Formalizing Paradoxes in Grounded Arithmetic using Isabelle/HOL

Ananthajit Srikanth
ananthajit.srikanth@epfl.ch
EPFL

Bryan Ford
bryan.ford@epfl.ch
EPFL

May 2026

Abstract

Standard logical foundations in theorem proving constrain the set of recursive functions that are directly expressible to avoid inconsistencies. However, this prevents us from expressing all Turing-complete computations via direct recursive definitions. We consider Grounded Arithmetic, a reasoning framework that avoids inconsistency from unconstrained recursive definitions by "dynamically type-checking" terms. Using the formalization of GA in Isabelle/HOL, we prove three self-referential statements to be nonterminating computation: the Liar Paradox, the Truthteller sentence, and Curry's paradox. To do so, we model each statement in GA as a function taking a certain number of steps of computation, and metalogically derive a contradiction if these terms were to terminate. In turn, this allows us to show that these statements have no concrete value in our framework. Using our non-standard inference rules, terms with no concrete value cannot be used in a proof by contradiction, and thus are inert when reasoning about other computations. We then discuss the accessibility of our approach, and how it permits us to avoid common inconsistencies that would otherwise occur when removing constraints from direct recursive definitions in other formal systems.

1 Introduction

Modern mathematical proofs, handwritten or otherwise, fall into one of two schools of thought: classical or intuitionistic. While both have demonstrated immense usefulness in the formalization of computer systems, they impose strict constraints on recursive definitions. This is, in part, to ensure that the system does not admit any nonterminating recursive definition, which can result in inconsistencies [7, 6, 16]. As a result of these constraints, many functions that we express with relative ease in programming languages outside a formal theorem-proving environment are difficult to express in such systems.

We consider the following example: a function `bad(x) = bad(x) + 1` defined within some formal system. This is a function that calls itself with no base case. If it were admissible as a direct recursive definition in a traditional formal system, it would imply $0 = 1$. Via the principle of explosion, any single inconsistency would render a formal system useless when distinguishing truth from falsehood. Avoiding self-referential – or better yet paradoxical – definitions

has been a long-standing issue when reasoning *within* and *about* formal systems [26, 14].

To expand expressiveness while defusing these inconsistencies, we consider Grounded Deduction (GD), a formal calculus that is based on Kripke's theory of truth [9]. GD relaxes constraints on recursive definitions in its syntax: definitions may be partially-terminating or nonterminating. Its solution to self-referential statements that do not terminate (or more appropriately, *ungrounded* statements) is the *habeas quid* principle: to be able to use some term in a proof, we must show *a priori* that it denotes some well-defined value. As we are interested in reasoning about computation in particular, we focus on Grounded Arithmetic (GA), the embedding of arithmetic within the inference rules of GD.

In this work, we formalize three self-referential statements as non-terminating computation in GA, namely: the Liar Paradox ("this sentence is false"), the Truthteller Sentence ("this sentence is true") and Curry's Paradox ("if this sentence is true, then pigs fly"). As an immediate consequence, they do not satisfy the *habeas quid* principle. Due to GA's non-classical inference rules, we cannot use these statements to derive anything. Hence, these sentences are effectively harmless when reasoning about other computations.

To prove these computations nonterminating, we devise a general approach for nontermination within the GA framework. Reasoning metalogically, we assume an environment where we can "work backwards" through our computation. Prior work has established an operational semantics for GA, from which we derive *inversion lemmas*, allowing us to reason backwards from more- to less-evaluated terms. The main insight is as follows: for some term to have a well-defined value, it would need to reduce within a certain number of "steps" of computation. If self-referential sentences such as the Liar were to denote some well-defined value, they must reduce in a certain number of steps. By making certain reasonable assumptions about our operational semantics (and the corresponding inference rules of GA), we can derive a contradiction and show that paradoxical statements do not denote a well-defined value.

In our formulation, we make extensive use of Isabelle's locale system, as well as the Isar language for structured proofs. These locales allow us to reason about function application, negation, and implication, the key component for each of the three paradoxes. For the Truthteller sentence, this will be because any function application has to reduce to its body, and this body should reduce in fewer steps. In other words, function application takes 'steps' to reduce. Similarly, negation also takes 'steps' to reduce. To ensure that the negation of a sentence $\neg s$ reduces in some step count, the original sentence s should take strictly fewer steps.

We will see that implication is more involved, as we assume strong Kripke semantics for the propositional operators. If the sentence $a \rightarrow b$ denotes some well-defined value, we can only infer that at least one of a or b is well-defined. Nevertheless, we can defuse Curry's paradox using additional case analysis.

In all three cases, we reinforce two key ideas. The first is that we illustrate how GA exhibits resistance to certain classes of self-referential (ungrounded) recursive definitions that express nonterminating computations. We do this by

showing that the said ungrounded sentences have no useful value (i.e., they denote $\perp$ in our semantics).

The second, and more important takeaway, is that GA, as the logical foundation for future proof assistants, can use a more direct (and potentially simpler), program-semantics-based approach to reasoning over a broader class of programs. Put simply, we aim to investigate GA's potential as a reasoning framework that is more in line with intuitive notions of computation i.e., the "Fregian paradise of 'type-free' functions" that Church, Curry and Scott were after [26].

We also consider GA's potential with respect to defusing more complex paradoxes. To begin with, we have Yablo's paradox, whose nontermination relies on the semantics of universal quantification, rather than negation or implication. We discuss our representation of universal quantifiers as computation and how we may show Yablo's paradox to be nonterminating. Furthermore, GA can be used to reflect on computation within itself [2]. With this in mind, we discuss reflection as a general method to resolve paradoxes such as those of Quine or Berry.

