Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

# A Foundational Formalization of Grounded Arithmetic 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

24.09.2025

# 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 $\Lambda$ 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 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].

To get an intuition of the ideas of *GA*, consider again the definition $L \equiv \neg L$ and how it behaves in *GA*, after already having discussed how it behaves in a classical logic. The definition $L \equiv \neg L$ is perfectly valid in *GA*. However, when trying to assign a truth value to it, i.e. to prove either $L$ or $\neg L$, it is quickly apparent that this is not possible 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.

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

Similarly, many other inference rules in *GA* demand a *grounding* of the involved values as additional premises compared to their classical counterparts. The authors refer to these premises as *habeas quid* conditions.

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. It is not clear whether *grounded* reasoning is feasible when aiming to formalize even basic arithmetic. The *GA* formalization in the mature HOL logic enables studying meta-logical properties such as consistency. However, it is not suitable for providing *GD*/*GA* 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. The next goal is to formalize large chunks of basic arithmetic to evaluate the feasibility of GA as a foundation of mathematical reasoning and provide tooling and proof automation to try and make *grounded* reasoning as small of an inconvenience over classical reasoning as possible.

# 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**

| | |
|---|---|
| $\tau ::= \alpha$ | type variable |
| $\mid \tau \Rightarrow \tau$ | function type |
| $\mid \mathrm{prop}$ | type of propositions |

**Term Syntax**

| | |
|---|---|
| $t ::= x$ | variable |
| $\mid c$ | constant |
| $\mid t\ t$ | application |
| $\mid \lambda x :: \tau . t$ | lambda abstraction |
| $\mid t \Longrightarrow t$ | implication |
| $\mid t \equiv t$ | equality |
| $\mid \bigwedge x :: \tau . t$ | universal quantification |

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 $\sigma$, 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 $\alpha \in \text{A}$ 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 $\lambda$-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 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'**

| | |
|---|---|
| $\tau ::= \alpha$ | type variable |
| $\mid \tau \Rightarrow \tau$ | function type |
| $\mid \mathrm{prop}$ | type of propositions |
| $\mid o$ | type of object logic propositions |

**Term Syntax of Pure'**

| | |
|---|---|
| $t ::= x$ | variable |
| $\mid c$ | constant |
| $\mid t\ t$ | application |
| $\mid \lambda x :: \tau . t$ | lambda abstraction |
| $\mid t \Longrightarrow t$ | implication |
| $\mid t \equiv t$ | equality |
| $\mid \bigwedge x :: \tau . t$ | universal quantification |
| $\mid \mathrm{True}$ | `o`-typed true constant |
| $\mid t \vee t$ | `o`-typed logical or |
| $\mid \mathrm{Trueprop}\ t$ | conversion function `o` to `prop` |

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

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

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

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 |
| $\mid t \vee t$ | logical disjunction |
| $\mid t = t$ | natural-number equality |
| $\mid$ if $t$ then $t$ 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 $S(x) = x \vee x$ 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 |
| $p \to q \equiv \neg p \vee q$ | implication |
| $p \leftrightarrow q \equiv (p \to q) \wedge (q \to p)$ | 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, $\Gamma$ denotes a set of background assumptions. For completeness sake, the explicit structural rules governing this set of assumptions is listed here. Since $\Gamma$ 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**

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

$$\frac{\Gamma \vdash p}{\Gamma \vdash \neg\neg p}\ (\neg\neg\ \text{IE}) \qquad \frac{\Gamma \vdash p \quad \Gamma \vdash \neg p}{\Gamma \vdash q}\ (\neg E) \qquad \frac{\Gamma \vdash \neg(p \vee q)}{\Gamma \vdash \neg p}\ (\vee\ \text{E1})$$

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

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.**

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

■

**Grounded Implication** Impliciation is not a primitive in $GA$, but rather the shorthand $a \rightarrow b \equiv \neg a \lor b$. 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*

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

**Proof.**

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

■

**Theorem 3.** *Implication Introduction*

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

**Proof.**

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

■

**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. The axioms correspond to the inference rules that would be derivable from encoded quantifiers [1].

**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/ Pure. Formally, this means that *GA* is embedded into the *Pure* calculus, using it as a meta-logic. The typed lambda calculus *Pure* is fully characterized in Section 2.1. In the following, all types mentioned are part of the *Pure* type system. *GA* itself has no real (conventional) notion of types, although the 'inherited' types of *Pure* can also be interpreted as syntactic *GA* types.

## 3.1 Proposotional Axioms

We first declare a syntactic Isabelle type for truth values in *GA* `o` and a function to convert from `o` to the type of truth values of the *Pure* calculus `prop`, as explained in Section 2.1.4.

```
typedecl o

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

Now, we can declare constants disj and not, with the propositional axioms from Section 2.2.1.

```
 axiomatization
   disj :: ‹o ⇒ o ⇒ o›  (infixr ‹∨› 30) and
   not :: ‹o ⇒ o› (‹¬ _› [40] 40)
 where
   disjI1: ‹P ⟹ P ∨ Q› and
   disjI2: ‹Q ⟹ P ∨ Q› and
   disjI3: ‹⟦¬P; ¬Q⟧ ⟹ ¬(P ∨ Q)› and
   disjE1: ‹⟦P ∨ Q; P ⟹ R; Q ⟹ R⟧ ⟹ R› and
   disjE2: ‹¬(P ∨ Q) ⟹ ¬P› and
   disjE3: ‹¬(P ∨ Q) ⟹ ¬Q› and
   dNegI: ‹P ⟹ (¬¬P)› and
   dNegE: ‹(¬¬P) ⟹ P› and
   exF: ‹⟦P; ¬P⟧ ⟹ Q›
```
Isabelle

This immediately introduces the infix notation $a \vee b$ for disj $a$ $b$ and $\neg a$ for not $a$, where `infixr` means that the provided syntax is an infix operator that associates to the right. The precedence of the ¬ operator is 40, it thus binds stronger than the ∨ operator with precedence 30.

In general, a rule in the natural deduction style formalization from Section 2.2 of the following shape:

$$\frac{\Gamma \cup A_1 \vdash P_1 \quad \Gamma \cup A_2 \vdash P_2 \quad \dots \quad \Gamma \cup A_n \vdash P_n}{\Gamma \vdash C}$$

Is translated into the following shape in Isabelle:

