Informatics

Automated Induction by Reflection

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Johannes Schoisswohl, BSc
Matrikelnummer 132784

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Prof.Dr. Laura Ildikó Kovács

Wien, 24. November 2020

Johannes Schoisswohl Laura Ildikó Kovács

TU WIEN Informatics

Automated Induction by Reflection

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Johannes Schoisswohl, BSc
Registration Number 132784

to the Faculty of Informatics

at the TU Wien

Advisor: Prof.Dr. Laura Ildikó Kovács

Vienna, 24th November, 2020

Johannes Schoisswohl Laura Ildikó Kovács

Erklärung zur Verfassung der Arbeit

Johannes Schoisswohl, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 24. November 2020

Johannes Schoisswohl

Acknowledgements

Partial results of this thesis have been supported by the ERC starting grant 2014 SYMCAR 639270, the EPSRC grant EP/P03408X/1, and the ERC proof of concept grant 2018 SYMELS 842066.

Abstract

Despite the advances in automated theorem proving in the last decades, making it practically feasible to reason about full first-order logic with interpreted equality and more, inductive reasoning still poses a serious challenge to state-of-the-art theorem provers. The reason for that is that in first-order logic induction requires an infinite number of axioms, which is not feasible as an input for a theorem prover that is a computer program, requiring a finite input. Mathematical practice is to specify these infinite sets of axioms as axiom schemes. Unfortunately these schematic definitions are not part of the syntax of first-order logic, and therefore not supported as an input for modern theorem provers.

In this thesis we introduce a new method, inspired by the field of axiomatic theories of truth, that allows to the express schematic definitions needed for first-order induction, in the standard syntax of multi-sorted first-order logic. Further the practical feasibility of this method is tested with state-of-the-art theorem provers, by comparing it to solvers native techniques for handling induction.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Automated reasoning has advanced tremendously in the last decades, pushing the limits of what machines can prove. Recent progress in this area features techniques such as first-order reasoning with term algebras [KRV17a], embedding programming control structures in first-order logic [KKRV16], the Avatar architecture[Vor14], combining theory instantiation and unification with abstraction in saturation-based proof search [RSV18], automating proof search for higher-order logic [BR20a, BBCW20], and first-order logic with rank-1 polymorphism [BR20b]. As expressive power of logics that can be handled practically, has dramatically increased, from propositional, and quantifier free first-order logic, to full first-order logic with equality, and beyond, the range of applications of automated theorem proving has progressed, from model checking of finite state transition systems using Binary Decision Diagrams, and SAT solving [DKW08], to verification of software with an unbounded state space by translating programs to first-order logic [GKR18], and assisting mathematicians in finding formal proofs using so-called hammers to close gaps in formal proofs in higher-order logic[HUW14].

Despite the fact that first-order logic with equality can be handled very well in many practical cases, there is one fundamental mathematical concept most first-order theorem provers lack; namely inductive reasoning. Not only is induction a very interesting theoretical concept, it is also of high practical importance, since it is required to reason about many concepts that are needed for verifying software. Theses concepts include natural number arithmetic, recursion, unbounded loop iterations, or recursive data structures like lists, and trees.

As we will examine in more detail in section 2.2 inductive reasoning can be understood as an uninterpreted reasoning problem in first-order, or second-order logic with equality. As Gödel's incompleteness[Bar82] theorem teaches us that there is no sound, complete and effective proof system for second-order logic, the hardness of the latter is not surprising. In contrast the first-order induction problem is “only” an ordinary first-order theory with a recursively enumerable number of axioms. Therefore the first-order problem can be in theory solved by enumerating all possible proofs in any sound, complete and effective proof system of first-order logic.

This theoretical observation, does of course not help in practice, since we do not want to enumerate all proofs until we find the right one “by chance”. Instead we want to find a proof by systematically deducing facts from the goal formula and our axioms.

Therefore we usually want our proof-system to be analytic, in order to perform automated proofs effectively. In the context of sequent-style calculi [Tak13] an analytic proof is a proof that does not involve the cut rule. In a more general sense we can think of an analytic proof as one where all formulas, occurring in the proof are (substitution instances of) subformulas of the goal or an axiom[Pfe84]. This means that no arbitrary lemmata, or auxiliary definitions are required to prove the goal. One of the most prominent of these proof systems is the resolution calculus[dN98].

In fact there are many analytic proof systems for arbitrary first-order theories. But analyticity does not help a lot in case of first-order induction. The problem is that the set of axioms of a first-order inductive theory contains an induction axiom $\mathbf{I}[\phi]$ for every formula ϕ in the language of the theory, which means every formula is a subformula of some axiom, hence every proof is trivially an analytic one.

This non-analyticity is a theoretical insight explaining why modern theorem provers struggle with the problem of induction, but there is also a rather practical problem that theorem provers have with induction as a first-order theory: First-order induction requires an infinite number of axioms, while first-order theorem provers are programs on computers finite memory and therefore require a finite set of input formulas. It is mathematical practice to give a finite representation of the infinite collection of axioms, by specifying them as an axiom scheme, which is unfortunately not supported by state of the art theorem provers. In this thesis, we want to address the problem of handling schematic definitions, and therefore automating induction in first-order theorem provers.

1.2 Goals

The goal of this thesis is firstly to investigate different approaches to induction, from both a theoretical (chapter 1), and a practical perspective (chapter 3). Based on this investigation a new approach to induction will be introduced in section 4.3. By replacing the infinite number of axioms used in first-order inductive theories, like Peano Arithmetic (**PA**), for a finite number of axioms that yield a conservative extension of that theory. By showing that our extension is indeed a conservative one we will show that the method is complete in some sense in subsection 4.3.1. Further in chapter 5 the feasibility of this technique is to be tested by running benchmarks on the state-of-the-art theorem provers.

In addition further potential applications of this approach to finitely axiomatizing a theory based on schematic definitions will be discussed in section 4.4.

The contributions of this thesis can be summarized as follows:

- a method for conservatively extending an arbitrary theory to have a truth predicate (which allows quantification over formulas) (section 4.2)
- showing how this method can be used to replace the induction scheme of **PA**, and other theories involving inductive datatypes (section 4.3)
- listing other applications of these kind of conservative extension (section 4.4)
- contributing a new set of benchmarks to the automated theorem proving community (section 5.1)
- conducting experiments on this set of benchmarks with state of the art theorem provers (section 5.4)
- comparing state of the art theorem provers on simple inductive problems (section 5.4)

CHAPTER 2

Preliminaries

2.1 Mathematical Logic

We assume basic knowledge of multi-sorted first-order logic, automated reasoning, proof theory and mathematical logic in general.

Within this thesis we will reason on three levels: the object, the meta, and the reflective level. Therefore we will use different symbols for logic on all of these levels. Meta and object level notation will be defined in this section, while reflective level logic will be defined in chapter 4, where axiomatic theories of truth are introduced.

We use the symbols $\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$ for logical negation, conjunction, disjunction, implication and equivalence respectively, and write $Qx : \sigma.\phi$ with $Q \in \{\exists, \forall\}$ for existential and universal quantification over the sort σ on the object level. If it is deducible from the context of a formula we will drop the sort declarations for quantifier variables. Further we will write $\underset{x_1,\dots,x_n}{\forall\!\!\!\forall} \phi$ for the formula $\forall x_1. \dots \forall x_n.\phi$, and $\forall\!\!\!\forall\phi$ to denote the universal closure of ϕ. Object level equality is denoted by $\approx$.

On the meta-level we will use !, &, $\|$, $\implies$, and $\iff$ for negation, conjunction, disjunction, implication and equivalence and $\mathbb{V}$, and $\exists$ for quantification. For meta-level equality $=$ is used. Meta-level logical formulas will only be used where they help to improve readability and precision, and otherwise natural language will be used.

By $\mathbf{Term}^{\Sigma}_{\sigma}$, $\mathbf{Var}^{\Sigma}_{\sigma}$, and $\mathbf{Form}^{\Sigma}$, we denote the sets of variables of sort σ, terms of sort σ and object-level formulas over a signature Σ respectively. Overloading notation we will sometimes use theories as superscripts, meaning the same thing as if the theories signature would have been used. Further we will drop the superscript if it is not necessary. As $\mathbf{Var}_{\sigma}$ is a countably infinite set, we will sometimes assume without loss of generality that it is composed of the variables $\{\mathsf{x}^{\sigma}_{i} \mid i \in \mathbb{N}\}$, and leave away the sort superscript σ, if it is deducible from the context.

We will use the symbols x, y, and z for variables, σ, and τ for sorts, P, Q, and R for predicates, f, g, and h for functions, and a, b and c for constants symbols. For unary functions in terms (like $s(s(n))$) we will sometimes drop the brackets (to obtain ssn) for the sake of readability.

For a function symbol f of a signature Σ, we write $s :: \sigma_1 \times \ldots \times \sigma_n \rightsquigarrow \sigma \in \Sigma$, to denote that $\mathbf{dom}(f) = \sigma_1 \times \ldots \times \sigma_n$, is the domain and $\mathbf{codom}(f) = \sigma$ is the codomain, and $\mathbf{arity}(f) = n$ is the arity of f. We consider constants being functions of arity 0, and we write $c :: \sigma$ for $c :: \rightsquigarrow \sigma$. Further we write $P : \mathsf{Pred}(\sigma_n \times \ldots \times \sigma_n)$ to denote that P is an predicate with domain $\mathbf{dom}(P) = \sigma_1 \times \ldots \times \sigma_n$ and arity $\mathbf{arity}(P) = n$. Further we will write $\mathbf{dom}(s, i)$ for the ith component of the domain of s.

If ϕ is a formula, or a term, x is a variable and t or a term, we write $\phi[x \mapsto t]$ to denote the formula, or term respectively resulting from replacing all occurences of x by t in ϕ. Similarly if x is a variable, and $\phi[x]$ is a formula (or a term), we denote the formula (or term) resulting from replacing all occurences of x for t by $\phi[t]$.

If Γ is a set of formulas, we write $\Gamma \vdash_X \phi$ to denote that ϕ is derivable from the axioms Γ in a proof system X, and we will drop the subscript X if it is not necessary. For I being a first-order interpretation we write $I \vDash \phi$ to denote that I is a model of ϕ. We write $\Gamma \vDash \phi$ to denote that ϕ is a semantic consequence of Γ.

A formula is open if it contains free variables, and closed otherwise. We consider a theory to be a set of closed formulas.

The semantics of formulas, and terms over a signature Σ is defined in terms of multi-sorted first-order interpretations $\mathcal{M}$, consisting of $\langle\langle\Delta_{\sigma_1}, \ldots, \Delta_{\sigma_n}\rangle, \mathcal{I}\rangle$ consisting of a domain Δ_{σ_i} for each sort $\sigma \in \mathbf{sorts}_\Sigma$, and an interpretation function $\mathcal{I}$, that freely interprets variables, function symbols, and predicate symbols, respecting the sorts, and is extended to terms in the usual manner. By Δ we denote $\langle\Delta_{\sigma_1}, \ldots, \Delta_{\sigma_n}\rangle$. We write $\mathcal{M} \vDash \phi$ for the structure $\mathcal{M}$ satisfying the formula ϕ. For that we also say $\mathcal{M}$ is a model of ϕ. We sometimes use $\mathcal{I} \vDash \phi$, instead of $\langle\langle\Delta_{\sigma_1}\rangle, \ldots, \Delta_{\sigma_n}, \mathcal{I}\rangle \vDash \phi$ if the domains are clear in the context. By $\mathbf{Interpret}_\Sigma$ we denote the class of all interpretation functions over a signature Σ.

Further we will $\mathcal{P}(S)$ to denote the power set of S, and $\mathbf{lfp}(f)$ for the least fixed point of a function, that maps sets to sets.

For specifying different versions of induction we will come across the concept of axiom schemes. We define an axiom scheme $\mathbf{S}$ to be a formula over the signature Σ extended with a special unary predicate P. We can instantiate the scheme $\mathbf{S}$ with an formula σ to get a formula over the original signature Σ by replacing all occurences of an atom $\mathsf{P}(t)$ by $\phi[x_0 \mapsto t]$. We write $\mathbf{S}[\phi]$ for the instantiation of $\mathbf{S}$ with ϕ, and call the set $\{\mathbf{S}[\phi] \mid \phi \text{ is a first order formula over } \Sigma\}$ "all first order instances of $\mathbf{S}$".

When defining our method for reflective reasoning we will need the concept of a conservative extension. A conservative extension of a theory $\mathcal{T}$ is a theory $\mathcal{T}'$, such that for all for $\forall \phi \in \mathbf{Form}^{\mathcal{T}}.(\mathcal{T} \vDash \phi \iff \mathcal{T}' \vDash \phi)$

2.2 Induction

The principle of induction is one of the most fundamental principles in mathematics. It often permits very simple, and elegant proofs once the right invariant has been found and it seems so inexplicably fundamental, that the parts of proofs where inductive strengthening is needed are often skipped as "obvious" by mathematicians. Induction is a very general principle, that comes in many different flavours, and can be generalized, and reformulated in many different ways. In this section we will investigate what different kinds of induction are out there and how they relate.

2.2.1 Peano Arithmetic vs. True Arithmetic

Almost every student gets in touch with the principle of mathematical induction, proving fundamental results in number theory. What is often not pointed out is that there are two variants of number theory, that are of different strength. Firstly there is a theory we will call True Arithmetic (**TA**), which is the kind of arithmetic we normally have in mind when we think about arithmetic, and secondly there is **PA**, which is usually used to formalized arithmetic as a first-order theory[Sha91]. Both **PA** and **TA**, feature only one sort, nat, and share a common core of axioms, which we will refer to as **Q**, namely the universal closure of the following formulas:

$$\mathsf{s}(x) \approx \mathsf{s}(y) \rightarrow x \approx y \tag{Inj}$$

$$\mathsf{0} \not\approx \mathsf{s}(x) \tag{Disj}$$

$$x + \mathsf{0} \approx x \tag{Add1}$$

$$x + \mathsf{s}(y) \approx \mathsf{s}(x + y) \tag{Add2}$$

$$x \times \mathsf{0} \approx \mathsf{0} \tag{Mul1}$$

$$x \times \mathsf{s}(y) \approx x + (x \times y) \tag{Mul2}$$

The difference between the two theories is how induction is axiomatised. In the case of **TA**, induction is defined as a second-order axiom.

$$\forall P.\Big(P\mathsf{0} \land \forall n.(Pn \rightarrow P\mathsf{s}n) \rightarrow \forall n.Pn\Big) \tag{Ind}$$

while in the case of **PA** induction is defined as a scheme of first-order formulas. In exact induction is defined by all first-order instances of the scheme ($\mathbf{I}_{\mathsf{nat}}$).

$$\mathsf{P}(\mathsf{0}) \land \forall x.(\mathsf{P}(x) \rightarrow \mathsf{P}(\mathsf{s}x)) \rightarrow \forall x.\mathsf{P}(x) \tag{$\mathbf{I}_{\mathsf{nat}}$}$$

Although the two theories look very similar, at the first sight, the differ dramatically in reality. While **TA** captures our intuition of mathematical induction – namely defining the model $\mathbb{N} = \langle \omega, I \rangle$, with $\omega = \{0, 1, 2, ...\}$, uniquely up to isomorphism – it is a theory in second-order logic, and therefore does not permit a sound, complete, and effective proof system. In contrast **PA** is a first-order theory, for which sound, complete and effective proof systems exists. However **PA** does not define the structure $\mathbb{N}$, but a class of models, that includes $\mathbb{N}$. Since $\mathbb{N}$ is the intended semantics of **PA** we call it the "standard model" the theory. The existence of non-standard models for **PA** follows directly from the Theorem of Löwenheim-Skolem, as well as from a combination of Gödel's completeness theorem, and his first incompleteness theorem [Bar82].

