ETH
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

19.09.2025

Abstract

Contents

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, these logics would be inconsistent.

To see this for the case of classical logic, 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). If the logic allows us to deduce either of L or $\neg L$, the other can be decuded as well by unfolding the definition and making use of double negation elimination, making the logic inconsistent.

Thanks to the LEM, we can 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.

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:

$$\text{prove\_anything} \coloneqq \Lambda\alpha.\ \text{prove\_anything}\ \alpha$$

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.

1.2 Enter GD

Grounded deduction (GD) is a logical framework developed recently at EPFL and whose development was motivated by precisely the observation made above. The project aims to axiomatize a consistent (free from contradiction) formal system, in which arbitrary recursion in definitions is permitted and which is still as expressive as possible.

In Section 2.2, *Grounded Arithmetic* (*GA*), a first-order theory of arithmetic based on *grounded* principles, is fully formalized based on a formalization by the authors of *GD* [1].

The definition $L \equiv \neg L$ is perfectly valid in *GA*. However, it is not possible to assign a boolean truth value to L using the *GA* inference rules. For example the derived contradiction rule in *GA* provides no help, as opposed to the classical version. The reason for this is an additional premise of $p \vee \neg p$, a circular proof obligation, since it asks for the very truth value assignment we are currently trying to prove. The truth value of L is not *grounded* in anything.

$$\frac{\Gamma \vdash p \ \mathsf{B} \qquad \Gamma \cup \{\neg p\} \vdash q \qquad \Gamma \cup \{\neg p\} \vdash \neg q}{\Gamma \vdash p}$$

There is an ongoing formalization project of *GD*/*GA* in the proof assistant Isabelle/HOL, which already yielded a consistency proof of the quantifier-free fragment of GD, showing great promise for GD as a reasoning framework. 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 a few 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.
- The logical primitives and axioms being embedded within the primitives of another logic (HOL in this case) makes reasoning within it contrived and needlessly complicated.
- 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, is formalized atop Pure.

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

Background 2

2.1 Isabelle/Pure

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

This subsection provides a formalization of the *Pure* calculus.

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 [2], [3] and the Isabelle reference manual [4].

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

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

Term Syntax

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

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 A any axiom $\alpha \in \mathrm{A}$ can always be derived, as can any assumption $\gamma \in \Gamma$.

Basic Rules

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

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

Implication Deduction Rules

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

Universal Quantification Deduction Rules

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

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

Equality Deduction Rules

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

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

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

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

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

Finally, the equivalence substitution rule:

Equivalence Elimination

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

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.

Isabelle

```
axiomatization
  True :: ‹o› and
  disj :: ‹o ⇒ o ⇒ o›  (infixr ‹∨› 30)
where
  true:   ‹True›
  disjI1: ‹P ⇒ P ∨ Q› and
  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 of the correct type (which is `o` for both P and Q here). Now, the type `o` has knows 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`.

Isabelle

```
judgment
  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.

