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

29.09.2025

Abstract

This thesis presents a foundational formalization of Grounded Arithmetic (GA), a first-order arithmetic based on the principles of Grounded Deduction (GD), directly within the Isabelle/Pure framework. Unlike classical and constructive logics, which impose strict termination requirements on definitions to preserve consistency, GD admits arbitrary recursion at the definitional level while weakening other inference rules to preserve its own consistency. The goal of this thesis is to investigate the feasibility of GA as a basis for mathematical reasoning by fully axiomatizing it in Pure.

The contributions of this thesis are the following. First, GA is fully axomatized in Pure, including axioms for propositional reasoning, natural numbers, quantifiers, conditional evaluation, and unrestricted definitional mechanisms. Second, core arithmetic functions are defined, their termination properties established, and many basic properties about them proved, supported by tooling written in Standard ML (SML), such as a subgoal solver, syntactic extensions, and methods for case analysis and induction. Third, the expressive power of GA is demonstrated by introducing a general framework for encoding inductive datatypes via Cantor tuples, and by establishing their fundamental properties (distinctness, injectivity, exhaustiveness, closure, and induction), which are explicitly stated and proved for the List datatype.

The resulting system, Isabelle/GA, enables interactive reasoning directly in GA with proof automation adapted to the grounded setting. While the implementation of a fully general definitional mechanism for inductive datatypes remains future work, the foundations laid in this thesis provide a robust platform for further exploration of grounded reasoning as a viable alternative to classical foundations of mathematics.

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:

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

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].

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.

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

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.

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
$\mid \text{prop}$	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 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 $\alpha \in \text{A}$ can always be derived, as can any assumption $\gamma \in \Gamma$.

Basic Rules

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

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

Implication Deduction Rules

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

Universal Quantification Deduction Rules

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

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

Equality Deduction Rules

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

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

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

Lambda Conversion Rules

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

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

Finally, the equivalence substitution rule:

Equivalence Elimination

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

2.1.4 Formalizing Object Logics in Pure

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

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

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

typedecl o

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

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

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

The axiomatized rules here simply state that True holds and that from either P or Q, $P \vee Q$ can be derived. Here, P and Q are implicitly universally quantified, ranging over all terms of type `prop`. That is, P and Q can be substituted for any term 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`.

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

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

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

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

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

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

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

Type Syntax of Pure'

$\tau ::= \alpha$	type variable
$\mid \tau \Rightarrow \tau$	function type
$\mid$ 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
$\mid$ True	o-typed true constant
$\mid t \vee t$	o-typed logical or
$\mid$ Trueprop t	conversion function o to prop

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

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

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

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

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, 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

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

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

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

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.

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

■

GROUNDED IMPLICATION Impliciation is not a primitive in *GA*, but rather the shorthand $a \to b \equiv \neg a \vee b$. 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*

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

Proof.

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

■

Theorem 3. *Implication Introduction*

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

Proof.

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

■

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

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

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
$\mid$ if t then t 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)
```

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›
```

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 $\neg$ operator is 40, it thus binds stronger than the $\vee$ operator with precedence 30.

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

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

Is translated into the following shape in Isabelle:

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

Or, equivalently:

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

Potential background assumptions Γ do not need to be explicitly managed in Isabelle. What we consider a derivation, i.e. $\vdash$ 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 $\vdash$, 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.

```
1  typedecl num
```
Isabelle

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.

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

Transitivity can be proved using the substitution axiom.

```
1  lemma eq_trans: "a = b ⇒ b = c ⇒ a = c"                           Isabelle
2  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.

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

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

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

Isabelle

```
1   lemma conjE1:
2     assumes p_and_q: "p ∧ q"
3     shows "p"
4   apply (rule dNegE)
5   apply (rule disjE2[where Q="¬q"])
6   apply (fold conj_def)
7   apply (rule p_and_q)
8   done
9
10  lemma conjE2:
11    assumes p_and_q: "p ∧ q"
12    shows "q"
13  apply (rule dNegE)
14  apply (rule disjE3[where P="¬p"])
15  apply (fold conj_def)
16  apply (rule p_and_q)
17  done
18
19  lemma conjI:
20    assumes p: "p"
21    assumes q: "q"
22    shows "p ∧ q"
23  apply (unfold conj_def)
24  apply (rule disjI3)
25  apply (rule dNegI)
26  apply (rule p)
27  apply (rule dNegI)
28  apply (rule q)
29  done
```

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, $P(n) \leq n$.

```
1   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 proved from these axioms, whose proof is given directly in the Isabelle formalization.

Theorem 4. *Natural Number Typing Rules*

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

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 $\equiv (S(0) = 0)$.

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

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

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

```
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
```
Isabelle

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 num $\Rightarrow$ 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 N nor False N can be proved, 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 B instead of N. Now, if True then True else False = True can be proved 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 $\equiv$ 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.

```
axiomatization                                                    Isabelle
  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 $3 = 0 \vee L := L$ 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.

```
lemma                                                             Isabelle
```

```
2    assumes l_def: "l := 0"
3    shows "l = 0"
4  apply (rule defE[OF l_def])
5  apply (fold isNat_def)
6  apply (rule nat0)
7  done
```

The proof first unfolds the definition of `l` using the `defE` axiom, yielding the goal state $0 = 0$, which can be proved by folding the definition of N, resulting in $0\ \mathsf{N}$, 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).

```
1  axiomatization                                                    Isabelle
2    add   :: "num ⇒ num ⇒ num"  (infixl "+" 60) and
3    sub   :: "num ⇒ num ⇒ num"  (infixl "-" 60) and
4    mult  :: "num ⇒ num ⇒ num"  (infixl "*" 70) and
5    div   :: "num ⇒ num ⇒ num"                  and
6    less  :: "num ⇒ num ⇒ num"  (infix "<" 50)  and
7    leq   :: "num ⇒ num ⇒ num"  (infix "≤" 50)
8  where
9    add_def:   "add x y  := if y = 0 then x else S(add x (P y))"       and
10   sub_def:   "sub x y  := if y = 0 then x else P(sub x (P y))"       and
11   mult_def:  "mult x y := if y = 0 then 0 else (x + mult x (P y))"   and
12   leq_def:   "leq x y  := if x = 0 then 1
13                           else if y = 0 then 0
14                           else (leq (P x) (P y))"                    and
15   less_def:  "less x y := if y = 0 then 0
16                           else if x = 0 then 1
17                           else (less (P x) (P y))"                   and
18   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 **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 $\leq$.

```
1 definition greater :: "num ⇒ num ⇒ num" (infix ">" 50) where
2   "greater x y ≡ 1 - (x ≤ y)"
3
4 definition geq :: "num ⇒ num ⇒ num" (infix "≥" 50) where
5   "geq x y ≡ 1 - (x < y)"
```
Isabelle

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`.

```
1 lemma less_0_false: "(x < 0) = 0"
2 apply (rule defE[OF less_def])
3 apply (rule condI1)
4 apply (rule zeroRefl)
5 apply (rule nat0)
6 done
```
Isabelle

■

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`

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

Proof.

By induction on the second argument.

Isabelle

```
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)
```

```
39              apply (rule BC)
40              apply (rule BC)
41              done
42          qed
43      qed
44  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 on 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 on 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 on the second argument in the strengthened proposition $\forall x.\ x \leq y\ \mathsf{N}$.

Isabelle

```
1   lemma leq_terminates:
2     shows "x N ⇒ y N ⇒ x ≤ y N"
3   proof -
4     have H: "y N ⇒ ∀x. x ≤ y N"
5     proof (rule ind[where a="y"], simp)
6       show "∀x'. x' ≤ 0 N"
7         apply (rule forallI)
8         apply (rule defE[OF leq_def])
9         apply (rule condT)
10        apply (rule eqBool)
11        apply (assumption)
12        apply (rule nat0)
13        apply (rule natS)
14        apply (rule nat0)
15        apply (rule eqSubst[where a="0"])
16        apply (rule eqSym)
17        apply (rule condI1)
18        apply (rule zeroRefl)
19        apply (rule nat0)+
20        done
21      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 on 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: $x\ \mathsf{N} \Longrightarrow y\ \mathsf{N} \Longrightarrow \mathrm{div}\ x\ \mathbf{S}(y)\ \mathsf{N}$. 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*

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

Proof.

By induction on 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 -
```

```
26            fix xa
27            assume xa_nat: "xa N"
28            assume hyp: "∀x'. x' ≤ x = 1 ⟶ Q x'"
29            assume step: "(∧x. x N ⇒ (∧y. y N ⇒ y ≤ x = 1 ⇒ Q y) ⇒ Q (S x))"
30            assume xa_leq_sx: "xa ≤ S x = 1"
31            have H: "xa ≤ x = 1 ⟶ Q xa"
32              by (rule forallE[where a="xa"], rule hyp, rule xa_nat)
33            show "x N ⇒ Q xa"
34              apply (rule disjE1[where P="xa ≤ x = 1" and Q="¬ xa ≤ x = 1"])
35              apply (fold GD.bJudg_def)
36              apply (rule eqBool)
37              apply (rule leq_terminates)
38              apply (rule xa_nat, assumption)
39              apply (rule natS, rule nat0)
40              apply (rule implE[where a="xa ≤ x = 1"])
41              apply (rule H)
42              apply (assumption)
43              proof -
44                assume xa_not_leq_x: "¬ xa ≤ x = 1"
45                have xa_eq_sx: "x N ⇒ xa = S x"
46                  apply (rule leq_suc_not_leq_implies_eq)
47                  apply (rule xa_nat, assumption)
48                  apply (rule xa_not_leq_x)
49                  apply (rule xa_leq_sx)
50                  done
51                have q_sx: "x N ⇒ Q S(x)"
52                  apply (rule step)
53                  apply (assumption)
54                  apply (rule implE)
55                  apply (rule forallE)
56                  apply (rule hyp, assumption+)
57                  done
58                show "x N ⇒ Q xa"
59                  apply (rule eqSubst[where a="S x" and b="xa"])
60                  apply (rule eqSym)
61                  apply (rule xa_eq_sx, assumption)
62                  apply (rule q_sx, assumption)
63                  done
64              qed
65          qed
66      qed
67      assume step: "(∧x. x N ⇒ (∧y. y N ⇒ y≤x = 1 ⇒ Q y) ⇒ (Q S(x)))"
68      show "a N ⇒ Q 0 ⇒ Q a"
69        apply (rule implE[where a="a ≤ a = 1"])
70        apply (rule forallE[where Q="λx. (x ≤ a = 1) ⟶ Q x"])
```

```
71        apply (rule q)
72        apply (assumption+)
73        apply (rule step)
74        apply (assumption+)
75        apply (rule leq_refl)
76        apply (assumption)
77        done
78  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 proved.

Theorem 9. *Termination of* `div`

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

Proof.

By strong induction on the second argument.

Isabelle

```
1   lemma div_terminates:
2     shows "x N ⇒ y N ⇒ div x S(y) N"
3   apply (rule strong_induction[where a="x"], assumption)
4   apply (rule defE[OF div_def], simp)
5   proof -
6     fix x
7     assume hyp: "∧ya. ya N ⇒ ya ≤ x = 1 ⇒ (div ya (S y) N)"
8     show "x N ⇒ y N ⇒ div (S x) (S y) N"
9       apply (rule defE[OF div_def])
10      apply (rule condT, simp)
11      apply (rule hyp, simp+)
12      done
13  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 (which are omitted from this document) 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 such as *GA* 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*.

