Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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

01.08.2025

Abstract

Contents

Introduction 1

Background 2

2.1 Isabelle/Pure

Any object logics in Isabelle, for example the popular Isabelle/HOL fragment, are formalized atop a logic framework called Pure. Pure is designed to be as generic as possible to allow implementing a wide range of object logics atop it. It provides infrastructure to define types and axioms to facilitate the formalization of object logics.

Unfortunately, there is no single document that lays out the syntax, axioms, and derivation rules of Pure 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

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

Term Syntax

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

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.

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

The arguments of $\equiv$ are the two operands to compare, the arguments for $\Longrightarrow$ 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 σ, there is a constant $\underset{\sigma}{\equiv} :: \sigma \Rightarrow \sigma \Rightarrow \text{prop}$.

2.1.3 Deduction Rules

The operational semantics of the underlying lamdba 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 Ω, any axiom $\omega \in \Omega$ can always be derived, as can any assumption $\gamma \in \Gamma$.

Basic Rules

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

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

Implication Deduction Rules

$$\frac{\Gamma \cup A \vdash B}{\Gamma \vdash A \Longrightarrow B} \quad (\Longrightarrow I) \qquad\qquad \frac{\Gamma_1 \vdash A \Longrightarrow B \quad \Gamma_2 \vdash A}{\Gamma_1 \cup \Gamma_2 \vdash B} \quad (\Longrightarrow E)$$

Universal Quantification Deduction Rules

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

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 ($a \Longrightarrow b$ and $b \Longrightarrow a$).

Equality Deduction Rules

$$\frac{}{\Gamma \vdash a \equiv a} \ (\equiv \text{Refl}) \qquad \frac{\Gamma \vdash b \equiv a}{\Gamma \vdash a \equiv b} \ (\equiv \text{Sym}) \qquad \frac{\Gamma \vdash a \equiv b \quad \Gamma \vdash b \equiv c}{\Gamma \vdash a \equiv c} \ (\equiv \text{Trans})$$

$$\frac{\Gamma \vdash a \equiv b}{\Gamma \vdash (\lambda x.a) \equiv (\lambda x.b)} \ (\equiv \text{Lam}) \qquad \frac{\Gamma \vdash a \Longrightarrow b \quad \Gamma \vdash b \Longrightarrow a}{\Gamma \vdash a \equiv b} \ (\equiv \text{Prop})$$

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

Lambda Conversion Rules

$$\frac{\text{y not free in } a}{\Gamma \vdash (\lambda x.a) \equiv (\lambda y.a[y/x])} \ (\alpha\text{-Conv}) \qquad \frac{}{\Gamma \vdash (\lambda x.a)\ b \equiv a[b/x]} \ (\beta\text{ -Conv})$$

$$\frac{\Gamma \vdash f\ x \equiv g\ x \quad \text{x not free in } \Gamma, f, \text{and } g}{\Gamma \vdash f \equiv g} \ (\text{Ext})$$

Finally, the equivalence substitution rule:

Equivalence Elimination

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

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 typedecl keyword, which declares a syntactic type in the Pure calculus. This type has no known inhabitants or any other information yet.

typedecl 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                                                    Isabelle
2   True :: ‹o› and
3   disj :: ‹o ⇒ o ⇒ o›  (infixr ‹∨› 30)
```

```
4 where
5   true:   ‹True›
6   disjI1: ‹P ⇒ P ∨ Q› and
7   disjI2: ‹Q ⇒ P ∨ Q› and
```

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. Now, the type o has knows inhabitants and structure. However, Isabelle (or rather, Pure) cannot reason about it, because it can only reason about terms of type Prop. To resolve this, a judgment must translate from the object-level proposition type o to the meta-level type Prop.

```
1 judgment                                                    Isabelle
2   Trueprop :: ‹o ⇒ prop›  (‹_› 5)
```

The syntax annotation (‹_› 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 $x \vee$ 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 ∨ True"                                            Isabelle
2 apply (rule disjI2)
3 apply (rule true)
4 done
```

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'

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

Term Syntax of Pure'

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

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

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

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

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.

This implicit structure reduces the control one has over the logic, and as a result, the best practice is to take a clean-slate approach, declare a new type for object-level propositions, and define a separate set of connectives and inference rules for it.

2.2 Grounded Deduction (GD)

2.2.1 Motivation: Recursive Definitions in Classical and Constructive Logic

Grounded deduction is a logical framework developed recently at EPFL. Its development is motivated by the observation that in logics of both classical and constructive tradition, there is inherently no definitional freedom. That is, definitions must describe provably terminating expressions. The reason for this is that without such restrictions in place, logics built on either tradition would be immediately 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 the definition 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 Λ 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.

Having shown that the presence of arbitrarily recursive definitions leads to logical inconsistency in both widely recognized schools of logic, let us now motivate why a formal system resistant to arbitrary definitional recursion would be desirable in the first place. In computer science in particular, the need for an ability to define arbitrary recursion is immediate; virtually every popular programming language is turing complete, which requires arbitrary recursion. Also, many programs are not designed to terminate at all, like operating system kernels or web servers. Additionally, the ability to state paradoxical definitions and use a formal system to reason about them is of interest in and of itself. A less trivial example of such a paradoxical recursive definition than the Liar's paradox stated above would be Yablo's paradox. This paradox is notable, because unlike the classical liar sentence ("This

sentence is false"), it constructs a self-referential paradox without direct self-reference, using an infinite sequence of sentences.

Let us consider an infinite list of sentences $(Y_n)_{n \in N}$, where each sentence Y_n is defined as follows:

$$Y_n \equiv \forall k.k > n \Rightarrow \neg Y_k$$

That is, each sentence Y_n asserts that all sentences following it are untrue.

Now suppose that one of the sentences Y_i is true. Then, by its definition, all Y_j with $j > i$ must be false. In particular, $Y_{i+1}$is false, which means that there must exist some $k > i + 1$ such that Y_k is true. But this contradicts the statement by Y_i that all Y_j with $j > i$ are false, in particular Y_k. Hence, no $Y_i$can be true.

But then, $\neg Y_i$ holds for all $i \in N$. This however means that Y_0 must be true, as its claim of $\neg Y_i$ for any $i > 0$ is indeed fulfilled. A seemingly paradoxical setup, even though there is no direct self-reference. No truth value can be assigned to any of the Y_i.

2.2.2 An Overview of GD

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

To prevent immediate inconsistency, GD must of course weaken other deduction rules. Specifically, GD adds a so-called *habeas quid* sequent to many inference rules. Intuitively, this means that in certain inference rules, a term must first be shown to terminate in order to be used. This is the basic idea of GD. 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.

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