2.2.2 Induction and Least Fixed Points

As mentioned in the previous section, the intended meaning of mathematical induction is to define the set of natural numbers ω (together and some operations on them). This aim can also be achieved with the more general concept of an inductive definition:

Definition 1 (Natural numbers)**.** *Let ω be the least set such that*

- $\mathsf{0} \in \omega$
- $x \in \omega \implies \mathsf{s}(x) \in \omega$

More formally we can define a monotonous operation X_ω, and define ω as the least fixed point of this operator.

$$\begin{aligned} X_\omega &: \mathcal{P}(\mathcal{T}) \mapsto \mathcal{P}(\mathcal{T}) \\ X_\omega(ts) &= ts \cup \{\mathsf{0}\} \cup \{\mathsf{s}(t) \mid t \in ts\} \\ \omega &: \mathcal{P}(\mathcal{T}) \\ \omega &= \mathbf{lfp}(X_\omega) \end{aligned}$$

As we see ω is "constructed" from the function symbols $\{\mathsf{s}, \mathsf{0}\}$. Therefore we call these functions the constructors $\mathbf{ctors}(\mathsf{nat})$ of the sort nat. This notion of designated constructor functions will help us to generalize the notion of a standard-model to arbitrary datatypes, in subsection 2.2.3.

It is worth to note mentioning that the core of Isabelle proof assistant [BW99] is built upon such such inductive definitions, and that an in-depth investigation of the relation between least fixed points and induction can be found in [BS11].

Furthermore least fixed points play an important role in model checking, where a program is seen as an initial set of states $I : \mathcal{P}(\mathcal{S})$, and a transition relation $T : \mathcal{P}(\mathcal{S} \times \mathcal{S})$, on the set of states $\mathcal{S}$ in which the system (i.e. the computer) can be. The set of states reachable when executing the program can then be defined as the least fixed point of the operator $Y_{I,T}$.

$$\begin{aligned} Y_{I,T} &: \mathcal{P}(\mathcal{S}) \mapsto \mathcal{P}(\mathcal{S}) \\ Y_{I,T}(s) &= s \cup I \cup \{x' \mid x' \in s, sTs'\} \end{aligned}$$

Therefore we can view the transition relation T and the set of initial states I as the constructors of the set or reachable program states, and we can view proving properties about our program, as inductive theorem proving on state spaces of programs. This is why in model checking inductive invariants are used, despite the fact that there are no induction axioms or inductive datatypes present, at the first sight.

2.2.3 Structural Induction

As we saw in the previous section, the unique model of **TA**, which is the standard-model of **PA**, can be generated from the set of constructor symbols $\mathbf{ctors}_{\mathsf{nat}} = \{\mathsf{s}, \mathsf{0}\}$. We now want to generalize this idea to other datastructures, like lists or binary trees. In order to define the new generalized notion of induction we will need the notion of an inductive datatype.

Definition 2. *An inductive datatype $\mathcal{D}$ with respect to a signature Σ is a pair $\langle \tau, \mathbf{ctors}_\tau \rangle$ such that*

$$\forall c \in \mathbf{ctors}_\tau . \mathbf{codom}_\Sigma(c) = \tau$$

.

As inductive datatypes are well-known for anyone using functional programming, we will borrow Haskell syntax for specifying inductive datatypes. Therefore we will informally write

$$\begin{aligned} \mathsf{data}\ \ \tau = & c_1(\alpha_{1,1}, \alpha_{1,2}, ..., \alpha_{1,m_1}) \\ & | c_2(\alpha_{2,1}, \alpha_{2,2}, ..., \alpha_{2,m_2}) \\ & \vdots \\ & | c_n(\alpha_{n,1}, \alpha_{n,2}, ..., \alpha_{n,m_n}) \end{aligned}$$

for the inductive datatype $\mathcal{D}_\tau$ over a signature Σ.

$$\begin{aligned} \mathcal{D}_\tau &= \langle \tau, \{c_1, ...c_n\} \rangle \\ \Sigma &\supseteq \{ c_i :: (\alpha_{i,1}, ...\alpha_{i,m_i}) \rightsquigarrow \tau \mid i \in \{1..n\} \} \end{aligned}$$

In this syntax we can define the natural numbers, lists, and binary trees.

$$\begin{array}{llcrcl} \mathsf{data} & \mathsf{nat} & = & \mathsf{0} & | & \mathsf{s}(\mathsf{nat}) \\ \mathsf{data} & \mathsf{lst}_\alpha & = & \mathsf{nil} & | & \mathsf{cons}(\alpha, \mathsf{lst}_\alpha) \\ \mathsf{data} & \mathsf{tree}_\alpha & = & \mathsf{leaf} & | & \mathsf{internal}(\mathsf{tree}_\alpha, \alpha, \mathsf{tree}_\alpha) \end{array}$$

Now that we made the notion of an inductive datatype precise, we can generalize mathematical induction, to structural induction. The basic idea of structural induction is to show a property of a datatype, by verify the property for every case how an element could have been constructed. We will now make this generalized notion of induction formal, and as we had two ways of defining induction for natural numbers – First-order induction, as in **PA**, and second-order induction, as in **TA**– we will generalize both of them.

Definition 3 (First-order Induction)**.** *Let $\mathcal{D}_\tau = \langle \tau, \mathbf{ctors} \rangle$ be an inductive datatype. The first-order structural induction scheme of τ is defined by*

$$\bigwedge_{c \in \mathbf{ctors}} case_c \to \forall x \mathsf{P}(x) \tag{$\mathbf{I}_\tau$}$$

where

$$case_c = \mathop{\forall\!\!\!\forall}_{x_1,\ldots,x_n} \Big(\bigwedge_{i \in recursive_c} \mathsf{P}(x_i) \to \mathsf{P}(c(x_1, ..., x_n))\Big)$$
$$recursive_c = \{i \mid \mathbf{dom}_\Sigma(c,i) = \tau\}$$

For a (first-order) instance $\mathbf{I}_\tau[\theta]$ of the scheme $\mathbf{I}_\tau$, we call θ, the induction invariant, $\bigwedge_{c \in \mathbf{ctors}} case_c$ the induction premise, and $\forall x \phi[x]$ the induction conclusion of $\mathbf{I}_\tau[\theta]$.

Analogous to **TA** we can define the second order formula induction formula that entails all instances of the first-order induction scheme as follows:

Definition 4 (Second-order Induction)**.** *Let $\mathcal{D}_\tau = \langle \tau, \mathbf{ctors} \rangle$ be an inductive datatype. Then we define the formula $\mathbf{I}^2_\tau$, called the second-order induction axiom of τ, as follows:*

$$\forall P.\Big(\bigwedge_{c \in \mathbf{ctors}} case_{P,c} \to \forall x.Px\Big) \tag{$\mathbf{I}^2_\tau$}$$

where

$$case_{P,c} = \mathop{\forall\!\!\!\forall}_{x_1 \ldots x_n} \Big(\bigwedge_{i \in recursive_c} Px_i \to P(c(x_1, ..., x_n))\Big)$$
$$recursive_c = \{i \mid \mathbf{dom}_\Sigma(c,i) = \tau\}$$

It is easy to verify that indeed these notions of induction for the datatype $\mathcal{D}_{\mathsf{nat}}$ are equivalent to mathematical induction, as defined in section 2.2.1.

As the theories of arithmetic, **PA** and **TA** contained some additional axioms besides induction, we need some additional axioms general case as well. For every inductive datatype $\mathcal{D}_\tau$ need to have the axioms of disjointness, and injectivity of the constructor functions:

$$\mathop{\forall\!\!\!\forall} \bigwedge_{c,d \in \mathbf{ctors}_\tau, c \neq d} c(x_{c,1}, ..., x_{c,\mathbf{arity}c}) \not\approx d(x_{d,1}, ..., x_{d,\mathbf{arity}d}) \tag{Disj$_\tau$}$$

$$\mathop{\forall\!\!\!\forall} \bigwedge_{c \in \mathbf{ctors}_\tau} \Big(c(x_1, ..., x_{\mathbf{arity}c}) \approx c(y_1, ..., y_{\mathbf{arity}c}) \to \bigwedge_{i \in \{1,...,\mathbf{arity}c\}} x_i \approx y_i\Big) \tag{Inj$_\tau$}$$

As in the case of natural numbers $\mathbf{I}_\tau$ is a weaker approximation $\mathbf{I}^2_\tau$. In order to refine this approximation it is common to add axioms or rules for finite term algebras, like acyclicity, to the proof system as it was done in [Cru17].

As in the case of natural number arithmetic there is a specific intended semantics of inductive datatypes. As in the case of ω as intended domain of **PA** and **TA**, the semantics of inductive datatypes can be defined in terms of least fixed point operators defined by the constructor symbols [BS11, BW99].

2.2.4 Induction schemes

It is well-known that one can formalize induction different using alternative schemes to mathematical induction, while retaining a theory with the same deductive closure. In this section we will give an overview about some of the most well-known examples of induction schemes.

Strong induction

The probably most well-known alternative to mathematical induction ($\mathbf{I}_{\mathsf{nat}}$) is the principle of strong induction ($\mathbf{I}^<$) [HW17], that can be used in order to define **PA**.

$$\forall x.\Big(\forall y.(y < x \to \mathsf{P}(y)) \to \mathsf{P}(x)\Big) \to \forall x.\mathsf{P}(x) \tag{$\mathbf{I}^<$}$$

Unlike the name suggests, strong induction is not really stronger than ordinary induction. If we restrict the inductive theories to specific classes of instances to the scheme (e.g. we only allow quantifier-free formulas for induction), $\mathbf{I}^<$ may yield a stronger theory than $\mathbf{I}_{\mathsf{nat}}$, but in the presence of all first-order instances of the induction scheme, the entailed theories are equal [HW17].

The careful reader may have already noticed that $<$ is not part of the signature of **PA** as we defined it in this thesis. This can be fixed in two ways. The straight forward way would be to add the symbol $<$ to the signature, and axiomatise it appropriately:

$$\begin{aligned} &\forall x.\neg x < 0 \\ &\forall x.0 < s(x) \\ &\forall x, y.\Big(s(x) < s(y) \leftrightarrow x < y\Big) \end{aligned}$$

The second way is to not introduce any new symbols to the signature, but to define

$$x < y = \exists z.x + \mathsf{s}(z) \approx y \qquad (\text{ where } z \text{ is a variable different from } x \text{ and } y\ )$$

The latter definition is very elegant since it does not need for a change in the signature, and shows the expressive power of pure **PA**. Another advantage is that it does not involve transitivity, which is known to be bad for saturation based theorem provers. Nevertheless we will stick with the former definition of $<$ since it can be generalized more nicely to arbitrary datatypes.

The $<$ relation can be thought of as a special case of the (proper) subterm relation $\lhd$ as presented in [KRV17b] and [RV19]. Given a proper axiomatisation of the subterm relation we can define the generalized strong induction principle for a inductive datatype $\mathcal{D}_\tau$ as

$$\forall x.\Big(\forall y.(y \lhd x \to \mathsf{P}(y)) \to \mathsf{P}(x)\Big) \to \forall x.\mathsf{P}(x) \tag{$\mathbf{I}^{\lhd}$}$$

Well-founded Induction

All kinds of induction we have seen so far (and in fact all we will encounter in this thesis) can be seen as a special form of one induction principle: Well-founded induction, or also sometimes called Noetherian induction [Str12]. The principle is based on an arbitrary well-founded relation $\prec$.

$$\forall x.\Big(\forall y.(y \prec x \to \mathsf{P}(y)) \to \mathsf{P}(x)\Big) \to \forall x.\mathsf{P}(x) \tag{$\mathbf{I}^{\prec}$}$$

Syntactically well-founded induction $\mathbf{I}^{\prec}$ looks very similar to strong induction $\mathbf{I}^{\lhd}$, but semantically it is a much more general principle. The major difference is that in the case of $\mathbf{I}^{\prec}$ there is no need for any inductive datatypes. $\mathbf{I}^{\prec}$ can be used for any well-founded relation, which does not even need to be an ordering [RV19].

Intuitively the well-founded induction principle can be understood as follows: For every x we can proof that $\phi[x]$ is the case, by assuming that for all predecessors y of x, $\phi[y]$ holds. This is sound since the relation is well-founded, hence there will be a “starting point” for the inductive reasoning chain, namely an x_0, that does not have any predecessors, an has therefore to be proved without any assumption about other elements.

It is obvious that strong induction $\mathbf{I}^{\lhd}$, as well as $\mathbf{I}^{<}$ are instances of $\mathbf{I}^{\prec}$. Maybe not as obvious is that structural induction $\mathbf{I}_\tau$ can be seen an instance of $\mathbf{I}^{\prec}$ as well. The corresponding relation is the immediate (or one-step) subterm relation $\lhd_1$, as described in [KRV17b] and [RV19], which is indeed a non-transitive relation hence not an ordering.

An induction principle that is also often mentioned in the literature is the “least counterexample principle”:

$$\exists x.\neg\mathsf{P}(x) \to \exists x.\Big(\forall y.(y \prec x \to \mathsf{P}(y)) \land \neg\mathsf{P}(x)\Big)$$

The least counterexample principle is actually just the contrapositive of the well-founded induction principle, hence there is no essential difference between the two axiom schemes.

Other induction schemes

In [HW17] multiple different induction schemes for **PA** are presented, and logical relations among those schemes are investigated. In addition to the $\mathbf{I}_{\mathsf{nat}}$, and $\mathbf{I}^{<}$ the following induction schemes for **PA** are being considered in [HW17]:

n-step induction

$$\bigwedge_{k<n} \mathsf{P}(k) \land \forall x.(\mathsf{P}(x) \to \mathsf{P}(\mathsf{s}x + n)) \to \forall x.\mathsf{P}(x) \tag{$\mathbf{I}^{n\text{-step}}$}$$

n induction

$$\bigwedge_{k<n} \mathsf{P}(k) \land \forall x.(\bigwedge_{k<n} \mathsf{P}(x+k) \to \mathsf{P}(\mathsf{s}x + n)) \to \forall x.\mathsf{P}(x) \tag{$\mathbf{I}^{n}$}$$

Polynomial induction

$$\mathsf{P}(0) \land \forall x.(\mathsf{P}(x) \to \bigwedge_{k<n} \mathsf{P}((n \times x) + k)) \to \forall x.\mathsf{P}(x) \tag{$\mathbf{I}^{n\text{-poly}}$}$$

As these many different induction schemes are pretty hard to grasp, presented as formally, (instances of) all the induction schemes defined for natural numbers are visualized in table 2.1.

Relations between schemes

It is natural to ask whether the set of induction axioms I needed to prove a theorem ψ can be restricted, in order to bound the search space for proofs. We could restrict I

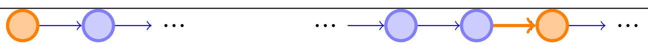
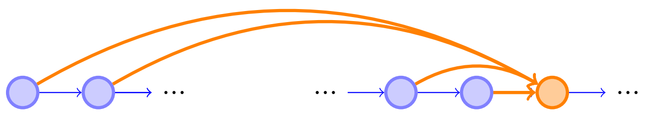
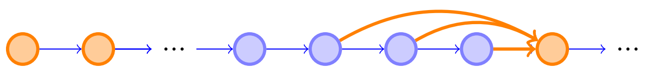
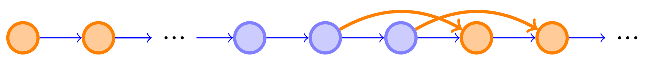
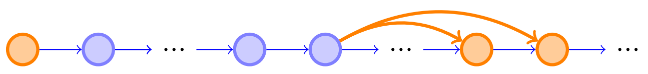
Scheme	Name	Visualization
$\mathbf{I}_{\mathsf{nat}}$	Mathematical Induction	
$\mathbf{I}^{<}$	Strong Induction	
$\mathbf{I}^{3}$	n-Induction	
$\mathbf{I}^{2\text{-step}}$	n-Step Induction	
$\mathbf{I}^{2\text{-poly}}$	Polynomial Induction	

Table 2.1: A visualization of different induction schemes on the natural numbers. The orange circles represent the numbers for which the property ϕ has to be proven, in order to prove ϕ for all numbers. The orange arrow represent for which numbers ϕ may be assumed when proving ϕ for the node on the end of the arrow. The blue arrows represent the s function.

in different ways: either by bounding the number, or by restricting shape of induction formulas. Both of these ideas are investigated in [HW17]. The first – probably surprising – result is that for a given provable formula ψ one induction axiom is sufficient to proof ψ. Still this one induction axiom can be a very complex formula, containing many other induction formulas. In fact the second result shows how complex this formula has to be: Given an arbitrary provable purely universal formula ψ, we cannot give a bound on the number of quantifier-alternations that is needed in the induction formula required to proof ψ. In fact both of these facts hold for any of the induction schemes for **PA** presented in this thesis.