```
1  axiomatization                                                    Isabelle
2    ack :: "num ⇒ num ⇒ num"
3  where
4    ack_def: "ack x y := if x = 0 then y + 1
5                         else if y = 0 then ack (P x) 1
6                         else ack (P x) (ack x (P y))"
```

Theorem 10. *Termination of* `ack`

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

Proof.

By nested induction and by using a helper lemma.

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

```
1   lemma [simp]: "n N ⇒ ack 0 n = n + 1"                           Isabelle
2   by (unfold_def ack_def, simp)
3
4   lemma "n N ⇒ m N ⇒ ack m n N"
5   apply (rule forallE[where a="n"])
6   apply (induct m)
7   apply (rule forallI, simp)
8   apply (rule forallI)
9   proof -
10    fix x z
11    show "x N ⇒ ∀y. ack x y N ⇒ z N ⇒ ack (S x) z N"
```

```
12      apply (induct z)
13      apply (unfold_def ack_def)
14      apply (subst rule: condI2, simp)
15      apply (subst rule: condI1, simp+)
16      apply (rule forallE[where a="1"], simp)
17      apply (rule forallE[where a="1"], simp)
18      apply (subst rule: condI1, simp+)
19      apply (rule forallE[where a="1"], simp)
20      apply (rule forallE[where a="1"], simp)
21      apply (unfold_def ack_def)
22      apply (subst rule: condI2, simp+)
23      apply (subst rule: condI2, simp+)
24      apply (rule forallE, simp)+
25      apply (rule forallI, simp+)
26      apply (rule forallE, simp)
27      done
28  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:

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

This corresponds exactly to:

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

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:

```
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 =
```
SML

```
13      SIMPLE_METHOD' (fn i =>
14        let
15          val defE_thm = Proof_Context.get_thm ctxt "defE"
16          val target_thm = Proof_Context.get_thm ctxt thm_name
17        in
18          CHANGED (resolve_tac ctxt [defE_thm OF [target_thm]] i)
19        end)
20  end
21
22  val _ =
23    Theory.setup
24      (Method.setup @{binding unfold_def}
25        (Scan.lift Args.name >> Unfold_Def.unfold_def_method)
26        "Unfold a definition using defE"
27    )
28
29  val _ =
30    Theory.setup
31      (Method.setup @{binding fold_def}
32        (Scan.lift Args.name >> Unfold_Def.fold_def_method)
33        "Fold a definition using defI"
34    )
```

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:

$$\texttt{complicated\_expression} \equiv \texttt{simple\_expression}$$

That is, the *Pure* simplifier works on meta-level equations. When invoked, 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 Axiom

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

This inference rule 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.

That is, the following lemma is provable in *GA*, and thus the axiom is admissible:

Theorem 11.

$$\frac{\Gamma \vdash a = b \quad \Gamma \vdash K\ b}{\Gamma \vdash K\ a}$$

Proof.

By equality substitution.

```
1 lemma "a = b ⇒ Q b ⇒ Q a"                                    Isabelle
2 apply (rule eqSubst[where a="b" and b="a"])
3 apply (rule eqSym)
4 apply (assumption+)
5 done
```

■

If the simplifier proves a theorem, having substituted a term b for a (due to a meta-equality theorem), the proof of the original theorem (with no substitution) can be constructed using the above theorem and the equality $a = b$. Since the simplifier constructs the meta-equality from exactly such an equality $a = b$, no new theorems of type `o` can be proved from this axiom.

If a rewrite theorem (that is, a theorem tagged with [simp]) has any premises, the simplifier only rewrites if it can discharge all its premises. Thus, a key step in configuring the simplifier for *GA* is to provide a competent solver that can discharge a wide range of commonly occuring premises such that the simplifier can apply more rewrites. Precisely such a solver is engineered in Section 4.3. For now, we already assume its existence at `GDAuto.gd_auto_tac` and use it in the following SML structure that configures the simplifier for *GA*. It achieves two main objectives:

1. Convert object equality theorems to meta equality theorems on the fly.
2. Set the solver for the simplifier.

```SML
structure GD_Simp =
struct
  fun convert_eq_to_meta_eq th: thm = th RS @{thm eq_reflection}

  fun match_object_rule th trm =
    case trm of
      Const (@{const_name GD.eq},  _) $ _ $ _  => [convert_eq_to_meta_eq th]
    | _ => []

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

let
  val gd_solver =
    Raw_Simplifier.mk_solver "GD_solver" GDAuto.gd_auto_tac
  fun set_solver ctxt =
    Raw_Simplifier.setSolver (ctxt, gd_solver)
  fun set_ssolver ctxt =
    Raw_Simplifier.setSSolver (ctxt, gd_solver)
  fun configure ctxt =
    ctxt
    |> Simplifier.set_mksimps GD_Simp.th_to_meta_eq_th
    |> set_solver
    |> set_ssolver
in
  Theory.setup (Simplifier.map_theory_simpset configure)
end;
```

Now, theorems with either a top-level meta equality ($\equiv$) or object equality ($=$) can be tagged with [simp] and the simplifier will use them to rewrite subexpressions when invoked.

For example when invoking the simplifier from the following goal state:

1. $a\ \mathsf{N} \Longrightarrow x\ \mathsf{N} \Longrightarrow x * \mathbf{P}(\mathbf{S}(a))\ \mathsf{N}$

```Isabelle
apply (simp)
```

The new goal state is:

1. $a\ \mathsf{N} \Longrightarrow x\ \mathsf{N} \Longrightarrow x * a\ \mathsf{N}$

The simplifier used the axiom `predSucInv` stating $a\ \mathsf{N} \Longrightarrow \mathbf{P}(\mathbf{S}(a))\ \mathsf{N} = a$ to rewrite $\mathbf{P}(\mathbf{S}(a))$ to a. This is because the `predSucInv` axiom was retroactively tagged with [simp].

4.3 A Subgoal Solver for *GA*

The main idea of the subgoal solver is very simple – keep a set of lemmas of the following structure that should always be applied if the current subgoal matches their consequent:

`simple_premise` $\Longrightarrow$ `complicated_consequent`

Since this is not an equality, it cannot be applied to a subexpression. It can only be applied like a normal theorem, when the consequent actually matches the current goal. The instantiated premise(s) then become(s) the new goal. If there is no premise, the goal is solved immediately, and if there are multiple premises, the number of subgoals increases, but they are expected to be easier to solve than the consequent.

Some example lemmas that are suitable:

```
1 lemma [auto]: "x N ⇒ ¬ S(x) = 0"                                    Isabelle
2 lemma natS [auto]: "a N ⇒ S a N"
3 lemma conjI [auto]: "p ⇒ q ⇒ p ∧ q"
4 lemma true [auto]: "True"
5 lemma true_bool [auto]: "True B"
6 lemma neq_bool [auto]: "a N ⇒ b N ⇒ (a ≠ b) B"
7 lemma if_trueI [auto]: "c ⇒ if c then True else False"
```

As can already be seen in this set of lemmas, the solver uses theorems with the [auto] tag.

The solver works in the following way.

1. Fetch the theorems with the [auto] tag using the function `Named_Theorems.get`.
2. Perform one iteration of the solver: Try to solve the current subgoal with the assumptions in context, else try to solve it with any other assumptions carried around in the simplifier context, or else try to apply any of the theorems tagged with [auto]. The iteration succeeds if and only if the goal state changed.
3. Recursively call another iteration on all new subgoals resulting from step 2, but succeed overall even if the recursive call fails. That is, the solver succeeds if at least one iteration succeeds.

The solver is extremely simple by design. It is easy to get stuck in a loop for the simplifier, which is why its subgoal solver should be simple, predictable, and most importantly, terminating. The solver tactic could be even more concise by using the `REPEAT_ALL_NEW` tactic combinator, which implements precisely the recursive structure of `solver_tac`. However, `REPEAT_ALL_NEW` is not bounded, which resulted in unpredictably occurring infinite loops of the simplifier when using it in the solver. Thus, it was replaced by this explicit recursion with a bound on the number of iterations.

The iteration bound is controlled by a global attribute in Isabelle and can be modified if necessary. The default value is 6, which has proved sufficient for all automated proofs in the GA formalization, while still maintaining acceptable performance. The `auto` solver can also be invoked directly. However, it is less powerful than the simplifier on its own, as it

does not apply the [simp] rewrite rules and does not rewrite subexpressions. When invoking `auto` directly, an optional argument can be supplied to override the iteration bound for that invocation only.

SML

```
val gd_auto_depth_limit =
  Attrib.setup_config_int @{binding gd_auto_depth_limit} (K 6)

fun TRY' tac i = TRY (tac i)

structure GDAuto =
struct
  fun uncond_rules ctxt = Named_Theorems.get ctxt @{named_theorems auto}

  fun solver_tac _    0 = K no_tac
    | solver_tac ctxt k =
      let
        val tac = assume_tac ctxt ORELSE'
                  resolve_tac ctxt (Simplifier.prems_of ctxt) ORELSE'
                  resolve_tac ctxt (uncond_rules ctxt)

        val one_iter = CHANGED o tac
        val recurse = solver_tac ctxt (k-1)
      in
        one_iter THEN_ALL_NEW TRY' recurse
    end

  fun gd_auto_tac ctxt i =
    let
      val fuel = Config.get ctxt gd_auto_depth_limit
    in
      CHANGED (REPEAT (CHANGED (solver_tac ctxt fuel i)))
    end
end

val parse_nat =
  Scan.optional (Scan.lift Parse.nat >> SOME) NONE

val _ =
  Theory.setup (
    Method.setup @{binding auto}
      (parse_nat >> (fn opt_n => fn ctxt =>
        let
          val ctxt' = (case opt_n of NONE => ctxt | SOME n => Config.put
          gd_auto_depth_limit n ctxt)
        in
          SIMPLE_METHOD' (GDAuto.gd_auto_tac ctxt')
```

```
42          end))
43        "Simple proof automation for GA"
44    )
```

There is not much to gain in trying to prune the set of rewrites at each point for the solver, since it only applies unconditional rewrites that are always the 'right choice', i.e. it doesn't perform a proof search.

The power of the solver comes not from its sophistication, but from a wealth of useful theorems that are added to its set of facts. With this simple solver (and a rich set of theorems tagged [auto] and [simp] respectively), the simplifier becomes capable of solving many kinds of subgoals entirely. For example, the termination proof of the multiplication function used to exactly mirror the one for the addition function from Section 3.9. With the new solver, almost everything can be solved by the simplifier. In particular, no manual reasoning to solve any *habeas quid* premises is required.

```
1  lemma mult_terminates [auto]:                                    Isabelle
2    shows ‹x N ⇒ y N ⇒ mult x y N›
3  proof (rule ind[where a=y])
4    show "y N ⇒ y N" by simp
5    show "x N ⇒ mult x 0 N"
6      by (unfold_def mult_def, simp)
7    fix a
8    show "a N ⇒ x N ⇒ mult x a N ⇒ mult x (S a) N"
9      by (unfold_def mult_def, rule condT, simp+)
10 qed
```

The lemma is itself immediately tagged [auto], such that the simplifier becomes even more powerful in subsequent proofs.

4.4 Conditional Rewrites

The `auto` solver exclusively applies assumptions and unconditional rewrites. However, it would be desirable to have a means to declare certain lemmas as conditional rewrites. This can mean multiple things, for example could there be a theorem that should only be applied by an automatic solver if some of its premises can be automatically solved, or a theorem should be applied entirely for proof search, i.e. it is generally uncertain whether applying it is productive towards solving the subgoal.