We begin with a background and motivation (Section 2.1), an overview of GD and GA (Section 2.2), and then discuss the formalization of our 3 main paradoxes (Sections 3, 4, and 5). At the beginning of each section, a motivating example of how unconstrained recursion can result in each paradox is introduced. We then discuss future work in Section 6, and Section 7 concludes.

2 Background

This section first motivates GA and our study of paradoxes. We then briefly outline the syntax and properties of GA, as well as its quantifier-free fragment, Basic GA (BGA).

2.1 Motivation

In the construction of most proofs, computer-checked or otherwise, we traditionally tend to restrict ourselves to two classes of logics. The first of these is classical, i.e., logics that assume the law of excluded middle as an axiom. Isabelle/HOL, among others such as HOL family of provers, or E.[1] can be considered to be part of this class [22, 10, 25] While these provers differ in their underlying implementation details, their assumption of the excluded middle remains invariant. The main benefit of some of these provers is the ease with which certain fragments of logic can be automated.

The second class is intuitionistic: provers such as Agda, Rocq and Lean that do not admit the excluded middle[2], and rely on dependent type theories instead [13, 21, 3]. These logics are appealing due to their use of the proofs-as-programs correspondence, with inference rules corresponding to typing rules.

[1] this is by no means an exhaustive list.

[2] without axiomatization

While both schools of thought have allowed for notable progress in the formalization of systems , modern theorem provers using these logics impose constraints on the expressivity of recursive functions[11, 18]. These constraints come at the cost of Turing-completeness, with respect to the set of recursive functions that are directly admissible.

When we do not impose these constraints in a formal system, we can express a greater number of "well-behaved" functions with relative ease, closer to those seen in day-to-day programming outside a prover. While it is possible to introduce self-referential definitions that are non-terminating, these serve as merely curiosities to be debugged in day-to-day programming. Yet, these definitions can easily produce inconsistencies in standard formal systems. Prior attempts at designing formal systems for computational reasoning have encountered the same issue, especially with Church and Curry's original formulation of the untyped λ-calculus [6, 7, 16].

For example, we may want to define mutually recursive functions that call each other across different modules. In most theorem provers, mutually recursive functions (and dually, mutually inductive datatypes) are tightly coupled. Usually, formal systems require that these kinds functions be defined in the same module, primarily due to syntactic constraints ensuring consistency and termination. We may also consider infinite sequences of computation that are not meant to terminate, such as streams. While most formulations of streams rely on the theories of co-induction and co-recursion, the class of functions that can be defined to operate on co-data are even more restrictive [1, 5].

A well-known approach for handling paradoxical definitions is paracomplete logics, which do not admit the excluded middle. While intuitionistic logic is paracomplete, it is not immune to the inconsistencies introduced by unconstrained recursive definitions (see Section 5). Unlike intuitionistic logic, however, many paracomplete logics are 3- or many-valued [8]. However, these logics are usually too weak for reasoning about computation, and the semantics of multiple truth values remains somewhat unclear, if not unintuitive. Nevertheless, the notion of paracompleteness yields a promising impetus to handle nonterminating computation. Thus, Grounded Deduction was devised to address the issues concerning paracompleteness.

Grounded Deduction (GD) is a first-order predicate calculus bearing similarity to many of the inference rules of NJ and NK [20], albeit with certain "weakenings" [9]. Given that any weakening of inference rules is usually a cause for concern, we justify our decisions below with a motivating example, namely the inference rule for contradiction.

The key modification that GD attaches, as introduced in Section 1 is the *habeas quid* principle:

We must have a "thing" before we use it.

The "thing" in question informally corresponds to a "dynamic type-check" for an inference rule. Consider the example of proof by contradiction in GD:

$$\frac{a\ \mathbf{B} \qquad \overset{[a]}{\underset{c}{\vdots}} \qquad \overset{[a]}{\underset{\neg c}{\vdots}}}{\neg a}$$

Similar to classical logic, if a hypothetically proves some arbitrary statement c, but a also proves $\neg c$, we have a contradiction, and therefore a must be false. The main difference is the *habeas quid* precondition: a must be shown to denote some boolean value before invoking this rule.

This rule states that "we must show a to be boolean" before we can use it. For most contradiction proofs, however, we do have a concrete 'boolean' term from which we can derive a contradiction. Similarly, GD adds *habeas quid* rules for implication, and universal quantification, which will be discussed in Sections 5 and 6.1 respectively.

As an example, habeas quid can be used to defuse the inconsistencies that Liar sentence would create in a classical environment, were it directly admissible. Define the Liar as $L \equiv \neg L$. If we were to use L to derive a contradiction, we would need to show that is a "thing" that denotes a well-typed boolean ($L\ \mathbf{B}$). In the case of the Liar L, to show that L is boolean, we would have to show that its definition $\neg L$ is boolean, but the only way to show $\neg L$ is boolean is to show L is boolean, resulting in a circular proof obligation.

Since our main goal is to reason about computation, we motivate our investigation by restricting our domains of discourse to the natural numbers. Hence, we define grounded arithmetic (GA), a system that allows us to reason about computation over natural numbers. Given that natural numbers are, to some extent, an arbitrary choice for models of computation, we can consider GA-as-a-logic, looking to Peano and Heyting arithmetic as inspiration [12]. From a computational perspective, we observe that GA's expressivity is on par with LCF, PCF and PPλ [19, 24, 23].