2.2.5 Induction as an inference rule

So far we have considered induction in two ways. First semantically, as theories in which the domains for the inductive sorts are interpreted as terms built from designated constructor symbols, and then syntactically, as theories which axioms that contain induction formulas. Next we will take a look at another syntactic approach: considering induction as additional rule of inference in a proof system.

Sequent calculus

One example for a replacement of induction by a new rule is given in [HW17]: in Genzen's sequent calculus the first-order mathematical induction axiom scheme $\mathbf{I}_{\mathsf{nat}}$ can be elided if the following induction rule is added to the calculus:

$$\frac{\Gamma, \vdash \Delta, \phi[\mathsf{0}] \qquad \Gamma, \phi[x] \vdash \Delta, \phi[\mathsf{s}x]}{\Gamma \vdash \Delta, \forall x.\phi[x]} \quad \text{(where } \phi \text{ is arbitrary and } x \text{ does not occur in } \Delta, \text{ or } \Gamma)$$

The resulting system can proof the same formulas, but from a finite set of axioms. This nice property of finite axiomatisations comes at a cost: The cut-rule cannot be eliminated in this calculus, which is an undesirable property for automated proof search.

The ω-rule

An alternative approach, that permits cut-elimination is the ω-rule[BIS92], which can be added to Hilbert-style calculi.

$$\frac{\phi[\mathsf{0}] \qquad \phi[\mathsf{s}(\mathsf{0})] \qquad \phi[\mathsf{s}(\mathsf{s}(\mathsf{0}))] \qquad \ldots}{\forall x.\phi[x]}$$

The ω-rule is as strong as second order induction $\mathbf{I}^2_{\mathsf{nat}}$, and does not require to reason in second order logic. The obvious problem of this rule is that it requires a infinite number of hypotheses to be proven. Nevertheless methods for utilizing (weaker forms of) the ω-rule for automated reasoning have been proposed [BIS92].

2.2.6 Implicit induction

The syntactic approaches to induction we considered so far were adding induction axioms, or induction rules to a base axiom, or proof system. Both approaches can be classified as “explicit induction” for induction being present as an explicit, and identifiable component of our system. In contrast to theses explicit approaches one can also modify the base system to implicitly yield the same consequences as if one would have added induction explicitly.

Infinite descent

The first approach to implicit induction is “Infinite descent”. This method dates back to Fermat [BGP12], and drops an explicit induction rule or axiom in favour of “unrolling” inductive definitions by case splits. In the paper [BS11] this idea is explored in-depth: a sequent style calculus in which features proofs with branches of infinite depth, as long as these infinite branches obey some global soundness conditions, is defined and cut-free soundness, and completeness of this calculus with respect to standard models is proven. Unsurprisingly the theorems of this powerful calculus are not recursively enumerable. Nevertheless based on this theoretical tool a recursively enumerable subset of that calculus, which is known as cyclic proofs, is presented in the same paper, which was later shown to be more powerful than explicit first-order induction [BT19].

Proof by consistency

Another implicit induction method is so-called induction-less induction, also known as proof by consistency [Com94]. This method targets at proving validity in standard models, and does it by reducing the proof of validity of $\mathcal{E} \vdash \phi$ a consistency check of the set of formulas $\{\phi\} \cup \mathcal{A} \cup \mathcal{E}$, where $\mathcal{A}$ is a first-order axiomatisation of the standard model for some induction constructors. Finding such an $\mathcal{A}$ is not possible in general, which is the major caveat of this method.

CHAPTER 3

State of the Art

As induction can be understood in many different ways from a theoretical perspective, it also has been tackled with a variety of approaches practically. Approaches include provers verifying equational properties of functional programs [PCI$^+$20, SDE12, CJRS12], systems that improve automation for inductive proofs in interactive theorem provers [DJ07], equipping SMT-solvers with inductive reasoning capabilities [RK15, Lei12], and solvers supporting full first-order logic with equality, and induction[Cru17, RV19].

In this chapter we will have a closer look at some of the techniques that have been developed.

3.0.1 Induction and saturation

As the best state of the art theorem provers featuring first-order logic are using saturation algorithms, there have been various attempts on integrating induction into saturation algorithms. The superposition based theorem prover VAMPIRE, considers induction as an additional inference rule, that introduces the induction axiom and immediately resolves it with another clause if the clause matches certain heuristics, that keep the search space from blowing up [RV19].

ZIPPERPOSITION [Cru17] integrates induction with superposition and rewrite rules, relying on the AVATAR architecture [Vor14] for handling case splits on variables of inductive datatypes efficiently.

3.0.2 Cyclic proofs

A different approach is taken in [EP20, KP13], where the ideas of cyclic reasoning described in section 2.2.6 are lifted from the goal-oriented sequent-style calculus to saturation algorithms. Although experiments are mentioned in [KP13], to the authors knowledge there is no implementation of either of the two calculi publicly available.

An implementation of the goal oriented cyclic reasoning described in [BGP12] is Cyclist, a generic theorem prover that supports cyclic reasoning, not only for inductive definitions, but also for entailments in pure separation logic, and Hoare style termination proofs.

3.0.3 Rippling

A famous heuristic used for automating induction is rippling, a strategy for transforming one formula into another, using a set of rewrite rules. In the context of inductive theorem proving it is used for proving the inductive step, after fixing the induction invariant and scheme [DJ07]. IsaPlanner [DJ07] is a practical framework supporting this heuristic. It provides a domain specific language for writing so-called proof plans to automate reasoning in the proof assistant Isabelle.

3.0.4 Theory exploration

As mentioned in Section 1.1, inductive proofs are not analytic in some proof systems, which means that it may be required to introduce arbitrary lemmas, or induction formulas during proof search. There are different techniques for finding potentially useful formulas, often summarized under the name theory exploration. One system implementing theory exploration, is HipSpec [CJRS12]. HipSpec aims at proving properties about Haskell programs inductively. In order to find potential lemmas randomly generated equational formulas are evaluated on random inputs. If those formulas hold for a reasonably number of inputs, HipSpec makes a proof attempt with a certain timeout, and adds the formula as an axiom on success.

Another method for theory exploration is implemented in Cvc4 [RK15]. There potential lemmas are found by enumerating formulas that match certain heuristics, ensuring the generated lemmas can be used efficiently, and are probable to hold due to the current state of the solver. A potential lemma ϕ is then added to the search space as an instance of the law of the excluded middle axiom $\phi \vee \neg\phi$.

Zipperposition uses an approach that is a combination of the two previous ones. When one of its heuristics proposes an induction lemma, the solver will try to evaluate it on some inputs. If the evaluations confirm the lemma proposed by the heuristic, it adds a version of the law of the excluded middle for this formula to the search space.

3.0.5 Recursion analysis

The heuristics used in the theorem provers Acl2 [Moo19], and Imandra [PCI+20], in their basic form, go back to ideas from the 70ies [BM75]. The technique aims at proving universal properties of functional programs in Lisp. The basic idea is that the solver evaluates the goal by unfolding inductive definitions, until it fails due to some uninterpreted symbols (corresponding to universally quantified variables) occur. If those symbols are used for case splits in recursive definitions, induction on this symbol is applied to the goal. For proving the induction step evaluation as well as so-called “fertilization” is used, meaning that the induction hypothesis is used to rewrite the goal.

3.0.6 Generalization

As described in [HHK$^+$20] applying induction to all occurrences of a variable in a formula at once is not always sufficient; in many cases a step of generalization is needed before selecting the induction variable. Aubin’s work [Aub79] builds upon the heuristics of Boyer and Moore by introducing a generalization heuristic. Boyer and Moores heuristic evaluates terms and only applies induction to uninterpreted symbols where evaluation fails. In addition to that Aubin suggests to apply induction to terms who’s evaluation would eventually lead to such a symbol. Furthermore Aubin gives generalization heuristics depending on how an argument is used in recursive function definitions. A modern solver implementing these heuristics is Zipperposition [Cru17].

Vampire features heuristics for generalizing induction as well [HHK$^+$20]. Its approach is to apply induction not only to all positions of one variable at once, but also to apply induction on an arbitrary subset of these positions. This generalization technique can not only be used in the context of functional programs with fully specified function definitions, but also in pure first-order logic.

3.0.7 SMT solvers

Due to the huge progress in SMT solving, SMT-solvers have been used for inductive reasoning as well. The approaches taken by Dafny [Lei12] and Cvc4 [RK15] both rely on inductive strengthening, which means universal formulas are replaced by the induction premise of the corresponding induction axiom. Cvc4 takes this approach during proof search, while Dafny employs inductive strengthening as a step of preprocessing.

CHAPTER 4

Induction by Reflection

In this chapter we will introduce a different approach to induction, which is based on the idea of Gödel encodings. The major difference to most other approaches we saw so far is that is does not require any adjustments of the proof system, and can therefore be combined with any state of the art theorem prover that supports sorted first-logic. Since second-order induction is not even semi-decidable, our aim is to only handle first-order induction, but to do this in a complete manner.

Given a theory $\mathcal{T}$ and a set of inductive data types $\mathcal{D}$, the idea of our approach is to build a finite conservative extension $\dot{\mathcal{T}}$ of the first-order inductive theory of $\mathcal{T}$ over the datatypes $\mathcal{D}$. Formally this means we want to find a finite set of formulas $\dot{\mathcal{T}}$ such that

$$\forall \phi \in \mathbf{Form}^{\mathcal{T}}. \Big(\dot{\mathcal{T}} \vdash \phi \iff \mathcal{T} \cup \{ \mathbf{I}_\tau[\psi] \mid \mathcal{D}_\tau \in \mathcal{D}, \psi \in \mathbf{Form}^{\mathcal{T}} \} \vdash \phi \Big)$$

As a means to this end we introduce some basic ideas from the field of axiomatic theories of truth, in order enable us to extend a base theory with the vocabulary to talk about formulas, this extension will be called the reflective extension of a theory. In another step we can add induction a single formula that contains quantifier over formulas in order to express the induction schema in a single formula.

4.1 Axiomatic theories of truth

Axiomatic theories of truth is a field of mathematical logic that originates from the undefinability theorem discovered by Tarski [Hor11]. This theorem is based on Gödel's encoding used for his famous incompleteness theorems. Therefore Gödel encodes formulas as numbers, and shows that **PA** is powerful enough to express and proof certain facts about these encodings. Tarski's discovery about **PA** was that there is no formula $\mathbf{T}[x]$, that expresses truth in **PA**.

Theorem 1 (Tarski's undefinability theorm)**.**
Let $\ulcorner\cdot\urcorner : \mathbf{Form^{PA}} \mapsto \mathbf{Term^{PA}}$ *be any Gödel encoding of the language of* **PA***.*

Then there is no formula $\mathbf{T}[x] \in \mathbf{Form^{PA}}$ *such that*

$$\forall \phi \in \mathbf{Form^{PA}}.\mathbf{PA} \vdash \mathbf{T}[\ulcorner\phi\urcorner] \leftrightarrow \phi$$

Based on this theorem there were different approaches of extending **PA** with new predicates and axioms, in order to be able to express the truth predicate of **PA**, and even other theories [Hor11, Bek05]. In order to build our finite axiomatisation of induction, we build on the idea of a compositional theory of truth, as defined in [Hor11] chapter 6. There Peano arithmetic is extended to **TC** by an additional predicate symbol $\mathsf{T} :: \mathsf{Pred}(\mathsf{nat})$ which is axiomatised as follows:

$$\forall \text{ atomic } \phi \in \mathcal{L}_{\mathbf{PA}}.\mathsf{T}(\phi) \leftrightarrow val^{+}(\phi) \tag{4.1}$$
$$\forall \phi \in \mathcal{L}_{\mathbf{PA}}.\mathsf{T}(\neg\phi) \leftrightarrow \neg\mathsf{T}(\phi) \tag{4.2}$$
$$\forall \phi, \psi \in \mathcal{L}_{\mathbf{PA}}.\mathsf{T}(\phi \wedge \psi) \leftrightarrow \mathsf{T}(\phi) \wedge \mathsf{T}(\psi) \tag{4.3}$$
$$\forall \phi(x) \in \mathcal{L}_{\mathbf{PA}}.\mathsf{T}(\forall x.\phi) \leftrightarrow \forall x.\mathsf{T}(\phi(x)) \tag{4.4}$$

where $val^{+}(x)$ is some formula defining the truth of the atomic formula. Note that this axiomatisation is not pure first order logic but contains a lot of notation conventions defined in chapter 3 of the book. Dropping these conventions the axiomatisation looks something like this:

$$\forall x.(Atom(x) \wedge Form(x) \rightarrow \mathsf{T}(x) \leftrightarrow val^{+}(x))$$
$$\forall x.(Form(x) \rightarrow \exists y.(Negation(y, x) \wedge (\mathsf{T}(z) \leftrightarrow \neg\mathsf{T}(x))))$$
$$\forall x, y.(Form(x) \wedge Form(y) \rightarrow \exists z.Conj(z, x, y) \wedge \mathsf{T}(z) \leftrightarrow \mathsf{T}(x) \wedge \mathsf{T}(y))$$
$$\forall p, v.(Form(p) \wedge Var(v) \rightarrow (\exists z.For(z, p, v) \wedge (\mathsf{T}(z) \leftrightarrow \forall v'.(Var(v') \rightarrow \exists p'.Subs(p', p, v, v') \wedge \mathsf{T}(p')))))$$

where $Form(x)$, $Var(x)$, $Atom(x)$ are formulas in **PA** that express that x is the Gödel-encoding of a formula, variable, or atom respectively. $Negation(y, x)$ is a formula that encodes y is the negation of the formula encoded by x. Analogously $Conj(...)$, $For(...)$, and $Subs(...)$ express logical conjunction, universal quantification, and syntactic substitution.

Basically **TC** fits for our purpose, since it has the nice property that it has a truth predicate T for **PA**, and it allows us to quantify over formulas, which allows us to express the induction scheme in a single formula. But this approach still has some problems: Firstly it still requires to reason in **PA** which includes the infinite induction scheme, and secondly would only extend **PA** with finite induction. The first problem would probably be resolvable by just dropping the induction scheme from **TC** in favor of an induction formula quantifying over formula codes, but we will take another approach resolving both problems at once.

4.2 Reflection in an arbitrary theory

In order to finitely axiomatise induction we will have to extend our base theory, with the vocabulary to talk about formulas. Instead of the classical approach from axiomatic theories of truth we will not rely on numbers to encode formulas, but we will directly introduce a new sort of formulas to our theory.

Firstly we need extend our signature Σ with the sorts needed to talk about first-order formulas. The basic idea is that each meta-level construction that is needed in order to define the syntax and the semantics of first-order logic, needs a "reflective" counter part on the object level.