- An example for the former is the following lemma:

  ```
  1  lemma "¬c ⇒ b N ⇒ d N ⇒ b = d ⇒ (if c then a else b) = d"    Isabelle
  ```

  If the current goal is of the shape of the consequent and the first three premises can be solved, the goal should be reduced to the fourth premise, $b = d$. Solving only the first premise is not sufficient, since b and d could be of type `o`, in which case b N and d N would not be solvable and the following lemma should have been applied instead:

  ```
  1  lemma "¬c ⇒ b B ⇒ d B ⇒ b = d ⇒ (if c then a else b) = d"    Isabelle
  ```

- A simple example for the latter kind would be the `eqSym` axiom. A goal might be solvable after applying it, but it generally does not simplify the goal:

```
1 eqSym: "a = b ⇒ b = a"
```
Isabelle

4.4.1 Using `simp` for conditional rewrites

The first approach towards conditional rewrites is to use the simplifier. Since it only applies a rewrite if it can solve all premises, it is a natural choice for conditional rewriting. It is also highly efficient, much more so than any custom solution anyone could hope to implement single handedly.

This approach works very well for lemmas like the following:

```
1 lemma [simp]: "c ⇒ (if c then True else b) = True"
2 lemma [simp]: "¬c ⇒ (if c then a else True) = True"
3 lemma suc_pred_inv [simp]: "x N ⇒ ¬ x = 0 ⇒ S P x = x"
```
Isabelle

The amount of complexity that can be moved into solving a premise is arbitrary. For example does the simplifier try to rewrite $\mathbf{P}(x) \leq \mathbf{P}(y)$ to 1 if the premises x N, y N, and $x \leq y = 1$ can be solved by the `auto` solver (and recursive simplification on the subgoals) when trying to apply the following lemma. However, $x \leq y = 1$ might be just as difficult to solve as $\mathbf{P}(x) \leq \mathbf{P}(y) = 1$.

```
1 lemma leq_monotone_pred [simp]:
2   assumes x_nat: "x N"
3   assumes y_nat: "y N"
4   assumes H: "x ≤ y = 1"
5   shows "P x ≤ P y = 1"
```
Isabelle

There are some key limitations to note for using the simplifier for conditional rewrites:

1. Adding too many rules like `leq_monotone_pred` (with low probability of the solver succeeding in solving its premises) to the simpset makes the performance of the simplifier deteriorate quickly, as it essentially ends up performing a proof search.
2. If the premise and consequent can have the same instantiation, the simplifier **will** run into an infinite loop. For instance, tagging the `eqSym` axiom with [simp] makes it enter an infinite loop on every invocation immediately.
3. The simplifier always tries to solve all premises and does not support reducing the goal to one or more premises (to be solved manually or by another tactic).
4. The simplifier operates on equalities only. While this is largely solved by the `auto` solver, it can only do unconditional applications.

Point (1) suggests moving low-probability-of-success rewrites to a different tool, but points (2), (3), and (4) ask for a different approach altogether.

4.4.2 Extending the auto Solver with Conditional Rewrites

Problem (4) asks for an extension of the `auto` solver to also handle conditional rewrites, which also solves problem (3). Thus, the next approach was to add another tag [cond] for lemmas intended to be used as conditional rewrites and let the `auto` solver use such a tagged theorem for rewriting if it can solve its first premise. Since any number of premises can be 'encoded' into the first premise by simply conjoining them ($\wedge$), conditioning on solving the first premise is sufficient.

Some lemmas that can be handled using this approach and not by the simplifier:

```
lemma [cond]: "a ⇒ a B"
lemma [cond]: "¬a ⇒ a B"
lemma [cond]: "((c B) ∧ (a N) ∧ (b N)) ⇒ if c then a else b N"
lemma [cond]: "((c B) ∧ (a B) ∧ (b B)) ⇒ if c then a else b B"
```
Isabelle

Especially the booleanness conditions a B are vital to solve effectively in the subgoal solver of the simplifier, since they are common *habeas quid* premises in *GA*. The first two lemmas effectively do a proof search and try to solve for both the positive and negative case.

The last two lemmas are equivalent to the axioms `condT` and `condTB`, just with the three premises conjoined into one, such that the `auto` solver only applies them if all three are solved. These lemmas show the limitations of using exclusively the simplifier for automation, since this lemma cannot be expressed as an equality. One would expect that the following lemmas could be used by the simplifier:

```
lemma [simp]: "c B ⇒ a N ⇒ b N ⇒ if c then a else b N = True"
lemma [simp]: "c B ⇒ a B ⇒ b B ⇒ if c then a else b B = True"
```
Isabelle

However, due to the extraordinarily syntactic nature of equality in *GA*, these propositions are not provable. `True` is defined as the canonical truth $0 = 0$ in *GA* and $\mathrm{True} = a$ cannot be derived for any a, as there is no axiom that allows deriving equality of equalities (in particular, no reflexivity axiom).

With these lemmas presented so far, the approach is the following. Previously, the `auto` solver was designed to apply a theorem if it is the single canonical way to solve a goal of a certain shape. The conditional extension now added the capability of trying multiple different solution strategies to definitely solve a goal. For examle, if premise c can be solved, the goal c B is solved completely and no new subgoals are generated. There is however another category of lemmas that can be integrated with the conditional extension to the `auto` solver. Namely, trying different solution strategies to reduce the current goal to a new subgoal that is likely easier to solve. For example the following lemmas implement this strategy for reducing if c then a else b constructs:

```
lemma [cond]: "(¬c ∧ (b N) ∧ (d N)) ⇒ b = d ⇒ (if c then a else b) =
d"
lemma [cond]: "(c ∧ (a N) ∧ (d N)) ⇒ a = d ⇒ (if c then a else b) = d"
```
Isabelle

With this extension, the simplifier gets an even more competent subgoal solver, providing another boost to proof automation. Problems (3) and (4) of the initial approach of using only the simplifier are now solved and the two approaches coexist.

The additions in the `auto` implementation are straightforward. The required additions are highlighted:

```SML
val gd_auto_depth_limit =
  Attrib.setup_config_int @{binding gd_auto_depth_limit} (K 6)

fun TRY' tac i = TRY (tac i)

structure GDAuto =
struct

  fun uncond_rules ctxt = Named_Theorems.get ctxt @{named_theorems auto}
  fun cond_rules   ctxt = Named_Theorems.get ctxt @{named_theorems cond}

  fun solver_tac _    0 = K no_tac
    | solver_tac ctxt k =
      let
        fun apply_and_solve_subgoal i th =
          match_tac ctxt [th] i
          THEN SOLVED' (solver_tac ctxt (k-1)) i
        fun cond_tacs i =
          FIRST (map (apply_and_solve_subgoal i)
            (cond_rules ctxt))
        val tac = assume_tac ctxt ORELSE'
                  resolve_tac ctxt (Simplifier.prems_of ctxt) ORELSE'
                  resolve_tac ctxt (uncond_rules ctxt) ORELSE'
                  cond_tacs
        val one_iter = CHANGED o tac
        val recurse = solver_tac ctxt (k-1)
      in
        REPEAT_ALL_NEW one_iter
    end

  fun gd_auto_tac ctxt i =
    let
      val fuel = Config.get ctxt gd_auto_depth_limit
    in
      CHANGED (REPEAT (CHANGED (solver_tac ctxt fuel i)))
    end
end

val parse_nat =
```

```
40    Scan.optional (Scan.lift Parse.nat >> SOME) NONE
41
42  val _ =
43    Theory.setup (
44      Method.setup @{binding auto}
45        (parse_nat >> (fn opt_n => fn ctxt =>
46          let
47            val ctxt' = (case opt_n of NONE => ctxt | SOME n => Config.put
              gd_auto_depth_limit n ctxt)
48          in
49            SIMPLE_METHOD' (GDAuto.gd_auto_tac ctxt')
50          end))
51        "Simple proof automation for GD logic"
52    )
```

There is one massive problem with this `cond` extension however, namely that problem (1) of the simplifier approach is made even worse here. The `auto` solver is the subgoal solver of the simplifier and it now performs quite a bit of proof search, and a completely unoptimized one at that, which makes the performance of the simplifier suffer.

4.4.3 Circumventing Weak Equality

The automation boost the `cond` extension from the previous Section 4.4.2 provides is satisfactory, but it slowed down the simplifier significantly. Although computationally the same problem, moving the rudimentary proof search capabilities of the `cond` extension from the `auto` solver into the simplifier itself should be much faster, simply due to its (presumably) more efficient implementation.

Although an equality like the following (which the simplifier could use as a rewrite rule) does not hold in *GA*:

```
1  lemma "¬c ⇒ b N ⇒ d N ⇒ ((if c then a else b) = d) = (b = d)"   Isabelle
```

The following proposition is provable:

```
1  lemma "¬c ⇒ b N ⇒ d N ⇒ (if c then a else b) = d ↔ b = d"   Isabelle
```

Adding a conversion axiom from $\leftrightarrow$ to $\equiv$ makes such lemmas usable as rewrites for the simplifier.

Iff Reflection Axiom

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

If-and-only-if ($\leftrightarrow$) is effectively an equality for propositions of type `o`, since object equalities ($=$) are not derivable for them (only for expressions of type `num`). It is unclear whether it is admissible. However, the following theorem shows that, while for admissibility the theorem

$a \leftrightarrow b \Longrightarrow K\ b \Longrightarrow K\ a$ would be required, the following inference rule that very closely resembles the axiom itself is readily provable in *GA*.

Theorem 12. *Iff Trueprop Reflection*

$$\frac{\Gamma \vdash a \leftrightarrow b}{\Gamma \vdash (\text{Trueprop}\ a) \equiv (\text{Trueprop}\ b)}$$

Proof.

Using the `equal_intr_rule` *Pure* axiom, derive both $a \Longrightarrow b$ and $b \Longrightarrow a$.

```
lemma "a ↔ b ⇒ (Trueprop a) ≡ (Trueprop b)"
apply (rule Pure.equal_intr_rule)
apply (unfold iff_def)
apply (rule implE, rule conjE1, assumption+)
apply (rule implE, rule conjE2, assumption+)
done
```
Isabelle

■

The same proof does not work when trying to prove `iff_reflection`, since the *Pure* `equal_intr_rule` only works for meta equalities ($\equiv$) where both sides are of type `prop`, whereas in `iff_reflection`, the two sides are of type `o`.

The proof of the theorem shows that the `iff_reflection` axiom is essentially an object-level version of the meta level `equal_intr_rule`, which allows deriving equality $\equiv$ when $a \Longrightarrow b$ and $b \Longrightarrow a$ can be derived. While the simplifier could in principle work with lemmas of the shape Trueprop $a \equiv$ Trueprop b, the left-hand side never matches any subterm in a *GA* expression, as it is of type `prop`, while *GA* terms are of type `o` or `num`. The admissibility of the axiom might be provable meta-logically by induction on the inference rules of *GA*, but *Pure* is not sufficiently strong as a meta-logic to do so, as it does not have an (meta-level) induction scheme.

The new axiom allows restating most of the lemmas previously tagged with [cond], such that the simplifier can rewrite them. An example is the following lemma:

```
lemma [cond]: "a < b = 1 ⇒ a N ⇒ b N ⇒ ¬ a = b"
```
Isabelle

Which was subsequently restated and reproven as:

```
lemma [simp]: "a < b = 1 ⇒ a N ⇒ b N ⇒ ¬ a = b ↔ True"
```
Isabelle

In fact, this is more powerful, since even subexpressions can be rewritten using the latter.