Having now introduced GA as a system of computation and reasoning, we return to our discussion of the Liar. It may be more useful (and relevant) to think of the Liar sentence in terms of nontermination. As with the example in Section 1, consider L not as a statement, but as a function. Supposing L does terminate, then so does $\neg L$, $\neg\neg L$, ... and so on.[3] It can be argued in an intuitive fashion that this is not the case, since we are unfolding a recursive definition infinitely with no "base case" (we will see this proof structure and its formalization in more detail starting in Section 3).

GA and GD would not be particularly useful if it could only show terms to be nonterminating. We consider the following simple example of a function that checks if a natural number is even:

$$\text{even}(a) \equiv a\ \mathbf{0}?\ \ \mathbf{T} : \mathbf{P}(a)\ \mathbf{0}?\ \ \mathbf{F} : \text{even}(\mathbf{P}(\mathbf{P}(a)))$$

Theorem 2.1. *Given that a* $\mathbf{N}$*,* `even`*(a) denotes a well-typed boolean value.*

Proof. By induction on n. □

[3] We may also observe that this is a recursive definition whose least fix-point is $\perp$

It is possible to make such a statement in GA, since it contains a first-order induction schema, with *habeas quid* ensuring that any n for which we say n is even must be shown *a priori* to be a natural number. We can consider this a "trivial typecheck", although the full scope of the overhead this incurs is yet to be investigated. Nevertheless, work by Kehrli on GD as a usable formal system has shown that such habeas quid conditions can be easily discharged [15]. However, given our focus on nontermination, we will not pursue this detail in this paper.

2.2 The Syntax and Properties of GA

GA is the more "general" system we plan to work with in the long run. There is ongoing work formalizing different fragments of GA. For the sake of simplicity, we focus on the quantifier-free fragment of GA, Basic GA (BGA). This is because the current formalization of BGA's metatheory is the most well-developed so far.

In the syntax of GA. we distinguish terms (denoted by the metavariable t) and formulae (denoted by the metavariable f):

$$t \equiv v_i \mid \mathbf{0} \mid \mathbf{S}(t) \mid \mathbf{P}(t) \mid t\ \mathbf{0}?\ t : t \mid d(t,t)$$
$$f \equiv t = t \mid t \neq t$$

The terms represent arbitrary variables, zero, successor, predecessor, if-zero-else and two argument function application, while the two formulas check for (dis)equality across terms. Our current formulation of BGA's semantics takes an operational approach to reduce terms. Nevertheless, GA was heavily inspired by Scott-style denotational semantics.

In the formulation of BGA, the set of functions that we can define are two-argument μ-recursive functions. Contrast this to GA, which allows for arbitrary n-ary function definitions.[4] This includes functions of arity 0 - the most direct formulation of our paradoxes. In either case, function definitions in (B)GA permit direct unconstrained recursion in their syntax.

We now discuss the main metalogical properties of BGA. To begin with, prior work has shown that BGA is truth-preserving, i.e., all inference rules preserve truth) and semantically complete (any well formed formula that is true is provable in BGA) [2]. BGA is not syntactically complete, as this is the key feature of paracompleteness: there are formulae φ such that neither φ nor $\neg\varphi$ are provable. This corresponds to the existence of computations that never terminate, e.g., paradoxes.

The key difference between BGA and other similar systems is in its ease of expressivity. We can define Turing-complete computations directly using recursive definitions, while avoiding issues that stem from nonterminating functions. In that sense, BGA is similar in expressivity to PCF or PPλ, mainly differing

[4]This design decision primarily stems from not wanting to deal with arbitrary arities for the sake of simplicity. This is because any n-ary function can be encoded and decoded using a 2-ary function, via Cantor pairs.

in its lack of types and higher-order functions [23, 19]. One key difference in our formulation, however, is that we assume the existence of an arbitrary list of recursive function definitions, forming an "environment" of fixed length.[5] This definition list can be seen as a "standard library" of functions to be specified by a user when "instantiating" BGA.

Having discussed the main metalogical properties of BGA, we next show how we can formalize nonterminating statements as violations of *habeas quid*, starting with the Truthteller Sentence. We delay our discussion of its more famous relative, the Liar Paradox, to introduce some tooling that will be necessary for both statements.

3 The Truthteller Sentence

We motivate our formalization with a simple example, the Truthteller sentence, i.e., "this sentence is true" [17]. While this sentence seems non-paradoxical, it is definitely *ungrounded*, in Kripke's terminology. We will discuss ungroundedness soon that corresponds strongly with the notion of non-termination.

We consider the following example: given a theorem prover with unconstrained recursion, one could define the following function:
`looper(x, y) := looper(x, y)`.[6] This is the same as an unchanging process that never terminates. If such a nonterminating statement were directly admissible via recursive definitions, it may be considered an underspecified function. For some standard theorem provers, however, nonterminating functions can be used to derive a contradiction [4, 5].

The Truthteller has no value that we can denote to it: as Kripke describes it, the sentence is a "process" that does not terminate, and is therefore considered *ungrounded*. This means it cannot yield any value, let one one of truth or falsehood. This notion is embedded into GA as computation. Any function that terminates with a boolean value can have its output be denoted with a corresponding value for truth or falsehood, while nonterminating functions are $\perp$, or ungrounded.

Due to BGA's technical constraint of supporting only 2-argument functions, a more precise formulation of the Truthteller would be $T_t(v_0, v_1) \equiv T_t(v_0, 0)$. We include two input arguments as all BGA functions are 2-ary. The arguments themselves are not of particular relevance for this paradox, since any function call to T_t resolves to a never-ending chain of function calls to T_t that never use the inputs passed in. When formulated in GA (and not just BGA), we can define the paradox more cleanly, as a 0-argument function.