4.2.1 Signature

Definition 5 (Reflective signature)**.** *Let Σ be an arbitrary signature. We define $\dot{\Sigma}$ to be the reflective extension of Σ.*

$$\begin{aligned}
\dot{\Sigma} = \Sigma &\cup \{\mathsf{v}_0^\sigma :: \mathsf{var}_\sigma \mid \sigma \in \mathbf{sorts}_\Sigma\} \\
&\cup \{\mathsf{next}_\sigma :: \mathsf{var}_\sigma \rightsquigarrow \mathsf{var}_\sigma \mid \sigma \in \mathbf{sorts}_\Sigma\} \\
&\cup \{\mathsf{inj}_\sigma :: \mathsf{var}_\sigma \rightsquigarrow \mathsf{term}_\sigma \mid \sigma \in \mathbf{sorts}_\Sigma\} \\
&\cup \{\dot{f} :: \mathsf{term}_{\sigma_1} \times \ldots \times \mathsf{term}_{\sigma_n} \rightsquigarrow \mathsf{term}_\sigma \mid f :: \sigma_1 \times \ldots \times \sigma_n \rightsquigarrow \sigma \in \Sigma\} \\
&\cup \{\dot{P} :: \mathsf{term}_{\sigma_1} \times \ldots \times \mathsf{term}_{\sigma_n} \rightsquigarrow \mathsf{form} \mid P :: \mathsf{Pred}(\sigma_1 \times \ldots \times \sigma_n) \in \Sigma\} \\
&\cup \{\dot{\approx}_\sigma :: \mathsf{term}_\sigma \times \mathsf{term}_\sigma \rightsquigarrow \mathsf{form} \mid \sigma \in \mathbf{sorts}_\Sigma\} \\
&\cup \{\dot{\bot}\} \\
&\cup \{\dot{\vee} :: \mathsf{form} \times \mathsf{form} \rightsquigarrow \mathsf{form}\} \\
&\cup \{\dot{\neg} :: \mathsf{form} \rightsquigarrow \mathsf{form}\} \\
&\cup \{\dot{\forall}_\sigma :: \mathsf{var}_\sigma \times \mathsf{form} \rightsquigarrow \mathsf{form} \mid \sigma \in \mathbf{sorts}_\Sigma\} \\
&\cup \{\mathsf{empty} :: \mathsf{env} \mid \sigma \in \mathbf{sorts}_\Sigma\} \\
&\cup \{\mathsf{push}_\sigma :: \mathsf{env} \times \mathsf{var}_\sigma \times \sigma \rightsquigarrow \mathsf{env} \mid \sigma \in \mathbf{sorts}_\Sigma\} \\
&\cup \{\mathsf{eval}_\sigma^v :: \mathsf{env} \times \mathsf{var}_\sigma \rightsquigarrow \sigma \mid \sigma \in \mathbf{sorts}_\Sigma\} \\
&\cup \{\mathsf{eval}_\sigma :: \mathsf{env} \times \mathsf{term}_\sigma \rightsquigarrow \sigma \mid \sigma \in \mathbf{sorts}_\Sigma\} \\
&\cup \{\dot{\models} :: \mathsf{Pred}(\mathsf{env} \times \mathsf{form})\}
\end{aligned}$$

where all newly introduced symbols are disjoint from the ones in Σ.

4.2.2 Intended semantics

As this definition is rather lengthy we will break down the intended semantics of all newly introduced symbols. We can split the definitions into two parts: one formalizing the syntax and one formalizing the semantics of our reflective first-order logic.

Reflective syntax

Variables The first thing that is needed to formalize the semantics of first order logic is a countably infinite set of variables $\mathbf{Var}_\sigma$ for each sort σ. Therefore we introduce a new sort var_σ that is intended to by interpreted as $\mathbf{Var}_\sigma$. The two functions v_0^σ, and next_σ that are added to the signature can be thought of as the constructors for this infinite set of variables. This means v_0^σ is intended to be interpreted as the variable x_0, $\mathsf{next}_\sigma(\mathsf{v}_0^\sigma)$ is meant to be interpreted as x_1, and so on. As it will become handy later we will introduce the following syntactic sugar for variables:

$$\mathsf{v}_{i+1}^\sigma = \mathsf{next}_\sigma(\mathsf{v}_i^\sigma) \qquad\qquad \text{for } i \geq 0$$

Terms The next step in the meta-level definition of the syntax of first-order logic is the set of terms $\mathbf{Term}_\sigma$ of sort σ. Therefore we introduce the sort term_σ for each sort σ in the signature Σ. Terms are defined inductively:

The base case is a variable. Since variables and terms are of different sorts, we need the function inj_σ to turn variables into terms. This function is intended to be interpreted as the identity function.

The step case of the inductive definition is building terms out of function symbols and other terms. Therefore we need to introduce a reflective function symbol $\dot{f}$ for every function f in the signature. The $\dot{f}$ is intended to be interpreted as the function symbol f, while f itself is interpreted as an actual function.

Formulas As for terms, formulas **Form** are defined inductively on the meta level. Therefore form id defined by functions that represent the different cases of the definition of formulas:

Atomic formulas are either equalities of predicate symbols "applied" to terms. Therefore we introduce a reflective equality symbol $\dot{\approx}_\sigma$ for each sort σ and a reflective version $\dot{P}$ for every predicate symbol P. $\dot{\approx}_\sigma$ is intended to be interpreted as $\approx$, which itself is interpreted as the actual equality relation $=$. Analogously $\dot{P}$ is intended to be interpreted as the predicate symbol P.

Even though it's not strictly necessary to have the nullary connective $\bot$ in order to get the expressive power of full first-order logic, we nevertheless introduce it to our language. The reflective symbol $\dot{\bot}$ is intended to be interpreted as the formula $\bot$. Complex formulas are built from atomic formulas and connectives, or quantifiers. Therefore we introduce a functionally complete set of reflective connectives, namely $\dot{\vee}$, and $\dot{\neg}$ that are to be interpreted as $\vee$, and $\neg$ respectively. As it will help in terms of readability, we will use infix notation for $\dot{\vee}$, and drop the parenthesis for $\dot{\neg}$ if there is no ambiguity.

In order to formalize quantification we introduce a function $\dot{\forall}_\sigma$ for each sort. As expected this function is intended to be interpreted as universal quantification over a variable. In order to visually tie our terms more closely to their intended interpretations we will write $\dot{\forall}x{:}\sigma.p$ for the term $\dot{\forall}_\sigma(x, p)$.

Reflective semantics

In order to be able to axiomatise the meaning of formulas correctly, we will need syntactic representations of the semantic structures needed to define the semantics of first-order logic.

Environment In order to define the meaning of a quantifier, we need to be able to redefine the meaning of a variable within the scope of the quantifier. Therefore we will use a stack of variable interpretations, we will call an environment. The idea is that a variable v_i^σ is freely interpreted in an empty environment empty, while it is interpreted as x if the tuple $\langle \mathsf{v}_i^\sigma, t\rangle$ was pushed on the stack using $\mathsf{push}_\sigma(e, \mathsf{v}_i^\sigma, t)$. The idea will become clearer in sections 4.2.3, and 4.2.4, where we will axiomatise the meaning and define a model of the reflective theory. Roughly one could also see the environment also as a partial definition of the variable interpretation of an interpretation of the theory.

Evaluation In order to make use of the environment we will need a reflective evaluation function for terms eval_σ and eval_σ^v that corresponds to interpreting terms and variables in some model $\mathcal{I}$ of the reflective theory.

Satisfaction Finally we have our reflective satisfaction relation $\dot{\models}$. We will write $(e \mathrel{\dot{\models}} p)$ for $\dot{\models}\ (e, p)$, which can roughly be interpreted as "the interpretation $\mathcal{I}$ partially defined by e satisfies p ($\mathcal{I} \models p$)". Our truth $\mathbf{T}$ predicate in the Tarskian sense, will finally be $\mathbf{T}(x) = (\mathsf{empty} \mathrel{\dot{\models}} x)$.

4.2.3 Axiomatisation

In this section we will formally define the intended semantics described in the previous section, relating reflective with non-reflective function and predicate symbols, by defining the meaning of the reflective satisfaction relation $\dot{\models}$, the meaning of the reflective evaluation functions eval_σ and eval_σ^v. All axioms we list will be implicitly universally quantified and be present for every sort $\sigma, \tau \in \mathbf{sorts}_\Sigma$. Finally the reflective extension $\dot{\mathcal{T}}$ of our base theory $\mathcal{T}$ will the union of all these axioms together with the $\mathcal{T}$ itself.

Reflective variable interpretation

As described in the previous section, the interpretation of variables in an empty environment empty is undefined. In contrast an environment to which a variable v, and a value x is pushed, evaluates the variable v to x.

$$\mathsf{eval}_\sigma^v(\mathsf{push}_\sigma(e, v, x), v) = x \qquad (\mathsf{Ax}_{\mathsf{eval}_0^v})$$

$$v \not\approx v' \to \mathsf{eval}_\sigma^v(\mathsf{push}_\sigma(e, v', x), v) = \mathsf{eval}_\sigma^v(e, v) \qquad (\mathsf{Ax}_{\mathsf{eval}_1^v})$$

$$\mathsf{eval}_\sigma^v(\mathsf{push}_\tau(e, w, x), v) = \mathsf{eval}_\sigma^v(e, v) \qquad \text{for } \sigma \neq \tau \qquad (\mathsf{Ax}_{\mathsf{eval}_2^v})$$

Reflective evaluation

The function symbol eval_σ defines the value of a reflective term t and thereby maps the reflective functions $\dot{f}$ to their non-reflective counter parts f. For variables eval_σ, just forwards the evaluation to eval^v_σ.

$$\mathsf{eval}_\sigma(e, \mathsf{inj}_\sigma(v)) = \mathsf{eval}^v_\sigma(e, v) \qquad (\mathsf{Ax}_{\mathsf{eval}_{var}})$$
$$\mathsf{eval}_\sigma(e, \dot{f}(t_1, ..., t_n)) = f(\mathsf{eval}_{\sigma_1}(e, t_1), ..., \mathsf{eval}_{\sigma_n}(e, t_n)) \qquad (\mathsf{Ax}_{\mathsf{eval}_f})$$
$$\text{for } f : \sigma_1 \times ... \times \sigma_n \rightsquigarrow \sigma \in \Sigma$$

Reflective satisfaction

The predicate symbol $\dot{\vDash}\, \sigma$ defines the truth of a formula with respect to some variable interpretation, by defining the meaning of the reflective connectives and the quantifier in terms of their object-level counter parts:

$$(e \dot{\vDash} x \dot{\approx}_\sigma y) \leftrightarrow \mathsf{eval}_\sigma(e, x) \approx \mathsf{eval}_\sigma(e, y) \qquad (\mathsf{Ax}_{\dot{\approx}})$$
$$(e \dot{\vDash} \dot{P}(t_1, ..., t_n)) \leftrightarrow P(\mathsf{eval}_{\sigma_1}(e, t_1), ..., \mathsf{eval}_{\sigma_n}(e, t_n)) \quad \text{for } P : \mathsf{Pred}(\sigma_1 \times ... \times \sigma_n) \qquad (\mathsf{Ax}_P)$$
$$(e \dot{\vDash} \dot{\bot}) \leftrightarrow \bot \qquad (\mathsf{Ax}_{\dot{\bot}})$$
$$(e \dot{\vDash} \dot{\neg}\, \phi) \leftrightarrow \neg(e \dot{\vDash} \phi) \qquad (\mathsf{Ax}_{\dot{\neg}})$$
$$(e \dot{\vDash} \phi \,\dot{\vee}\, \psi) \leftrightarrow (e \dot{\vDash} \phi) \vee (e \dot{\vDash} \psi) \qquad (\mathsf{Ax}_{\dot{\vee}})$$
$$(e \dot{\vDash} \dot{\forall} v{:}\sigma.\phi) \leftrightarrow \forall x : \sigma.(\mathsf{push}_\sigma(e, v, x) \dot{\vDash} \phi) \qquad (\mathsf{Ax}_{\dot{\forall}})$$

4.2.4 Consistency and Conservativeness

Now that we have specified our theory we need to ensure that it indeed models what we intended to. Therefore we have to ensure that $\dot{\mathcal{T}}$ is a conservative extension of $\mathcal{T}$, and secondly that $\dot{\mathcal{T}}$ is consistent. In general we cannot ensure that $\dot{\mathcal{T}}$ is consistent, since already the base theory $\mathcal{T}$ could have been inconsistent. Hence we will show that $\dot{\mathcal{T}}$ is consistent if $\mathcal{T}$ is consistent.

In order to proof both, conservativeness and consistency, we introduce the notion of an reflective interpretation $\dot{\mathcal{M}}$, that is based on $\mathcal{M}$. The idea is that $\dot{\mathcal{M}}$ interprets every symbol in the base theory $\mathcal{T}$ as it would be interpreted in $\mathcal{M}$, hence all for every formula in $\mathbf{Form}^{\mathcal{T}}$ is true in $\dot{\mathcal{M}}$ iff it is true in $\mathcal{M}$. Due to soundness and completeness of first-order logic we get that $\dot{\mathcal{T}}$ is indeed a conservative extension of $\mathcal{T}$. Further due to the fact that for every model of $\mathcal{M}$ of $\mathcal{T}$ we have model $\dot{\mathcal{M}}$ of $\dot{\mathcal{T}}$, we also have that the theory is consistent if $\mathcal{T}$ is consistent. In order to ensure this reasoning is correct we need to ensure that $\dot{\mathcal{M}}$ also satisfies the axioms we introduced for reflective theories. This will be done by interpreting the new reflective sort form as the set of first order formulas $\mathbf{Form}$, and interpreting the sort term_σ as terms of sort $\mathbf{Term}_\sigma$.

Formally the reflective model can be defined as follows:

Let $\mathcal{M} = \langle\langle \Delta_{\sigma_1}, ..., \Delta_{\sigma_n}\rangle, \mathcal{I}\rangle$ be a first-order interpretation over the signature Σ. We define the reflective model $\dot{\mathcal{M}}$ to be

$$\dot{\mathcal{M}} = \langle\langle \Delta_{\sigma_1}, ..., \Delta_{\sigma_n}, \mathbf{Term}_{\sigma_1}, ...\mathbf{Term}_{\sigma_n}, \mathbf{Form}\rangle, \dot{\mathcal{I}}\rangle$$

$$\dot{\mathcal{I}}(f) : \Delta_{\sigma_1} \times ... \times \Delta_{\sigma_n} \mapsto \Delta_\sigma \qquad \text{for } f :: \sigma_1 \times ... \times \sigma_n \rightsquigarrow \sigma \in \Sigma$$
$$\dot{\mathcal{I}}(f) = \mathcal{I}(f)$$

$$\dot{\mathcal{I}}(P) : \mathcal{P}(\Delta_{\sigma_1} \times ... \times \Delta_{\sigma_n}) \qquad \text{for } P :: \mathsf{Pred}(\sigma_1 \times ... \times \sigma_n) \in \Sigma$$
$$\dot{\mathcal{I}}(P) = \mathcal{I}(P)$$

$$\dot{\mathcal{I}}(\mathsf{v}_0^\sigma) : \mathbf{Var}_\sigma \qquad \text{for } \sigma \in \mathbf{sorts}_\Sigma$$
$$\dot{\mathcal{I}}(\mathsf{v}_0^\sigma) = \mathsf{x}_0$$

$$\dot{\mathcal{I}}(\mathsf{next}_\sigma) : \mathbf{Var}_\sigma \mapsto \mathbf{Var}_\sigma \qquad \text{for } \sigma \in \mathbf{sorts}_\Sigma$$
$$\dot{\mathcal{I}}(\mathsf{next}_\sigma)(\mathsf{x}_i) = \mathsf{x}_{i+1}$$

$$\dot{\mathcal{I}}(\mathsf{inj}_\sigma) : \mathbf{Var}_\sigma \mapsto \mathbf{Term}_\sigma \qquad \text{for } \sigma \in \mathbf{sorts}_\Sigma$$
$$\dot{\mathcal{I}}(\mathsf{inj}_\sigma)(x) = x$$

$$\dot{\mathcal{I}}(\dot{f}) : \mathbf{Term}_{\sigma_1} \times ... \times \mathbf{Term}_{\sigma_n} \mapsto \mathbf{Term}_\sigma \qquad \text{for } f :: \sigma_1 \times ... \times \sigma_n \rightsquigarrow \sigma \in \Sigma$$
$$\dot{\mathcal{I}}(\dot{f})(t_1, ..., t_n) = f(\dot{\mathcal{I}}(t_1), ..., \dot{\mathcal{I}}(t_n))$$

