hardware.

10.5 A Functional Self-Reflective System

Taking the last sentence above to its logical conclusion, let us now investigate the consequences of ending the regress in self-modifiability already at the functions implementing the interpreter. If the interpreter interprets a Turing-complete instruction set, then having the ability to inspect and modify the lexical environment and the continuation at any point is already enough to attain *functionally* complete reflective control.

The reason that Queinnec mentions modifiability of the functions implementing the interpreter is probably not for functional reasons, but for timing and efficiency reasons. As we have seen, the ability to “grab the computational context” is enough to attain functional reflection, but it does not allow a program to speed up its own interpreter (*if* it would know how). In this section we will introduce a new interpreter for a self-reflective system that takes a middle road. The functions implementing this interpreter are primitives (i.e., black boxes); however, they are also (1) very few in number, (2) very small, and (3) fast in execution.⁶ We will show that a program being run by this interpreter can inspect and modify itself (including speed upgrades) in a sufficiently powerful way to be self-reflective.

First of all, we need a programming language. Here we use a syntax that is even simpler than classical λ -calculus, specified by the following recursive grammar.

$$\Lambda_3 ::= c \mid n \mid (\Lambda_3 \Lambda_3) \quad (10.24)$$

where c are constants (symbols for booleans and primitive functions) and n are numbers (sequences of digits). There are no special forms (such as `quote`, `lambda`, `set!`, and `if` in Scheme), just function application, where all functions are unary. Primitive binary functions are invoked as, e.g., `((+ 1) 2)`. Under the hood, the only compound data structure is the pair⁷ and an application $(f x)$ is simply represented as a pair $f:x$. An instance of a number takes as much space as one pair (say, 64 bits); constants do not take any space.

For representing what happens under the hood, we extensively use the notation $a:d$, meaning a pair with head a and tail d . The colon associates from right to left, so $a:ad:dd$ is the same as $(a:(ad:dd))$. \perp is a constant used to represent false or an error or undefined value; for example, the result of an integer division by zero is \perp (programs are *never* halted due to an error). The empty list (Scheme: `'()`) is represented by the constant \emptyset .

To look up a value stored in a storage σ at location p , we use the notation $\sigma[p \rightarrow n]$ for numbers and $\sigma[p \rightarrow a:d]$ for pairs. Allocation is denoted as $\sigma[q \stackrel{*}{\leftarrow} n]$ for numbers and $\sigma[q \stackrel{*}{\leftarrow} a:d]$ for pairs, where q is a fresh location where the allocated number or pair can be found. \mathbb{N}_σ is used to denote the set of locations where numbers are stored; all other locations store pairs. The set of constants is denoted as \mathbb{P} . We refer the reader to the appendix for more details on notation and storage whenever something appears unclear.

The crux of the interpreter to be specified in this section is that not only programs are stored as pairs, but also the lexical environment and the continuation. Since programs can build (`cons`), inspect (`car`, `cdr`),⁸ and modify (`set-car!`, `set-cdr!`) pairs, they can then also build, inspect, and modify lexical environments and continuations using these same functions. That is, provided that they are accessible. The interpreter presented below will see to that.

As is evident from (10.24), the language lacks variables, as well as a construct for lambda abstraction. Still, we stay close to λ -calculus by using the technique of De Bruijn indices [4]. Instead of named variables, numbers are used to refer to bound values. For example, where in Scheme the λ -calculus expression $\lambda x.\lambda y.x$ is written as `(lambda (x y) x)`, with De Bruijn indices one would write `(lambda (lambda 1))`. That is, a number n evaluates to the value bound by the

⁶The last point of course depends on the exact implementation, but since they are so small, it will be clear that they can be implemented very efficiently. In our not-so-optimized reference implementation written in C++, one primitive step of the evaluator takes about twenty nanoseconds on a 2009-average PC. Since reflection is possible after every such step, there are about fifty million chances per second to interrupt evaluation, grab the computational context, and inspect and possibly modify it.

⁷Our reference implementation also supports vectors (fixed-size arrays) though.

⁸In Scheme and Λ_3 , `car` and `cdr` are primitive unary functions that return the head and tail of a pair, respectively.

n th enclosing lambda operator (counting the first as zero). A number has to be quoted to be used as a number (e.g., to do arithmetics). But as we will see shortly, neither `lambda` nor `quote` exists in Λ_3 ; however, a different, more general mechanism is employed that achieves the same effect.