To analyze the paradox formally (and prove it nonterminating), we require some tooling to begin with. Given an environment expressive enough to handle GA assumptions, we define the predicate `evi`: `evi A t s` means that an arbitrary GA term t reduces to some value under assignment A within a certain

[5] The full details of definition lists are discussed in [2, Section 4.3].

[6] The arguments are mainly for consistency with other examples in BGA and have no significant bearing on the meaning of the sentence.

number of computation steps s. Note that `evi` is type-agnostic.

Returning to the denotational semantics of GA,[7] we have that the denotation t is not equivalent to $\bot_\tau$ in the corresponding Scott domain $D(\tau)$. In BGA, `evi` represents the most general form of *habeas quid*: we simply "have something", with no regard to its type.

Since `evi` is step-dependent, it consists of two main conditions: no term reduces in 0 steps, and a term that has reduced in n steps stays reduced for any greater number of steps (i.e., `evi` is monotone).

locale model_evi =
  fixes evi :: "asn ⇒ trm ⇒ nat ⇒ bool"
  assumes evi_0: "¬evi A t 0"
    and evi_mono: "evi A t i ⟹ i ≤ j ⟹ evi A t j"

Since the Truthteller sentence is a function that takes in two arguments, we also need to describe the semantics of function application. We can base this on the operational semantics of BGA, as described in prior work [2]. We consider some function $\mathbf{d}_i(a_0, a_1) \equiv D\langle n_0, n_1\rangle$, where $\mathbf{d}_i$ is the function name and D is its body (with arguments appropriately substituted). We have the following forward-evaluation rule in BGA's semantics, which allows us to reason from the evaluation of the function's arguments and body to the ultimate evaluation of the function call:

$$\frac{D\langle n_0, n_1\rangle \Downarrow m \qquad a_0 \Downarrow n_0 \qquad a_1 \Downarrow n_1}{\mathbf{d}_i(a_0, a_1) \Downarrow m}$$

However, to use `evi` to prove functions non-terminating, we need to reason "backwards" through our computation. So, we need to formulate an inversion lemma. Assuming that the function call $\mathbf{d}_i(a_0, a_1)$ reduces to a value for some assignment A, and the definition of $\mathbf{d}_i$, it must be the case that the function body D reduces to a value for the specific assignment $[a_0 \mapsto n_0, a_1 \mapsto n_1]$ *in strictly fewer steps*. One may ask why the value itself is considered irrelevant as far as `evi` is concerned. Since our primary interest is nontermination, it is usually enough to derive a contradiction from the hypothetical premise of our term having some value; we do not need to know what this value is. We then obtain this locale, encoding our assumptions:

locale model_evi_app2 = model_nat + trm_dfns +
trm_app2 + trm_nat_cons + model_boolin +
assumes evi_app2 :
 "dfn_is df 2 t ⟹
   evi A (df[**n**x, **n**y]) s ⟹ ∃s' < s. (evi (asn2 (natv x) (natv y)) t s') "

[7] A detailed discussion of this is beyond the scope of this paper, and may be found in [9, Section 8].

In this locale, **n**x, **n**y are natural number terms in GA, with (natv x), (natv y) denoting the GA numbers' respective values as natural numbers in HOL. df is a 2-argument function, with dfn_is defining df to have body t. In general, for each `evi` locale, we have an assumption that shows that any term that evaluates in s steps must have one or more subterms that evaluate in s' < s steps.

These two facts are sufficient to demonstrate the nontermination of the Truthteller in GA. We write out a proof sketch for its nontermination as follows:

Lemma 3.1. *T_t is nonterminating in GA, i.e., $\neg$ `evi A` $T_t(v_0, v_1)$ `s`*

Proof. We prove this by (strong) induction on the step count. The base case is trivial as nothing terminates in 0 steps.

For our inductive case, assume that some arbitrary $x \in \mathbb{N}$, `A` an assignment, and v_0, v_1 **N** such that $\forall y < x$, $\neg$ `evi A` $T_t(v_0, v_1)$ y. We want to show $\neg$ `evi A` $T_t(v_0, v_1)$ `x` , that is, $T_t(v_0, v_1)$ does not evaluate to anything in x steps. Assume, for the sake of contradiction, that it does evaluate in x steps. Then, by inversion, consider the $x' < x$ such that `evi` $[v_0 \mapsto v, v_1 \mapsto 0]$ $T_t(v_0, v_1)$ `x'` for some v. This contradicts our induction hypothesis, and therefore T_t cannot reduce to *any* value. □

Theorem 3.2. *T_t has no boolean value, i.e., it is ungrounded in GA.*

Proof. By Lemma 3.1, we have that T_t denotes no value. It follows trivially that T_t cannot denote any boolean value either. □

We observe that the proof of the Truthteller is quite simple, "reversing" computation to construct a term we know to not terminate in strictly fewer steps. In doing this, GA defuses programs such as `looper := looper`. This brings us back to the two key ideas referenced in Section 1. We have shown how we can prove ungrounded self-referential statements in BGA to be nonterminating computation. Given the proof of BGA's consistency, we can demonstrate how ungrounded statements cannot be proven true or false in BGA [2]. It follows that ungrounded sentences are harmless, rather than leading to contradictions as they would in classical logic. The second idea is that our nontermination proof can be shown using only a few rules from our operational semantics. The Isabelle/HOL formalization can be found in the Appendix 7.

Having proven that the Truthteller is nonterminating (and hence ungrounded), we now shift our focus to the Liar Paradox.