Isabelle

```
lemma "x ∨ True"
apply (rule disjI2)
apply (rule true)
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'

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

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

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

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

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.

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

2.2 Grounded Arithmetic (GA)

This subsection provides a full characterization of GA, a first-order formalization of arithmetic based on the principles of GD. This is the fragment that is later formalized in Isabelle.

GA 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, GA must weaken other deduction rules commonly seen in classical logic. Specifically, GA 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.

2.2.1 BGA Formalization

We start by formalizing the syntax and axioms of *Basic Grounded Arithmetic* (*BGA*), the quantifier-free fragment of GA, based on the formalization in [1]. This formulation later adds quantifiers by encoding them as unbounded computations in *BGA*, yielding full *GA*. This however requires a sophisticated encoding using Gödel-style reflection, i.e. encoding its own term syntax into natural numbers, which is out of scope for a formalization in *Pure*. Thus, we will later add quantifiers by simply axiomatizing them.

The primitive term syntax of BGA is the following.

BGA Primitive Term Syntax

$t ::= x$	variable
$\mid 0$	natural-number constant zero
$\mid \mathbf{S}(t)$	natural-number successor
$\mid \mathbf{P}(t)$	natural-number predecessor
$\mid \neg t$	logical negation
$\mid t \lor t$	logical disjunction
$\mid t = t$	natural-number equality
$\mid \text{if } t \text{ then } t \text{ else } t$	conditional evaluation
$\mid d(t, ..., t)$	application of recursive definition

It is noteworthy that the GA term syntax mixes expressions that are natural numbers and expressions that are formulas into the same syntactic category. For example, the expression $S(x) = x \lor x$ is a valid term according to the syntax, despite the left-hand side shape clearly indicating a natural number, while the right hand side shape indicates a truth value.

Besides the primitives, other constants and logical connectives are defined as notational shorthands using the primitives.

Notational Shorthands

$\text{True} \equiv 0 = 0$	true constant
$\text{False} \equiv 0 = S(0)$	false constant
$a\ \mathsf{N} \equiv a = a$	number type
$p\ \mathsf{B} \equiv p \vee \neg p$	boolean type
$p \wedge q \equiv \neg(\neg p \vee \neg q)$	logical conjunction
$p \rightarrow q \equiv \neg p \vee q$	implication
$p \leftrightarrow q \equiv (p \rightarrow q) \wedge (q \rightarrow p)$	biconditional
$a \neq b \equiv \neg(a = b)$	inequality

The surprising shorthands are $a\ \mathsf{N}$ and $p\ \mathsf{B}$. The latter is a predicate over p deciding whether it is a binary truth value. In a logic with the law of excluded middle, $p\ \mathsf{B}$ would be a tautology for any p, but in a logic without it, it can be interpreted as a termination certificate for truth values. Similarly, $a\ \mathsf{N}$ can be interpreted as a termination certificate for natural number expressions. The shorthand itself is surprising, because if equality is reflexive, $a = a$ is true for any a. In GA however, equality is not reflexive as we will soon see, and a proof of $a = a$ is equivalent to a termination proof of the expression.

The syntax does not mean much without a set of axioms giving them meaning. We start with listing the propositional logic axioms.

In the following, Γ denotes a set of background assumptions. For completeness sake, the explicit structural rules governing this set of assumptions is listed here. Since Γ is a set, the usually explicit rules for permuting and duplicating assumptions are not needed.

Structural Rules

$$\frac{}{\Gamma \cup \{p\} \vdash p}\ (\text{H}) \qquad \frac{\Gamma \vdash q}{\Gamma \cup \{p\} \vdash q}\ (\text{W})$$

Propositional Logic Axioms

$$\frac{\Gamma \vdash p}{\Gamma \vdash p \vee q}\ (\vee\ \text{I1}) \qquad \frac{\Gamma \vdash q}{\Gamma \vdash p \vee q}\ (\vee\ \text{I2}) \qquad \frac{\Gamma \vdash \neg p \quad \Gamma \vdash \neg q}{\Gamma \vdash \neg(p \vee q)}\ (\vee\ \text{I3})$$

$$\begin{array}{c} \Gamma \vdash p \\ \hline\hline \Gamma \vdash \neg\neg p \end{array}\ (\neg\neg\ \text{IE}) \qquad \frac{\Gamma \vdash p \quad \Gamma \vdash \neg p}{\Gamma \vdash q}\ (\neg E) \qquad \frac{\Gamma \vdash \neg(p \vee q)}{\Gamma \vdash \neg p}\ (\vee\ \text{E1})$$

