



Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

*Distributed  
Computing*



# **A Foundational Formalization of Grounded Deduction in Isabelle/Pure**

BSc Thesis

Sascha Kehrli  
skehrli@ethz.ch

Distributed Computing Group  
Computer Engineering and Networks Laboratory  
ETH Zürich

**Supervisors:**  
Prof. Dr. Bryan Ford  
Prof. Dr. Roger Wattenhofer

31.08.2025

# Abstract

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation: Recursive Definitions in Classical and Constructive Logic .....	1
1.2	Enter GD .....	2
<b>2</b>	<b>Background</b>	<b>3</b>
2.1	Isabelle/Pure .....	3
2.1.1	Syntax of Pure .....	3
2.1.2	Equality, Implication, and Quantification as Type Constructors .....	4
2.1.3	Deduction Rules .....	4
2.1.4	Formalizing Object Logics in Pure .....	5
2.2	Grounded Deduction (GD) .....	7
<b>3</b>	<b>Formalizing GD in Pure</b>	<b>9</b>
<b>4</b>	<b>Tooling for Isabelle/GD</b>	<b>10</b>
<b>5</b>	<b>Encoding Inductive Datatypes in GD</b>	<b>12</b>
<b>A</b>	<b>References</b>	<b>13</b>

# Introduction

1

## 1.1 Motivation: Recursive Definitions in Classical and Constructive Logic

We start with the simple observation that in logics of both classical and constructive tradition, there is a seemingly inherent lack of definitional freedom. That is, definitions must describe provably terminating expressions. The reason for these restrictions is that, without them in place, logics built on either tradition would be inconsistent.

To see this, consider the definition

$$L \equiv \neg L.$$

Let us imagine that this is a valid definition in a classical logic (that is, a logic that at least has the law of excluded middle (LEM) and double negation elimination). We may then reason about its truth value using the LEM.

Let us first prove that  $L$  holds by contradiction.

Assuming  $\neg L$ , we can derive  $\neg\neg L$  by unfolding the definition once and then  $L$  via double negation elimination. Since we derived both  $L$  and  $\neg L$  from hypothetically assuming  $\neg L$ , a contradiction, this allows us to definitely conclude  $L$ .

However, having proven  $L$ , we can also derive  $\neg L$  by applying the definition, and thus derived a contradiction in the logic itself, making it inconsistent.

What went wrong? The law of excluded middle forces a truth value on any term in classical logic, thus circular or non-sensical definitions such as  $L \equiv \neg L$ , for which no truth value can or should be assigned, cannot be admitted.

Constructive logics discard the law of excluded middle and are thus safe from a proof by contradiction like the one shown above. However, in intuitionistic tradition, lambda calculus terms are interpreted as proof terms, witnessing the truth of the proposition encoded by their type. Lambda functions of type  $A \Rightarrow B$  are then interpreted as producing a proof of  $B$  given a proof of  $A$ , which however means that they must always terminate.

To see this, consider the following attempt at a definition of an (ill-founded) term of type  $\forall\alpha.\alpha$ , i.e., a proof of every proposition:

$$\text{prove\_anything} := \Lambda\alpha. \text{prove\_anything } \alpha$$

Here, the construct  $\Lambda$  is the type-level analogue of lambda abstraction: it abstracts over a type variable and substitutes it in the body. That is, if  $e$  has type  $T$ , then  $\Lambda\alpha.e$  has type  $\forall\alpha.T$ .

If such a term were permitted in the logic, it would type-check as having type  $\forall\alpha.\alpha$ . Instantiating it at any type  $P$  yields a term of type  $P$ , i.e., a proof of  $P$  for arbitrary  $P$ , making every proposition in the logic trivially provable.

What went wrong this time? Functions in constructive logics represent logical implication. If a function has type  $A \Rightarrow B$ , the function must provide proof of  $B$ , that is, return a term  $b : B$ , given any term  $a : A$ . The function *witnesses* the implication of  $A$  to  $B$ . If the function

does not terminate on an input however, this proof is not actually constructed and assuming the hypothetical resulting proof term leads to inconsistency.

## 1.2 Enter GD

Grounded deduction (GD) is a logical framework developed recently at EPFL, whose development was motivated by precisely the observation made above. The project aims to axiomatize a consistent formal system, in which arbitrary recursion in definitions is permitted, which is still as expressive as possible. There is an ongoing formalization project of GD in the proof assistant Isabelle/HOL, which already yielded a consistency proof of the quantifier-free fragment of GD, showing great promise for its credibility. However, the other aim of GD is to show that it is also expressive and importantly, usable as a tool for formalizing mathematics itself. The formalization in the mature HOL logic enables studying meta-logical properties of GD, such as consistency. However, it is not suitable for providing GD as a tool for formal reasoning itself for two main reasons.