4 The Liar Paradox

Consider the following sentence: "This sentence is false". By naive classical reasoning, the Liar is false if and only if it is true. When viewed as computation, we can think of the Liar paradox as a function that constantly alternates between true and false for each step of computation. Returning to our example from Section 3, we can consider a similar statement for the Liar:

`bad(x,y) = 1 - bad(x,y)`. If admitted, $0 = 1$ can be proven from this function. Using the principle of explosion in a classical setting, this function would allow us to prove anything.

The proof of the Liar's ungroundedness is similar to that of the Truthteller, so we will not repeat our proof approach for the sake of brevity.

Nevertheless, our current formulation is insufficient to represent the Liar. Notice that the Liar would be the following sentence $L \equiv \neg L$ – or as a function, $L(v_0, v_1) \equiv \neg L(v_0, v_1)$ – yet we have not defined any assumptions to reason about the nontermination of statements containing negation.

We can consider two approaches to handling negation: one by considering the operational semantics of GA, and the other by considering its typing rules, both of which converge to the same idea.

For negation, we have the following inference rule:

$$\frac{a\ \mathbf{B}}{(\neg a)\ \mathbf{B}}$$

That is, if a is boolean, so is $\neg a$. Similarly, if we know that $(\neg a)$ reduces to a well-defined value, then a must have reduced in strictly fewer steps. This is shown in the following locale:

locale model_evi_neg = model_boolin + model_neg +
  assumes evi_neg :
    "evi A (¬t) s ⟹ ∃s' < s. evi A t s' "

Theorem 4.1. *$L \equiv \neg L$ has no boolean value, it is ungrounded in GA.*

The proof is similar, except we have to find two smaller step counts: one for function application and one for negation.

Returning to our big picture, since we have that the liar sentence L is nonterminating, it has no value. Thus, it cannot be used in a proof by contradiction. We have shown how GA can defuse this paradox with ease using its operational semantics. The Liar can alternatively be considered a function of the form $f(x, y) = 1 - f(x, y)$. This suggests that functions with oscillatory behavior to the Liar can be approached similarly and shown to be ungrounded. Hence, they cannot be used to prove the truth or falsehood of any arbitrary term.

5 Curry's Paradox

Curry's paradox informally states the following: "If this sentence is true, then pigs fly". We can define it as follows: $C \equiv C \to P$, for an arbitrary P [27, 8]. Importantly, the structure of P is irrelevant to the formulation of the paradox.

For a more concrete example, we divert our focus from unconstrained recursive functions, to unconstrained inductive types.

In a traditional proof environment, Curry's paradox shows up if we allow unconstrained recursive definitions in type constructors, such as a naive formulation of the untyped λ-calculus. A simple example would be an inductive constructor that violates Isabelle's positivity conditions on recursive occurrences of function types:

datatype bad = e | badcons "bad $\Longrightarrow$ nat"

If such a type constructor could be defined and admitted, we could construct nonterminating functions similar to the Ω-combinator [5].

Considering the semantics of BGA, Implication is defined in terms of disjunction, so $a \to b \equiv \neg a \vee b$ [2]. Having already discussed negation, we primarily focus on disjunction, since the `evi` predicate between $a \to b$ and $a \vee b$ are similar.

We have the following four typing rules for GA:

$$\frac{a\ \mathbf{B}}{(a \vee b)\ \mathbf{B}} \qquad \frac{b\ \mathbf{B}}{(a \vee b)\ \mathbf{B}}$$

$$\frac{a\ \mathbf{B}}{(a \to b)\ \mathbf{B}} \qquad \frac{b\ \mathbf{B}}{(a \to b)\ \mathbf{B}}$$

The inversion rule says that if we have $a \vee b\ \mathbf{B}$, then at least one of a or b has a boolean value, so it is necessary to consider both cases.

locale model_evi_imp = model_boolin + trm_imp +
assumes evi_imp :
"evi A (a $\longrightarrow$ b) s $\Longrightarrow$ $\exists$ s' < s. (evi A a s') $\vee$ (evi A b s') "

locale model_evi_or = model_boolin + trm_or +
assumes evi_or :
"evi A (a $\vee$ b) s $\Longrightarrow$ $\exists$ s' < s. (evi A a s') $\vee$ (evi A b s') "

Much like Kleene's strong 3-valued logic, where we consider "unknown" rather than "ungrounded", if $t \equiv a \vee b$ is known to be true or false, then we only know that at least one of a or b is boolean. If the disjunct t is true, we only know that at least one of a or b is true. Otherwise, when t is false, both a and b must be false. The rules for the conditional are derived in a similar fashion, since we do not use Łukasiewicz's non-standard implication operator [27].

The proof for Curry's paradox also follows quite similarly, with one minor difference: we must perform case analysis to consider whether it is the antecedent or the consequent that is "well-typed". Since this is notably different from our previous two paradoxes, we discuss the proof in more detail. As with the Truthteller, we represent the paradox $C(v_0, v_1) = C(v_0, 0) \to P$ for all P.

Lemma 5.1. *C does not evaluate to any value:* $\neg$ `evi A` $C(v_0, v_1)$ `s`

Proof. By (strong) induction on the step count. The base case is trivial as nothing terminates in 0 steps.

For our inductive case, assume that some arbitrary $x \in \mathbb{N}$, `A` an assignment and v_0, v_1 $\mathbf{N}$, $\forall y < x$, $\neg$ `evi A` $C(v_0, v_1)$ `y`. We want to show $\neg$ `evi A` $C(v_0, v_1)$ `x`, that is, $C(v_0, v_1)$ does not evaluate in x steps.