$$\dot{\mathcal{I}}(\dot{P}) : \mathbf{Term}_{\sigma_1} \times ... \times \mathbf{Term}_{\sigma_n} \mapsto \mathbf{Form} \qquad \text{for } P :: \mathsf{Pred}(\sigma_1 \times ... \times \sigma_n) \in \Sigma$$
$$\dot{\mathcal{I}}(\dot{P})(t_1, ..., t_n) = P(\dot{\mathcal{I}}(t_1), ..., \dot{\mathcal{I}}(t_n))$$

$$\dot{\mathcal{I}}(\dot{\approx}_\sigma) : \mathbf{Term}_\sigma \times \mathbf{Term}_\sigma \mapsto \mathbf{Form} \qquad \text{for } \sigma \in \mathbf{sorts}_\Sigma$$
$$\dot{\mathcal{I}}(\dot{\approx}_\sigma)(s, t) = \dot{\mathcal{I}}(s) \approx \dot{\mathcal{I}}(t)$$

$$\dot{\mathcal{I}}(\dot{\bot}) : \mathbf{Form}$$
$$\dot{\mathcal{I}}(\dot{\bot}) = \bot$$

$$\dot{\mathcal{I}}(\dot{\vee}) : \mathbf{Form} \times \mathbf{Form} \mapsto \mathbf{Form}$$
$$\dot{\mathcal{I}}(\dot{\vee})(\phi, \psi) = \phi \vee \psi$$

$$\dot{\mathcal{I}}(\dot{\neg}) : \mathbf{Form} \mapsto \mathbf{Form}$$
$$\dot{\mathcal{I}}(\dot{\neg})(\phi) = \neg\phi$$

$$\dot{\mathcal{I}}(\dot{\forall}_\sigma) : \mathbf{Var}_\sigma \times \mathbf{Form} \mapsto \mathbf{Form} \qquad \text{for } \sigma \in \mathbf{sorts}_\Sigma$$
$$\dot{\mathcal{I}}(\dot{\forall}_\sigma)(\mathsf{x}_i, \phi) = \forall \mathsf{x}_i : \sigma.\phi$$

$$\dot{\mathcal{I}}(\mathsf{empty}) : \mathbf{Interpret}_\Sigma$$
$$\dot{\mathcal{I}}(\mathsf{empty}) = \mathcal{I}$$

$$\dot{\mathcal{I}}(\mathsf{push}_\sigma) : \mathbf{Interpret}_\Sigma \times \mathbf{Var}_\sigma \times \sigma \mapsto \mathbf{Interpret}_\Sigma \qquad \text{for } \sigma \in \mathbf{sorts}_\Sigma$$
$$\dot{\mathcal{I}}(\mathsf{push}_\sigma)(\mathcal{J}, \mathsf{x}_i, v)(x) = \begin{cases} v & \text{if } x = \mathsf{x}_i \\ \mathcal{J}(x) & \text{otherwise} \end{cases}$$

$$\dot{\mathcal{I}}(\mathsf{eval}^v_\sigma) : \mathbf{Interpret}_\Sigma \times \mathbf{Var}_\sigma \mapsto \Delta_\sigma \qquad \text{for } \sigma \in \mathbf{sorts}_\Sigma$$
$$\dot{\mathcal{I}}(\mathsf{eval}^v_\sigma)(\mathcal{J}, \mathsf{x}_i) = \mathcal{J}(\mathsf{x}_i)$$

$$\dot{\mathcal{I}}(\mathsf{eval}_\sigma) : \mathbf{Interpret}_\Sigma \times \mathbf{Term}_\sigma \mapsto \Delta_\sigma \qquad \text{for } \sigma \in \mathbf{sorts}_\Sigma$$
$$\dot{\mathcal{I}}(\mathsf{eval}_\sigma)(\mathcal{J}, t) = \mathcal{J}(t)$$

$$\dot{\mathcal{I}}(\dot{\vDash}) : \mathcal{P}(\mathbf{Interpret}_\Sigma \times \mathbf{Form})$$
$$\dot{\mathcal{I}}(\dot{\vDash}) = \{\langle \mathcal{J}, \phi\rangle \in \mathbf{Interpret}_\Sigma \times \mathbf{Form} \mid \mathcal{J} \vDash \phi\}$$

In order to ensure that $\dot{\mathcal{M}}$ is indeed a model of $\dot{\mathcal{T}}$ we actually would need to ensure that $\dot{\mathcal{M}}$ satisfies all axioms we introduced for the reflective theory. This would only involve repeating the axioms in natural language, defining the semantics of first-order logic on the meta level. Since this is not very insightful, and does not give any more confidence in the correctness of the axiomatisation, we will skip this part.

Truth predicate

Now that we ensured that $\dot{\mathcal{T}}$ is a conservative extension of $\mathcal{T}$, we need to ensure that the theory really is a theory of truth. This means we need to ensure that the theory has a truth predicate. In the definition we gave before we needed a Gödel encoding in order to define our truth predicate. This Gödel encoding maps variables, terms, and formulas to numerals. Since our theory $\dot{\mathcal{T}}$ does not necessarily contain any number symbols, we need to use a generalized definition of a Gödel encoding, namely that it maps variables, terms, and formulas in our base language $\mathbf{Form}^{\mathcal{T}}$ to terms in our extended language $\mathbf{Form}^{\dot{\mathcal{T}}}$. In exact we map formulas $\mathbf{Form}$ to terms of sort form, variables $\mathbf{Var}_\sigma$ to var_σ and $\mathbf{Term}_\sigma$ to term_σ. Formally we define our Gödel encoding as follows:

$$\begin{aligned}
\ulcorner \phi \vee \psi \urcorner &= \ulcorner \phi \urcorner \mathbin{\dot{\vee}} \ulcorner \psi \urcorner && && (\,\mathsf{Gdl}_\vee) \\
\ulcorner \neg \phi \urcorner &= \dot{\neg}\, \ulcorner \phi \urcorner && && (\,\mathsf{Gdl}_\neg) \\
\ulcorner \bot \urcorner &= \dot{\bot} && && (\,\mathsf{Gdl}_\bot) \\
\ulcorner \forall \mathsf{x}_i : \sigma . \phi \urcorner &= \dot{\forall} \mathsf{v}_i^\sigma {:} \sigma . \ulcorner \phi \urcorner && && (\,\mathsf{Gdl}_\forall) \\
\ulcorner \mathsf{x}_n \urcorner &= \mathsf{inj}_\sigma(\mathsf{v}_n^\sigma) && \text{where } \mathsf{x}_n \in \mathbf{Var}_\sigma && (\,\mathsf{Gdl}_\mathsf{x}) \\
\ulcorner s \approx t \urcorner &= \ulcorner s \urcorner \mathbin{\dot{\approx}_\sigma} \ulcorner t \urcorner && \text{where } s,t \in \mathbf{Term}_\sigma && (\,\mathsf{Gdl}_\approx) \\
\ulcorner f(t_1, ..., t_n) \urcorner &= \dot{f}(\ulcorner t_1 \urcorner, ..., \ulcorner t_n \urcorner) && && (\,\mathsf{Gdl}_f) \\
\ulcorner P(t_1, ..., t_n) \urcorner &= \dot{P}(\ulcorner t_1 \urcorner, ..., \ulcorner t_n \urcorner) && && (\,\mathsf{Gdl}_P)
\end{aligned}$$

Now that we have defined our Gödel encoding we can show that $\dot{\mathcal{T}}$ contains a truth predicate $\mathbf{T}[\phi]$ for $\mathcal{T}$, namely the formula $\mathsf{empty} \mathrel{\dot{\models}} \ulcorner \phi \urcorner$. Therefore we need to show that the following theorem holds.

Theorem 2 (Truth Predicate)**.**

$$\forall \phi \in \mathbf{Form}^{\mathcal{T}}. \Big(\dot{\mathcal{T}} \vdash \phi \leftrightarrow (\mathsf{empty} \mathrel{\dot{\models}} \ulcorner \phi \urcorner) \Big)$$

Proof. In order to proof this theorem inductively we will need to strengthen our goal to:

$$\forall e \in \mathbf{stack}. \dot{\mathcal{T}} \vdash \phi \leftrightarrow (e \mathrel{\dot{\models}} \ulcorner \phi \urcorner)$$

where we define the set $\mathbf{stack}$ inductively as the least set such that

- $\mathsf{empty} \in \mathbf{stack}$
- $e \in \mathbf{stack} \;\&\; \sigma \in \mathbf{sorts} \;\&\; i \in \mathbb{N} \implies \mathsf{push}_\sigma(e, \mathsf{v}_i^\sigma, \mathsf{x}_i) \in \mathbf{stack}$

Next we will rewrite our goal to to:

$$\forall e \in \mathbf{stack}. \dot{\mathcal{T}} \vdash \phi \iff \dot{\mathcal{T}} \vdash (e \mathrel{\dot{\models}} \ulcorner \phi \urcorner)$$

We will now prove the theorem by induction on the structure of ϕ. Most cases can be proven by simply unfolding of definitions of the Gödel encoding, applying the axioms of $\dot{\mathcal{T}}$, and applying the induction hypothesis.

case $\alpha \vee \beta$**.**

$$\begin{aligned}
\dot{\mathcal{T}} \vdash (e \mathrel{\dot{\models}} \ulcorner \alpha \vee \beta \urcorner) &\iff \dot{\mathcal{T}} \vdash (e \mathrel{\dot{\models}} (\ulcorner \alpha \urcorner \mathbin{\dot{\vee}} \ulcorner \beta \urcorner)) && \text{by } (\ \mathsf{Gdl}_{\vee}\ ) \\
&\iff \dot{\mathcal{T}} \vdash (e \mathrel{\dot{\models}} \ulcorner \alpha \urcorner) \vee (e \mathrel{\dot{\models}} \ulcorner \beta \urcorner) && \text{by } (\mathsf{Ax}_{\dot{\vee}}) \\
&\iff \dot{\mathcal{T}} \vdash \alpha \vee (e \mathrel{\dot{\models}} \ulcorner \beta \urcorner) && \text{by I.H.} \\
&\iff \dot{\mathcal{T}} \vdash \alpha \vee \beta && \text{by I.H.} \\
&\quad \square
\end{aligned}$$

case $\neg\psi$**.**

$$\begin{aligned}
\dot{\mathcal{T}} \vdash (e \mathrel{\dot{\models}} \ulcorner \neg\psi \urcorner) &\iff \dot{\mathcal{T}} \vdash (e \mathrel{\dot{\models}} \dot{\neg} \ulcorner \psi \urcorner) && \text{by } (\ \mathsf{Gdl}_{\neg}\ ) \\
&\iff \dot{\mathcal{T}} \vdash \neg(e \mathrel{\dot{\models}} \ulcorner \psi \urcorner) && \text{by } (\mathsf{Ax}_{\dot{\neg}}) \\
&\iff \dot{\mathcal{T}} \vdash \neg\psi && \text{by I.H.} \\
&\quad \square
\end{aligned}$$

case $\bot$**.**

$$\begin{aligned}
\dot{\mathcal{T}} \vdash (e \mathrel{\dot{\models}} \ulcorner \bot \urcorner) &\iff \dot{\mathcal{T}} \vdash (e \mathrel{\dot{\models}} \dot{\bot}) && \text{by } (\ \mathsf{Gdl}_{\bot}\ ) \\
&\iff \dot{\mathcal{T}} \vdash \bot && \text{by } (\mathsf{Ax}_{\dot{\bot}}) \\
&\quad \square
\end{aligned}$$

case $\forall \mathsf{x}_i : \sigma.\phi$**.**

$$\begin{aligned}
\dot{\mathcal{T}} \vdash (e \mathrel{\dot{\models}} \ulcorner \forall \mathsf{x}_i : \sigma.\phi \urcorner) &\iff \dot{\mathcal{T}} \vdash (e \mathrel{\dot{\models}} \dot{\forall} \mathsf{v}_i^{\sigma}{:}\sigma.\ulcorner \phi \urcorner) && \text{by } (\ \mathsf{Gdl}_{\forall}\ ) \\
&\iff \dot{\mathcal{T}} \vdash \forall \mathsf{x}_i.(\mathsf{push}_{\sigma}(e, \mathsf{v}_i^{\sigma}, \mathsf{x}_i) \mathrel{\dot{\models}} \ulcorner \phi \urcorner) && \text{by } (\mathsf{Ax}_{\dot{\forall}}) \\
&\iff \dot{\mathcal{T}} \vdash \forall \mathsf{x}_i.\phi && \text{by I.H.} \\
&\quad \square
\end{aligned}$$

The more involved cases are the ones dealing with atomic formulas. The reason why this cannot be dealt with by simple unfolding of definitions, is that we cannot ensure that every object level variable v_i^{σ} is interpreted as the variable x_i of sort σ, since this would have required adding an infinite number of axioms to our base theory $\mathcal{T}$.

case $P(t_1, ..., t_n)$**.** By the existence of a sound and complete proof system for first-order logic, we can rewrite our induction hypothesis as

$$\dot{\mathcal{T}} \models P(t_1, ..., t_n) \iff \dot{\mathcal{T}} \models (e \mathrel{\dot{\models}} \ulcorner P(t_1 ... t_n) \urcorner)$$

which is equivalent to the statement

$$\forall \mathcal{M} \models \dot{\mathcal{T}}.\big(\mathcal{M} \models P(t_1, ..., t_n)\big) \iff \forall \mathcal{M} \models \dot{\mathcal{T}}.\big(\mathcal{M} \models (e \mathrel{\dot{\models}} \ulcorner P(t_1 ... t_n) \urcorner)\big)$$

which can again be rewritten to

$$\exists\, \mathcal{M} \vDash \dot{\mathcal{T}}.\Big(\mathcal{M} \nvDash P(t_1, ..., t_n)\Big) \iff \exists\, \mathcal{M} \vDash \dot{\mathcal{T}}.\Big(\mathcal{M} \nvDash (e \,\dot{\vDash}\, \ulcorner P(t_1 ... t_n)\urcorner)\Big)$$

We will proof both directions of the biconditional separately:

case "$\Longleftarrow$" In order to show that this implication holds we will show, that if there is a model $\langle\mathcal{D},\mathcal{I}\rangle$ such that $\langle\mathcal{D},\mathcal{I}\rangle \nvDash (e \,\dot{\vDash}\, \ulcorner P(t_1,...,t_n)\urcorner)$, then there is another model $\langle\mathcal{D},\hat{\mathcal{I}}\rangle$ such that $\langle\mathcal{D},\hat{\mathcal{I}}\rangle \nvDash P(t_1,...,t_n)$.
The idea is that $\hat{\mathcal{I}}$ differs from $\mathcal{I}$ only in the interpretation of the variables. In exact the variables in $\hat{\mathcal{I}}$ are interpreted in the same way as reflective variables v_i^σ are interpreted in $\mathcal{I}$. Therefore the interpretation of the evaluation of a term $\ulcorner t\urcorner$ in $\hat{\mathcal{I}}$ will be the same as the interpretation of t in $\mathcal{I}$, hence $\langle\mathcal{D},\hat{\mathcal{I}}\rangle$ will satisfy $(e \,\dot{\vDash}\, \ulcorner\phi\urcorner)$ iff $\langle\mathcal{D},\mathcal{I}\rangle$ satisfies ϕ, which implies what we want to show.
More formally:
Let $\langle D, I\rangle$ be a model of $\dot{\mathcal{T}}$. We define $\hat{\mathcal{I}}$ as follows.

$$\hat{\mathcal{I}}(x) = \begin{cases} \mathcal{I}(\mathsf{eval}_\sigma^v(\mathsf{empty}, \mathsf{v}_i^\sigma)) & \text{if } x = \mathsf{x}_i \ \&\ \mathsf{x}_i \in \mathbf{Var}_\sigma \\ \mathcal{I}(x) & \text{otherwise} \end{cases}$$