While all lemmas intended for conditional rewriting can be reformulated as biconditional, not all such reformulated propositions can actually be proven. The reason is the *habeas quid* premise of the derivable implication introduction rule in *GA*, which requires proving the antecedent of the implication boolean (B). This is not possible for example in the following lemma:

```
lemma [cond]: "((c B) ∧ (a N) ∧ (b N)) ⇒ if c then a else b N"
```
Isabelle

While it can be restated as:

```
lemma [simp]: "((c B) ∧ (a N) ∧ (b N)) ⇒ if c then a else b N ↔
True"
```
Isabelle

Proving this proposition requires showing that (if c then a else b N) B, which is equivalent to solving the halting problem in *GA*, as it requires proving that the proposition a N is decidable in general. Thus, rewrites of this shape remain in the domain of the `cond` extension of the `auto` solver.

The configuration of the simplifier needs to be extended in order for it to correctly apply biconditional theorems. The minimal additional logic is hightlighted in green:

```
structure GD_Simp =
struct
  fun convert_eq_to_meta_eq th: thm = th RS @{thm eq_reflection}
  fun convert_iff_to_meta_eq th: thm = th RS @{thm iff_reflection}

  fun match_object_rule th trm =
    case trm of
      Const (@{const_name GD.eq},  _) $ _ $ _  => [convert_eq_to_meta_eq th]
    | Const (@{const_name GD.iff},  _) $ _ $ _ => [convert_iff_to_meta_eq th]
    | _ => []
...
```
SML

4.4.4 Proof Search

The initial approach of using the simplifier to handle conditional rewrites maybe unsurprisingly ended up being the most effective solution, although it did take an extension to make better use of it. The reason is its sophistication and efficient implementation, which makes it hard to beat even though it is not designed to be used for even rudimentary proof search. An approach that might provide additional value over the setup constructed in this section is a true proof search (implemented on top of the simplifier, and not as its subgoal solver). However, since a naive implementation is exponential in the number of theorems in the rewrite set, this requires a sophisticated datastructure to search for applicable rules in order to be tractable at the very least. This might be a valuable addition to the tooling of *GA*, but has not been implemented yet for two reasons.

- An efficient implementation is a significant time investment and seemed out of the scope for this thesis.
- The automation with the existing setup was surprisingly effective and a real proof search for the sake of better proof automation never seemed exceptionally desirable.

4.5 Manual Substitution

Finally, if the simplifier is not powerful enough to solve all premises of a theorem that would actually be the right one to apply, a simple substitution command that allows rewriting

(including subexpressions) with a specific theorem and solving some of its premises manually is very helpful.

For example, consider the following goal state reached by an invocation of the simplifier.

1. $x\ \mathsf{N} \Longrightarrow x'\ \mathsf{N} \Longrightarrow (\text{if } \mathbf{S}(x') =$
   $0 \text{ then } 0 \text{ else if cpx } x' = 0 \text{ then } \mathbf{S}(\text{cpy } x') \text{ else } \mathbf{P}(\text{cpx } x'))\ \mathsf{N}$

Another invocation of `simp` fails.

```
1 apply (simp)
```
Isabelle

The simplifier is not able to take the second branch of the top-level conditional, since to do this rewrite, it would have to show that the second branch terminates, i.e. show that it is N, which it is unable to do.

Now, a command like the following would come in very handy:

```
1 apply (subst rule: condI2)
```
Isabelle

This performs the rewrite and leaves the unsolved premises as new subgoals, resulting in the goal state:

1. $x\ \mathsf{N} \Longrightarrow x'\ \mathsf{N} \Longrightarrow (\text{if cpx } x' = 0 \text{ then } \mathbf{S}(\text{cpy } x') \text{ else } \mathbf{P}(\text{cpx } x'))\ \mathsf{N}$
2. $x\ \mathsf{N} \Longrightarrow x'\ \mathsf{N} \Longrightarrow (\text{if cpx } x' = 0 \text{ then } \mathbf{S}(\text{cpy } x') \text{ else } \mathbf{P}(\text{cpx } x'))\ \mathsf{N}$

Here, the same goal appears twice, since the premise of the `condI2` rule, to show the second branch terminates, precisely coincides with the remaining goal after substituting the second branch for the entire conditional.

Without the `subst` command, the `condI2` theorem couldn't have been applied directly, since the substituted conditional is a subexpression; it would have had to be preceded by an equality substitution and an application of equality symmetry first (which is more or less what the `subst` command does). Thus, the `subst` command is a convenient way of manually advancing the proof when the simplifier gets stuck.

In fact, more occurences of `eqSubst` applications can be replaced by a more convenient method. Consider the following goal state.

1. $x\ \mathsf{N} \Longrightarrow x'\ \mathsf{N} \Longrightarrow \neg(\mathbf{P}(x') = \mathbf{P}(\mathbf{S}(x))) \Longrightarrow \neg(\mathbf{P}(x') = x)$

So far, this was solved the following way:

```
1 apply (rule eqSubst[where a="P S x" and b="x"])
2 apply (simp)
```
Isabelle

Where the simplifier solves the first resulting subgoal after the equality substitution $\mathbf{P}(\mathbf{S}(x)) = x$ automatically and then the second subgoal by assumption.

The `eqSubst` axiom itself should be hidden and the rewrite should be done automatically with a simple:

```
1 apply (subst "P S x = x")
```
Isabelle

This is not a huge difference, but it makes the proof cleaner.

The `subst` command thus has two different modes of operating, one accepting a theorem name (which is expected to have a top-level equality) and using it to rewrite a match for the left-hand side to the right-hand side, concluding the first subgoal with the theorem itself, and the other accepting an equality itself, which it uses to rewrite, and then tries to solve using the simplifier.

This is implemented in the following SML code, which works as follows:

1. Parses either a theorem name or a term, whichever succeeds.
2. If the argument is a theorem name, it fetches the theorem and extracts the left-hand side and right-hand side of its top-level equality (assuming it is of the expected shape), instantiates the `eqSubst` theorem with those extracted terms, and then finally applies it. Then, it applies the `eqSym` axiom to flip the equality from the first subgoal, which can then be resolved by the passed theorem itself.
3. If the argument is a term, the code extracts the left-hand side and right-hand side of its top-level equality (assuming it is of the expected shape), instantiates the `eqSubst` theorem with those extracted terms, and then finally applies it. Then, it tries to solve the resulting first subgoal (the equality itself) by applying the simplifier.

SML

```
datatype input = AsThm of thm | AsTrm of term

structure GD_Subst =
struct
  fun strip_asms t =
    case t of
      @{term "(==>)"} $ _ $ t' => strip_asms t'
    | _ => t

  fun get_lhs_rhs_of_eq t =
      case t of
        @{term "(Trueprop)"} $ t'                  => get_lhs_rhs_of_eq t'
      | Const (@{const_name GD.eq}, _) $ lhs $ rhs => (SOME lhs, SOME rhs)
      | _                                          => (NONE, NONE)

  fun get_eq (AsTrm t)   = let val (l, r) = get_lhs_rhs_of_eq t in (r, l) end
    | get_eq (AsThm thm) = get_lhs_rhs_of_eq (strip_asms (Thm.prop_of thm))

  fun eq_subst_tac input ctxt =
    case (get_eq input) of
      (SOME pat, SOME rhs) =>
        let
          val l = (Thm.cterm_of ctxt pat)
          val r = (Thm.cterm_of ctxt rhs)
          val eqSub = Proof_Context.get_thm ctxt "eqSubst"
          val eqSub' =
            Drule.infer_instantiate' ctxt
            [SOME r, SOME l]
```

```
29              eqSub
30          in
31            resolve_tac ctxt [eqSub']
32          end
33        | (_,_) => K no_tac
34
35    fun gd_subst_tac (AsThm thm) ctxt =
36      let val eqSym = Proof_Context.get_thm ctxt "eqSym" in
37        (eq_subst_tac (AsThm thm) ctxt) THEN'
38        resolve_tac ctxt [eqSym] THEN'
39        resolve_tac ctxt [thm]
40      end
41    | gd_subst_tac (AsTrm trm) ctxt =
42      (eq_subst_tac (AsTrm trm) ctxt) THEN'
43      (fn i => TRY (SOLVED' (Simplifier.asm_full_simp_tac ctxt) i))
44
45  end
46
47  val parse_subst_args : input context_parser =
48    (Scan.lift (Args.$$$ "rule" |-- Args.colon) |-- Attrib.thm >> AsThm)
49    || (Args.term >> AsTrm)
50
51  val _ =
52  Theory.setup
53      (Method.setup @{binding subst}
54      (parse_subst_args >>
55        (fn inp => fn ctxt => SIMPLE_METHOD' (GD_Subst.gd_subst_tac inp ctxt)))
56      "Substitute using the given theorem name or term."
57  )
```

4.6 Case Distinction

What is missing so far is a good way to do case distinction over natural numbers and truth values. The foundation for these case distinctions is given by the following two lemmas.

Theorem 13. *Cases Bool*

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

Proof.

By the 'case distinction' axiom `disjE1` using the fact that $q \vee \neg q$ by $q\ \mathsf{B}$.

Isabelle

```
1  lemma cases_bool:
2    assumes q_bool: "q B"
```

```
3     assumes H: "q ⇒ p"
4     assumes H1: "¬q ⇒ p"
5     shows "p"
6   apply (rule disjE1[where P="q" and Q="¬q"])
7   apply (fold bJudg_def)
8   apply (rule q_bool)
9   apply (rule H)
10  apply (assumption)
11  apply (rule H1)
12  apply (assumption)
13  done
```

■

Theorem 14. *Cases Nat*

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

Proof.

By the 'case distinction' axiom `disjE1` using the fact that $x = 0 \vee \neg(x = 0)$ and using the helper lemma `num_nonzero`, stating that $a\ \mathsf{N} \Longrightarrow \neg(a = 0) \Longrightarrow \exists x.\ a = \mathbf{S}(x)$

Isabelle

```
1   lemma cases_nat: "x N ⇒ (x = 0 ⇒ Q 0) ⇒ (∧y. y N ⇒ x = S(y) ⇒ Q
    S(y)) ⇒ Q x"
2   apply (rule disjE1[where P="x = 0" and Q="¬ x = 0"])
3   apply (fold bJudg_def, simp)
4   apply (subst "0 = x", assumption)
5   apply (rule existsE[where Q="λc. x = S(c)"])
6   apply (rule num_nonzero)
7   proof -
8     fix a
9     show "(∧y. y N ⇒ x = S y ⇒ Q (S y)) ⇒ a N ⇒ x = S a ⇒ Q x"
10     by (subst "S a = x", assumption)
11  qed
```

■

Using these theorems, the job of the `cases` method is very simple – parse the argument and decide which theorem to apply:

SML

```
1   datatype input = BoolCaseTac of term | NatCaseTac of term
2
3   structure GDCases =
4   struct
5
6   fun try_inst_thm ctxt t th =
```

```
7     let val ct = Thm.cterm_of ctxt t in
8       try (fn th => Drule.infer_instantiate' ctxt [SOME ct] th) th
9     end
10
11  fun gd_bool_cases_tac ctxt x =
12    case (try_inst_thm ctxt x @{thm cases_bool}) of
13      SOME th => resolve_tac ctxt [th]
14    | NONE    => K no_tac
15
16  fun gd_nat_cases_tac ctxt x =
17    case (try_inst_thm ctxt x @{thm cases_nat}) of
18      SOME th => resolve_tac ctxt [th]
19    | NONE    => K no_tac
20
21  fun gd_cases_tac ctxt input =
22    case input of
23      BoolCaseTac t   => SIMPLE_METHOD' (gd_bool_cases_tac ctxt t)
24    | NatCaseTac t   => SIMPLE_METHOD' (gd_nat_cases_tac ctxt t)
25
26  val parse_cases_args : input context_parser =
27    (Scan.lift (Args.$$$ "bool" |-- Args.colon) |-- Args.term >> BoolCaseTac)
28    || (Args.term >> NatCaseTac)
29
30  val _ =
31    Theory.setup
32      (Method.setup @{binding cases}
33        (parse_cases_args >> (fn inp => fn ctxt => gd_cases_tac ctxt inp))
34        "case analysis")
35
36  end
```

4.7 Induction Method

Although there is an induction axiom, this is another opportunity to make *GA* consistent with the sorts of commands/methods a user would expect of a proof assistant and provide a method that wraps the axiom application.