```
(A1 ⇒ P1) ⇒ (A2 ⇒ P2) ⇒ ... ⇒ (An ⇒ Pn) ⇒ C
```

Or, equivalently:

```
⟦A1 ⇒ P1; A2 ⇒ P2; ...; An ⇒ Pn⟧ ⇒ C
```

Potential background assumptions Γ do not need to be explicitly managed in Isabelle. What we consider a derivation, i.e. ⊢ in *GA* is just implication $\Longrightarrow$ in Pure.

Theorems in this section that are explicitly presented in a *theorem block* keep using the natural deduction style to be consistent with Section 2.1 and Section 2.2. In this notation, deducability from assumptions is denoted ⊢, while in Isabelle the corresponding symbol is $\Longrightarrow$, i.e. meta-level implication.

## 3.2 Natural Number Axioms

We declare a type `num` for natural numbers.

```
typedecl num
```

Before the majority of natural number axioms, we define equality and some connectives derived from the primitives. We axiomatize equality with unrestricted substitution and symmetry. Equality is not just defined for `num`, but for any type `'a`, where `'a` is a generic type variable. As expected, equality is a binary infix operator associating to the left.

```
axiomatization
  eq :: ‹'a ⇒ 'a ⇒ o›  (infixl ‹=› 45)
where
  eqSubst: ‹⟦a = b; Q a⟧ ⇒ Q b› and
  eqSym: ‹a = b ⇒ b = a›
```

Transitivity can be proven using the substitution axiom.

```
lemma eq_trans: "a = b ⇒ b = c ⇒ a = c"
by (rule eqSubst[where a="b" and b="c"], assumption)
```

However, transitivity is not a very useful lemma, as it just as efficient to directly use equality substitution.

Inequality, the B and N judgments, and other logical operators can now be defined in terms of the axiomatized primitives. Notably, B is explicitly typed at $o \Rightarrow o$, meaning that e.g. the term **S**(zero) B is rejected by the Isabelle parser, as it is not well-typed. Similarly, N is explicitly typed at num $\Rightarrow o$, only accepting an argument of type num.

```
 definition neq :: ‹num ⇒ num ⇒ o› (infixl ‹≠› 45)
   where ‹a ≠ b ≡ ¬ (a = b)›
 definition bJudg :: ‹o ⇒ o› (‹_ B› [21] 20)
   where ‹(p B) ≡ (p ∨ ¬p)›
 definition isNat :: ‹num ⇒ o› (‹_ N› [21] 20)
```

```
where "x N ≡ x = x"
definition conj :: ‹o ⇒ o ⇒ o› (infixl ‹∧› 35)
  where ‹p ∧ q ≡ ¬(¬p ∨ ¬q)›
definition impl :: ‹o ⇒ o ⇒ o› (infixr ‹⟶› 25)
  where ‹p ⟶ q ≡ ¬p ∨ q›
definition iff :: ‹o ⇒ o ⇒ o› (infixl ‹⟷› 25)
  where ‹p ⟷ q ≡ (p ⟶ q) ∧ (q ⟶ p)›
```

As an example, all the introduction and elimination rules for conjunction $\wedge$ can be proven now:

```
lemma conjE1:
  assumes p_and_q: "p ∧ q"
  shows "p"
apply (rule dNegE)
apply (rule disjE2[where Q="¬q"])
apply (fold conj_def)
apply (rule p_and_q)
done

lemma conjE2:
  assumes p_and_q: "p ∧ q"
  shows "q"
apply (rule dNegE)
apply (rule disjE3[where P="¬p"])
apply (fold conj_def)
apply (rule p_and_q)
done

lemma conjI:
  assumes p: "p"
  assumes q: "q"
  shows "p ∧ q"
apply (unfold conj_def)
apply (rule disjI3)
apply (rule dNegI)
apply (rule p)
apply (rule dNegI)
apply (rule q)
done
```
Isabelle

We can now axiomatize the `zero` constant and `pred` and `suc` functions. Compared to the formalization in Section 2.2, there is an additional axiom here stating $P(0) = 0$. This will allow stating some convenient lemmas later, for example that for any $n$ with $n$ N, $P(n) \leq n$.

```
Isabelle
```

```
axiomatization
  zero :: ‹num›                           and
  suc :: ‹num ⇒ num›     (‹S(_)› [800]) and
  pred :: ‹num ⇒ num›    (‹P(_)› [800])
where
  nat0: ‹zero N› and
  sucInj: ‹S a = S b ⇒ a = b› and
  sucCong: ‹a = b ⇒ S a = S b› and
  predCong: ‹a = b ⇒ P a = P b› and
  eqBool: ‹⟦a N; b N⟧ ⇒ (a = b) B› and
  eqBoolB: ‹⟦x B; y B⟧ ⇒ (x = y) B› and
  sucNonZero: ‹a N ⇒ S a ≠ zero› and
  predSucInv: ‹a N ⇒ P(S(a)) = a› and
  pred0: ‹P(zero) = zero› and
  ind: ‹⟦a N; Q zero; ⋀x. x N ⇒ Q x ⇒ Q S(x)⟧ ⇒ Q a›
```

The following two crucial inference rules can be proven from these axioms, whose proof is given directly in the Isabelle formalization.

**Theorem 4.** *Natural Number Typing Rules*

$$\frac{\Gamma \vdash a\ \mathsf{N}}{\Gamma \vdash \mathbf{S}(a)\ \mathsf{N}}\ (\mathbf{S}\ \mathsf{N}) \qquad\qquad \frac{\Gamma \vdash a\ \mathsf{N}}{\Gamma \vdash \mathbf{P}(a)\ \mathsf{N}}\ (\mathbf{P}\ \mathsf{N})$$

**Proof.**

```
Isabelle
lemma natS:
  assumes a_nat: "a N"
  shows "S a N"
apply (unfold isNat_def)
apply (rule sucCong)
apply (fold isNat_def)
apply (rule a_nat)
done

lemma natP:
  assumes a_nat: "a N"
  shows "P a N"
apply (unfold isNat_def)
apply (rule predCong)
apply (fold isNat_def)
apply (rule a_nat)
done
```

∎

The constants True and False can now be defined as canonical truth and falsehood True $\equiv$ $(0 = 0)$ and False $\equiv$ $(S(0) = 0)$.