By using De Bruijn indices the lexical environment can be represented as a list,⁹ with “variable” lookup being simply a matter of indexing the lexical environment. The continuation will also be represented as a list; specifically, a list of functions, where each function specifies what has to be done as the next step in evaluating the program. So at the heart of the interpreter is a stepper function, which pops the next function from the continuation and invokes it on the current program state. This is performed by the primitive unary function `step`, which is specified as follows. (Note that every n -ary primitive function takes $n + 2$ arguments, the extra two being the continuation and the storage.)

$$\mathbf{step}(\pi)(\kappa, \sigma) = \begin{cases} \varphi(\pi)(\kappa', \sigma) & \text{if } \sigma[\kappa \rightarrow \varphi:\kappa'] \text{ and } \varphi \in \mathbb{F}_1 \\ \mathbf{step}(\varphi')(\kappa', \sigma[\varphi' \stackrel{*}{\leftarrow} \varphi:\pi]) & \text{if } \sigma[\kappa \rightarrow \varphi:\kappa'] \text{ and } \varphi \in \mathbb{F}_2 \\ \varphi(\pi', \pi)(\kappa', \sigma) & \text{if } \sigma[\kappa \rightarrow (\varphi:\pi'):\kappa'] \text{ and } \varphi \in \mathbb{F}_2 \\ \mathbf{step}(\perp)(\kappa', \sigma) & \text{if } \sigma[\kappa \rightarrow \varphi:\kappa'] \\ \pi & \text{otherwise} \end{cases} \quad (10.25)$$

where \mathbb{F}_1 and \mathbb{F}_2 are the sets of all primitive unary and binary functions, respectively. In the first case, the top of the continuation is a primitive unary function, and `step` supplies π to that function. In the second case, the top is a primitive binary function, but since we need a second argument to invoke that function, `step` makes a pair of the function and π as its first argument. In the third case, where such a pair with a binary function is found at the top of the continuation, π is taken to be the second argument and the binary function is invoked. In the fourth case, the top of the continuation is found not to be a function (because the first three cases are tried first); this is an error, so the non-function is popped from the continuation and `step` continues with \perp to indicate the error. Finally, in the fifth case, when the continuation is not a pair at all, `step` cannot continue with anything and π is returned as the final value.

It is crucial that all primitive functions invoke `step`, for it is the heart of the evaluator, processing the continuation. For example, the primitive unary function `cdr`, which takes the second element of a pair, is specified as follows.

$$\mathbf{cdr}(p)(\kappa, \sigma) = \begin{cases} \mathbf{step}(d)(\kappa, \sigma) & \text{if } \sigma[p \rightarrow a:d] \\ \mathbf{step}(\perp)(\kappa, \sigma) & \text{otherwise} \end{cases} \quad (10.26)$$

Henceforth we will not explicitly write the “`step`(\perp)(κ, σ) otherwise” part anymore. As a second example, consider addition, which is a primitive binary function using the underlying implementation’s addition operator.

$$+(n, n')(\kappa, \sigma) = \mathbf{step}(n'')(\kappa, \sigma[n'' \stackrel{*}{\leftarrow} (m + m')]) \quad \text{if } n, n' \in \mathbb{N}_\sigma, \sigma[n \rightarrow m] \text{ and } \sigma[n' \rightarrow m'] \quad (10.27)$$

Notice that, like every expression, n and n' are mere indices of the storage, so first their values (m and m' , respectively) have to be retrieved. Then a fresh index n'' is allocated and the sum of m and m' is stored there. Here $\overset{64}{+}$ may be the 64-bit integer addition operator of the underlying implementation’s language.

Now we turn to the actual evaluation function: `eval`. It is a unary function taking a *closure*, denoted here using the letter θ . A closure is a pair whose head is a lexical environment and whose tail is an (unevaluated) expression. Given these facts we can immediately answer the question of how to start the evaluation of a Λ_3 program π stored in storage σ : initialize $\sigma' = \sigma[\theta \stackrel{*}{\leftarrow} \varnothing:\pi][\kappa \stackrel{*}{\leftarrow} \mathbf{eval}:\varnothing]$ and determine the value of `step`(θ)(κ, σ'). So we form a pair (closure) of the lexical environment (which is initially empty: \varnothing) and the program π to be evaluated in

⁹In Scheme and Λ_3 , a *list* is a tail-linked, \varnothing -terminated sequence of pairs, where the elements of the list are held by the heads of the pairs.