- Formalizing GD within a mature metalogic such as HOL adds the axioms of the metalogic to the trusted base of GD, which is undesirable from a meta-logical perspective.
- A logic is developed largely for idealistic reasons; the authors believe its reasoning principles are the right ones for at least some domain. Formalizing such a logic within another rich logic means that its reasoning principles are simply embedded in the, likely very different principles, of the meta-logic, defeating that purpose.

It is thus highly desirable to formalize a foundational formal system like GD atop a very minimal reasoning framework.

This is exactly what Isabelle provides with the Pure framework: A minimal, generic logical calculus to formalize object logics on top of. Any object logic in Isabelle, including Isabelle/HOL, are formalized atop Pure.

This thesis aims to fully axiomatize GD in Pure, yielding essentially an interactive theorem prover Isabelle/GD, which can be used for formal reasoning based directly on the reasoning principles and axioms of GD.

# Background

2

## 2.1 Isabelle/Pure

Isabelle provides a barebones logical framework called *Pure*. It contains a minimal meta-logic, which is a typed lambda calculus with few additional connectives, some keywords to add types and constants to said calculus, and a structured proof language called Isar.

Any object logic in Isabelle, for example the highly mature Isabelle/HOL fragment, are formalized atop Pure.

Isabelle is implemented in Standard ML (SML), and implementing an object logic directly in Pure almost always requires writing SML for things like proof automation, providing keywords or definitional mechanisms for users, or various other tooling such as code extraction.

Unfortunately, there is no single document that lays out the syntax, axioms, and derivation rules of the Pure calculus in their entirety. The following is an attempt at providing such a characterization, combining information from two Isabelle papers [1], [2] and the Isabelle reference manual [3].

### 2.1.1 Syntax of Pure

The core syntax of Pure is a typed lambda calculus, augmented with type variables, universal quantification, equality, and implication.

Propositions are terms of the distinct type `prop`. Propositions in Pure are thus terms and not types, like they are in type-theory based provers like Rocq or Lean.

#### Type Syntax

$$\begin{aligned} \tau &::= \alpha \text{ (type variable)} \\ &| \tau \Rightarrow \tau \text{ (function type)} \\ &| \text{prop} \text{ (type of propositions)} \end{aligned}$$

#### Term Syntax

$$\begin{aligned} t &::= x \text{ (variable)} \\ &| c \text{ (constant)} \\ &| t \ t \text{ (application)} \\ &| \lambda x :: \tau. t \text{ (lambda abstraction)} \\ &| t \Rightarrow t \text{ (implication)} \\ &| t \equiv t \text{ (equality)} \\ &| \bigwedge x :: \tau. t \text{ (universal quantification)} \end{aligned}$$

The symbols used for implication, equality, and universal quantification are non-standard to leave the standard symbols free for object logics.

Even though Pure has type variables, it provides no construct to capture them as an argument, and thus also has no for-all type like the polymorphic lambda calculus System F.

### 2.1.2 Equality, Implication, and Quantification as Type Constructors

The connectives equality, implication, and universal quantification are all type constructors of the `prop` type with the following polymorphic type signatures.

$$\begin{aligned} \equiv &:: \alpha \Rightarrow \alpha \Rightarrow \text{prop} \\ \Rightarrow &:: \text{prop} \Rightarrow \text{prop} \Rightarrow \text{prop} \\ \bigwedge &:: (\alpha \Rightarrow \text{prop}) \Rightarrow \text{prop} \end{aligned}$$

The arguments of  $\equiv$  are the two operands to compare, the arguments for  $\Rightarrow$  are the sequent and consequent respectively, while the argument of  $\bigwedge$  is a function from the type whose inhabitants are quantified over to the term that represents the body of the quantifier.

Since type variables denote only a single, albeit arbitrary, type, there is technically one instance of each polymorphic connective for every given type. For example, for any type  $\sigma$ , there is a constant  $\equiv_{\sigma} :: \sigma \Rightarrow \sigma \Rightarrow \text{prop}$ .

### 2.1.3 Deduction Rules

The operational semantics of the underlying lambda calculus and its typing rules are standard and thus omitted. The following discusses the more interesting deduction rules, which make Pure a logical framework.