```
Isabelle
definition True
  where ‹True ≡ zero = zero›
definition False
  where ‹False ≡ S(zero) = zero›
```

## 3.3 Grounded Contradiction

The grounded contradiction lemma proven as a derivation tree in Section 2.2.1.1 can now be proven in the Isabelle formalization.

**Grounded Contradiction**

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

```
Isabelle
lemma grounded_contradiction:
  assumes p_bool: ‹p B›
  assumes notp_q: ‹¬p ⇒ q›
  assumes notp_notq: ‹¬p ⇒ ¬q›
  shows ‹p›
proof (rule disjE1[where P="p" and Q="¬p"])
  show "p ∨ ¬p"
    using p_bool unfolding GD.bJudg_def .
  show "p ⇒ p" by assumption
  show "¬p ⇒ p"
  proof -
    assume not_p: "¬p"
    have q: "q" using notp_q[OF not_p] .
    have not_q: "¬q" using notp_notq[OF not_p] .
    from q and not_q show "p"
      by (rule exF)
  qed
qed
```

## 3.4 Syntax Translation for Natural Numbers

The axiomatized symbols for natural numbers are of the shape ([SP]$^+$ zero) as a regex. For example, `zero` and `S(S(P zero))` are natural numbers in this GA formalization. It would be better to use the familiar base 10 system, such that the user can write a base-10 number and it is correctly interpreted as the corresponding `num` expression by Isabelle. Luckily, Isabelle provides powerful syntax translation support.

The following snippet achieves this by first declaring an implicit conversion function from numerical tokens (resulting from user input) to the natural number type `num`. The annotation `("_")` means it is applied to all such tokens implicitly. Then, an SML file called *peano_numerals.ML* provides the function *parse_gd_numeral*, which translates a number in base 10 to the `num` version (for example 3 to **S**(**S**(**S**(zero)))).

Finally, the command *parse_translation* specifies that the previously declared `_gd_num` constant performs the action specified by the *parse_gd_numeral* function.

```
Isabelle
syntax
  "_gd_num" :: "num_token ⇒ num"    ("_")

ML_file "peano_numerals.ML"

parse_translation ‹
  [(@{syntax_const "_gd_num"}, Peano_Syntax.parse_gd_numeral)]
›
```

As mentioned before, Isabelle is implemented mostly in (a dialect of) Standard ML (SML). The SML infrastructure of Isabelle is not meant to be completely abstracted away from the (advanced) user, but rather does Isabelle provide a rich API of SML functions and types, which is collectively referred to as Isabelle/ML. A syntax translation is precisely the kind of task that can be implemented in SML, hooking into the Isabelle implementation itself.

The following are the contents of the *peano_numerals.ML* file, providing the translation logic from a string representation of a base-10 number to a `num` expression.

```
SML
structure Peano_Syntax = struct

  fun nat_to_peano 0 = Syntax.const @{const_syntax "zero"}
    | nat_to_peano n = Syntax.const @{const_syntax "suc"} $ nat_to_peano (n -
    1)

  fun parse_gd_numeral _ [Free (s, _)] =
        (case Int.fromString s of
           SOME n => nat_to_peano n
         | NONE => error ("Not a numeral: " ^ s))
    | parse_gd_numeral _ _ = error "Unexpected numeral syntax"

end
```

The function `Int.fromString` is part of the Isabelle/ML API and tries to convert the input string to an (SML) `Int`. The translation from `Int` is then straightforward – `0` is translated to `zero` and `n` to **S** prepended to the recursive translation of `n-1`.

## 3.5 Quantifier Axiomatization

The quantifier axioms exactly implement the ones from the pen-and-paper formalization in Section 2.2.2, displayed again here for reference.

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

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

A crucial detail is the explicit typing of the quantifiers at the type $(\mathrm{num} \Rightarrow o) \Rightarrow o$. It makes the quantifiers range only over num, crucial as *GA* aims to be a first-order theory over the natural numbers. In a higher-order theory, the quantifiers would range over any type and be typed at $('a \Rightarrow o) \Rightarrow o$, where $'a$ is a generic type variable.

```
axiomatization                                                    Isabelle
  forall :: "(num ⇒ o) ⇒ o"  (binder "∀" [8] 9) and
  exists :: "(num ⇒ o) ⇒ o"  (binder "∃" [8] 9)
where
  forallI: "⟦⋀x. x N ⇒ Q x⟧ ⇒ ∀x. Q x" and
  forallE: "⟦∀c'. Q c'; a N⟧ ⇒ Q a" and
  existsI: "⟦a N; Q a⟧ ⇒ ∃x. Q x" and
  existsE: "⟦∃i. Q i; ⋀a. a N ⇒ Q a ⇒ R⟧ ⇒ R"
```

## 3.6 Conditional Evaluation Axiomatization

Conditional evaluation cannot seem to be derived from primitives axiomatized so far and must thus be an axiomatized primitive itself.

The syntax of the conditional operator, with the desired shape of the application cond $a\ b\ c$ being if $a$ then $b$ else $c$, seems complicated, but Isabelle is well-equipped to handle syntax like this, with the ability to specify the 'holes' in a syntax expression like in the following declaration. This type of notation is called mixfix in Isabelle, as it mixes infix and prefix notation.

Notably, the two branches of the operator and the return type are typed at the generic $'a$, indicating that the conditional operator in this formalization is polymorphic. Contrary to the pen-and-paper formalization in Section 2.2.1, this allows for returning truth values in a conditional evaluator (or any other type at that, for example functions of type $\mathrm{num} \Rightarrow \mathrm{num}$), for example the constants `True` and `False`. Although these could be encoded with natural numbers just as well, this requires equality checks of a result and is less elegant. Due to the *habeas quid* premises of the two branching axioms, making the arguments of the syntactic construct generic is not sufficient, it just means that the parser won't reject a term such as if $c$ then True else False. Since neither True $\mathsf{N}$ nor False $\mathsf{N}$ can be proven, this term cannot be 'reduced' using either `condI1` or `condI2`. Thus, there are three additional axioms for conditional evaluation over values of type `o`, mirroring the ones for conditional evaluation over `num`, but with the *habeas quid* premise of $\mathsf{B}$ instead of $\mathsf{N}$. Now, if True then True else False $=$ True can be proven using the axiom `condI1B`.