that environment. What $\text{step}(\theta)(\kappa, \sigma')$ will do is go to the first case in (10.25), namely to call the primitive unary function eval with θ as argument. Then eval must distinguish three cases because a program can either be a primitive constant (i.e., \perp , \emptyset , or a function), a number, or a pair. Constants are self-evaluating, numbers are treated as indices of the lexical environment, and pairs are treated as applications. eval is then specified as follows.

$$\text{eval}(\theta)(\kappa, \sigma) = \begin{cases} \text{step}(\pi)(\kappa, \sigma) & \text{if } \sigma[\theta \rightarrow \rho:\pi] \text{ and } \pi \in \mathbb{P} \\ \text{step}(\rho \downarrow_{\sigma} n)(\kappa, \sigma) & \text{if } \sigma[\theta \rightarrow \rho:\pi] \text{ and } \pi \in \mathbb{N}_{\sigma} \text{ and } \sigma[\pi \rightarrow n] \\ \text{step}(\theta_1)(\kappa', \sigma') & \text{if } \sigma[\theta \rightarrow \rho:\pi:\pi'] \end{cases} \quad (10.28)$$

where $\sigma' = \sigma[\theta_1 \stackrel{*}{\leftarrow} \rho:\pi][\theta_2 \stackrel{*}{\leftarrow} \rho:\pi'][\kappa' \stackrel{*}{\leftarrow} \text{eval}:(\text{next}:\theta_2):\kappa]$. In the first case the constant is simply extracted from the closure (θ). In the second case, the De Bruijn index inside the closure is used as an index of the lexical environment ρ , also contained in θ . (Definitions of \mathbb{P} , \mathbb{N} , and \downarrow are provided in the appendix). In the third case, eval makes two closures: θ_1 is the operator, to be evaluated immediately; θ_2 is the operand, to be evaluated “next.” Both closures get the same lexical environment (ρ). The new continuation κ' reflects the order of evaluation of first π then π' : eval itself is placed at the top of the continuation, followed by the primitive binary function next with its first argument (θ_2) already filled in. The second argument of next will become the value of π , i.e., the operator of the application. next then simply has to save the operator on the continuation and evaluate the operand, which is its first argument (a closure). This behavior is specified in the *second* case below.

$$\text{next}(\theta, \varphi)(\kappa, \sigma) = \begin{cases} \text{step}(\theta)(\kappa', \sigma[\kappa' \stackrel{*}{\leftarrow} \varphi':\kappa]) & \text{if } \sigma[\varphi \rightarrow \text{quote2}:\varphi'] \\ \text{step}(\theta)(\kappa', \sigma[\kappa' \stackrel{*}{\leftarrow} \text{eval}:\varphi:\kappa]) & \text{otherwise} \end{cases} \quad (10.29)$$

What does the first case do? It handles a generalized form of quoting. The reason that quote , lambda , set! , and if are special forms in Scheme (as we have seen in Λ_2) is because their arguments are not to be evaluated immediately. This common theme can be handled by just one mechanism. We introduce the binary primitive function quote2 and add a hook to next (the first case above) to prevent quote2 's second argument from being evaluated. Instead, the closure (θ , which represented the unevaluated second argument of quote2) is supplied to the first argument of quote2 (which must therefore be a function).¹⁰

So where in Scheme one would write $(\text{quote } (1 \ 2))$, here we have to write $((\text{quote2 } \text{cdr}) (1 \ 2))$. Since quote2 supplies a closure (an environment–expression pair) to its first argument, cdr can be used to select the (unevaluated) expression from that closure. Also, we have to quote numbers explicitly, otherwise they are taken to be De Bruijn indices. For example, Scheme's $(+ \ 1 \ 2)$ here becomes $((+ ((\text{quote2 } \text{cdr}) \ 1)) ((\text{quote2 } \text{cdr}) \ 2))$. This may all seem cumbersome, but it should be kept in mind that the Scheme representation can easily be converted automatically, so a programmer does not have to notice any difference and can just continue programming using Scheme syntax.