Proposition 1.

$$\mathcal{I}(\mathsf{eval}_\sigma(e, \ulcorner t\urcorner)) = \hat{\mathcal{I}}(t)$$

Proof. We apply induction on t.

case $f(t_1,...,t_n)$**.**

$$\begin{aligned}
&\hat{\mathcal{I}}(f(t_1,...,t_n)) \\
&= \hat{\mathcal{I}}(f)(\hat{\mathcal{I}}(t_1),...,\hat{\mathcal{I}}(t_n)) \\
&= \mathcal{I}(f)(\hat{\mathcal{I}}(t_1),...,\hat{\mathcal{I}}(t_n)) && \text{by definition of } \hat{\mathcal{I}} \\
&= \mathcal{I}(f)(\mathcal{I}(\mathsf{eval}_\sigma(e,\ulcorner t_1\urcorner)),...,\mathcal{I}(\mathsf{eval}_\sigma(e,\ulcorner t_n\urcorner))) && \text{by I.H.} \\
&= \mathcal{I}(\mathsf{eval}_\sigma(e,\ulcorner f(t_1,...,t_n)\urcorner)) && \text{by } (\ \mathsf{Gdl}_f\ ) \text{ and } (\mathsf{Ax}_{\mathsf{eval}_f})
\end{aligned}$$

□

case $\mathsf{x}_i \in \mathbf{Var}_\sigma$**.**

$$\begin{aligned}
&\hat{\mathcal{I}}(\mathsf{x}_i) \\
&= \mathcal{I}(\mathsf{eval}_\sigma^v(e, \mathsf{v}_i^\sigma)) && \text{by definition of } \hat{\mathcal{I}} \\
&= \mathcal{I}(\mathsf{eval}_\sigma(e, \mathsf{inj}_\sigma(\mathsf{v}_i^\sigma))) && \text{by } (\mathsf{Ax}_{\mathsf{eval}_{var}}) \\
&= \mathcal{I}(\mathsf{eval}_\sigma(e, \ulcorner \mathsf{x}_i\urcorner)) && \text{by } (\ \mathsf{Gdl}_\mathsf{x}\ )
\end{aligned}$$

□

□

Now that we have showed that Proposition 1 holds we can reason as follows:

$$\begin{aligned}
&\hat{\mathcal{I}} \models P(t_1, \ldots, t_n) \\
\iff& \hat{\mathcal{I}}(P) \ni \langle \hat{\mathcal{I}}(t_1), \ldots, \hat{\mathcal{I}}(t_n) \rangle \\
\iff& \hat{\mathcal{I}}(P) \ni \langle \mathcal{I}(\mathsf{eval}_\sigma(e, \ulcorner t_1 \urcorner)), \ldots, \mathcal{I}(\mathsf{eval}_\sigma(e, \ulcorner t_n \urcorner)) \rangle && \text{by Proposition 1} \\
\iff& \mathcal{I}(P) \ni \langle \mathcal{I}(\mathsf{eval}_\sigma(e, \ulcorner t_1 \urcorner)), \ldots, \mathcal{I}(\mathsf{eval}_\sigma(e, \ulcorner t_n \urcorner)) \rangle && \text{by definition of } \hat{\mathcal{I}} \\
\iff& \mathcal{I} \models P(\mathsf{eval}_\sigma(e, \ulcorner t_1 \urcorner), \ldots, \mathsf{eval}_\sigma(e, \ulcorner t_n \urcorner)) \\
\iff& \mathcal{I} \models (e \mathrel{\dot{\models}} \dot{P}(\ulcorner t_1 \urcorner, \ldots, \ulcorner t_n \urcorner)) && \text{by } (\mathsf{Ax}_P) \\
\iff& \mathcal{I} \models (e \mathrel{\dot{\models}} \ulcorner P(t_1, \ldots, t_n) \urcorner) && \text{by } (\ \mathsf{GdI}_P\ )
\end{aligned}$$

From this we can conclude that if there is a model $\langle \mathcal{D}, \mathcal{I} \rangle$ such that $\langle \mathcal{D}, \mathcal{I} \rangle \not\models (e \mathrel{\dot{\models}} \ulcorner P(t_1, \ldots, t_n) \urcorner)$, then the model $\langle \mathcal{D}, \hat{\mathcal{I}} \rangle \not\models P(t_1, \ldots, t_n)$. □

case " $\Longrightarrow$ " The idea for this case similar to the idea for the case before: We assume there is a model $\langle \mathcal{D}, \mathcal{I} \rangle$ that makes $P(t_1, \ldots, t_n)$ false, and from that build another model $\langle \mathcal{D}, \hat{\mathcal{I}} \rangle$ that makes the $(e \mathrel{\dot{\models}} \ulcorner P(t_1, \ldots, t_n) \urcorner)$ false. In this case our new model will differ from the old one not in the interpretation of the variables x_i, but in the interpretation of the evaluation of the reflective variables v_i^σ, in such a way that the evaluation of v_i^σ in $\hat{\mathcal{I}}$ will always be interpreted as the same value as the interpretation of x_i in $\mathcal{I}$.
Let $\langle D, I \rangle$ be a model of $\dot{\mathcal{T}}$. We define the interpretation $\hat{\mathcal{I}}$ as follows.

$$\hat{\mathcal{I}}(x) = \mathcal{I}(x) \qquad \text{for } x \neq \mathsf{eval}_\sigma^v$$

$$\hat{\mathcal{I}}(\mathsf{eval}_\sigma^v)(e, v) = \begin{cases} \mathcal{I}(\mathsf{x}_i) & \text{if } e = \mathsf{empty} \;\&\; v = \mathsf{v}_i^\sigma \\ \hat{\mathcal{I}}(t) & \text{if } e = \mathsf{push}_\sigma(e', v, t) \\ \hat{\mathcal{I}}(\mathsf{eval}_\sigma^v)(e', v) & \text{if } e = \mathsf{push}_\sigma(e', v', t) \;\&\; v \neq v' \\ \hat{\mathcal{I}}(\mathsf{eval}_\sigma^v)(e', v) & \text{if } e = \mathsf{push}_\tau(e', v', t) \;\&\; \tau \neq \sigma \end{cases}$$

Note that the definition of $\hat{\mathcal{I}}(\mathsf{eval}_\sigma^v)$, is not a partial definition, since we defined $e \in$ **stack** inductively.

Proposition 2.

$$\mathcal{I}(t) = \hat{\mathcal{I}}(\mathsf{eval}_\sigma(e, \ulcorner t \urcorner)) \tag{4.5}$$

Proof. We apply induction on t:

case $f(t_1, ..., t_n)$**.**

$$\begin{aligned}
\mathcal{I}(f(t_1,...,t_n)) &= \mathcal{I}(f)(\mathcal{I}(t_1),...,\mathcal{I}(t_n)) \\
&= \hat{\mathcal{I}}(f)(\mathcal{I}(t_1),...,\mathcal{I}(t_n)) && \text{by definition of } \hat{\mathcal{I}} \\
&= \hat{\mathcal{I}}(f)(\mathcal{I}(\mathsf{eval}_\sigma(e,\ulcorner t_1\urcorner)),...,\mathcal{I}(\mathsf{eval}_\sigma(e,\ulcorner t_n\urcorner))) && \text{by I.H.} \\
&= \hat{\mathcal{I}}(f(\mathsf{eval}_\sigma(e,\ulcorner t_1\urcorner),...,\mathsf{eval}_\sigma(e,\ulcorner t_n\urcorner))) \\
&= \hat{\mathcal{I}}(\mathsf{eval}_\sigma(e,\dot{f}(\ulcorner t_1\urcorner,...,\ulcorner t_n\urcorner))) && \text{by } (\mathsf{Ax}_{\mathsf{eval}_f}) \\
&= \hat{\mathcal{I}}(\mathsf{eval}_\sigma(e,\ulcorner f(t_1,...,t_n)\urcorner)) && \text{by } (\ \mathsf{Gdl}_f\ )
\end{aligned}$$

$\square$

case $\mathsf{x}_i \in \mathbf{Var}_\sigma$**.** Induction on e:

case empty

$$\begin{aligned}
\mathcal{I}(\mathsf{x}_i) &= \hat{\mathcal{I}}(\mathsf{eval}^v_\sigma)(\mathsf{empty},\mathsf{v}^\sigma_i) && \text{by definition of } \hat{\mathcal{I}} \\
&= \hat{\mathcal{I}}(\mathsf{eval}^v_\sigma)(\mathsf{empty},\ulcorner \mathsf{x}_i\urcorner) && \text{by } (\ \mathsf{Gdl}_\mathsf{x}\ ) \\
&= \hat{\mathcal{I}}(\mathsf{eval}^v_\sigma(\mathsf{empty},\ulcorner \mathsf{x}_i\urcorner))
\end{aligned}$$

$\square$

case $\mathsf{push}_\tau(e', \mathsf{v}^\tau_j, \mathsf{x}_j)$

case $\tau \neq \sigma$

$$\begin{aligned}
&\hat{\mathcal{I}}(\mathsf{eval}^v_\sigma(\mathsf{push}_\tau(e',\mathsf{v}^\tau_j,\mathsf{x}_j),\ulcorner \mathsf{x}_i\urcorner)) \\
&= \hat{\mathcal{I}}(\mathsf{eval}^v_\sigma)(\mathsf{push}_\tau(e',\mathsf{v}^\tau_j,\mathsf{x}_j),\ulcorner \mathsf{x}_i\urcorner) \\
&= \hat{\mathcal{I}}(\mathsf{eval}^v_\sigma)(e',\ulcorner \mathsf{x}_i\urcorner) && \text{by definition of } \hat{\mathcal{I}} \\
&= \hat{\mathcal{I}}(\mathsf{eval}^v_\sigma(e',\ulcorner \mathsf{x}_i\urcorner)) \\
&= \mathcal{I}(\mathsf{x}_i) && \text{by I.H.}
\end{aligned}$$

$\square$

case $\tau = \sigma \ \&\ i \neq j$

$$\begin{aligned}
&\hat{\mathcal{I}}(\mathsf{eval}^v_\sigma(\mathsf{push}_\tau(e',\mathsf{v}^\tau_j,\mathsf{x}_j),\ulcorner \mathsf{x}_i\urcorner)) \\
&= \hat{\mathcal{I}}(\mathsf{eval}^v_\sigma(\mathsf{push}_\sigma(e',\mathsf{v}^\sigma_j,\mathsf{x}_j),\ulcorner \mathsf{x}_i\urcorner)) \\
&= \hat{\mathcal{I}}(\mathsf{eval}^v_\sigma)(\mathsf{push}_\sigma(e',\mathsf{v}^\sigma_j,\mathsf{x}_j),\ulcorner \mathsf{x}_i\urcorner) \\
&= \hat{\mathcal{I}}(\mathsf{eval}^v_\sigma)(e',\ulcorner \mathsf{x}_i\urcorner) && \text{by definition of } \hat{\mathcal{I}} \\
&= \hat{\mathcal{I}}(\mathsf{eval}^v_\sigma(e',\ulcorner \mathsf{x}_i\urcorner)) \\
&= \mathcal{I}(\mathsf{x}_i) && \text{by I.H.}
\end{aligned}$$

$\square$

case $\tau = \sigma \,\&\, i = j$

$$\begin{aligned}
&\hat{\mathcal{I}}(\mathsf{eval}^v_\sigma(\mathsf{push}_\tau(e', \mathsf{v}^\tau_j, \mathsf{x}_j), \ulcorner \mathsf{x}_i \urcorner)) \\
&= \hat{\mathcal{I}}(\mathsf{eval}^v_\sigma(\mathsf{push}_\sigma(e', \mathsf{v}^\sigma_i, \mathsf{x}_i), \ulcorner \mathsf{x}_i \urcorner)) \\
&= \hat{\mathcal{I}}(\mathsf{eval}^v_\sigma)(\mathsf{push}_\sigma(e', \mathsf{v}^\sigma_i, \mathsf{x}_i), \ulcorner \mathsf{x}_i \urcorner) \\
&= \hat{\mathcal{I}}(\mathsf{x}_i) && \text{by definition of } \hat{\mathcal{I}} \\
&= \mathcal{I}(\mathsf{x}_i) && \text{by definition of } \hat{\mathcal{I}}
\end{aligned}$$

$\square$

This concludes the end of the proof of Proposition 2. $\square$

Therefore we can reason analogous to before.

$$\begin{aligned}
&\mathcal{I} \not\models P(t_1, ..., t_n) \\
\iff &\mathcal{I}(P) \not\ni P(t_1, ..., t_n) \\
\iff &\mathcal{I}(P) \not\ni \langle \hat{\mathcal{I}}(\mathsf{eval}_\sigma(e, \ulcorner t_1 \urcorner)), ..., \hat{\mathcal{I}}(\mathsf{eval}_\sigma(e, \ulcorner t_n \urcorner)) \rangle && \text{by Proposition 2} \\
\iff &\hat{\mathcal{I}}(P) \not\ni \langle \hat{\mathcal{I}}(\mathsf{eval}_\sigma(e, \ulcorner t_1 \urcorner)), ..., \hat{\mathcal{I}}(\mathsf{eval}_\sigma(e, \ulcorner t_n \urcorner)) \rangle && \text{by definition of } \hat{\mathcal{I}} \\
\iff &\hat{\mathcal{I}} \not\models P(\mathsf{eval}_\sigma(e, \ulcorner t_1 \urcorner), ..., \mathsf{eval}_\sigma(e, \ulcorner t_n \urcorner)) \\
\iff &\hat{\mathcal{I}} \not\models (e \mathrel{\dot{\models}} \dot{P}(\ulcorner t_1 \urcorner, ..., \ulcorner t_n \urcorner)) && \text{by } (\mathsf{Ax}_P) \\
\iff &\hat{\mathcal{I}} \not\models (e \mathrel{\dot{\models}} \ulcorner P(t_1, ..., t_n) \urcorner) && \text{by } (\ \mathsf{GdI}_P\ )
\end{aligned}$$

Now that we have established this know that if there is a model $\langle \Delta, \mathcal{I} \rangle$ such that $\langle \Delta, \mathcal{I} \rangle \not\models P(t_1, ..., t_n)$, then there is a model $\langle \Delta, \hat{\mathcal{I}} \rangle$ such that $\hat{\mathcal{I}} \not\models (e \mathrel{\dot{\models}} \ulcorner P(t_1, ..., t_n) \urcorner)$. This concludes our proof of the case " $\implies$ " of the biconditional, and therefore as well the induction case for $\phi = P(t_1, ..., t_n)$ in the proof of our main Theorem 2. $\square$

case $s \approx t$**.** This case is exactly the same as for uninterpreted predicates, since equality can be thought of as a binary predicate.

We have therefore established that Theorem 2 indeed holds. $\square$

Let us summarize the contents of this rather formal section: Given an arbitrary finitely axiomatizable theory $\mathcal{T}$, we defined its reflective extension $\dot{\mathcal{T}}$, obtained by adding a finite number of sorts, and symbols, and axioms to the theory. We showed that every model $\mathcal{M}$ of $\mathcal{T}$ can be extended to a model $\dot{\mathcal{M}}$ of $\dot{\mathcal{T}}$, and which implies conservativeness and consistency of $\dot{\mathcal{T}}$ (given $\mathcal{T}$ itself is consistent). Further we showed that $\dot{\mathcal{T}}$ has a truth predicate for $\mathcal{T}$, namely $\lambda x.(\mathsf{empty} \mathrel{\dot{\models}} x)$ which will be vital for our applications of extending theories.

4.3 Finitely axiomatizing induction

Based on the observations of the last section we will now show how to build a finite theory that entails the first order induction scheme. The key idea here is the following: When defining **PA**, mathematicians do not write down the infinite number of induction axioms, as their amount of time and paper is finite. Instead they quantify over formulas on the meta-level. This is what we will do as well. Given a theory $\mathcal{T}$, one can think of the reflective extension $\dot{\mathcal{T}}$ as the meta-level language mathematicians use, which allows them to quantify over formulas.