Instead of applying the axiom:

```
1  apply (rule ind[where a="x"])                                    Isabelle
```

The `induct` method is applied:

```
1  apply (induct x)                                                 Isabelle
```

And instead of the strong induction lemma:

```
apply (rule strong_induction[where a="x"])
```
Isabelle

A flag can be passed to the `induct` method:

```
apply (induct strong x)
```
Isabelle

For much of this thesis, Isabelle proofs were presented in apply-style scripts. However, Isabelle also provides a structured proof language called *Isar*, which aims for emulating natural language proofs and overall better legibility compared to apply-style scripts. One nice feature of *Isar* are named cases; The premises of a theorem can be named and when applying it, the subgoals (and their respective assumptions) are bound to a case name that can be invoked with the following general syntax:

```
proof (method_name)
  case (case_1_name case_1_arg_1 ...)
    show ?case
    ...
next
  case (case_2_name case_2_arg_1 ...)
    show ?case
    ...
...
qed
```
Isabelle

For example, the goal is to be able to provide the following syntax for applying induction:

```
proof (induct y)
  case Base
    show case?
    ...
next
  case (Step xa)
    ...
qed
```
Isabelle

Precisely this functionality is provided by the following SML implementation of the `induct` method:

```
structure GD_Induct =
struct
  val induct_thm = @{thm ind}
  val strong_induct_thm = @{thm strong_induction}

  fun try_inst_thm ctxt t th =
    let val ct = Thm.cterm_of ctxt t in
      try (fn th => Drule.infer_instantiate' ctxt [SOME ct] th) th
    end

```
SML

```
11    fun closes_first_prem ctxt i th st =
12      let
13        val tac =
14        DETERM (
15          resolve_tac ctxt [th] i
16          THEN ((SOLVED' (assume_tac ctxt)) i)
17        )
18      in
19        Option.isSome (Seq.pull (tac st))
20      end
21
22    fun apply_tac tac st =
23      let
24        val res = DETERM tac st
25      in
26        case Seq.pull res of
27          SOME (st', _) => st'
28        | NONE => raise THM ("tactic failed", 0, [st])
29      end
30
31    fun induct_tac strong t =
32      CONTEXT_SUBGOAL (fn (_, i) => fn (ctxt, st) =>
33      let
34        val th  = if strong then strong_induct_thm else induct_thm
35        val th' = try_inst_thm ctxt t th
36        val tac =
37          case th' of
38            SOME th'' => DETERM (match_tac ctxt [th''] i)
39          | NONE      => no_tac
40        val st' = apply_tac tac st
41        val (spec, _) = Rule_Cases.get th
42        val cases_prop = Thm.prop_of (Rule_Cases.internalize_params st')
43        val cases = Rule_Cases.make_common ctxt cases_prop spec
44        val post_tac = TRY (SOLVED' (assume_tac ctxt) i)
45      in
46        CONTEXT_CASES cases post_tac (ctxt, st')
47      end)
48
49    fun gd_induct_method (strong, t) _ =
50      Method.CONTEXT_METHOD (K (induct_tac strong t 1))
51  end
52
53  val parse_induct_args =
54    Scan.lift (Scan.optional ((Args.$$$ "strong") >> K true) false)
55    -- Args.term
```

```
56
57 val _ =
58   Theory.setup
59     (Method.setup @{binding induct}
60     (parse_induct_args >> GD_Induct.gd_induct_method)
61     "Apply rule ind with where a = <term>"
62   )
```

4.8 A Case Study: Proving Strict Monotonicity of `cpy`

The following case study reviews the tooling and automation introduced in this section by presenting a proof of strict monotonicity for the `cpy` function, which extracts the second component of a *Cantor pair* and is central in Section 5.

The `cpy` function itself is not important here, but the proof is a great example of how the few methods introduced in this section make up the majority of commands used in a proof and are a huge step up compared to the axiom level reasoning required before this chapter.

Isabelle

```
1  lemma cpy_strict_mono [simp]: "x N ⇒ cpy (S x) < (S x) = 1"
2  proof (induct strong x)
3    case Base
4      from Base show ?case
5        by (unfold_def cpy_def, simp)
6  next
7    case (Step xa)
8      fix y
9      assume hyp: "∧y. y N ⇒ y ≤ xa = 1 ⇒ cpy (S y) < (S y) = 1"
10     from Step show ?case
11       apply (unfold_def cpy_def, simp)
12       apply (cases bool: "cpx (S xa) = 0")
13       apply (simp add: cpx_suc)+
14       apply (cases bool: "cpx xa = 0")
15       apply (simp add: cpx_suc cpy_suc)+
16       apply (subst "S P xa = xa", simp)
17       apply (rule cpx_nz_arg_nz, simp)
18       apply (rule le_monotone_suc)+
19       apply (rule hyp, simp)
20       done
21 qed
```

Encoding Inductive Datatypes in GA 5

With *Isabelle/GA* now being a 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* or Rocq, contain powerful definitional mechanisms that allow for straightforward specification of things like inductive datatypes, recursive predicates, infinitary sets, and so on.

These definitional packages are effectively *theory compilers*, as they take a 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 the key steps towards such a definitional mechanism for inductive datatypes in *Isabelle/GA* 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 the key 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 on the datatype, and so on).
- Plan out a semantic type system consisting of encoded types embedded within the single syntactic type of `num` in *Pure* and introduce tooling for it.
- Write a definitional package that parses an inductive datatype declaration and compiles it into the necessary definitions, lemmas, and accompanying proofs.

5.1 Inductive Datatypes In General

In general, an inductive datatype is specificed by a list of constructors, where each constructor has a finite number of arguments (possibly zero), each constrained by a type (which may itself be an inductive datatype, and in particular may be the datatype currently being defined). The datatype itself is then given by the least fixed point of the monotone operator that closes a set under these constructors [6].

For example, the following is an inductive definition of a list datatype:

```
datatype List =                                                    Pseudo
  Nil
| Cons Nat List
```

There are two constructors, one called `Nil` with no arguments (i.e. a constant), and one called `Cons` with two arguments, one of type `Nat` and one of type `List` itself. The set of `List`s is the least fixed point of the operator that closes under these two constructors. Intuitively,

this least fixed point is the limit of successive approximations; starting with the empty set, the first closure step adds `Nil`, the next adds all `List`s of the form `Cons n Nil`, the next adds all lists of the form `Cons n (Cons m Nil)`, and so on, eventually producing all finite lists.

An inductive datatype defined in this way satisfies the following properties:

- **Closure** (generation): applying a constructor to arguments (that are valid elements of their respective types) yields a valid element of the type, e.g. $n\ \mathsf{N} \Longrightarrow$ is_list xs $\Longrightarrow$ is_list (Cons n xs) and is_list Nil.
- **Exhaustiveness**: every element of the datatype must be built from some constructor; there are no "extra" elements beyond the closure.
- **Distinctness**: different constructors build different elements, e.g. Nil $\neq$ Cons n xs for any n, xs.
- **Injectivity**: each constructor is injective in its arguments, e.g. Cons n xs = Cons n xs $\Longrightarrow$ $n = m \wedge$ xs = ys.
- **Induction principle**: properties of elements of the datatype can be proved by showing they hold for each constructor case, assuming the property for recursive arguments.

The goal now is to find an encoding of an inductive datatype into the natural numbers such that all these properties are fulfilled and can be proved in *GA* itself without adding any axioms.

5.2 Encoding: Constructors

The constructor encoding is responsible for ensuring the latter three properties **distinctness**, **injectivity**, and the **induction principle**. The first two can be ensured by an injective encoding function, and the third is ensured by an encoding function that is strictly monotonous in all recursive arguments (i.e. the ones of the same type).

The encoding of choice is a right-associative extension to the Cantor pairing function to Cantor tuples, where each constructor with arguments $a_1, ..., a_n$ is encoded as follows:

$$\langle \text{type\_tag}, \text{constructor\_tag}, a_1, ..., a_{n-1}, a_n \rangle$$

Due to right-associativity, this is equivalent to :

$$\langle \text{type\_tag}, \langle \text{constructor\_tag}, \langle a_1, \langle ..., \langle a_{n-1}, a_n \rangle ... \rangle \rangle \rangle \rangle$$

Where the notation $\langle \cdot, \cdot \rangle$ is the well-known Cantor pairing function, which is a bijection on the natural numbers and strictly monotonous in both arguments for $n \geq 2$. It is defined as follows [7]:

Cantor Pairing Function

$$\langle x, y \rangle = \frac{((x+y)^2+3x+y)}{2}$$

5.3 Encoding: Type Membership Predicates

Since the values of the inductive datatypes are encoded as natural numbers (`num`), they must be of the syntactic Isabelle type `num` themselves. Thus, to determine 'type membership', e.g.

whether a given `num` is considered an encoded `List`, there has to be a predicate that decides this. For `List`, such a type membership predicate shall be called `is_list`, and the idea is that is_list a is a proposition-level (`o`) type membership certificate, similar to how x N is a certificate for a terminating natural number. Thus, *GA* can be viewed as having a 'dynamic' type system embedded within the propositional syntax itself, where the types are B, N, and now also inductive datatypes such as `List`.

Since the type membership predicate effectively determines the inhabitants of the type, it is responsible for the first two properties, **closure** and **exhaustiveness**. Formally, the type membership predicate, which is called is_τ for a given coded type τ, should fulfill the following properties. For each τ constructor C_i and its arguments $a_{i,1}, ..., a_{i,n_i}$ with their respective type constraints $\tau_{i,1}, ..., \tau_{i,n_i}$:

1. **Closure of the type membership predicate**: For each constructor C_i, if all its arguments fulfill their corresponding type membership predicates, then C_i applied to these arguments must fulfill its type membership predicate:

$$\text{is\_}\tau_{\text{i1}}\ a_{\text{i1}} \Longrightarrow ... \Longrightarrow \text{is\_}\tau_{\text{in}}\ a_{\text{in}} \Longrightarrow \text{is\_}\tau\ (C_i\ a_{\text{i1}}\ ...\ a_{\text{in}})$$

2. **Exhaustiveness of the type membership predicate**: If the type membership predicate is_τ is fulfilled by a value x, then there must exist a constructor and a set of corresponding arguments fulfilling their type membership predicates, such that x equals the constructor applied to these arguments.

$$\begin{aligned}
&\text{is_}\tau\ x \Longrightarrow \\
&\exists a_{1,1}\ ...\ a_{1,n_1}.\ \ \text{is_}\tau_{1,1}\ a_{1,1} \wedge ... \wedge \text{is_}\tau_{1,n_1}\ a_{1,n_1} \wedge x = \left(C_1\ a_{1,1}\ ...\ a_{1,n_1}\right) \quad \vee \\
&... \quad \vee \\
&\exists a_{m,1}\ ...\ a_{m,n_m}.\ \ \text{is_}\tau_{m,1}\ a_{m,1} \wedge ... \wedge \text{is_}\tau_{m,n_m}\ a_{m,n_m} \wedge x = \left(C_m\ a_{m,1}\ ...\ a_{m,n_m}\right)
\end{aligned}$$

For the `List` type, these two criteria evaluate to:

- For `Nil`: is_list Nil
  For `Cons`: n N $\Longrightarrow$ is_list xs $\Longrightarrow$ is_list Cons n xs
- is_list x $\Longrightarrow$ x = Nil $\vee$ $\exists n$ xs. n N $\wedge$ is_list xs $\wedge$ x = (Cons n xs)

For an inductive datatype τ, the type membership predicate is_τ must invert the encoding for each constructor, i.e. treat it like a Cantor tuple, extract its elements, and check if it matches the encoding. This guarantees closure, while the bijectivity of the encoding guarantees exhaustiveness. We will make this more precise and prove it explicitly later.