$$\frac{\Gamma \vdash \neg(p \vee q)}{\Gamma \vdash \neg q}\ (\vee\ \text{E2}) \qquad \frac{\Gamma \vdash p \vee q \quad \Gamma \cup \{p\} \vdash r \quad \Gamma \cup \{q\} \vdash r}{\Gamma \vdash r}\ (\vee\ \text{E3})$$

Rules such as $\neg\neg$ IE with a double-line are bidirectional, i.e. they serve as both an introduction and elimination rule.

The propositional axioms are fairly standard, but inclusion of double negation elimination is notable, as this is common in classical logics, but omitted in computational logics.

Equality Axioms

$$\frac{\Gamma \vdash a = b}{\Gamma \vdash b = a}\ (= S) \qquad \frac{\Gamma \vdash a = b \quad \Gamma \vdash K\ a}{\Gamma \vdash K\ b}\ (= E)$$

The equality axioms notably omit reflexivity. Symmetry of equality is an axiom, as is equality substitution in an arbitrary context K. Transitivity of equality can be deduced using equality substitution.

Natural Number Axioms

$$\frac{}{\Gamma \vdash 0\ \mathsf{N}}\ (0I) \qquad \frac{\Gamma \vdash a = b}{\Gamma \vdash \mathbf{S}(a) = \mathbf{S}(b)}\ (\mathbf{S} = IE) \qquad \frac{\Gamma \vdash a = b}{\Gamma \vdash \mathbf{P}(a) = \mathbf{P}(b)}\ (\mathbf{P} = I)$$

$$\frac{\Gamma \vdash a\ \mathsf{N}}{\Gamma \vdash \mathbf{P}(\mathbf{S}(a)) = a}\ (\mathbf{P} = I2) \qquad \frac{\Gamma \vdash a\ \mathsf{N}}{\Gamma \vdash \mathbf{S}(a) \neq 0}\ (\mathbf{S} \neq 0I) \qquad \frac{\Gamma \vdash c \quad \Gamma \vdash a\ \mathsf{N}}{\Gamma \vdash (\text{if } c \text{ then } a \text{ else } b) = a}\ (?\,I1)$$

$$\frac{\Gamma \vdash \neg c \quad \Gamma \vdash b\ \mathsf{N}}{\Gamma \vdash (\text{if } c \text{ then } a \text{ else } b) = b}\ (?\,I2) \qquad \frac{\Gamma \vdash K\ 0 \quad \Gamma \cup \{x\ \mathsf{N}, K\ x\} \vdash K\ \mathbf{S}(x) \quad \Gamma \vdash a\ \mathsf{N}}{\Gamma \vdash K\ a}\ (\text{Ind})$$

$$\frac{\Gamma \vdash a\ \mathsf{N} \quad \Gamma \vdash b\ \mathsf{N}}{\Gamma \vdash a = b\ \mathsf{N}}\ (= \text{TI}) \qquad \frac{\Gamma \vdash c\ \mathsf{B} \quad \Gamma \vdash a\ \mathsf{N} \quad \Gamma \vdash b\ \mathsf{N}}{\Gamma \vdash \text{if } c \text{ then } a \text{ else } b\ \mathsf{N}}\ (?\,\text{TI})$$

The natural number axioms are fairly close to the standard Peano axioms, with some notable exceptions.

The *grounding* equality is the $0I$ axiom, postulating that $0\ \mathsf{N}$, or, by unfolding the definition, $0 = 0$. Using the $S = IE$ axiom, $\mathbf{S}(a)\ \mathsf{N}$ can be deduced for any a for which $a\ \mathsf{N}$ is already known. The induction axiom ind has an additional premise of $a\ \mathsf{N}$, i.e. it requires proof that the expression induction is performed over is indeed a (terminating) natural number.

Conditional evaluation is a primitive in GA and its behavior must thus be axiomatized. The two inference rules correspond to the positive and negative evaluation of the condition, and they both require that the expression from the corresponding branch is shown to be terminating (i.e. $a\ \mathsf{N}$ and $b\ \mathsf{N}$ respectively). This additional premise prevents equalities of potentially non-terminating expressions to be deduced.

Grounded Contradiction Although GA is not classical, a contradiction rule can be derived. The resulting inference rule has an additional $p\ \mathsf{B}$ premise not present in the classical version, which demands p is first shown to have a truth value. To get a feeling for the logic, we construct the proof explicitly in a natural deduction style derivation tree.

Theorem 1. Grounded Contradiction

$$\frac{\Gamma \vdash p\ \mathsf{B} \quad \Gamma \cup \{\neg p\} \vdash q \quad \Gamma \cup \{\neg p\} \vdash \neg q}{\Gamma \vdash p}$$

Proof.

$$\dfrac{\dfrac{\Gamma \vdash p\ \mathsf{B}}{\Gamma \vdash p \lor \neg p}\,\mathsf{B}\text{ def} \qquad \dfrac{}{\Gamma \cup \{p\} \vdash p}\,\text{H} \qquad \dfrac{\Gamma \cup \{\neg p\} \vdash q \qquad \Gamma \cup \{\neg p\} \vdash \neg q}{\Gamma \cup \{\neg p\} \vdash p}\,\neg E}{\Gamma \vdash p}\,\lor\text{E3}$$

GROUNDED IMPLICATION Impliciation is not a primitive in GA, but rather the shorthand $a \to b \equiv \neg a \lor b$. From this definition, the classical elimination rule *modus ponens* can be derived. However, only a weakened introduction rule, with the now familiar additional *habeas quid* premise, can be derived.

Theorem 2. Modus Ponens

$$\dfrac{\Gamma \vdash p \qquad \Gamma \vdash p \to q}{\Gamma \vdash q} \quad (\to E)$$

Proof.