```
 consts                                                           Isabelle
```

```
  cond :: ‹o ⇒ 'a ⇒ 'a ⇒ 'a› (‹if _ then _ else _› [25, 24, 24] 24)

axiomatization where
  condI1: ‹⟦c; a N⟧ ⇒ (if c then a else b) = a› and
  condI2: ‹⟦¬c; b N⟧ ⇒ (if c then a else b) = b› and
  condT: ‹⟦c B; a N; b N⟧ ⇒ if c then a else b N› and
  condI1B: ‹⟦c; d B⟧ ⇒ (if c then d else e) = d› and
  condI2B: ‹⟦¬c; e B⟧ ⇒ (if c then d else e) = e› and
  condTB: ‹⟦c B; d B; e B⟧ ⇒ if c then d else e B›
```

## 3.7 Definitional Mechanism Axiomatization

The axioms in the formalization in Section 2.2.1 don't make it entirely clear what the definitional operator with the symbol ≡ is. It is not part of the primitive syntax, making it meta-logical, in the sense that it is outside the logical calculus. However, it is also a premise in the definitional axioms, implying it behaves like a proposition:

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

The most straightforward implementation of such a mechanism in Isabelle is to just treat it as any other logical connective. Its definition and axiomatization make it conceptually very similar to equality $=$.

Like equality, the definitional mechanism is polymorphic, allowing the left-hand side and right-hand side of the definition to be of any (albeit the same) type. The first axiom, called `defE`, is exactly the same as equality substitution, and allows the left-hand side of a definition (the *symbol*), to be substituted by its right-hand side (the *expansion*) in any context. Instead of symmetry, the second equality axiom, the definitional operator has the introduction axiom `defI`, which 'folds' a definition, i.e. replaces the right-hand side of a definition with the left-hand-side in any context.

```isabelle
axiomatization
  def :: ‹'a ⇒ 'a ⇒ o› (infix ‹:=› 10)
where
  defE: ‹⟦a := b; Q b⟧ ⇒ Q a› and
  defI: ‹⟦a := b; Q a⟧ ⇒ Q b›
```

This means that a definition $a := b$ is just another proposition of type `o`. For example is the sentence $3 = 0 \lor L := L$ a well-formed term of type `o`, as long as $L$ is a previously declared constant.

However, with the current set of axioms, a statement of the shape $a := b$ cannot possibly be proven, as there is no rule that introduces the definitional operator. However, the 'truth' of a definition can be assumed, for example in the following lemma.

```isabelle
lemma
```

```
  assumes l_def: "l := 0"
  shows "l = 0"
apply (rule defE[OF l_def])
apply (fold isNat_def)
apply (rule nat0)
done
```

The proof first unfolds the definition of `l` using the `defE` axiom, yielding the goal state $0 = 0$, which can be proven by folding the definition of N, resulting in 0 N, whose truth is postulated by the `nat0` axiom.

To get a 'globally visible' definition, the definition must be axiomatized. This does not axiomatize any properties about the defined symbol, it just axiomatizes the 'equivalence' of the left-hand side with the right-hand side, which, together with the axioms `defE` and `defI` means they can be substituted for each other in any context. The formalization of *GA* in Isabelle/HOL by the authors includes a consistency proof of the axioms given any fixed finite set of definitions [1]. Thus, axiomatizing definitions maintains consistency, as long as there is only a single definition per symbol.

## 3.8 Defining Arithmetic Functions in GA

Having axiomatized full *GA* in Isabelle/Pure, the next step is to define basic arithmetic functions and prove some lemmas about them.

The recursive definitions of the arithmetic functions are straightforward and correspond to case distinctions over the second argument (of zero and non-zero).

```
axiomatization                                                         Isabelle
  add   :: "num ⇒ num ⇒ num"  (infixl "+" 60) and
  sub   :: "num ⇒ num ⇒ num"  (infixl "-" 60) and
  mult  :: "num ⇒ num ⇒ num"  (infixl "*" 70) and
  div   :: "num ⇒ num ⇒ num"                  and
  less  :: "num ⇒ num ⇒ num"  (infix "<" 50)  and
  leq   :: "num ⇒ num ⇒ num"  (infix "≤" 50)
where
  add_def:   "add x y  := if y = 0 then x else S(add x (P y))"       and
  sub_def:   "sub x y  := if y = 0 then x else P(sub x (P y))"       and
  mult_def:  "mult x y := if y = 0 then 0 else (x + mult x (P y))"   and
  leq_def:   "leq x y  := if x = 0 then 1
                          else if y = 0 then 0
                          else (leq (P x) (P y))"                    and
  less_def:  "less x y := if y = 0 then 0
                          else if x = 0 then 1
                          else (less (P x) (P y))"                   and
  div_def:   "div x y  := if x < y = 1 then 0 else S(div (x - y) y)"
```

These definitions show the necessity of the predecessor function $\mathbf{P}$ in $GA$. The comparison functions $\leq$ and $<$ are defined to compute natural numbers, where 1 encodes truth and 0 falsehood. Note that the division function does not terminate for $y = 0$.

The functions $>$ and $\geq$ can now be defined in terms of $<$ and $\leq$.