What about lambda abstractions? For that we have the primitive binary function lambda2 , which also needs the help of quote2 in order to prevent its argument from being evaluated prematurely. For example, where in Scheme the λ -calculus expression $\lambda x.\lambda y.x$ is written as $(\text{lambda } (x \ y) \ x)$, here we have to write $((\text{quote2 } \text{lambda2}) ((\text{quote2 } \text{lambda2}) \ 1))$. lambda2 is specified as follows (*cf.* (10.17)).

$$\text{lambda2}(\theta, \varepsilon)(\kappa, \sigma) = \text{step}(\theta')(\kappa', \sigma[\theta' \stackrel{*}{\leftarrow} (\varepsilon:\rho):\pi][\kappa' \stackrel{*}{\leftarrow} \text{eval}:\kappa]) \quad \text{if } \sigma[\theta \rightarrow \rho:\pi] \quad (10.30)$$

So lambda2 takes a closure and a value, extends the lexical environment captured by the closure with the provided value, and then signals that it wants the new closure to be evaluated by pushing eval onto the continuation.

As we saw earlier, conditional expressions are ternary, taking a condition, a *then*-part, and an *else*-part. In the very simple system presented here, if is just a primitive unary function, which

¹⁰ quote2 may also be seen as a kind of *feexpr* builder.

pops from the continuation both the *then*-part and the *else*-part, and sets up one of them for evaluation, depending on the value of the condition.

$$\text{if}(\varepsilon)(\kappa, \sigma) = \begin{cases} \text{step}(\theta)(\kappa'', \sigma[\kappa'' \stackrel{*}{\leftarrow} \text{eval}:\kappa']) & \text{if } \varepsilon \neq \perp \text{ and } \sigma[\kappa \rightarrow (\text{next}:\theta):\varphi:\kappa'] \\ \text{step}(\theta)(\kappa'', \sigma[\kappa'' \stackrel{*}{\leftarrow} \text{eval}:\kappa']) & \text{if } \varepsilon = \perp \text{ and } \sigma[\kappa \rightarrow \varphi:(\text{next}:\theta):\kappa'] \end{cases} \quad (10.31)$$

So any value other than \perp is interpreted as “true.” Where in Scheme one would write `(if c t f)`, here we simply write `((if c) t) f`.

An implementation of Scheme’s `set!` in terms of `quote2` can be found in the appendix.

Now we have seen that the lexical environment and the program currently under evaluation are easily obtained using `quote2`. What remains is reflection on the continuation. Both inspection and modification of the continuation can be achieved with just one very simple primitive: `swap-continuation`. It simply swaps the current continuation with its argument.

$$\text{swap-continuation}(\kappa')(\kappa, \sigma) = \text{step}(\kappa)(\kappa', \sigma) \quad (10.32)$$

The interested reader can find an implementation of Scheme’s `call-with-current-continuation` in terms of `swap-continuation` in the appendix.

This was the last of the core functions implementing the interpreter for Λ_3 . All that is missing is some more (trivial) primitive function like `cons`, `pair?`, `*`, `eq?`, and `function(s)` for communicating with the external environment.

10.6 Discussion

So is Λ_3 the language a self-reflective Gödel Machine can be programmed in? It certainly is a suitable core for one. It is simple and small to the extreme (so it is easy to reason about), yet it allows for full functional self-reflection. It is also important to note that we have left nothing unspecified; it is exactly known how programs, functions, and numbers are represented structurally and in memory. Even the number of memory allocations that each primitive function performs is known in advance and predictable. This is important information for self-reasoning systems such as the Gödel Machine. In that sense Λ_3 ’s evaluator solves all the issues involved in self-reflection that we had uncovered while studying the pure λ -calculus-based tower of meta-circular evaluators.

We are currently using a Λ_3 -like language for our ongoing actual implementation of a Gödel Machine. The most important extension that we have made is that, instead of maintaining one continuation, we keep a stack of continuations, reminiscent of the poetically beautiful reflective tower with its summit lost in gray and cloudy skies—except this time with solid foundations and none of those misty marshes! Although space does not permit us to go into the details, this construction allows for an easy and efficient way to perform interleaved computations (including critical sections), as called for by the specification of the Gödel Machine’s *scheduler*.

Acknowledgments

This research was funded in part by Swiss National Science Foundation grant #200020-138219.

Appendix: Details of Notation Used

After parsing and before evaluation, a Λ_3 program has the following structure:

$$\begin{array}{ll} \#f, \text{error} \implies \perp & (\pi \pi') \implies \pi:\pi' \\ \#t, '() \implies \oslash & \text{lambda } \pi \implies (\text{quote2}:\text{lambda2}):\pi \\ x \implies x \text{ for } x \in \mathbb{P} \cup \mathbb{F} \cup \mathbb{N} & \text{quote } \pi \implies (\text{quote2}:\text{cdr}):\pi \end{array}$$

For historical reasons [12], π is a typical variable denoting a program or expression (same thing), ρ is a typical variable denoting an environment, κ is a typical variable denoting a continuation (or ‘call stack’), and σ is a typical variable denoting a storage (or working memory).