For the `List` datatype, the `is_list` predicate fulfilling these properties is the following:

```
is_list_def: "is_list x := if x = 0                                    Isabelle
                             then False
                           else if x = Nil
                             then True
                           else if is_cons x
                             then True
                           else False"
and
is_cons_def: "is_cons x := (cpi 1 x = list_type_tag)
```

```
10                                ∧ (cpi 2 x = list_cons_tag)
11                                ∧ ((cpi 3 x) N)
12                                ∧ (is_list (cpi' 4 x))"
```

Where cpi ix extracts the i'th element of a Cantor tuple (with at least $i+1$ elements) and cpi ix extracts the i'th element of a Cantor tuple with exactly i elements.

The general idea of the type membership predicate is to check for each constructor, whether the argument matches its encoding shape, and if so, whether all (encoded) arguments recursively fulfill their respective predicates.

5.4 Cantor Tuples in GA

So far, we have identified the required properties to make an inductive datatype encoding work and have then identified a scheme for defining constructors and a type membership predicate that are expected to fulfill all these properties. Next, we have to formalize the basis for this encoding, namely Cantor pairings and the associated infrastructure to be able to 'extract' elements from one.

Since the Cantor pairing function is bijective, there is an inverse function mapping each natural number z to the unique pair (x, y) with $z = \langle x, y \rangle$. In the following, let $\mathrm{cpx}(z)$ denote the first component of this inverse, i.e. the unique x such that there exists an x' with $z = \langle x, x' \rangle$. Analogously, let $\mathrm{cpy}(z)$ denote the second component of the inverse.

The standard definition of Cantor pairs and the inverses $\mathrm{cpx}(z)$ and $\mathrm{cpy}(z)$ are analytic closed form expressions, which is what the initial formalization in GA used as well. However, it turns out that in order to prove properties about these functions when they are defined in such a way, a highly mature library of arithmetic lemmas is required. This was especially apparent when trying to prove the growth property $x < \langle x, y \rangle$ (for $x \geq 2$). However, maybe unexpectedly, many of these properties turned out to be much easier to prove in *GA* when these functions are defined recursively.

Thus, the following recursive *GA* definition of a Cantor pair is used from now on:

```
1  cpair_def: "cpair x y := if y = 0 then div (x * S(x)) 2                Isabelle
2                           else cpair x P(y) + x + y + 2"
```

Termination follows by induction on the second argument.

Theorem 15. *Termination of* `cpair`

$$\frac{\Gamma \vdash x\ \mathsf{N} \quad \Gamma \vdash y\ \mathsf{N}}{\Gamma \vdash \langle x, y \rangle\ \mathsf{N}}$$

Proof.

```
1  lemma cpair_terminates [auto]: "x N ⇒ y N ⇒ ⟨x, y⟩ N"                  Isabelle
2  apply (induct y, simp)
3  apply (unfold_def cpair_def, simp)+
4  done
```

■

To provide syntax for general Cantor k-tuples ($\langle a_1, a_2, ..., a_k \rangle$), the following snippet translates such tuples into right associative nested `cpair`s.

```
nonterminal cpair_args                                                    Isabelle

syntax
  "_cpair"      :: "num ⇒ cpair_args ⇒ num"        ("⟨_,_⟩")
  "_cpair_arg"  :: "num ⇒ cpair_args"               ("_")
  "_cpair_args" :: "num ⇒ cpair_args ⇒ cpair_args" ("_,_")
translations
  "⟨x, y⟩" == "CONST cpair x y"
  "_cpair x (_cpair_args y z)" == "_cpair x (_cpair_arg (_cpair y z))"
```

The inverse functions $\mathrm{cpx}(z)$ and $\mathrm{cpy}(z)$ can also be defined mutually recursively:

```
cpx_def: "cpx x := if x = 0 then 0                                        Isabelle
                   else if cpx (P x) = 0 then S(cpy P(x))
                   else P(cpx (P x))" and
cpy_def: "cpy x := if cpx (P x) = 0 then 0
                   else S(cpy (P x))"
```

Despite the mutually inductive structure, termination is quite straightforward, as each recursive call is on a smaller argument.

Theorem 16. *Termination of cpx and cpy*

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

Proof.

The termination proof is mutual as well, by induction on $\mathrm{cpx}\ x\ \mathsf{N} \wedge \mathrm{cpy}\ x\ \mathsf{N}$.

```
lemma cpx_cpy_terminate: "x N ⇒ (cpx x N) ∧ (cpy x N)"                    Isabelle
apply (induct x, simp)
apply (unfold_def cpx_def, simp)
apply (subst rule: condI2)
apply (rule condT, simp)
apply (rule conjE1, simp)
apply (rule conjE2, simp)
apply (rule conjE1, simp)
apply (rule condT, simp)
apply (rule conjE1, simp)
apply (rule conjE2, simp)
apply (rule conjE1, simp)
apply (unfold_def cpy_def, simp)
apply (rule condT, simp)
```

```
15  apply (rule conjE1, simp)
16  apply (rule conjE2, simp)
17  done
```

■

Before proving the critical injectivity property of the `cpair` function, the two following lemmas are required:

Theorem 17. *Projection Lemmas for `cpx` and `cpy`*

$$\frac{\Gamma \vdash x\ \mathsf{N} \quad \Gamma \vdash y\ \mathsf{N}}{\Gamma \vdash \mathrm{cpx}\ \langle x, y\rangle = x} \qquad \frac{\Gamma \vdash x\ \mathsf{N} \quad \Gamma \vdash y\ \mathsf{N}}{\Gamma \vdash \mathrm{cpy}\ \langle x, y\rangle = y}$$

These are well-known properties of the encoding and are given without explicit proof in *GA*, mostly due to time constraint for this thesis. A lemma in Isabelle can be stated and subsequently used without proof by using the `sorry` keyword.

Isabelle
```
1  lemma cpx_proj [simp]: "a N ⇒ b N ⇒ cpx ⟨a, b⟩ = a"
2  sorry
3
4  lemma cpy_proj [simp]: "a N ⇒ b N ⇒ cpy ⟨a, b⟩ = b"
5  sorry
```

Now, injectivity can be proved:

Theorem 18. *Injectivity of `cpair`*

$$\frac{\Gamma \vdash a\ \mathsf{N} \quad \Gamma \vdash b\ \mathsf{N} \quad \Gamma \vdash c\ \mathsf{N} \quad \Gamma \vdash d\ \mathsf{N} \quad \Gamma \vdash \langle a, b\rangle = \langle c, d\rangle}{\Gamma \vdash a = c \wedge b = d}$$

Proof.

Isabelle
```
1   lemma cpair_inj:
2     assumes eq: "⟨a, b⟩ = ⟨c, d⟩"
3     shows "a N ⇒ b N ⇒ c N ⇒ d N ⇒ a = c ∧ b = d"
4   proof -
5     have H: "a N ⇒ b N ⇒ cpx ⟨a, b⟩ = cpx ⟨c, d⟩"
6       by (rule eqSubst[OF eq], simp)
7     have a_eq_c: "a N ⇒ b N ⇒ c N ⇒ d N ⇒ a = c"
8       apply (rule eqSubst[where a="cpx ⟨a, b⟩" and b="a"], simp)
9       apply (rule eqSubst[where a="cpx ⟨c, d⟩" and b="c"], simp)
10      apply (rule H, simp)
11      done
12    have H2: "a N ⇒ b N ⇒ cpy ⟨a, b⟩ = cpy ⟨c, d⟩"
13      by (rule eqSubst[OF eq], simp)
14    have b_eq_d: "a N ⇒ b N ⇒ c N ⇒ d N ⇒ b = d"
15      apply (rule eqSubst[where a="cpy ⟨a, b⟩" and b="b"])
```

```
16      apply (rule cpy_proj, assumption+)
17      apply (rule eqSubst[where a="cpy ⟨c, d⟩" and b="d"])
18      apply (rule cpy_proj, assumption+)
19      apply (rule H2, assumption+)
20      done
21    show "a N ⇒ b N ⇒ c N ⇒ d N ⇒ a = c ∧ b = d"
22      apply (rule conjI)
23      apply (rule a_eq_c)
24      apply (simp)
25      apply (rule b_eq_d)
26      apply (simp)
27      done
28  qed
```

■

The next key property is that a Cantor pair is strictly larger than both its argument (for $x \geq 2$ and $y \geq 1$). This is critical for proving the induction lemma for `Lists` later.

Theorem 19. *Cantor pairing strictly dominates components*

$$\frac{\Gamma \vdash x\ \mathsf{N} \quad \Gamma \vdash y\ \mathsf{N}}{\Gamma \vdash \mathbf{S}(y) < \langle x, \mathbf{S}(y)\rangle = x} \qquad \frac{\Gamma \vdash x\ \mathsf{N} \quad \Gamma \vdash y\ \mathsf{N}}{\Gamma \vdash \mathbf{S}(\mathbf{S}(x)) < \langle \mathbf{S}(\mathbf{S}(x)), y\rangle = x}$$

Proof.

Isabelle

```
lemma cpair_strict_mono_r [simp]: "x N ⇒ y N ⇒ (S y) < ⟨x, (S y)⟩ =
1"
proof (induct y)
  case Base
    show "x N ⇒ y N ⇒ 1 < ⟨x,1⟩ = 1"
      apply (rule less_le_trans[where b="2"], simp)
      apply (unfold_def cpair_def, simp)
      done
next
  case (Step xa)
  show "x N ⇒ y N ⇒ xa N ⇒ S xa < ⟨x, (S xa)⟩ = 1 ⇒ S S xa < ⟨x, (S S xa)⟩ =
1"
    apply (unfold_def cpair_def, simp)
    apply (rule less_le_trans[where b="S S xa + 2"])
    apply (simp add: add_assoc)+
    done
qed
```

Isabelle

```
lemma [simp]:
  "x N ⇒ y N ⇒ S S x < ⟨(S S x), y⟩ = 1"
apply (induct y)
```

```
4  apply (unfold_def cpair_def, simp)
5  apply (rule less_le_trans[where b="div (2 * (S S S x)) 2"], simp)
6  apply (simp add: mult_div_inv)
7  apply (rule leq_mono_div, simp+)
8  apply (unfold_def cpair_def, simp)
9  done
```

■

The next key lemma to prove is the reconstruction lemma, stating that the pair $(\text{cpx } z, \text{cpy } z)$ constitutes the inverse of the Cantor pairing.

Theorem 20. *Reconstruction Lemma*

$$\frac{\Gamma \vdash z\ \mathsf{N}}{\Gamma \vdash z = \langle \text{cpx } z, \text{cpy } z \rangle}$$

Proof.

Using surjectivity of the Cantor pairing function and the projection lemmas.