4.3.1 Natural Numbers

In order to finitely axiomatise **PA**, we need a finite fragment of **PA** to start with. The obvious choice is **Q**, which we defined as **PA** without the induction formulas. In the next step we build the reflective extension $\dot{\mathbf{Q}}$. This theory has two essential properties. Firstly it has a sort of formulas form, hence we can quantify over this sort. Secondly we have a truth predicate for **Q** in $\dot{\mathbf{Q}}$, which means we can represent an arbitrary formula of **Q** in a single term in $\dot{\mathbf{Q}}$.

Now we can define **PA**′, a theory that is a conservative extension of **PA**. Therefore we will add one axiom to $\dot{\mathbf{Q}}$, called the reflective induction axiom:

$$\begin{aligned}\forall\phi : \mathsf{form}.\Big(&\mathbf{True}[\phi, 0] \wedge \\ &\forall n : \mathsf{nat}.(\mathbf{True}[\phi, n] \to \mathbf{True}[\phi, \mathsf{s}n]) \\ &\to \forall n : \mathsf{nat}.\mathbf{True}[\phi, n]\Big)\end{aligned} \tag{$\dot{\mathbf{I}}_{\mathsf{nat}}$}$$

where

$$\mathbf{True}[\phi, n] := (\mathsf{push}_{\mathsf{nat}}(\mathsf{empty}, \mathsf{v}_0^{\mathsf{nat}}, n) \mathrel{\dot{\vDash}} \phi)$$

$$\mathbf{PA}' = \dot{\mathbf{Q}} \cup \{\dot{\mathbf{I}}_\tau\}$$

We now need to establish the fact that **PA**′ is indeed a conservative extension of **PA**. Therefore we will need the following auxiliary lemma:

$$\dot{\mathcal{T}} \vDash (\mathsf{push}_\sigma(e, \ulcorner \mathsf{x}_i \urcorner, t) \mathrel{\dot{\vDash}} \ulcorner \phi[\mathsf{x}_i] \urcorner) \leftrightarrow (e \mathrel{\dot{\vDash}} \ulcorner \phi[t] \urcorner) \tag{4.6}$$

This lemma can be proven rather straight forward by induction on the formula ϕ, and since it will only involve a lot of technical details, and no real insights, we will skip proving it.

Proving that **PA**′is a conservative extension of **PA** means need to show that every formula in $\mathbf{Form}^{\mathbf{PA}}$ is provable in **PA**′ iff it is provable in **PA**. We will prove both directions of this biconditional separately:

$\forall \phi \in \mathbf{Form}^{\mathbf{PA}}.(\mathbf{PA} \vDash \phi \implies \mathbf{PA}' \vDash \phi)$ We will show this by showing that all axioms of **PA** are derivable in **PA′**. Since **Q** is a subset of both **PA**, and **PA′**we only need to deal with the induction axioms. This is rather straight forward. Let $\phi[0] \wedge \forall n.(\phi[n] \rightarrow \phi[n+1]) \rightarrow \forall n.\phi[n]$ be an arbitrary instance of the first order mathematical induction scheme. So let us instantiate the reflective induction axiom $(\dot{\mathbf{I}}_{\mathsf{nat}})$ with $\ulcorner\phi[\mathsf{x}_0]\urcorner$. Hence we get

$$\begin{aligned}\mathbf{PA}' \vdash &\mathbf{True}[\ulcorner\phi[\mathsf{x}_0]\urcorner, 0] \wedge \\ &\quad \forall n : \mathsf{nat}.(\mathbf{True}[\ulcorner\phi[\mathsf{x}_0]\urcorner, n] \rightarrow \mathbf{True}[\ulcorner\phi[\mathsf{x}_0]\urcorner, \mathsf{s}n]) \\ &\quad \rightarrow \forall n : \mathsf{nat}.\mathbf{True}[\ulcorner\phi[\mathsf{x}_0]\urcorner, n]\end{aligned}$$

which we can expand to

$$\begin{aligned}\mathbf{PA}' \vdash &(\mathsf{push}_{\mathsf{nat}}(\mathsf{empty}, \mathsf{v}_0^{\mathsf{nat}}, 0) \dot{\vDash} \ulcorner\phi[\mathsf{x}_0]\urcorner) \wedge \\ &\quad \forall n : \mathsf{nat}.((\mathsf{push}_{\mathsf{nat}}(\mathsf{empty}, \mathsf{v}_0^{\mathsf{nat}}, n) \dot{\vDash} \ulcorner\phi[\mathsf{x}_0]\urcorner) \rightarrow (\mathsf{push}_{\mathsf{nat}}(\mathsf{empty}, \mathsf{v}_0^{\mathsf{nat}}, \mathsf{s}n) \dot{\vDash} \ulcorner\phi[\mathsf{x}_0]\urcorner)) \\ &\rightarrow \forall n : \mathsf{nat}.(\mathsf{push}_{\mathsf{nat}}(\mathsf{empty}, \mathsf{v}_0^{\mathsf{nat}}, n) \dot{\vDash} \ulcorner\phi[\mathsf{x}_0]\urcorner)\end{aligned}$$

By lemma (4.6) we can derive

$$\begin{aligned}\mathbf{PA}' \vdash &(\mathsf{empty} \dot{\vDash} \ulcorner\phi[0]\urcorner) \wedge \\ &\quad \forall n : \mathsf{nat}.((\mathsf{empty} \dot{\vDash} \ulcorner\phi[n]\urcorner) \rightarrow (\mathsf{empty} \dot{\vDash} \ulcorner\phi[\mathsf{s}n]\urcorner)) \\ &\rightarrow \forall n : \mathsf{nat}.(\mathsf{empty} \dot{\vDash} \ulcorner\phi[n]\urcorner)\end{aligned}$$

Applying Theorem 2, the fact that $\lambda x.(\mathsf{empty} \dot{\vDash} x)$ is our truth predicate, we get

$$\mathbf{PA}' \vdash \phi[0] \wedge \forall n : \mathsf{nat}.(\phi[n] \rightarrow \phi[\mathsf{s}n]) \rightarrow \forall n : \mathsf{nat}.\phi[n]$$

Which concludes the first part of the proof. □

$\forall \phi.(\mathbf{PA}' \vDash \phi \implies \mathbf{PA} \vDash \phi)$ We will show this show this contrapositive. Suppose we have some formula ϕ such that $\mathbf{PA} \nvDash \phi$. Hence there is a counter-model $\mathcal{M}$, such that $\mathcal{M} \vDash \mathbf{PA}$ but $\mathcal{M} \nvDash \phi$. Since $\mathbf{Q} \subset \mathbf{PA}$, it holds that $\mathcal{M} \vDash \mathbf{Q}$. As we established in Section 4.2.4 we can extend the model $\mathcal{M}$ to the reflective model $\dot{\mathcal{M}}$ such that $\dot{\mathcal{M}} \vDash \dot{\mathbf{Q}}$, and that $\dot{\mathcal{M}} \nvDash \phi$. We now just need to establish that $\dot{\mathcal{M}}$ is a model of **PA′**, in order to show our goal.

In $\dot{\mathcal{M}}$ the sort form is interpreted as the actual set of formulas **Form**. Therefore let $\phi[\mathsf{x}_0]$ be an arbitrary of these formulas. Since $\mathcal{M} \vDash \mathbf{PA}$, we have that $\dot{\mathcal{M}} \vDash \mathbf{PA}$, which implies that

$$\dot{\mathcal{M}} \vDash \phi[0] \wedge \forall n : \mathsf{nat}.(\phi[n] \rightarrow \phi[\mathsf{s}n]) \rightarrow \forall n : \mathsf{nat}.\phi[n]$$

By Theorem 2 we can establish that

$$\begin{aligned}\dot{\mathcal{M}} \models &(\mathsf{empty} \dot{\models} \ulcorner \phi[0] \urcorner) \land \\ &\quad \forall n : \mathsf{nat}.((\mathsf{empty} \dot{\models} \ulcorner \phi[n] \urcorner) \to (\mathsf{empty} \dot{\models} \ulcorner \phi[\mathsf{s}n] \urcorner)) \\ &\to \forall n : \mathsf{nat}.(\mathsf{empty} \dot{\models} \ulcorner \phi[n] \urcorner)\end{aligned}$$

which can according to Lemma 4.6 be rewritten to

$$\begin{aligned}\dot{\mathcal{M}} \models &\mathbf{True}[\phi[x_0], 0] \land \\ &\quad \forall n : \mathsf{nat}.(\mathbf{True}[\phi[x_0], n] \to \mathbf{True}[\phi[x_0], \mathsf{s}n]) \\ &\to \forall n : \mathsf{nat}.\mathbf{True}[\phi[x_0], n]\end{aligned}$$

Since $\dot{\mathcal{M}}$ interprets form formulas exactly as the set **Form**, and $\phi[x_0]$ was an arbitrary formula, this means that reflective induction axiom $\dot{\mathbf{I}}_\tau$ holds for $\dot{\mathcal{M}}$. Therefore $\dot{\mathcal{M}}$ models $\mathbf{PA}'$ but not ϕ, which concludes the second part of the proof. □

4.3.2 Arbitrary datatypes

The technique from the previous section can be lifted to arbitrary datatypes. Therefore, again we translate the meta-level definition of the induction schemes $\mathbf{I}_\tau$ for all our datatypes $\mathcal{D}_\tau$ to an equivalent reflective version. This means for a theory $\mathcal{T}$, we build $\mathcal{T}'$, by adding the axiom $\dot{\mathbf{I}}_\tau$ to $\dot{\mathcal{T}}$ for every datatype $\mathcal{D}_\tau$ in the theory.

$$\forall \phi : \mathsf{form}.\Big(\bigwedge_{c \in \mathbf{ctors}} case_{\phi,c} \to \forall x : \tau.\mathbf{True}[\phi, x]\Big) \qquad (\dot{\mathbf{I}}_\tau)$$

where

$$\begin{aligned} case_{\phi,c} &:= \mathop{\dot{\forall\!\!\!\forall}}_{x_1,\dots,x_n} \Big(\bigwedge_{i \in recursive_c} \mathbf{True}[\phi, \mathsf{x}_i] \to \mathbf{True}[\phi, c(x_1, \dots, x_n)]\Big) \\ recursive_c &:= \{ i \mid \mathbf{dom}_\Sigma(c, i) = \tau \} \\ \mathbf{True}[\phi, n] &:= (\mathsf{push}_\tau(\mathsf{empty}, \mathsf{v}_0^\tau, n) \dot{\models} \phi) \end{aligned}$$

The proof of the equivalence of **PA** and **PA**′can be generalized straight forward for this kind of theories. Therefore we will omit formalizing it.

In the case of extending **Q** to a conservative extension of **PA**, the axioms of constructor disjointness (Disj$_\tau$), and injectivity (Inj$_\tau$) were already present in **Q**. Thus for an arbitrary inductive theory $\mathcal{T}$ with inductive datatypes $\mathcal{D}_\mathcal{T}$ we define the reflective inductive extension $\ddot{\mathcal{T}}$ as follows:

$$\ddot{\mathcal{T}} = \mathcal{T} \cup \{(\dot{\mathbf{I}}_\tau), (\mathrm{Disj}_\tau), (\mathrm{Inj}_\tau) \mid \mathcal{D}_\tau \in \mathcal{D}_\mathcal{T}\}$$

4.4 Other applications

As we saw our reflective extension can be nicely used to formalize the induction schemata within pure first order logic. In the same way this approach can be taken further, in order to finitely axiomatise other theories using axiom schemes.

One example for such a theory is Zermelo-Fraenkel set theory (**ZFC**). It contains the axiom schema of replacement and the restricted comprehension schema, which can both be formalised in the same way as in **PA**. Another theoretical use case for reflective reasoning is finitely formalising modal logics, in which the class of Kripke frames for which the logic is defined, is defined as axiom scheme. Since the languages of modal logics are different from first-order languages, a translation to first-order logic, as presented in [vB06], must be applied before formalising the axiom schemata in a reflective extension of the translated theory.

Another interesting application we will briefly sketch out here, is using reflective reasoning in order to finitely axiomatise the Hoare caluculus in first order logic. The base first order language we need for that will need the following sorts:

- prog A sort of programs, since we want to reason about program correctness.
- expr A sort of integer expressions, since our programming language will only deal with integers.
- form A sort of formulas, over integers we need in order to express pre and post conditions for programs.

Therefore we first use a base theory of integers $\mathcal{I}$, and build it's reflective extension $\dot{\mathcal{I}}$. To the signature of this theory we add functions that can be thought of as constructors of programs:

$$\begin{aligned}
\mathsf{while} &:: \mathsf{expr} \times \mathsf{prog} \rightsquigarrow \mathsf{prog}\\
\mathsf{if} &:: \mathsf{expr} \times \mathsf{prog} \times \mathsf{prog} \rightsquigarrow \mathsf{prog}\\
\mathsf{seq} &:: \mathsf{prog} \times \mathsf{prog} \rightsquigarrow \mathsf{prog}\\
\mathsf{asign} &:: \mathsf{var}_{\mathsf{int}} \times \mathsf{expr} \rightsquigarrow \mathsf{prog}
\end{aligned}$$

Further we add a predicate that represents the Hoare triplet relation to our signature:

$$\mathsf{Hoare} :: \mathsf{Pred}(\mathsf{form} \times \mathsf{prog} \times \mathsf{form})$$

In order to axiomatise all rules of the Hoare calculus, we need another function symbol $\mathsf{subs}_{\mathsf{int}} :: \mathsf{form} \times \mathsf{var}_{\mathsf{int}} \times \mathsf{term}_{\mathsf{int}} \rightsquigarrow \mathsf{form}$. This function can, if the reflective signature is extended by some more helper functions, easily be axiomatised in order to behave like the substitution of variables by terms within a formula. However, since this function is not needed for implementing induction via reflection, we will omit the axiomatisation of this function.

Finally we can add the universal closure of the following formulas as axioms to our theory in order to define the Hoare calculus for partial correctness:

$$\mathsf{Hoare}(\phi \mathbin{\dot\wedge} b \not\approx \dot{0}, \pi_1, \psi) \wedge \mathsf{Hoare}(\phi \mathbin{\dot\wedge} b \approx \dot{0}, \pi_2, \psi) \to \mathsf{Hoare}(\phi, \mathsf{if}(b, \pi_1, \pi_2), \psi)$$

$$\mathsf{Hoare}(\phi, \pi_1, \xi) \wedge \mathsf{Hoare}(\xi, \pi_2, \psi) \to \mathsf{Hoare}(\phi, \mathsf{seq}(\pi_1, \pi_2), \psi)$$

$$\mathsf{Hoare}(\mathsf{subs}_{\mathsf{int}}(\phi, v, e), \pi, \psi)$$

$$(\mathsf{empty} \mathrel{\dot\vDash} \phi \mathbin{\dot\to} \phi') \wedge \mathsf{Hoare}(\phi', \pi, \psi') \wedge (\mathsf{empty} \mathrel{\dot\vDash} \psi' \mathbin{\dot\to} \psi) \to \mathsf{Hoare}(\phi, \pi, \psi)$$

$$\mathsf{Hoare}(I \mathbin{\dot\wedge} b \not\approx \dot{0}, \pi, I) \to \mathsf{Hoare}(I, \mathsf{while}(b, \pi), I)$$

CHAPTER 5

Experimental evaluation

Now that we have fixed our technique that allows us to approach induction in an arbitrary theory $\mathcal{T}$ via translation to a finitely axiomatised conservative extension $\ddot{\mathcal{T}}$, we will see how this approach compares to state-of-the-art solutions to inductive theorem proving. Since our way of axiomatising induction heavily relies on reasoning in the reflective extension $\dot{\mathcal{T}}$ of a theory, we will first assess how well different solvers perform in reasoning in the reflective setup, without the reflective induction axiom being present.