A storage $\sigma = \langle \mathbb{S}, \mathbb{N} \rangle$ consists of a fixed-size array \mathbb{S} of 64 bit chunks and a set \mathbb{N} of indices indicating at which positions numbers are stored. All other positions store pairs, i.e., two 32 bit indices to other positions in \mathbb{S} . To look up the value in a storage σ at location p , we use the notation $\sigma[p \rightarrow n]$ or $\sigma[p \rightarrow a:d]$. The former associates n with all 64 bits located at index p , whereas the latter associates a with the least significant 32 bits and d with the most significant 32 bits. We say that the value of p is the number n if $\sigma[p \rightarrow n]$ and $p \in \mathbb{N}_\sigma$ or the pair with head a and tail d if $\sigma[p \rightarrow a:d]$. Whenever we write $\sigma[p \rightarrow a:d]$, we tacitly assume that $p \notin \mathbb{N}_\sigma$.

A new pair is allocated in the storage using the notation $\sigma[p \stackrel{*}{\leftarrow} a:d]$, where a and d are locations already in use and p is a fresh variable pointing to a previously unused location in σ . Similarly, $\sigma[p \stackrel{*}{\leftarrow} n]$ is used to allocate the number n in storage σ , after which it can be referred to using the fresh variable p . Note that we do not specify where in \mathbb{S} new pairs and numbers are allocated, nor how and when unreachable ones are garbage collected. We only assume the existence of a function $free(\sigma)$ indicating how many free places are left (*after* garbage collection). If $free(\sigma) = 0$, allocation returns the error value \perp . Locating a primitive in a storage also returns \perp . These last two facts are formalized respectively as follows:

$$\sigma[\perp \stackrel{*}{\leftarrow} x] \text{ iff } free(\sigma) = 0 \quad (10.33)$$

$$\sigma[c \rightarrow \perp] \text{ for all } c \in \mathbb{P} \quad (10.34)$$

where the set of primitives \mathbb{P} is defined as follows:

$$\mathbb{P} = \mathbb{B} \cup \mathbb{F}, \quad \mathbb{B} = \{\perp, \emptyset\}, \quad \mathbb{F} = \mathbb{F}_1 \cup \mathbb{F}_2, \quad (10.35)$$

$$\mathbb{F}_1 = \{\text{step, eval, if, car, cdr, pair?, number?, swap-continuation}\}, \quad (10.36)$$

$$\mathbb{F}_2 = \{\text{next, lambda2, quote2, cons, set-car!, set-cdr!, eq?, =, +, -, *, /, >\} \quad (10.37)$$

These are the basic primitive functions. More can be added for speed reasons (for common operations) or introspection or other external data (such as amount of free memory in storage, current clock time, IO interaction, etc.). Note that `quote2` is just a dummy binary function, i.e., $\text{quote2}(\varphi, \varepsilon)(\kappa, \sigma) = \text{step}(\perp)(\kappa, \sigma)$.

For convenience we often shorten the allocation and looking up of nested pairs:

$$\sigma[p \stackrel{*}{\leftarrow} a:(ad:dd)] \stackrel{\text{def}}{=} \sigma[p' \stackrel{*}{\leftarrow} ad:dd][p \stackrel{*}{\leftarrow} a:p'] \quad (10.38)$$

$$\sigma[p \stackrel{*}{\leftarrow} (aa:da):d] \stackrel{\text{def}}{=} \sigma[p' \stackrel{*}{\leftarrow} aa:da][p \stackrel{*}{\leftarrow} p':d] \quad (10.39)$$

$$\sigma[p \rightarrow a:(ad:dd)] \stackrel{\text{def}}{=} (\sigma[p \rightarrow a:p'] \text{ and } \sigma[p' \rightarrow ad:dd]) \quad (10.40)$$

$$\sigma[p \rightarrow (aa:da):d] \stackrel{\text{def}}{=} (\sigma[p \rightarrow p':d] \text{ and } \sigma[p' \rightarrow aa:da]) \quad (10.41)$$

Note that the result of an allocation is always the modified storage, whereas the result of a lookup is true or false.