Isabelle

```
1   lemma [auto]: "z N ⇒ z = ⟨cpx z, cpy z⟩"
2   apply (rule existsE[where Q="λb. z=⟨cpx z,b⟩"])
3   apply (rule existsE[where Q="λc. ∃i. z=⟨c,i⟩"])
4   apply (simp)
5   proof -
6     fix a
7     show "z N ⇒ a N ⇒ ∃i. z = ⟨a,i⟩ ⇒ ∃i. z = ⟨cpx z,i⟩"
8       apply (subst "a = cpx z")
9       apply (rule existsE[where Q="λi. z=⟨a,i⟩"])
10      apply (simp+)
11      done
12    show "z N ⇒ a N ⇒ z = ⟨cpx z,a⟩ ⇒ z = ⟨cpx z,cpy z⟩"
13      apply (subst "a = cpy z")
14      apply (subst "⟨cpx z, a⟩ = z")
15      apply (subst rule: cpy_proj)
16      done
17  qed
```

■

The proof of this theorem relies on the surjectivity of the Cantor pairing function on the natural numbers, which is another of its well-known properties. In the GA formalization, this fact is stated without proof, as proving it would have exceeded the scope and time constraints of this thesis.

Theorem 21. *Surjectivity of the Cantor pairing function*

$$\frac{\Gamma \vdash z\ \mathsf{N}}{\Gamma \vdash \exists x\ y.\ z = \langle x, y\rangle}$$

```
lemma cpair_surjective [auto]: "a N ⇒ ∃b c. a = ⟨b,c⟩"
sorry
```
Isabelle

To project the i-th component of a Cantor k-tuple, we first define cpi' i to return the i-th element of a right-associated Cantor i-tuple.

```
cpi'_def: "cpi' n z := if n = 0 then 0
                       else if n = 1 then z
                       else cpy (cpi' (n-1) z)"
```
Isabelle

Intuitively, `cpi' i` performs i successive selections of the second component (`cpy`) of the outermost Cantor pair. If the value is a Cantor i-tuple, the result is exactly the i-th element. In the other case where the value is a Cantor k-tuple with $k > i$, the term cpi' i z yields the suffix Cantor $k - i + 1$-tuple beginning at the i-th position and thus an additional `cpx` application is required to extract the i-th position. This is what the `cpi` function does:

```
definition cpi :: "num ⇒ num ⇒ num" where
  "cpi i x ≡ cpx (cpi' i x)"
```
Isabelle

The termination proof of `cpi'` is by induction over the first argument and omitted here, although it is included in the Isabelle formalization. The next interesting lemma is a version of the reconstruction lemma for Cantor 4-tuples. The lemma is stated for 4-tuples specifically, since the `Cons` constructor of the `List` datatype is encoded as a Cantor 4-tuple.

Theorem 22. *Reconstruction lemma for Cantor 4-tuples*

$$\frac{\Gamma \vdash z\ \mathsf{N} \quad \Gamma \vdash a\ \mathsf{N} \quad \Gamma \vdash b\ \mathsf{N} \quad \Gamma \vdash c\ \mathsf{N} \quad \Gamma \vdash d\ \mathsf{N}}{\Gamma \vdash z = \langle \text{cpi } 1\ z, \text{cpi } 2\ z, \text{cpi } 3\ z, \text{cpi' } 4\ z\rangle}$$

Proof.

Stated in a slightly different way for ease of application. Using the reconstruction lemmas for `cpx` and `cpy`.

```
lemma cp4_reconstr: "x N ⇒ a N ⇒ b N ⇒ c N ⇒ d N ⇒ cpi 1 x = a ⇒
cpi 2 x = b ⇒
       cpi 3 x = c ⇒ cpi' 4 x = d ⇒
       x = ⟨a,b,c,d⟩"
apply (rule cpair_eq_I, simp)
apply (subst "cpi 1 x = cpx x", auto)
apply (rule cpair_eq_I, simp)
apply (subst "cpi 2 x = cpx (cpy x)", auto)
apply (rule cpair_eq_I, simp)
apply (subst "cpi 3 x = cpx (cpy (cpy x))", auto)
apply (subst "cpi' 4 x = cpy (cpy (cpy x))", auto)
```
Isabelle

```
done
```

■

5.5 The Encoded List Datatype

With the Cantor pairing infrastructure, everything is ready for defining the full `List` datatype and proving all the properties required for an inductive datatype in Section 5.1.

`List` is introduced as a type synonym for the `num` type. This means Isabelle treats `List` as `num` internally, but it can still be written as a type by the user. The definition of the constructors and the `is_list` predicate were already given previously and are restated here for completeness sake.

Isabelle

```
type_synonym List = num

definition list_type_tag where
  "list_type_tag ≡ 1"

definition list_nil_tag where
  "list_nil_tag ≡ 1"

definition list_cons_tag where
  "list_cons_tag ≡ 2"

definition Nil :: "List" where
  "Nil ≡ ⟨list_type_tag,list_nil_tag⟩"

definition Cons :: "num ⇒ List ⇒ List" where
  "Cons n xs ≡ ⟨list_type_tag,list_cons_tag,n,xs⟩"

axiomatization
  is_list :: "num ⇒ o" and
  is_cons :: "num ⇒ o"
where
  is_cons_def: "is_cons x := (cpi 1 x = list_type_tag)
                              ∧ (cpi 2 x = list_cons_tag)
                              ∧ ((cpi 3 x) N)
                              ∧ (is_list (cpi' 4 x))" and
  is_list_def: "is_list x := if x = 0
                               then False
                             else if x = Nil
                               then True
                             else if is_cons x
                               then True
                             else False"
```

The first important lemma states termination of the `is_list` predicate, which essentially means that type membership checking is decidable.

Theorem 23. *Termination of `is_list` and `is_cons`*

$$\frac{\Gamma \vdash x\ \mathsf{N}}{\Gamma \vdash \text{is\_list}\ x\ \mathsf{B}} \qquad \frac{\Gamma \vdash x\ \mathsf{N}}{\Gamma \vdash \text{is\_cons}\ x\ \mathsf{B}}$$

Proof.

First, prove that `Nil` and `Cons` are terminating natural numbers for any arguments (`nil_nat` and `cons_nat`). Then, prove termination of `is_list` and `is_cons` mutually inductively using the strong induction lemma.

```
lemma nil_nat [auto]: "Nil N"
unfolding Nil_def list_type_tag_def by simp

lemma cons_nat [auto]: "n N ⇒ xs N ⇒ Cons n xs N"
unfolding Cons_def list_type_tag_def by simp

lemma list_cons_term [auto]: "x N ⇒ (is_list x B) ∧ (is_cons x B)"
proof (induct strong x)
  case Base
    show "x N ⇒ (is_list 0 B) ∧ (is_cons 0 B)"
      apply (unfold_def is_list_def)
      apply (unfold_def is_cons_def)
      apply (unfold_def is_list_def)
      apply (simp)
      done
next
  case (Step xa)
    fix y
    assume hyp: "(∧y. y N ⇒ y ≤ xa = 1 ⇒ (is_list y B) ∧ (is_cons y B))"
    from Step show ?case
      apply (unfold_def is_list_def)
      apply (unfold_def is_cons_def)
      apply (simp)
      apply (rule condTB, simp)+
      apply (rule conjE1, rule hyp, simp, rule le_suc_implies_leq, simp)+
      done
qed
```

Isabelle

■

5.5.1 Proving Constructor Distinctness

Distinctness follows immediately from disjoint constructor tags in the encoding.

Theorem 24. *List constructor distinctness*

$$\frac{\Gamma \vdash n\ \mathsf{N} \quad \Gamma \vdash \mathrm{xs}\ \mathsf{N}}{\Gamma \vdash \neg\ \mathrm{Nil} = \mathrm{Cons}\ n\ \mathrm{xs}} \qquad \frac{\Gamma \vdash n\ \mathsf{N} \quad \Gamma \vdash \mathrm{xs}\ \mathsf{N}}{\Gamma \vdash \neg\ \mathrm{Cons}\ n\ \mathrm{xs} = \mathrm{Nil}}$$

Proof.

```
lemma [auto]: "n N ⇒ xs N ⇒ ¬ Nil = Cons n xs"
unfolding Nil_def Cons_def by simp

lemma [auto]: "n N ⇒ xs N ⇒ ¬ Cons n xs = Nil"
unfolding Nil_def Cons_def by simp
```
Isabelle

■

5.5.2 Proving Injectivity of `Cons`

Injectivity reduces to injectivity of the Cantor pairing at each nesting level.

Theorem 25. *Injectivity of Cons*

$$\frac{\Gamma \vdash n\ \mathsf{N} \quad \Gamma \vdash m\ \mathsf{N} \quad \Gamma \vdash \mathrm{xs}\ \mathsf{N} \quad \Gamma \vdash \mathrm{ys}\ \mathsf{N} \quad \Gamma \vdash \mathrm{Cons}\ n\ \mathrm{xs} = \mathrm{Cons}\ m\ \mathrm{ys}}{\Gamma \vdash n = m \wedge \mathrm{xs} = \mathrm{ys}}$$

Proof.

```
lemma
  "n N ⇒ m N ⇒ xs N ⇒ ys N ⇒ Cons n xs = Cons m ys ⇒ n = m ∧ xs = ys"
unfolding Cons_def
apply (rule cpair_inj)
apply (rule cpair_inj_r, rule cpair_inj_r, simp)
done
```
Isabelle

■

5.5.3 Proving Exhaustiveness

Exhaustiveness is essentially a case-distinction lemma and states that every element recognized by `is_list` is either `Nil` or has `Cons`-shape with well-typed arguments. We first prove a decoding lemma for `Cons`-shapes, then a decoding lemma for `List` in general, and finally the full case distinction rule.

Theorem 26. *List exhaustiveness*

$$\frac{\Gamma \vdash \mathrm{is\_cons}\ \mathrm{x} \quad \Gamma \vdash x\ \mathsf{N}}{\Gamma \vdash \exists n\ \mathrm{xs}.\ (n\ \mathsf{N}) \wedge (\mathrm{is\_list}\ \mathrm{xs}) \wedge x = (\mathrm{Cons}\ n\ \mathrm{xs})}$$

$$\frac{\Gamma \vdash \text{is\_list x} \quad \Gamma \vdash x\ \mathsf{N}}{\Gamma \vdash x = \text{Nil} \vee \exists n\ \text{xs}.\ (n\ \mathsf{N}) \wedge (\text{is\_list xs}) \wedge x = (\text{Cons}\ n\ \text{xs})}$$

Proof.

Isabelle

```
lemma cons_decode [auto]:
  "is_cons x ⇒ x N ⇒ ∃n xs. ((n N) ∧ is_list xs ∧ x = Cons n xs)"
apply (rule existsI[where a="cpi 3 x"], simp+)
apply (rule existsI[where a="cpi' 4 x"], simp+)
apply (unfold Cons_def)
apply (subst rule: cons_1_tag)
apply (subst rule: cons_2_2)
apply (rule cp4_reconstr, simp+)
done

lemma list_decode: "x N ⇒ is_list x ⇒ (x = Nil) ∨ (∃n xs. (n N) ∧ is_list xs
∧ (x = Cons n xs))"
apply (rule implE[where a="is_list x"])
apply (unfold_def is_list_def)
apply (cases bool: "x=Nil", simp+)
apply (rule implI, simp)
apply (rule disjI1, simp)
apply (cases bool: "is_cons x", simp+)
apply (rule implI)
apply (rule condTB, simp)+
apply (simp+)
apply (rule implI, simp)
apply (rule exF[where P="False"], simp)
done
```

■

Theorem 27. *List cases*

$$\frac{\Gamma \vdash \text{is\_list}\ x \quad \Gamma \vdash x\ \mathsf{N} \quad \Gamma \cup \{x = \text{Nil}\} \vdash p \quad \Gamma \cup \{n\ \mathsf{N}, \text{xs}\ \mathsf{N}, \text{is\_list xs}, x = \text{Cons}\ n\ \text{xs}\} \vdash p}{p}$$

Proof.