```isabelle
definition greater :: "num ⇒ num ⇒ num" (infix ">" 50) where
  "greater x y ≡ 1 - (x ≤ y)"

definition geq :: "num ⇒ num ⇒ num" (infix "≥" 50) where
  "geq x y ≡ 1 - (x < y)"
```

A first property provable about the `less` function is that `x < 0` is always false, i.e. `x < 0 = 0`.

**Theorem 5.** *Less than zero is false*

$$\frac{}{\Gamma \vdash x < 0 = 0}$$

**Proof.**

Unfolding the definition `less_def` using the `defE` axiom results in the goal state:

1. $(\text{if } 0 = 0 \text{ then } 0 \text{ else if } x = 0 \text{ then } 1 \text{ else } \mathbf{P}(x) < \mathbf{P}(0)) = 0$

As the condition holds, applying the `condI1` axiom results in the goal state:

1. $0 = 0$
2. $0 \ \mathsf{N}$

These can both be discharged easily. The latter is the axiom `nat0` and the former is the unfolded version of this axiom, stated by the lemma `zeroRefl`.

```isabelle
lemma less_0_false: "(x < 0) = 0"
apply (rule defE[OF less_def])
apply (rule condI1)
apply (rule zeroRefl)
apply (rule nat0)
done
```

■

## 3.9 Termination Proofs

Due to the *habeas quid* premises of so many axioms, an expression like $a + b$ becomes truly useful only if $a + b \ \mathsf{N}$ is provable. With the interpretation that $a \ \mathsf{N}$ is a termination certificate for $a$, $a \ \mathsf{N} \Longrightarrow b \ \mathsf{N} \Longrightarrow a + b \ \mathsf{N}$ is essentially a termination proof of the `add` function, conditioned on its operands also being terminating natural numbers themselves.

**Theorem 6.** *Termination of* `add`

$$\frac{\Gamma \vdash x\ \mathsf{N} \qquad \Gamma \vdash y\ \mathsf{N}}{\Gamma \vdash x + y\ \mathsf{N}}$$

**Proof.**

By induction over the second argument.

```
lemma add_terminates:
  assumes x_nat: ‹x N›
  assumes y_nat: ‹y N›
  shows ‹add x y N›
proof (rule ind[where a=y])
  show "y N" by (rule y_nat)
  show "add x 0 N"
    proof (rule defE[OF add_def])
      show "if (0 = 0) then x else S(add x P(0)) N"
        apply (rule eqSubst[where a="x"])
        apply (rule eqSym)
        apply (rule condI1)
        apply (rule zeroRefl)
        apply (rule x_nat)
        apply (rule x_nat)
        done
    qed
  show ind_step: "⋀a. a N ⇒ ((x + a) N) ⇒ ((x + S(a)) N)"
    proof (rule defE[OF add_def])
      fix a
      assume a_nat: "a N" and BC: "add x a N"
      show "if (S(a) = 0) then x else S(add x P(S(a))) N"
        proof (rule condT)
          show "S(a) = 0 B"
            apply (rule eqBool)
            apply (rule natS)
            apply (rule a_nat)
            apply (rule nat0)
            done
          show "x N" by (rule x_nat)
          show "S(add x P(S(a))) N"
            apply (rule GD.natS)
            apply (rule eqSubst[where a="x+a"])
            apply (rule eqSubst[where a="a" and b="P(S(a))"])
            apply (rule eqSym)
            apply (rule predSucInv)
            apply (rule a_nat)
            apply (fold isNat_def)
          apply (rule BC)
          apply (rule BC)
          done
        qed
    qed
qed
```

■

Termination proofs of subtraction and multiplication follow the same structure, as they also recurse to the immediate predecessor in the second argument. This recursive structure exactly mirrors induction over the corresponding argument, which is why these proofs are so straightforward, despite spelling them out at the axiom level at this point.

Things get a bit more interesting with the $\leq$ function, as it recurses in both arguments. The solution is to prove a stronger lemma, which universally quantifies over one argument, and then perform induction over the other argument.

**Theorem 7.** *Termination of* `leq`

$$\frac{\Gamma \vdash x\ \mathsf{N} \quad \Gamma \vdash y\ \mathsf{N}}{\Gamma \vdash x \leq y\ \mathsf{N}}$$

**Proof.**

By induction over the second argument in the strengthened proposition $\forall x.\ x \leq y\ \mathsf{N}$.

```
lemma leq_terminates:                                         Isabelle
  shows "x N ⇒ y N ⇒ x ≤ y N"
proof -
  have H: "y N ⇒ ∀x. x ≤ y N"
  proof (rule ind[where a="y"], simp)
    show "∀x'. x' ≤ 0 N"
      apply (rule forallI)
      apply (rule defE[OF leq_def])
      apply (rule condT)
      apply (rule eqBool)
      apply (assumption)
      apply (rule nat0)
      apply (rule natS)
      apply (rule nat0)
      apply (rule eqSubst[where a="0"])
      apply (rule eqSym)
      apply (rule condI1)
      apply (rule zeroRefl)
      apply (rule nat0)+
      done
    show "∧x. x N ⇒ (∀xa. xa ≤ x N) ⇒ (∀xa. xa ≤ S(x) N)"
```