Suppose, for the sake of contradiction that $C(v_0, v_1)$ evaluates in x steps. Then, by inversion, consider the $x' < x$ such that `evi` $[v_0 \mapsto v, v_1 \mapsto 0]$ `(`$C(v_0, v_1) \to P$`) x'` for some v.

There are two cases: at some step count $x'' < x'$, we have `evi` $[v_0 \mapsto v, v_1 \mapsto 0]$ $C(v_0, v_1)$ `x''`, which is false by the induction hypothesis, or `evi` $[v_0 \mapsto v, v_1 \mapsto 0]$ `P x''`. Since P can be any arbitrary statement, we have that any statement must evaluate to some value. Given that there are statements that do not evaluate to any value, we have a contradiction.

In either case, C does not reduce to *any* value. □

Theorem 5.2. *C has no boolean value.*

Proof. By Lemma 5.1, we have that C denotes no value. It follows trivially that C cannot denote any boolean value either. □

Returning to our constructor example at the start of this section, we reinforce our two key ideas. Given our demonstration of the ungroundedness of Curry's paradox, we expect to find that non-positive "ill-defined" types (and functions that we can admit from said types) can also be defused in GA with ease, using the same technique demonstrated in this section.

Notwithstanding the minor challenge from our inversion lemmas, we find the mitigation of Curry's paradox to have a similar and simple structure. Relying directly on the operational semantics of GA, we can demonstrate the Curry sentence's ungroundedness.

Having discussed our formalization of these paradoxes, we now discuss future work.

6 Other Paradoxes

This section first discusses Yablo's paradox, whose nontermination proof requires reasoning about the semantics of universal quantification. We then consider paradoxes involving reflective computation, where GA reasons about itself. Furthermore, we consider how these reflective paradoxes may be formalized.

6.1 Yablo's Paradox

Yablo's paradox is the following infinite set of sentences [28]:

$$\begin{aligned} (Y0) &\quad \text{For all } k > 0, Y_k \text{ is not true} \\ (Y1) &\quad \text{For all } k > 1, Y_k \text{ is not true} \\ &\quad \ldots \end{aligned}$$

We note a minor difference in formulation, albeit with no impact on our un-groundedness analysis: Yablo originally defined k to begin from 1 instead of 0.

We can write the definition for an arbitrary Y_i as follows:

$$Y(i) = \forall k\ \mathbf{N}, k > i \to \neg Y(k)$$

Since Yablo's Paradox uses universal quantification, we assume a fragment of GA that contains quantifiers as part of its syntax. We will need to formulate an inversion lemma for universal quantification. For GA, to show that some universally quantified formula holds, we must either be able to prove it true by an induction principle, or find a counterexample demonstrating that the predicate does not hold for all natural numbers, as seen from the two inference rules for universal quantification.

$$\frac{p\langle 0,...\rangle \qquad \dfrac{[x\ \mathbf{N}, p\langle x,...\rangle]}{p\langle S(x),...\rangle}}{\forall x\ \mathbf{N} p\langle x,...\rangle} \qquad \frac{a\ \mathbf{N} \qquad \neg p\langle a,...\rangle}{\neg\forall x\ \mathbf{N} p\langle x,...\rangle}$$

Our first observation is that if $\forall x\ \mathbf{N} p\langle x,...\rangle$ denotes some boolean value, we must have metalogically that for some x, $p\langle x,...\rangle$ reduces to a boolean value in strictly fewer steps.

With respect to BGA, we define universal quantification in terms of a "reflective proof-search" (see also section 6.2). This proof search tries to check whether or not the base and inductive cases can be shown to be true. This is done by finding a natural number corresponding to the Gödel code of a GA proof. Should the universally quantified statement be false, a counterexample must have been generated, if one can be found.

In the context of Yablo's paradox, we will use our inversion lemmas to show that neither the proof search can terminate, nor can finding a counterexample. The case for the counterexample is straightforward: since we must show that the inner term of Yablo's paradox is false, the implication must be false for some term a. From our implication rules, it follows that the antecedent $k > i$ and consequent $\neg Y(k)$ must reduce in fewer steps, a clear contradiction. The positive case, however, is more challenging: a naïve rule that does not take proof search into account will not suffice. If $Y(k)$ were to reduce, then the search procedure must reduce in strictly fewer steps. Since the Yablo proof term is quantified over natural numbers, we can assume that this proof was constructed by induction. It follows that for some proof-checking function $C(Pr, Co)$,[8] C must reduce in strictly fewer steps for both the base and inductive cases. We aim to use and investigate these observations to prove Yablo's paradox $Y(i)$ as nonterminating computation for any i.

6.2 Paradoxes of reflection

Quine's paradox is the statement:

> "Yields falsehood when preceded by its own quotation"
> yields falsehood when preceded by its own quotation.

Paradoxes such as those of Quine, Berry, and Grelling [8] require one additional component of BGA tooling: reflection. BGA already has a `coding` typeclass, which allows us to construct an arithmetization for arbitrary HOL types, provided we prove this encoding injective [2]. We will then use this coding to produce an "embedding" of BGA within itself, allowing us to (metalogically) discuss BGA's ability to prove statements about itself.

[8] With the two arguments of C representing premises and a conclusion, respectively.

With this tooling, we have also constructed a locale that allows us to model the proof search of BGA sequents within BGA. Proof search is modeled using a primitive-recursive step function: it terminates with a value whenever a term can be proven or refuted in BGA given `s` steps of fuel, and does not terminate otherwise. It should suffice to show that no amount of "fuel" provided to such a step function will result in the proof search terminating, and thus each of these paradoxes are ungrounded.