Relative to an object logic with a set of defined axioms  $\Omega$ , any axiom  $\omega \in \Omega$  can always be derived, as can any assumption  $\gamma \in \Gamma$ .

#### Basic Rules

$$\frac{A \text{ axiom}}{\Gamma \vdash A} \quad (\text{Axiom})$$

$$\frac{A \in \Gamma}{\Gamma \vdash A} \quad (\text{Ass})$$

The implication and universal quantification introduction and elimination rules are standard.

#### Implication Deduction Rules

$$\frac{\Gamma \cup A \vdash B}{\Gamma \vdash A \Rightarrow B} \quad (\Rightarrow I)$$

$$\frac{\Gamma_1 \vdash A \Rightarrow B \quad \Gamma_2 \vdash A}{\Gamma_1 \cup \Gamma_2 \vdash B} \quad (\Rightarrow E)$$

#### Universal Quantification Deduction Rules

$$\frac{\Gamma \vdash B(x) \quad x \text{ not free in } \Gamma}{\Gamma \vdash \bigwedge x.B(x)} \quad (\bigwedge I) \qquad \frac{\Gamma \vdash \bigwedge x.B(x)}{\Gamma \vdash B(a)} \quad (\bigwedge E)$$

For equality, besides the expected deduction rules corresponding to the equivalence relation properties, there are also deduction rules for equality of lambda abstractions and **prop**, the latter of which is defined as equivalence of truth values ( $a \Rightarrow b$  and  $b \Rightarrow a$ ).

#### Equality Deduction Rules

$$\begin{array}{ccc} \frac{}{\Gamma \vdash a \equiv a} \quad (\equiv \text{Refl}) & \frac{\Gamma \vdash b \equiv a}{\Gamma \vdash a \equiv b} \quad (\equiv \text{Sym}) & \frac{\Gamma \vdash a \equiv b \quad \Gamma \vdash b \equiv c}{\Gamma \vdash a \equiv c} \quad (\equiv \text{Trans}) \\ \\ \frac{\Gamma \vdash a \equiv b}{\Gamma \vdash (\lambda x.a) \equiv (\lambda x.b)} \quad (\equiv \text{Lam}) & \frac{\Gamma \vdash a \Rightarrow b \quad \Gamma \vdash b \Rightarrow a}{\Gamma \vdash a \equiv b} \quad (\equiv \text{Prop}) \end{array}$$

The  $\lambda$ -conversion rules facilitate equivalence reasoning for lambda abstractions. The rules are  $\alpha$ -conversion,  $\beta$ -conversion and extensionality. The notation  $a[y/x]$  expresses the substitution of  $x$  with  $y$  in  $a$ , that is, all occurrences of  $x$  in  $a$  are replaced with  $y$ .

#### Lambda Conversion Rules

$$\begin{array}{ccc} \frac{y \text{ not free in } a}{\Gamma \vdash (\lambda x.a) \equiv (\lambda y.a[y/x])} \quad (\alpha\text{-Conv}) & \frac{}{\Gamma \vdash (\lambda x.a) b \equiv a[b/x]} \quad (\beta\text{-Conv}) \\ \\ \frac{\Gamma \vdash f x \equiv g x \quad x \text{ not free in } \Gamma, f, \text{ and } g}{\Gamma \vdash f \equiv g} \quad (\text{Ext}) \end{array}$$

Finally, the equivalence substitution rule:

#### Equivalence Elimination

$$\frac{\Gamma \vdash a \equiv b \quad \Gamma \vdash a}{\Gamma \vdash b} \quad (\equiv E)$$

### 2.1.4 Formalizing Object Logics in Pure

An object logic in Pure is created by adding new types, constants and axioms. That is, the Pure logic is extended.

It is convention to define a new propositional type in an object logic, which is used as the type of propositions in the *object* logic, as opposed to the *meta* logic, which is Pure.