```
22    proof -
23     fix x
24     assume H: "∀xa. xa ≤ x N"
25     show "x N ⇒ ∀xa. xa ≤ S(x) N"
26      proof (rule forallI)
27       fix xa
28       show "x N ⇒ xa N ⇒ xa ≤ S(x) N"
29        apply (rule defE[OF leq_def])
30        apply (rule condT)
31        apply (rule eqBool)
32        apply (assumption)
33        apply (rule nat0)+
34        apply (rule natS, rule nat0)
35        apply (rule condT)
36        apply (rule eqBool)
37        apply (rule natS, assumption)
38        apply (rule nat0)+
39        apply (rule eqSubst[where a="x" and b="P S x"])
40        apply (rule eqSym, rule predSucInv, assumption)
41        apply (rule forallE[where a="P xa"])
42        apply (rule H)
43        apply (rule natP, assumption)
44        done
45      qed
46    qed
47  qed
48  then show "x N ⇒ y N ⇒ x ≤ y N"
49   by (rule forallE)
50 qed
```

■

Although the key ideas of the proof are straightforward – the strengthening of the proposition in line 4 and applying induction over the second argument on line 5 are only one line each – this is clouded by a lot of effort to discharge the *habeas quid* premises and other simple things like replacing a $\mathbf{P}(\mathbf{S}(x))$ with $x$ in a subexpression. The latter currently requires an appication of equality substitution, an application of equality symmetry, and then applying the `predSucInv` axiom.

This issue is tackled in Section 4, where a lot of proof automation and tactics are introduced to simplify reasoning.

The next goal is to prove termination of the division function. This poses two new challenges.

1. The function does not terminate for $y = 0$.
2. The recursive pattern is does not mirror induction, i.e. it does not recurse to the immediate predecessor.

The former is solved relatively easily. One option is to add $\neg y = 0$ as an explicit assumption to the termination proof. Another, more elegant, option is to restate the theorem in the following way: $x\ \mathsf{N} \Longrightarrow y\ \mathsf{N} \Longrightarrow \text{div}\ x\ \mathbf{S}(y)\ \mathsf{N}$. Now, it holds for any natural numbers $x$ and $y$.

To solve the second problem, a strong induction lemma needs to be proven first. This then allows assuming the induction hypothesis for any $y' \leq y$ when proving the statement for $\mathbf{S}(y)$.

The only difference to the induction axiom is that the hypothesis in the induction step is stronger – instead of $K\ x$ it is now $\bigwedge y.\ y \leq x = 1 \Longrightarrow K\ y$ (in Isabelle notation) or $y \leq x = 1 \vdash K\ y$ (in natural deduction style notation).

**Theorem 8.** *Strong Induction*

$$\frac{\Gamma \vdash a\ \mathsf{N} \qquad \Gamma \vdash K\ 0 \qquad \Gamma \cup \{x\ \mathsf{N},\ \{y\ \mathsf{N},\ y \leq x = 1\} \vdash K\ y)\} \vdash K\ \mathbf{S}(x)}{\Gamma \vdash K\ a}$$

**Proof.**

By induction over $a$ in the strengthened object-level proposition $\forall x.\ (x \leq a = 1) \longrightarrow K\ x$.

```isabelle
lemma strong_induction:
  shows "a N ⇒ Q 0 ⇒ (⋀x. x N ⇒ (⋀y. y N ⇒ y ≤ x = 1 ⇒ Q y) ⇒ (Q S(x))) ⇒ Q a"
proof -
  have q: "a N ⇒ Q 0 ⇒ (⋀x. x N ⇒ (⋀y. y N ⇒ y≤x = 1 ⇒ Q y) ⇒ (Q S(x))) ⇒
           ∀x. (x ≤ a = 1) → Q x"
    apply (rule ind[where a="a"], assumption)
    apply (rule forallI implI)+
    apply (rule eqBool, rule leq_terminates, assumption)
    apply (rule nat0, rule natS, rule nat0)
    proof -
      fix x
      show "x N ⇒ x ≤ 0 = 1 ⇒ Q 0 ⇒ Q x"
        apply (rule eqSubst[where a="0" and b="x"], rule eqSym)
        apply (rule leq_0, assumption+)
        done
      show "a N ⇒ x N ⇒
            Q 0 ⇒
            (⋀x. x N ⇒ (⋀y. y N ⇒ y ≤ x = 1 ⇒ Q y) ⇒ Q (S x)) ⇒
            ∀xa. xa ≤ x = 1 → Q xa ⇒
            ∀xa. xa ≤ (S x) = 1 → Q xa"
        apply (rule forallI implI)+
        apply (rule eqBool, rule leq_terminates, assumption)
        apply (rule natS, assumption)
        apply (rule natS, rule nat0)
        proof -
        fix xa
        assume xa_nat: "xa N"
        assume hyp: "∀x'. x' ≤ x = 1 → Q x'"
        assume step: "(∧x. x N ⇒ (∧y. y N ⇒ y ≤ x = 1 ⇒ Q y) ⇒ Q (S x))"
        assume xa_leq_sx: "xa ≤ S x = 1"
        have H: "xa ≤ x = 1 → Q xa"
          by (rule forallE[where a="xa"], rule hyp, rule xa_nat)
        show "x N ⇒ Q xa"
          apply (rule disjE1[where P="xa ≤ x = 1" and Q="¬ xa ≤ x = 1"])
          apply (fold GD.bJudg_def)
          apply (rule eqBool)
          apply (rule leq_terminates)
          apply (rule xa_nat, assumption)
          apply (rule natS, rule nat0)
          apply (rule implE[where a="xa ≤ x = 1"])
          apply (rule H)
          apply (assumption)
          proof -
            assume xa_not_leq_x: "¬ xa ≤ x = 1"
            have xa_eq_sx: "x N ⇒ xa = S x"
              apply (rule leq_suc_not_leq_implies_eq)
              apply (rule xa_nat, assumption)
              apply (rule xa_not_leq_x)
              apply (rule xa_leq_sx)
              done
            have q_sx: "x N ⇒ Q S(x)"
              apply (rule step)
              apply (assumption)
              apply (rule implE)
              apply (rule forallE)
              apply (rule hyp, assumption+)
              done
            show "x N ⇒ Q xa"
              apply (rule eqSubst[where a="S x" and b="xa"])
              apply (rule eqSym)
              apply (rule xa_eq_sx, assumption)
              apply (rule q_sx, assumption)
              done
          qed
      qed
    qed
    assume step: "(∧x. x N ⇒ (∧y. y N ⇒ y≤x = 1 ⇒ Q y) ⇒ (Q S(x)))"
    show "a N ⇒ Q 0 ⇒ Q a"
      apply (rule implE[where a="a ≤ a = 1"])
      apply (rule forallE[where Q="λx. (x ≤ a = 1) → Q x"])
      apply (rule q)
      apply (assumption+)
      apply (rule step)
      apply (assumption+)
      apply (rule leq_refl)
      apply (assumption)
      done
qed
```

■

It is necessary to use the object level connectives (i.e. $\forall$ instead of $\bigwedge$ and $\longrightarrow$ instead of $\Longrightarrow$) in the stronger proposition induction is performed over, since the induction axiom only works over expressions of type `o` (i.e. the object-level truth value type) and not of type `prop` (which the meta-level connectives $\Longrightarrow$ and $\bigwedge$ are defined on). Thus, applying the *GA* induction axiom on the corresponding meta-level proposition $\bigwedge x.\ (x \leq a = 1) \Longrightarrow K\ x$ would not work. It is simply a type error.

The idea of this proof is again very straightforward, but spelling it out using the axioms is lengthy and challenging.

Using the strong induction lemma, termination of the division function defined earlier can be proven.

**Theorem 9.** *Termination of* `div`

$$\frac{\Gamma \vdash x\ \mathsf{N} \qquad \Gamma \vdash y\ \mathsf{N}}{\Gamma \vdash \mathrm{div}\ x\ y\ \mathsf{N}}$$

**Proof.**

By strong induction over the second argument.