Isabelle

```
lemma cases_list [case_names _ HQ Nil Cons, cases]: "is_list x ⇒
        (x N) ⇒
        (x = Nil ⇒ Q) ⇒
        (∧n xs. n N ⇒ xs N ⇒ is_list xs ⇒ x = Cons n xs ⇒ Q)
        ⇒ Q"
apply (rule disjE1[OF list_cases], simp, assumption)
apply (rule existsE[where Q="λn. ∃xs. (n N) ∧ is_list xs ∧ x = Cons n xs"])
```

```
8   apply (assumption)
9   proof -
10    fix a
11    show "is_list x ⇒
12       (x = Nil ⇒ Q) ⇒
13       (∧n xs. n N ⇒ xs N ⇒ is_list xs ⇒ x = Cons n xs ⇒ Q) ⇒
14       ∃n xs. (n N) ∧ is_list xs ∧ x = Cons n xs ⇒
15       a N ⇒ ∃xs. (a N) ∧ is_list xs ∧ x = Cons a xs ⇒ Q"
16      apply (rule existsE[where Q="λxs. (a N) ∧ is_list xs ∧ x = Cons a xs"])
17      apply (assumption)
18      proof -
19        fix aa
20        show "
21        (∧n xs. n N ⇒ xs N ⇒ is_list xs ⇒ x = Cons n xs ⇒ Q) ⇒
22          aa N ⇒ (a N) ∧ is_list aa ∧ x = Cons a aa ⇒ Q"
23          apply (rule Pure.meta_mp[where P="a N"])
24          apply (rule Pure.meta_mp[where P="is_list aa"])
25          apply (rule Pure.meta_mp[where P="x = Cons a aa"])
26          apply (assumption)
27          apply (rule conjE2, simp)
28          apply (rule conjE2, rule conjE1, simp)
29          apply (rule conjE1, rule conjE1, simp)
30          done
31      qed
32  qed
```

■

5.5.4 Proving Closure

Closure states that every constructor application yields an element of the encoded datatype. For `List`, this is immediate from the definition of `is_list` and the `Cons`-shape predicate.

Theorem 28. *List closure*

$$\frac{}{\Gamma \vdash \text{is\_list Nil}} \qquad \frac{\Gamma \vdash n\ \mathsf{N} \quad \Gamma \vdash \text{xs}\ \mathsf{N} \quad \Gamma \vdash \text{is\_list xs}}{\Gamma \vdash \text{is\_list (Cons } n \text{ xs)}}$$

Proof.

Isabelle

```
1  lemma [auto]: "¬ Nil = 0"
2  unfolding Nil_def by simp
3
4  lemma [auto]: "is_list Nil"
5  by (unfold_def is_list_def, simp)
```

```
lemma [auto]: "n N ⇒ xs N ⇒ is_list xs ⇒ is_cons (Cons n xs)"
unfolding Cons_def by (unfold_def is_cons_def, simp)

lemma cons_is_list [auto]:
  "n N ⇒ xs N ⇒ is_list xs ⇒ is_list (Cons n xs)"
apply (unfold_def is_list_def)
apply (unfold_def is_cons_def)
apply (unfold Cons_def)
apply (simp)
done
```
Isabelle

■

5.5.5 Proving `List` Induction

We derive an induction principle for `List` by strong induction on the `num` code, using the growth property $\text{xs} < \text{Cons}\ n\ \text{xs} = 1$ to justify the recursive call.

Theorem 29. *List induction*

$$\frac{\Gamma \vdash \text{is\_list}\ a \quad \Gamma \vdash a\ \mathsf{N} \quad \Gamma \vdash K\ \text{Nil} \quad \Gamma \cup \{x\ \mathsf{N},\ \text{xs}\ \mathsf{N},\ \text{is\_list xs},\ K\ \text{xs}\} \vdash K\ (\text{Cons x xs})}{K\ \text{a}}$$

Proof.

```
lemma [simp]: "xs N ⇒ is_list xs ⇒ n N ⇒ xs < Cons n xs = 1"
unfolding Cons_def by simp
```
Isabelle

```
lemma [case_names _ HQ Nil Cons, induct]:
  "is_list a ⇒ a N ⇒ Q Nil ⇒ (∧x xs. x N ⇒ xs N ⇒ is_list xs ⇒ Q xs ⇒ Q (Cons
  x xs))
   ⇒ Q a"
apply (rule implE[where a="is_list a"])
apply (induct strong a)
apply (rule implI, simp)
apply (rule exF[where P="is_list 0"], simp)
apply (rule implI, simp)
proof -
  fix xa
  assume hyp: "(∧y. y N ⇒ y ≤ xa = 1 ⇒ is_list y → Q y)"
  assume cons: "(∧x xs. x N ⇒ xs N ⇒ is_list xs ⇒ Q xs ⇒ Q (Cons x xs))"
  show "a N ⇒ xa N ⇒ is_list S xa ⇒ Q Nil ⇒
        Q S xa"
    proof (cases "S xa", simp)
      case Nil
        from Nil show ?case
```
Isabelle

```
18            by (simp+)
19      next
20        case (Cons n xs)
21          from Cons and cons show ?case
22            apply (simp)
23            apply (rule cons, simp)
24            apply (rule obj_impl)
25            apply (rule hyp, simp)
26            apply (rule le_suc_implies_leq, simp+)
27            done
28      qed
29  qed
```

■

5.6 Tooling for Inductive Datatypes

To make the induction and case distinction mechanisms of `List` useful, they are integrated with the existing `induct` and `cases` tactics.

The idea is that the induction lemma of an inductive type is annotated with the [induct] tag, and the corresponding case distinction lemma is annotated with [cases]. To accommodate the new inductive types, the `induct` and `cases` tactics are extended accordingly. The behavior of the `induct` tactic is now as follows:

- If the argument `strong` is supplied, the strong induction lemma is applied.
- Otherwise, the tactic iterates over all theorems annotated with the [induct] attribute and proceeds as follows for each candidate theorem:
  - Attempt to instantiate the theorem with the given term x.
  - Check whether the first subgoal resulting from the instantiation can be discharged using only the assumptions currently available in the proof context (this check is performed without committing to the application).
  - If this succeeds, the theorem is selected, applied to the goal, and the first subgoal is solved.
  - Finally, the corresponding cases are generated from the theorem definition, making use of the `case_names` annotation.

The following is the final code for the `induct` method, implementing the logic described above:

SML

```
structure GD_Induct =
struct
  val induct_thm = @{thm ind}
  val strong_induct_thm = @{thm strong_induction}

  fun try_inst_thm ctxt t th =
    let val ct = Thm.cterm_of ctxt t in
      try (fn th => Drule.infer_instantiate' ctxt [SOME ct] th) th
    end
```

```
10
11  fun closes_first_prem ctxt i th st =
12    let
13      val tac =
14      DETERM (
15        resolve_tac ctxt [th] i
16        THEN ((SOLVED' (assume_tac ctxt)) i)
17      )
18    in
19      Option.isSome (Seq.pull (tac st))
20    end
21
22  fun select_induct_thm ctxt t i st =
23    let
24      val induct_thms = Named_Theorems.get ctxt @{named_theorems induct}
25      fun is_instantiable th =
26        case (try_inst_thm ctxt t th) of
27          NONE     => NONE
28        | SOME th' => if (closes_first_prem ctxt i th' st) then SOME th else
          NONE
29    in
30      case (get_first is_instantiable induct_thms) of
31        SOME th => th
32      | NONE    => induct_thm
33    end
34
35  fun apply_tac tac st =
36    let
37      val res = DETERM tac st
38    in
39      case Seq.pull res of
40        SOME (st', _) => st'
41      | NONE => raise THM ("tactic failed", 0, [st])
42    end
43
44  fun induct_tac strong t =
45    CONTEXT_SUBGOAL (fn (_, i) => fn (ctxt, st) =>
46    let
47      val th  = if strong then strong_induct_thm else (select_induct_thm ctxt
      t i st)
48      val th' = try_inst_thm ctxt t th
49      val tac =
50        case th' of
51          SOME th'' => DETERM (match_tac ctxt [th''] i)
52        | NONE      => no_tac
```

```
53        val st' = apply_tac tac st
54        val (spec, _) = Rule_Cases.get th
55        val cases_prop = Thm.prop_of (Rule_Cases.internalize_params st')
56        val cases = Rule_Cases.make_common ctxt cases_prop spec
57        val post_tac = TRY (SOLVED' (assume_tac ctxt) i)
58      in
59        CONTEXT_CASES cases post_tac (ctxt, st')
60      end)
61
62    fun gd_induct_method (strong, t) _ =
63      Method.CONTEXT_METHOD (K (induct_tac strong t 1))
64  end
65
66  val parse_induct_args =
67    Scan.lift (Scan.optional ((Args.$$$ "strong") >> K true) false)
68    -- Args.term
69
70  val _ =
71    Theory.setup
72      (Method.setup @{binding induct}
73      (parse_induct_args >> GD_Induct.gd_induct_method)
74      "Apply rule ind with where a = <term>"
75    )
```

5.7 Future Work

This concludes the proof of all the required properties stated in Section 5.1. Thus, the encoding of `Nil` and `Cons` together with the `is_list` predicate constitute a full inductive datatype in *GA*. Since most of the steps undertaken to get there followed a clear template, in the future, an inductive datatype should be able to be compiled from a simple description of the constructors:

```
1  declaretype List =                                                    Isabelle
2    Nil
3    | Cons of "num" "List"
```

To make inductive datatypes truly practical in *GA*, it would be desirable to provide an accompanying definitional mechanism, accepting high-level specifications such as the following:

```
1  fun sum :: "List ⇒ num" where                                         Isabelle
2   sum_nil: "sum Nil = 0" and
3   sum_cons: "sum (Cons n xs) = n + sum xs"
```

That is then compiled into the following encoding-aware recursive *GA* definition:

```
1  axiomatization                                                        Isabelle
```

```
2    sum :: "List ⇒ num"
3  where
4    sum_def: "sum x := if x = Nil then 0
5                       else if (is_cons x) then (cpi 3 x) + (sum (cpi' 4 x))
6                       else omega"
```

From which a termination proof and the defining equations can be derived:

Isabelle

```
1   lemma [auto]: "x N ⇒ is_list x ⇒ sum x N"
2   proof (induct x, simp)
3     case Nil
4     show ?case
5       by (unfold_def sum_def, simp add: sum_def)
6   next
7     case (Cons n xs)
8     from Cons show ?case
9       apply (unfold_def sum_def)
10      apply (subst rule: condI2)
11      apply (subst rule: condI1)
12      apply (unfold Cons_def, simp+)
13      apply (unfold_def is_cons_def, simp)
14      done
15  qed
16
17  lemma [simp]: "sum Nil = 0"
18  by (unfold_def sum_def, simp)
19
20  lemma [simp]: "n N ⇒ xs N ⇒ is_list xs ⇒ sum (Cons n xs) = n + sum xs"
21  apply (rule eqSym)
22  apply (unfold_def sum_def)
23  apply (unfold_def is_cons_def)
24  apply (unfold Cons_def Nil_def, simp)
25  done
```

While a full implementation of these mechanisms is outside the scope of this thesis, the foundations have been developed, and what remains is largely an engineering task left for future work.

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.

[6] L. C. Paulson, "A fixedpoint approach to (co)inductive and (co)datatype definitions," in *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, G. D. Plotkin, C. Stirling, and M. Tofte, Eds., The MIT Press, 2000, pp. 187–212.

[7] L. Meri, "Some remarks on the Cantor pairing function," *Le Matematiche*, vol. 62, p. , 2007.