This is achieved using the `typedcl` keyword, which declares a syntactic type in the Pure calculus. This type has no known inhabitants or any other information yet.



typed decl  $o$

Any information about  $o$  must be axiomatized. For example, the following declares typed constants `disj` and `True` and axiomatizes certain rules about them.

```
1 axiomatization
2   True :: <o> and
3   disj :: <o  $\Rightarrow$  o  $\Rightarrow$  o> (infixr <v> 30)
4 where
5   true:   <True>
6   disjI1: <P  $\Rightarrow$  P v Q> and
7   disjI2: <Q  $\Rightarrow$  P v Q> and
```

Isabelle

The axiomatized rules here simply state that `True` holds and that from either  $P$  or  $Q$ ,  $P \vee Q$  can be derived. Here,  $P$  and  $Q$  are implicitly universally quantified, ranging over all terms of type `prop`. That is,  $P$  and  $Q$  can be substituted for any term of the correct type (which is  $o$  for both  $P$  and  $Q$  here). Now, the type  $o$  has known inhabitants and structure. However, Isabelle (or rather, Pure) cannot reason about it, because it cannot connect the type  $o$  meaningfully with its meta-theory. To resolve this, a judgment must translate from the object-level proposition type  $o$  to the meta-level type `prop`.

```
1 judgment
2   Trueprop :: <o  $\Rightarrow$  prop> (<_> 5)
```

Isabelle

The syntax annotation  $(\langle\_ \rangle 5)$  means that any term of type  $o$  is implicitly augmented with the `Trueprop` judgment. The very low precedence value of 5 ensures that the `Trueprop` judgment is only applied to top-level terms. For example, the term  $x \vee \text{True}$  is the same as `Trueprop (x  $\vee$  True)` and both are of type `prop` due to the `Trueprop` predicate converting the formula to that type.

As you might have noticed, we have made use of this implicit conversion from  $o$  to `Prop` already in the axiomatization block from earlier. That is, the `Trueprop` judgment must be declared before the axiomatization block, else the latter will just report a typing error.

Now, we can state and prove a first lemma in this tiny object logic, using the previously defined axioms.

```
1 lemma "x v True"
2   apply (rule disjI2)
3   apply (rule true)
4   done
```

Isabelle

Applying `disjI2` ‘selects’ the second disjunct to prove, which results in the subgoal `True`, which in turn we can solve using the `true` axiom.

This short introduction suffices for now, as we will later implement a much richer logic, Grounded Deduction, using these same basic constructs. We can clearly see that implementing an object logic in Pure actually extends Pure, in the sense that it adds new types and deduction rules. For example, our extension added a type and three symbols to the existing syntax of Pure. If we call the tiny logic formalized above `Pure'`, the following is its type and term syntax:

**Type Syntax of Pure'**

$$\begin{aligned}
\tau &::= \alpha \text{ (type variable)} \\
&| \tau \Rightarrow \tau \text{ (function type)} \\
&| \text{prop} \text{ (type of propositions)} \\
&| \text{o} \text{ (type of object propositions)}
\end{aligned}$$
**Term Syntax of Pure'**

$$\begin{aligned}
t &::= x \text{ (variable)} \\
&| c \text{ (constant)} \\
&| t \ t \text{ (application)} \\
&| \lambda x :: \tau. t \text{ (lambda abstraction)} \\
&| t \Longrightarrow t \text{ (implication)} \\
&| t \equiv t \text{ (equality)} \\
&| \bigwedge x :: \tau. t \text{ (universal quantification)} \\
&| \text{True} \text{ (o-typed true constant)} \\
&| t \vee t \text{ (o-typed logical or connective)} \\
&| \text{Trueprop } t \text{ (conversion function o to Prop)}
\end{aligned}$$

Further, we can view the added axioms as new inference rules, with the explicit `Trueprop` function application.

$$\frac{\Gamma \vdash \text{Trueprop } P}{\Gamma \vdash \text{Trueprop } P \vee Q} \quad (\text{disjI1}) \qquad \frac{\Gamma \vdash \text{Trueprop } Q}{\Gamma \vdash \text{Trueprop } P \vee Q} \quad (\text{disjI2})$$

$$\frac{}{\Gamma \vdash \text{Trueprop True}} \quad (\text{true})$$

It is technically possible to avoid declaring a new proposition type for an object logic and instead use `prop` directly as the type of propositions. However, doing so means that the (object) logic immediately inherits the built-in connectives and deduction rules, such as implication ( $\Longrightarrow$ ) and universal quantification ( $\bigwedge$ ), and the sequent-style reasoning built into the kernel.

Such a structure reduces the control one has over the logic and keeps many reasoning principles implicit.

**2.2 Grounded Deduction (GD)**

This subsection provides a full characterization of the GD logic that is later formalized.

GD makes definitions first-class objects in the logic and allows arbitrary references of the symbol currently being defined or other, previously defined symbols, in the expanded term.