7 Conclusion

In this paper, we devise and explain formulations of locales within Isabelle/HOL to represent assumptions over the operational semantics of GA. More specifically, we model the semantics of function application, negation and implication in terms of step counts. Using this, we can show that ungrounded, but unconstrained direct recursive definitions can be "defused" with relative ease.

We have multiple directions for future work. We would like to continue in our paradox formalization and assess the extent to which rule inversion lemmas can facilitate our reasoning. Additionally, we would like to see if we can automate these classes of inversions.

Prior work has shown that GA and GD may be powerful enough to be used for reasoning about traditional computation [15]. We aim to continue assessing GD's ease-of-use as a logical framework, while also testing its resistance to paradoxes metalogically.

This current work acts as a "springboard", suggesting that with some automation of nontermination proofs, we can easily defuse certain inconsistencies within GA and BGA, while continuing to possess the tooling for unconstrained recursion.

We are also interested in investigating systems with non-local, mutual recursion and nested recursions that may otherwise be challenging to express via direct recursion. Finally, we hypothesize that reasoning about and writing proofs over codata and coinductive types using the *habeas quid* principle may be easier, e.g., formalizing streams without the need for friend functions as discussed in Blanchette *et al*'s work [1].

References

[1] Jasmin Christian Blanchette et al. "Defining Nonprimitively (Co) recursive Functions in Isabelle/HOL". In: *URL: https://isabelle. in. tum. de/dist/Isabelle2021/doc/corec. pdf* (2019).

[2] Elliot Bobrow, Bryan Ford, and Stefan Milenkovič. *Have a Thing? Reasoning around Recursion with Dynamic Typing in Grounded Arithmetic*. Jan. 24, 2026. DOI: 10.48550/arXiv.2510.25369. arXiv: 2510.25369 [cs]. URL: http://arxiv.org/abs/2510.25369 (visited on 05/06/2026). Pre-published.

[3] Ana Bove, Peter Dybjer, and Ulf Norell. "A Brief Overview of Agda - A Functional Language with Dependent Types". In: *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*. Munich, Germany: Springer-Verlag, 2009, pp. 73–78.

[4] Joachim Breitner. *Isabelle functions: Always total, sometimes undefined - Joachim Breitner's Homepage*. 2017. URL: https://www.joachim-breitner.de/blog/732-Isabelle_functions__Always_total,_sometimes_undefined#fnref4 (visited on 06/16/2026).

[5] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, Dec. 2013. 424 pp. ISBN: 978-0262026659.

[6] Alonzo Church. "A Set of Postulates for the Foundation of Logic". In: *Annals of Mathematics* 33.2 (Apr. 1932), pp. 346–366. DOI: 10.2307/1968337.

[7] H. B. Curry. "Grundlagen der kombinatorischen Logik". In: *American Journal of Mathematics* 52.3 (July 1930), pp. 509–536.

[8] Hartry Field. *Saving Truth From Paradox*. Oxford University Press, May 2008. 424 pp. ISBN: 978-0199230747.

[9] Bryan Ford. *Reasoning Around Paradox with Grounded Deduction*. arXiv preprint 2409.08243. Sept. 2024. URL: https://arxiv.org/abs/2409.08243.

[10] M. J. C. Gordon and T. F. Melham, eds. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, June 1993. 492 pp. ISBN: 978-0521441896.

[11] Gernot Heiser. "seL4: Operating Systems With the Reliability of Mathematics". In: *IEEE Reliability Magazine* (2026).

[12] Arend Heyting. *Intuitionism: An introduction*. Studies in logic and the foundations of mathematics. North-Holland, Jan. 1971. 147 pp. ISBN: 978-0720422399.

[13] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. *The Coq Proof Assistant: A Tutorial*. Version 8.5pl2. Apr. 2016.

[14] Antonius J. C. Hurkens. "A simplification of Girard's paradox". In: *International Conference on Typed Lambda Calculi and Applications*. Springer Berlin Heidelberg, 1995, pp. 266–278.

[15] Sascha Kehrli. "Formalizing Grounded Arithmetic atop Isabelle/Pure". ETH Zürich, Sept. 2025. URL: https://bford.info/thesis/2025-kehrli/.

[16] S. C. Kleene and J. B. Rosser. "The Inconsistency of Certain Formal Logics". In: *Annals of Mathematics* 36.3 (July 1935), pp. 630–636. DOI: 10.2307/1968646.

[17] Saul Kripke. "Outline of a Theory of Truth". In: *The Journal of Philosophy* 72.19 (Nov. 1975), pp. 690–716. DOI: 10.2307/2024634.

[18] Xavier Leroy et al. "CompCert – A Formally Verified Optimizing Compiler". In: *Embedded Real Time Software and Systems (ERTS)*. Toulouse, France, Jan. 2016.

[19] Robin Milner. *Models of LCF*. Tech. rep. Mathematical Centre Tracts. University of Edinburgh, 1976, pp. 49–63.

[20] Samuel Mimram. *Program = Proof*. 2020. ISBN: 979-8-6155-9183-9.

[21] Leonardo de Moura and Sebastian Ullrich. "The Lean 4 theorem prover and programming language". In: *28th International Conference on Automated Deduction (CADE)*. July 2021, pp. 625–635.

[22] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, 2283. Springer, Apr. 2002. 240 pp. ISBN: 978-3540433767.

[23] Lawrence C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge Tracts in Theoretical Computer Science 2. Cambridge ; New York: Cambridge University Press, 1987. ISBN: 978-0-521-34632-0.