```
lemma div_terminates:
  shows "x N ⇒ y N ⇒ div x S(y) N"
apply (rule strong_induction[where a="x"], assumption)
apply (rule defE[OF div_def], simp)
proof -
  fix x
  assume hyp: "∧ya. ya N ⇒ ya ≤ x = 1 ⇒ (div ya (S y) N)"
  show "x N ⇒ y N ⇒ div (S x) (S y) N"
    apply (rule defE[OF div_def])
    apply (rule condT, simp)
    apply (rule hyp, simp+)
    done
qed
```

■

This proof is much shorter than the previous ones despite significantly higher complexity. The magic lies in the `simp` method, which invokes the simplifier. This theorem was proven much later than the previous termination proofs and using a lot of automation. The simplifier applies numerous previously proven lemmas here, for example to automatically solve the base case of div $0\ \mathbf{S}(y)\ \mathsf{N}$.

Section 4 goes into how automation is introduced into an axiomatized logic in *Pure*.

The authors of *GA* place a lot of emphasis on primitive recursion as a 'benchmark' for the expressivitiy of *GA*. Namely, they proved that all primitive recursive functions can be expressed and proven terminating in *GA*. While such a proof is out of reach for this formalization, it is more fitting for this formalization to show that *GA* can actually go beyond that. The Ackermann function is famously not primitive recursive [5]. With the tooling from Section 4, a termination proof of the Ackermann function is surprisingly simple to spell out in *GA*, using the standard approach of nested induction.

Consider the following standard definition of the Ackermann function in *GA*.

```isabelle
axiomatization
  ack :: "num ⇒ num ⇒ num"
where
  ack_def: "ack x y := if x = 0 then y + 1
                       else if y = 0 then ack (P x) 1
                       else ack (P x) (ack x (P y))"
```

**Theorem 10.** *Termination of* ack

$$\frac{\Gamma \vdash x\ \mathsf{N} \qquad \Gamma \vdash y\ \mathsf{N}}{\Gamma \vdash \mathsf{ack}\ x\ y\ \mathsf{N}}$$

**Proof.**

By nested induction using a helper lemma.

The outer induction ranges over the second argument and proves the stronger statement $\forall n.\ \mathsf{ack}\ m\ n\ \mathsf{N}$.

```isabelle
lemma [simp]: "n N ⇒ ack 0 n = n + 1"
by (unfold_def ack_def, simp)

lemma "n N ⇒ m N ⇒ ack m n N"
apply (rule forallE[where a="n"])
apply (induct m)
apply (rule forallI, simp)
apply (rule forallI)
proof -
  fix x z
  show "x N ⇒ ∀y. ack x y N ⇒ z N ⇒ ack (S x) z N"
    apply (induct z)
    apply (unfold_def ack_def)
    apply (subst rule: condI2, simp)
    apply (subst rule: condI1, simp+)
    apply (rule forallE[where a="1"], simp)
    apply (rule forallE[where a="1"], simp)
    apply (subst rule: condI1, simp+)
    apply (rule forallE[where a="1"], simp)
    apply (rule forallE[where a="1"], simp)
    apply (unfold_def ack_def)
    apply (subst rule: condI2, simp+)
    apply (subst rule: condI2, simp+)
    apply (rule forallE, simp)+
    apply (rule forallI, simp+)
    apply (rule forallE, simp)
    done
qed
```

∎

# Tooling for Isabelle/GA 4

Having implemented *GA* in *Pure* effectively obtained an interactive theorem prover we term *Isabelle/GA*, based on the axioms of *GA*. In its current state however, *Isabelle/GA* is not a very useful 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 using axioms or previously proven lemmas, leading to highly verbose and cumbersome proofs, as seen in Section 3.

This chapter aims for making *Isabelle/GA* more usable as a proof assistant and, towards that end, introduces various methods for simpler and cleaner reasoning, a simple auto-solver, and most importantly, compatibility with the powerful simplifier built into *Pure*. The goal is to mostly hide the axiomatic system of *GA* behind abstract proof methods found in existing theorem provers and allow proofs to focus on their main idea, rather than being about mapping to specific axioms and discharging *habeas quid* premises.

## 4.1 (Un)folding (Recursive) Definitions

The first methods we implement are the `unfold_def` and `fold_def` methods, which take the name of a definition (:=) and (un)fold it once in the current goal state.

Example usage:

```Isabelle
apply (unfold_def mult_def)
apply (fold_def mult_def)
```

This corresponds exactly to:

```Isabelle
apply (rule defE[OF mult_def])
apply (rule defI[OF mult_def])
```

The method names are intentionally similar to the existing `unfold` and `fold`, which (un)fold an (non-recursive) Isabelle definition.

The implementation in SML is as follows:

```SML
structure Unfold_Def =
struct
  fun fold_def_method thm_name ctxt =
    SIMPLE_METHOD' (fn i =>
      let
        val defI_thm = Proof_Context.get_thm ctxt "defI"
        val target_thm = Proof_Context.get_thm ctxt thm_name
      in
        CHANGED (resolve_tac ctxt [defI_thm OF [target_thm]] i)
      end)

  fun unfold_def_method thm_name ctxt =
    SIMPLE_METHOD' (fn i =>
      let
        val defE_thm = Proof_Context.get_thm ctxt "defE"
        val target_thm = Proof_Context.get_thm ctxt thm_name
      in
        CHANGED (resolve_tac ctxt [defE_thm OF [target_thm]] i)
      end)
end

val _ =
  Theory.setup
    (Method.setup @{binding unfold_def}
      (Scan.lift Args.name >> Unfold_Def.unfold_def_method)
      "Unfold a definition using defE"
  )

val _ =
  Theory.setup
    (Method.setup @{binding fold_def}
      (Scan.lift Args.name >> Unfold_Def.fold_def_method)
      "Fold a definition using defI"
  )
```

The Isabelle/ML infrastructure is well-equipped to handle such method definitions and multiple components of the provided infrastructure are visible in the snippet:

1. **Parsing**: `Args.name` parses an identifier and the `>>` combinator passes the parsed argument on to the defined method.
2. **Tactic Combinators**: `resolve_tac` simply applies the given list of theorems (in this case a singleton list) to subgoal `i` of the given `context`. `CHANGED` takes a tactic and succeeds if and only if its argument tactic changed the goal state. That is, if there was no definition to fold/unfold, the method fails, even if the theorem application itself succeeds.
3. **Method registration**: `Method.setup` sets up the defined method at the given binding. `SIMPLE_METHOD'` converts a value of type int ⇒ tactic (usually a function applying a tactic to the `i`'th subgoal, where `i` is ita argument) to a method.

## 4.2 Configuring the Simplifier

*Pure* already contains a simplifier. The rewrite rules it uses are theorems of the shape:

```Isabelle
complicated_expression ≡ simple_expression
```

That is, the *Pure* simplifier works on meta-level equations. When applied, the simplifier tries to match the left-hand side of any rewrite equation in any subexpression and rewrites it to the right-hand side of the equation.

The simplifier uses theorems tagged with [simp] as its rewrite rules. Its exact inner workings are not of interest here, but it is highly sophisticated and sublinear in the number of rewrite rules.

The first problem with using the simplifier in *GA* is that the axioms allow deriving object-level equality ($=$), but not meta-level equality ($\equiv$). Either, *GA* needs its own simplifier, or it needs to seek 'compatibility' of object equality with meta equality.

The simple solution is to add an axiom to convert object equality to meta equality:

**Equality Reflection**

$$\frac{\Gamma \vdash a = b}{\Gamma \vdash a \equiv b}$$

This proposition is not provable from either the *Pure* axioms or the *GA* axioms, which is why it needs to be axiomatized. Using it for rewrites however is completely safe, as any such rewrite can be achieved using the existing `eqSubst` axiom.

If a rewrite theorem (that is, a theorem tagged with [simp]) has premises, the simplifier only rewrites if it can discharge all its premises.

The following SML structure configures the simplifier for *GA*. It implements three functionalities:

1. Converts object equality theorems to meta equality theorems on the fly.
2. Configures a solver tactic.
3. Sets a subgoaler.

```sml
structure GD_Simp =
struct
  fun reflect_eq th: thm = th RS @{thm eq_reflection};

  fun match_object_rule th trm =
    case trm of
      Const (@{const_name GD.eq},  _) $ _ $ _ =>
        (case try reflect_eq th of SOME th' => [th'] | NONE => [])
    | _ => []

  fun mksimps _ th =
    case Thm.concl_of th of
       Const (@{const_name "Pure.eq"}, _) $ _ $ _ => [th]
     | Const (@{const_name "GD.Trueprop"}, _) $ x => match_object_rule th x
     | _ => []

  fun step_tac ctxt i =
    let
      val close = reflexive_thm :: Simplifier.prems_of ctxt
      val tac =
        assume_tac ctxt
        ORELSE' resolve_tac ctxt close
        ORELSE' GDAuto.gd_auto_tac ctxt
    in
      REPEAT_DETERM (CHANGED (tac i))
    end

  val gd_solver =
    Raw_Simplifier.mk_solver "GD_solver" step_tac
end;

let
  fun set_solver ctxt =
    Raw_Simplifier.setSolver (ctxt, GD_Simp.gd_solver)
  fun set_ssolver ctxt =
    Raw_Simplifier.setSSolver (ctxt, GD_Simp.gd_solver)
  fun configure ctxt =
    ctxt
    |> Simplifier.set_mksimps GD_Simp.mksimps
    |> Simplifier.set_subgoaler asm_full_simp_tac
    |> set_solver
    |> set_ssolver
in
  Theory.setup (Simplifier.map_theory_simpset configure)
end;
```

# 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 beyond just natural numbers. 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 take steps towards such a definitional mechanism for inductive datatypes in Isabelle/GD and encode them into the existing natural number theory. 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 into the natural numbers using the cantor pairing infrastructure from the first step. 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).
- Plan out a semantic type system embedded within the single syntactic type of `num` in *Pure*.
- Write a definitional package that parses an inductive datatype declaration and compiles it into the necessary definitions, lemmas, and accompanying proofs.

## 5.1 Next

- cantor pairing formalization; cantor pairing and projection functions recursive formalizations and mutual termination proofs; key theorems

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)
```

# 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.

[5] R. Péter, *Recursive functions in computer theory.* Ellis Horwood, 1981.