To prevent immediate inconsistency, GD must weaken other deduction rules commonly seen in classical logic. Specifically, GD adds a so-called *habeas quid* sequent to many inference rules. Intuitively, this means that in certain inference rules, a (sub)term must first be shown to terminate. The following formalization is based on the GD formalization in [4].

## Formalizing GD in Pure

3

We now have all the tools to formalize the GD logic in Isabelle.

# Tooling for Isabelle/GD

## 4

There are two main motivations for formalizing GD in Pure. On one hand, it enables studying GD reflectively in an instance of itself. On the other hand, having implemented GD in Pure, we have effectively obtained an interactive theorem prover based on the axioms of GD. In its current state however, Isabelle/GD is not a very usable theorem prover. There is no proof automation, no term rewriting, and no easy way to formalize higher level mathematics. Users can only reason about natural numbers and only use axioms or previously proven lemmas in their proofs.

This chapter aims for making Isabelle/GD more usable as a proof assistant and, towards that end, introduces a rewrite engine and a proof-search procedure for automating simple proofs, as well as a multitude of simpler methods to facilitate even fully manual reasoning.

As an initial motivation, here is how cumbersome a simple proof looks like in the current version of GD.

Before:

Isabelle

```

1 lemma cons_is_list:
2   assumes n_nat: "n N"
3   assumes xs_list: "is_list xs"
4   shows "is_list (Cons n xs)"
5   apply (rule eqSubst[where a="True"])
6   apply (unfold_def is_list_def)
7   apply (rule eqSym)
8   apply (rule condI2BEq)
9   apply (gd_auto)
10  apply (rule n_nat)
11  apply (rule list_nat)
12  apply (rule xs_list)
13  apply (gd_auto)
14  proof -
15    show "(if Cons n xs = Cons n xs ∧ is_list xs ∧ (n N) then True else False)
16      = True"
17    apply (rule condI1B)
18    apply (rule conjI)+
19    apply (fold isNat_def)
20    apply (rule cons_nat)
21    apply (rule n_nat)
22    apply (rule list_nat)
23    apply (rule xs_list)
24    apply (rule xs_list)
25    apply (rule n_nat)
26    apply (rule true_bool)
27    done
28  show "True" by (rule true)

```

```
28 qed
```

After:

```
1 lemma cons_is_list [gd_auto]:  
2   shows "n N  $\Rightarrow$  xs N  $\Rightarrow$  is_list xs  $\Rightarrow$  is_list (Cons n xs)"  
3 by (unfold_def is_list_def, simp)
```

Isabelle

# Encoding Inductive Datatypes in GD 5

With Isabelle/GD now being a slightly more convenient proof assistant, the next goal is to make it easier to extend the domain of discourse. Modern proof assistants, like Isabelle/HOL, contain fancy definitional mechanisms that allow for easy definition of things like inductive datatypes, recursive predicates, infinitary sets, and so on.

These definitional packages are effectively *theory compilers*, as they take a simple high-level definition, like an inductive datatype declaration, and map it to definitions, axioms, and automatically proven lemmas, encoding the high-level definition in lower level existing primitives.

The goal of this chapter is to provide a definitional mechanism for inductive datatypes in Isabelle/GD and encode them under the hood into the existing formalization of natural numbers. That is, any inductive datatype should be definable and conveniently usable without adding any axioms.

The roadmap towards this lofty goal is as follows:

- Formalize enough basic number theory to be able to define cantor pairings and some basic properties about them.
- Manually encode an inductive datatype. Define a type membership predicate, define the constructors as cantor pairings of their arguments and prove the necessary lemmas (such as all constructors being disjoint, the type membership predicate returning true for all values of the constructors, induction over the datatype, and so on).
- Write a definitional package that parses an inductive datatype declaration and compiles it into the necessary definitions, lemmas, and accompanying proofs.

# A References

- [1] L. C. Paulson, “Isabelle: The Next 700 Theorem Provers.” [Online]. Available: <https://arxiv.org/abs/cs/9301106>
- [2] L. C. Paulson, “The foundation of a generic theorem prover,” *J. Autom. Reason.*, vol. 5, no. 3, pp. 363–397, 1989, doi: 10.1007/BF00248324.
- [3] L. Paulson, T. Nipkow, and M. Wenzel, “The Isabelle Reference Manual,” 1998.
- [4] B. Ford, “Reasoning Around Paradox with Grounded Deduction.” [Online]. Available: <https://arxiv.org/abs/2409.08243>