[24] G.D. Plotkin. "LCF Considered as a Programming Language". In: *Theoretical Computer Science* 5 (1977), pp. 223–255.

[25] Stephan Schulz, Simon Cruanes, and Petar Vukmirović. "Faster, Higher, Stronger: E 2.3". In: *Proc. of the 27th CADE, Natal, Brasil*. Ed. by Pascal Fontaine. LNAI 11716. Springer, 2019, pp. 495–507.

[26] Dana Scott. "Combinators and classes". In: *-Calculus and Computer Science Theory*. Ed. by C. Böhm. Berlin, Heidelberg: Springer, 1975, pp. 1–26. ISBN: 978-3-540-37944-7. DOI: 10.1007/BFb0029517.

[27] Peter Smith. "Curry's Paradox, Lukasiewicz, and Field". manuscript.

[28] Stephen Yablo. "Paradox without Self-Reference". In: *Analysis* 53.4 (Oct. 1993), pp. 251–252.

Appendices

A Appendix A: Formalization of the Truthteller Sentence

```
locale truthteller_ga =
      model_evi_app2 +  model_evi_eq +
      fixes tt :: dfn
      assumes tt_def : "dfn_is tt 2 (tt[nx, 0])"
    begin

    lemma ttp : "¬ evi A (tt[nx, 0]) s"
    proof (induction s arbitrary: A x rule: less_induct)
      fix s' A x
      show "(⋀y A x. y < s' ⟹ ¬ evi A (tt[nx, 0]) y) ⟹
          ¬ evi A (tt[nx, 0]) s'  "
      proof (cases s')
```

```
    case 0
    then show ?thesis using evi_0 by auto
  next
    case sdef:(Suc nat)
    assume ih:"(⋀y A x. y < s' "
    show "¬ evi A (tt[nx, 0]) s' "
    proof
      assume "evi A (tt[nx, 0]) s' "
      then obtain s' "
        by (metis Ex_less_Suc trm_num_0 sdef evi_app2_suc tt_def)
      then show "False" using ih by auto
    qed
  qed
qed
```

The Truthteller paradox cannot be represented with a boolean value.

```
theorem tt_paradox: "¬ tbool A (tt[nx, 0])"
using ttp tbool_ex by blast
end
```

B Appendix B: Formalization of the Liar Paradox

```
locale liar_ga =
    model_evi_neg + model_evi_app2 +
    fixes liar :: dfn
    assumes liar_def : " dfn_is liar 2 (¬(liar[nx, 0]))"
  begin
```

The liar paradox does not evaluate to any well-defined value

```
  theorem liar_paradox_simple : "¬ (evi A (liar[nx, 0]) s)"
  proof (induction s arbitrary: A x rule: less_induct)
    fix x A xa
    show "(⋀y A xa. y < x ⟹ ¬ evi A (liar[nxa, 0]) y) ⟹
        ¬ evi A (liar[nxa, 0]) x"
    proof (cases x)
      case 0
      then show ?thesis using evi_0 by auto
    next
      case xdef: (Suc nat)
      define A' "
      assume ih: "(⋀y A x' "
      show  "¬ evi A (liar[nxa, 0]) x"
      proof
        assume d0:"evi A (liar[nxa, 0]) x"
        then obtain x' " using
           liar_def evi_app2_suc trm_num_0 adef xdef by metis
        then obtain x' "        using liar_def evi_neg_suc trm_num_0 adef
```

```
xdef not0_implies_Suc evi_0 by metis
      show "False" using ih d1 d2 d0 by auto
    qed
  qed
qed

theorem liar_paradox: "¬ tbool A (liar[nx ,0])"
using liar_paradox_simple tbool_ex by blast
end
```

C Appendix C: Formalization of Curry's Paradox

```
locale curry_ga = model_evi_imp + model_evi_app2 + model_evi_eq +
  fixes curry :: dfn
  assumes curry_def : "dfn_is curry 2 ((curry[nx, 0]) ⟶ pigs)"
begin
```

Curry's paradox does not evaluate to any value in GA. As with the Liar and the Truthteller, this is proven by induction over the step count.

```
lemma curry_evi : "¬ evi A (curry[nx, 0]) s"

proof (induction s arbitrary: A x rule: less_induct)
  fix s A x
  show "(⋀y A xa. y < s ⟹ ¬ evi A (curry[nxa, 0]) y) ⟹
      ¬ evi A (curry[nx, 0]) s"
  proof (cases s)
    case 0
    then show ?thesis using evi_0 by auto
  next
    case sdef:(Suc nat)
    assume ih:"(⋀y A x. y < s ⟹ ¬ evi A (curry[nx, 0]) y)"
    show "¬ evi A (curry[nx, 0]) s"
    proof
      assume contra: "evi A (curry[nx, 0]) s"
      then obtain s' "
        using curry_def evi_app2_suc trm_num_0 sdef by metis
      then obtain s' " using evi_0 not0_implies_Suc by metis
      then consider
          (l) "evi (asn2 (natv x) (natv 0)) (curry[nx, 0]) s' "
        | (r) "⋀ pigs. evi (asn2 (natv x) (natv 0)) (pigs) s' "
        using curryp evi_imp_suc by metis
      then show "False"
      proof (cases)
        case l
        then show "False" using ih curryp contra sd by fastforce
      next
```

```
      case r
      then show "False" using curryp ih less_SucI sd by blast
    qed
  qed
 qed
qed
```

Curry's paradox has no boolean value

```
theorem curry_paradox: "¬ tbool A (curry [nx,0])"
using tbool_ex curry_evi by force
end
```