Taking the n th element of list ρ stored in σ is defined as follows.

$$\rho \downarrow_\sigma n = \begin{cases} \varepsilon & \text{if } n = 0 \text{ and } \sigma[\rho \rightarrow \varepsilon:\rho'] \\ \rho' \downarrow_\sigma (n-1) & \text{if } n > 0 \text{ and } \sigma[\rho \rightarrow \varepsilon:\rho'] \\ \perp & \text{otherwise} \end{cases} \quad (10.42)$$

Scheme’s `set!` and `call-with-current-continuation` can be implemented in Λ_3 as follows. They are written in Scheme syntax instead of Λ_3 for clarity, but this is no problem because there is a simple procedure for automatically converting almost any Scheme program to the much simpler Λ_3 .

```

1 (define set!
2   (quote2 (lambda (env-n)
3             (lambda (x)
4               (set-car! (list-tail (car env-n) (cdr env-n)) x))))))

5 (define (call-with-current-continuation f)
6   (get-continuation
7     (lambda (k)
8       (set-continuation (cons f k) (set-continuation k)))))

9 (define (get-continuation f)
10  (swap-continuation (list f)))

11 (define (set-continuation k x)
12  (swap-continuation (cons (lambda (old_k) x) k)))

```

where the functions `cons`, `list`, `car`, `cdr`, `set-car!`, and `list-tail` work as in Scheme [11].

Bibliography

- [1] Harold Abelson, Gerald J. Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, second edition, 1996.
- [2] Alan Bawden. Reification without evaluation. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, 1988.
- [3] Olivier Danvy and Karoline Malmkjær. Intensions and extensions in a reflective tower. In *Lisp and Functional Programming (LFP'88)*, 1988.
- [4] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indagationes Mathematicae*, 34:381–392, 1972.
- [5] Jim des Rivières and Brian Cantwell Smith. The implementation of procedurally reflective languages. In *1984 ACM Symposium on LISP and functional programming*, 1984.
- [6] Daniel P. Friedman and Mitchell Wand. Reification: Reflection without metaphysics. In *Proceedings of ACM Symposium on Lisp and Functional Programming*, 1984.
- [7] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.
- [8] Marcus Hutter. The fastest and shortest algorithm for all well-defined problems. *International Journal of Foundations of Computer Science*, 13(3):431–443, 2002.
- [9] Stanley Jefferson and Daniel P. Friedman. A simple reflective interpreter. *LISP and Symbolic Computation*, 9(2-3):181–202, 1996.
- [10] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: a survey. *Journal of AI research*, 4:237–285, 1996.

- [11] Richard Kelsey, William Clinger, Jonathan Rees, and (eds.). Revised⁵ report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1), August 1998.
- [12] Christian Queinnec. *Lisp in Small Pieces*. Cambridge University Press, 1996.
- [13] T. Schaul and J. Schmidhuber. Metalearning. *Scholarpedia*, 6(5):4650, 2010.
- [14] J. Schmidhuber. Completely self-referential optimal reinforcement learners. In W. Duch, J. Kacprzyk, E. Oja, and S. Zadrozny, editors, *Artificial Neural Networks: Biological Inspirations - ICANN 2005, LNCS 3697*, pages 223–233. Springer-Verlag Berlin Heidelberg, 2005. Plenary talk.
- [15] J. Schmidhuber. Gödel machines: Fully self-referential optimal universal self-improvers. In B. Goertzel and C. Pennachin, editors, *Artificial General Intelligence*, pages 199–226. Springer Verlag, 2006. Variant available as arXiv:cs.LO/0309048.
- [16] J. Schmidhuber. Ultimate cognition à la Gödel. *Cognitive Computation*, 1(2):177–193, 2009.
- [17] Juergen Schmidhuber. Gödel machines: Self-referential universal problem solvers making provably optimal self-improvements. Technical Report IDSIA-19-03, arXiv:cs.LO/0309048 v2, IDSIA, 2003.
- [18] Brian Cantwell Smith. Reflection and semantics in LISP. In *Principles of programming languages (POPL84)*, 1984.
- [19] Bas R. Steunebrink and Jürgen Schmidhuber. A family of Gödel Machine implementations. In *Proceedings of the 4th Conference on Artificial General Intelligence (AGI-11)*. Springer, 2011.
- [20] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In *Proceedings of ACM Symposium on Lisp and Functional Programming*, 1986.