$$\dfrac{\dfrac{}{\Gamma \cup \{q\} \vdash q}\,\text{H} \qquad \dfrac{\dfrac{}{\Gamma \cup \{\neg p\} \vdash \neg p}\,\text{H} \qquad \dfrac{\Gamma \vdash p}{\Gamma \cup \{\neg p\} \vdash p}\,\text{W}}{\Gamma \cup \{\neg p\} \vdash q}\,\neg E \qquad \dfrac{\Gamma \vdash p \to q}{\Gamma \vdash \neg p \lor q}\,\to\text{def}}{\Gamma \vdash q}\,\lor\text{E3}$$

Theorem 3. Implication Introduction

$$\dfrac{\Gamma \vdash p\ \mathsf{B} \qquad \Gamma \cup \{p\} \vdash q}{\Gamma \vdash p \to q} \quad (\to I)$$

Proof.

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash p\ \mathsf{B}}{\Gamma \vdash p \lor \neg p}\,\mathsf{B}\text{ def} \qquad \dfrac{\Gamma \cup \{p\} \vdash q}{\Gamma \cup \{p\} \vdash \neg p \lor q}\,\lor\text{E2} \qquad \dfrac{\dfrac{}{\Gamma \cup \{\neg p\} \vdash \neg p}\,\text{H}}{\Gamma \cup \{\neg p\} \vdash \neg p \lor q}\,\lor\text{I1}}{\Gamma \vdash \neg p \lor q}\,\lor\text{E3}}{\Gamma \vdash p \to q}\,\to\text{def}$$

DEFINITIONAL AXIOMS Finally, the axioms for definitions allow arbitrary substitution of a symbol with its definition body (and the other way around) in any context. The vector notation $\vec{a}$ denotes an argument vector for the defined function symbol.

Definition Axioms

$$\dfrac{\Gamma \vdash s(\vec{x}) \equiv d(\vec{x}) \qquad \Gamma \vdash K\ d(\vec{a})}{\Gamma \vdash K\ s(\vec{a})} \quad (\equiv E) \qquad\qquad \dfrac{\Gamma \vdash s(\vec{x}) \equiv d(\vec{x}) \qquad \Gamma \vdash K\ s(\vec{a})}{\Gamma \vdash K\ d(\vec{a})} \quad (\equiv I)$$

2.2.2 GA with Axiomatized Quantifiers

As already mentioned, the creators of *GA* claim that quantifiers can be encoded into BGA using the powerful definitional mechanism [1], yielding full *GA* "for free". However, as this will not be feasible in the formalization within *Pure*, the following axiomatizes the quantifiers instead.

Quantifier Axioms

$$\frac{\Gamma \cup \{x\ \mathsf{N}\} \vdash K\ x}{\Gamma \vdash \forall x. K\ x}\ (\forall I) \qquad \frac{\Gamma \vdash \forall x. K\ x \quad \Gamma \vdash a\ \mathsf{N}}{\Gamma \vdash K\ a}\ (\forall E)$$

$$\frac{\Gamma \vdash a\ \mathsf{N} \quad \Gamma \vdash K\ a}{\Gamma \vdash \exists x. K\ x}\ (\exists I) \qquad \frac{\Gamma \vdash \exists x. K\ x \quad \Gamma \cup \{x\ \mathsf{N}, K\ x\} \vdash q}{\Gamma \vdash q}\ (\exists E)$$

Besides the additional *habeas quid* premises, the quantifier axioms are standard.

Since the quantifiers are primitive here, they must be added to the primitive term syntax, yielding the full *GA* primitive term syntax.

GA Primitive Term Syntax

$t ::= x$	variable
$\mid 0$	natural-number constant zero
$\mid \mathbf{S}(t)$	natural-number successor
$\mid \mathbf{P}(t)$	natural-number predecessor
$\mid \neg t$	logical negation
$\mid t \vee t$	logical disjunction
$\mid t = t$	natural-number equality
$\mid \text{if } t \text{ then } t \text{ else } t$	conditional evaluation
$\mid d(t, ..., t)$	application of recursive definition
$\mid \forall x.t$	universal quantifier
$\mid \exists x.t$	existential quantifier

This set of axioms is now a full formalization of a grounded flavor of first-order arithmetic, which we just refer to as *GA* from now on.

Formalizing GA in Pure 3

This chapter aims to translate the full formalization of *GA* in Section 2.2 into 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
lemma cons_is_list:
  assumes n_nat: "n N"
  assumes xs_list: "is_list xs"
  shows "is_list (Cons n xs)"
apply (rule eqSubst[where a="True"])
apply (unfold_def is_list_def)
apply (rule eqSym)
apply (rule condI2BEq)
apply (gd_auto)
apply (rule n_nat)
apply (rule list_nat)
apply (rule xs_list)
apply (gd_auto)
proof -
  show "(if Cons n xs = Cons n xs ∧ is_list xs ∧ (n N) then True else False) = True"
    apply (rule condI1B)
    apply (rule conjI)+
    apply (fold isNat_def)
    apply (rule cons_nat)
    apply (rule n_nat)
    apply (rule list_nat)
    apply (rule xs_list)
    apply (rule xs_list)
    apply (rule n_nat)
    apply (rule true_bool)
    done
  show "True" by (rule true)
```

```
qed
```

After:

Isabelle

```
lemma cons_is_list [gd_auto]:
  shows "n N ⇒ xs N ⇒ is_list xs ⇒ is_list (Cons n xs)"
by (unfold_def is_list_def, simp)
```

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] B. Ford, “Reasoning Around Paradox with Grounded Deduction.” [Online]. Available: https://arxiv.org/abs/2409.08243

[2] L. C. Paulson, “Isabelle: The Next 700 Theorem Provers.” [Online]. Available: https://arxiv.org/abs/cs/9301106

[3] 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.

[4] L. Paulson, T. Nipkow, and M. Wenzel, “The Isabelle Reference Manual,” 1998.