5.1 Problems

As mentioned previously we will consider two classes of problems, which we will call **Refl** and **Ind**. **Refl** does contain simple problems that require reasoning in the reflective extension of some base theory. **Ind** is a set of problems that requires reasoning about inductive datatypes, without any need for reflective reasoning per se. Since many of the benchmarks have the same axioms, but different conclusions a list of all the base theories is given in table 5.1.

5.1.1 Reflective

This problem set aims at testing how well different state-of-the-art theorem provers perform reasoning in the reflective extension of theories. The problem set is split into two parts: $\mathbf{Refl_0}$, and $\mathbf{Refl_1}$.

The group $\mathbf{Refl_0}$ is the simplest one. For every theory $\mathcal{T} \in \{\mathbf{N}+\mathbf{Leq}+\mathbf{Add}+\mathbf{Mul}, \mathbf{N}+\mathbf{L}+\mathbf{Pref}+\mathbf{App}\}$, and any axiom $\alpha \in \mathcal{T}$ we try to proof the validity of $\dot{\mathcal{T}} \vdash (\mathsf{empty} \mathrel{\dot{\models}} \ulcorner\alpha\urcorner)$. Since we established that $\lambda x.(\mathsf{empty} \mathrel{\dot{\models}} x)$ is the truth predicate of $\mathcal{T}$, and the fact that α is an axiom, we know that these consequence assertions indeed hold.

$\mathbf{Refl_1}$, the second part of the benchmark set **Refl**, involves reasoning in the reflective extension $\dot{\mathcal{T}}$ of some theory as well. But in this case not the reflective version of the

Name	Theory
N	$\mathsf{data}\ \ \mathsf{nat} = \mathsf{zero} \mid \mathsf{s}(\mathsf{nat})$
Leq	$\leq :: \mathsf{Pred}(\mathsf{nat} \times \mathsf{nat})$ $\forall x.(x \leq x)$ (1) $\forall x, y.((x \leq y) \rightarrow (x \leq \mathsf{s}(y)))$ (2)
Add	$+ :: \mathsf{nat} \times \mathsf{nat} \rightsquigarrow \mathsf{nat}$ $\forall y.(\mathsf{zero} + y) \approx y$ (1) $\forall x, y.(\mathsf{s}(x) + y) \approx \mathsf{s}((x + y))$ (2)
Mul	$* :: \mathsf{nat} \times \mathsf{nat} \rightsquigarrow \mathsf{nat}$ $\forall y.(\mathsf{zero} * y) \approx \mathsf{zero}$ (1) $\forall x, y.(\mathsf{s}(x) * y) \approx (y + (x * y))$ (2)
L	$\mathsf{data}\ \ \mathsf{lst} = \mathsf{nil} \mid \mathsf{cons}(\mathsf{nat}, \mathsf{lst})$
Pref	$\mathsf{pref} :: \mathsf{Pred}(\mathsf{lst} \times \mathsf{lst})$ $\forall x.\mathsf{pref}(\mathsf{nil}, x)$ (1) $\forall a, x.\neg\mathsf{pref}(\mathsf{cons}(a, x), \mathsf{nil})$ (2) $\forall a, b, x, y.(\mathsf{pref}(\mathsf{cons}(a, x), \mathsf{cons}(b, y)) \leftrightarrow (a \approx b \wedge \mathsf{pref}(x, y)))$ (3)
App	$+\!\!+ :: \mathsf{lst} \times \mathsf{lst} \rightsquigarrow \mathsf{lst}$ $\forall r.(\mathsf{nil} +\!\!+ r) \approx r$ (1) $\forall a, l, r.(\mathsf{cons}(a, l) +\!\!+ r) \approx \mathsf{cons}(a, (l +\!\!+ r))$ (2)
E	$\mathsf{a} :: \alpha$ $\mathsf{b} :: \alpha$ $\mathsf{c} :: \alpha$ $\mathsf{p} :: \mathsf{Pred}(\alpha)$ $\mathsf{q} :: \mathsf{Pred}(\alpha)$ $\mathsf{r} :: \mathsf{Pred}(\alpha)$
Id	$\mathsf{id} :: \mathsf{nat} \rightsquigarrow \mathsf{nat}$ $\forall x.\mathsf{id}(x) \approx x$ (1)
Eq	$\mathsf{equal} :: \mathsf{Pred}(\mathsf{nat} \times \mathsf{nat} \times \mathsf{nat})$ $\mathsf{equal}(\mathsf{zero}, \mathsf{zero}, \mathsf{zero}) \leftrightarrow \top$ (1) $\forall y, z.(\mathsf{equal}(\mathsf{zero}, \mathsf{s}(y), z) \leftrightarrow \bot)$ (2) $\forall y, z.(\mathsf{equal}(\mathsf{zero}, y, \mathsf{s}(z)) \leftrightarrow \bot)$ (3) $\forall x, z.(\mathsf{equal}(\mathsf{s}(x), \mathsf{zero}, z) \leftrightarrow \bot)$ (4) $\forall x, y.(\mathsf{equal}(\mathsf{s}(x), y, \mathsf{zero}) \leftrightarrow \bot)$ (5) $\forall x, y, z.(\mathsf{equal}(\mathsf{s}(x), \mathsf{s}(y), \mathsf{s}(z)) \leftrightarrow \mathsf{equal}(x, y, z))$ (6)
Rev	$\mathsf{rev} :: \mathsf{lst} \rightsquigarrow \mathsf{lst}$ $\mathsf{rev}(\mathsf{nil}) \approx \mathsf{nil}$ (1) $\forall x, xs.\mathsf{rev}(\mathsf{cons}(x, xs)) \approx (\mathsf{rev}(xs) +\!\!+ \mathsf{cons}(x, \mathsf{nil}))$ (2)
Rev′	$\mathsf{rev}' :: \mathsf{lst} \rightsquigarrow \mathsf{lst}$ $\mathsf{revAcc} :: \mathsf{lst} \times \mathsf{lst} \rightsquigarrow \mathsf{lst}$ $\forall x.\mathsf{rev}'(x) \approx \mathsf{revAcc}(x, \mathsf{nil})$ (1) $\forall acc.\mathsf{revAcc}(\mathsf{nil}, acc) \approx acc$ (2) $\forall acc, x, xs.\mathsf{revAcc}(\mathsf{cons}(x, xs), acc) \approx \mathsf{revAcc}(xs, \mathsf{cons}(x, acc))$ (3)

Table 5.1: Theories used for the experiments.

Theory	Conjecture	id
E	$\ulcorner \forall x : \alpha . x \approx x \urcorner$	`eqRefl`
E	$\ulcorner \forall x, y, z : \alpha.((x \approx y \land y \approx z) \to x \approx z) \urcorner$	`eqTrans`
E	$\ulcorner \mathsf{p}(\mathsf{a}) \lor \neg \mathsf{p}(\mathsf{a}) \urcorner$	`excludedMiddle-0`
E	$\ulcorner \forall x.(\mathsf{p}(x) \lor \neg \mathsf{p}(x)) \urcorner$	`excludedMiddle-1`
E	$\ulcorner \forall x.\mathsf{p}(x) \to \mathsf{p}(\mathsf{a}) \urcorner$	`universalInstance`
E	$\ulcorner (\mathsf{p}(\mathsf{a}) \to \mathsf{q}(\mathsf{b})) \leftrightarrow (\neg \mathsf{q}(\mathsf{b}) \to \neg \mathsf{p}(\mathsf{a})) \urcorner$	`contraposition-0`
E	$\ulcorner \forall x, y.((\mathsf{p}(x) \to \mathsf{q}(y)) \leftrightarrow (\neg \mathsf{q}(y) \to \neg \mathsf{p}(x))) \urcorner$	`contraposition-1`
E	$\ulcorner ((\mathsf{p}(\mathsf{a}) \land \mathsf{q}(\mathsf{b})) \to \mathsf{r}(\mathsf{c})) \leftrightarrow (\mathsf{p}(\mathsf{a}) \to (\mathsf{q}(\mathsf{b}) \to \mathsf{r}(\mathsf{c}))) \urcorner$	`currying-0`
E	$\ulcorner \forall x, y, z.(((\mathsf{p}(x) \land \mathsf{q}(y)) \to \mathsf{r}(z)) \leftrightarrow (\mathsf{p}(x) \to (\mathsf{q}(y) \to \mathsf{r}(z)))) \urcorner$	`currying-1`
$\mathbf{N} + \mathbf{Add}$	$\ulcorner (1 + 2) \approx 3 \urcorner$	`addGround-0`
$\mathbf{N} + \mathbf{Add}$	$\ulcorner (8 + 5) \approx 13 \urcorner$	`addGround-1`
$\mathbf{N} + \mathbf{Add}$	$\ulcorner \exists x.(8 + x) \approx 13 \urcorner$	`addExists`
$\mathbf{N} + \mathbf{Add}$	$\ulcorner \exists z.\forall x.(z + x) \approx x \urcorner$	`existsZeroAdd`
$\mathbf{N} + \mathbf{Add} + \mathbf{Mul}$	$\ulcorner (3 * 4) \approx 12 \urcorner$	`mulGround`
$\mathbf{N} + \mathbf{Add} + \mathbf{Mul}$	$\ulcorner \exists x.(3 * x) \approx 12 \urcorner$	`mulExists`
$\mathbf{N} + \mathbf{Add} + \mathbf{Mul}$	$\ulcorner \exists z.\forall x.(z * x) \approx z \urcorner$	`existsZeroMul`
$\mathbf{N} + \mathbf{L} + \mathbf{App}$	$\ulcorner (\mathsf{nil} +\!\!+ \mathsf{cons}(7, \mathsf{nil})) \approx \mathsf{cons}(7, \mathsf{nil}) \urcorner$	`appendGround-0`
$\mathbf{N} + \mathbf{L} + \mathbf{App}$	$\ulcorner (\mathsf{cons}(3, \mathsf{nil}) +\!\!+ \mathsf{cons}(7, \mathsf{nil})) \approx \mathsf{cons}(3, \mathsf{cons}(7, \mathsf{nil})) \urcorner$	`appendGround-1`
$\mathbf{N} + \mathbf{L} + \mathbf{App}$	$\ulcorner \exists x.(\mathsf{cons}(3, \mathsf{nil}) +\!\!+ x) \approx \mathsf{cons}(3, \mathsf{cons}(7, \mathsf{nil})) \urcorner$	`appendExists`
$\mathbf{N} + \mathbf{L} + \mathbf{App}$	$\ulcorner \exists n.(n +\!\!+ \mathsf{cons}(7, \mathsf{nil})) \approx \mathsf{cons}(7, \mathsf{nil}) \urcorner$	`existsNil`

Table 5.2: Conjectures and theories used for the benchmark set $\mathbf{Refl_1}$. The number symbols used are abbreviations for the corresponding numerals.

axioms, but the reflective versions of some simple consequence of $\mathcal{T}$ are to be proven. Table 5.2, lists all conjectures and the related theories, that are to be proven in this set of benchmarks.

5.1.2 Inductive

Ind contains a set of simple properties, that require reasoning about inductive datatypes. Every problem $\mathcal{T} \vDash \phi$ in this set of benchmarks will be approached in two way. Firstly proving it directly for the solvers that support induction natively, and secondly translating the problem to $\ddot{\mathcal{T}} \vDash \phi$. Table 5.3 lists the set of conjectures used for this experiment.

5.2 Solvers

For the experimental evaluation two (non-disjoint) sets of solvers were considered. Firstly solvers that support induction natively, and secondly various general-purpose theorem provers that are able to deal with many-sorted quantified first-order formulas. The solvers considered where the SMT-solvers CVC4 and Z3, the superposition-based first-order theorem prover VAMPIRE, the higher-order theorem prover ZIPPERPOSITION that uses a combination of superposition and term rewriting, and the inductive theorem prover ZENO, that is designed to proof inductive properties of a HASKELL-like programming language.

Theory	Conjecture	id
$\mathbf{N} + \mathbf{Add}$	$\forall x, y.(x + y) \approx (y + x)$	`addCommut`
$\mathbf{N} + \mathbf{Add} + \mathbf{Mul}$	$\forall x, y.(x * y) \approx (y * x)$	`mulCommut`
$\mathbf{N} + \mathbf{Add}$	$\forall x, y, z.(x + (y + z)) \approx ((x + y) + z)$	`addAssoc`
$\mathbf{N} + \mathbf{Add} + \mathbf{Mul}$	$\forall x, y, z.(x * (y * z)) \approx ((x * y) * z)$	`mulAssoc`
$\mathbf{N} + \mathbf{Add}$	$\forall x.(x + \mathsf{zero}) \approx x$	`addNeutral`
$\mathbf{N} + \mathbf{Add} + \mathbf{Mul}$	$\forall x.(x * 1) \approx x$	`addNeutral-0`
$\mathbf{N} + \mathbf{Add} + \mathbf{Mul}$	$\forall x.(1 * x) \approx x$	`addNeutral-1`
$\mathbf{N} + \mathbf{Add} + \mathbf{Mul}$	$\forall x.(x * \mathsf{zero}) \approx \mathsf{zero}$	`mulZero`
$\mathbf{N} + \mathbf{Add} + \mathbf{Mul}$	$\forall x, y, z.(x * (y + z)) \approx ((x * y) + (x * z))$	`distr-0`
$\mathbf{N} + \mathbf{Add} + \mathbf{Mul}$	$\forall x, y, z.((y + z) * x) \approx ((y * x) + (z * x))$	`distr-1`
$\mathbf{N} + \mathbf{Leq}$	$\forall x, y, z.(((x \leq y) \wedge (y \leq z)) \rightarrow (x \leq z))$	`leqTrans`
$\mathbf{N} + \mathbf{Leq}$	$\forall x.(\mathsf{zero} \leq x)$	`zeroMin`
$\mathbf{N} + \mathbf{Leq} + \mathbf{Add}$	$\forall x, y.(x \leq (x + y))$	`addMonoton-0`
$\mathbf{N} + \mathbf{Leq} + \mathbf{Add}$	$\forall x.(x \leq (x + x))$	`addMonoton-1`
$\mathbf{N} + \mathbf{Add} + \mathbf{Id}$	$\forall x, y.(\mathsf{id}(x) + y) \approx (y + x)$	`addCommutId`
$\mathbf{N} + \mathbf{L} + \mathbf{App}$	$\forall x, y, z.(x \mathbin{+\!\!+} (y \mathbin{+\!\!+} z)) \approx ((x \mathbin{+\!\!+} y) \mathbin{+\!\!+} z)$	`appendAssoc`
$\mathbf{N} + \mathbf{L} + \mathbf{Pref} + \mathbf{App}$	$\forall x, y.\mathsf{pref}(x, (x \mathbin{+\!\!+} y))$	`appendMonoton`
$\mathbf{N} + \mathbf{Eq}$	$\forall x.\mathsf{equal}(x, x, x)$	`allEqRefl`
$\mathbf{N} + \mathbf{Eq}$	$\forall x, y, z.(\mathsf{equal}(x, y, z) \leftrightarrow (x \approx y \wedge y \approx z))$	`allEqDefsEquality`
$\mathbf{N} + \mathbf{L} + \mathbf{App} + \mathbf{Rev}$	$\forall x.\mathsf{rev}(\mathsf{rev}(x)) \approx x$	`revSelfInvers`
$\mathbf{N} + \mathbf{L} + \mathbf{App} + \mathbf{Rev}$	$\forall x.(x \mathbin{+\!\!+} (\mathsf{rev}(x) \mathbin{+\!\!+} x)) \approx ((x \mathbin{+\!\!+} \mathsf{rev}(x)) \mathbin{+\!\!+} x)$	`revAppend-0`
$\mathbf{N} + \mathbf{L} + \mathbf{App} + \mathbf{Rev}$	$\forall x.\mathsf{rev}((x \mathbin{+\!\!+} (x \mathbin{+\!\!+} x))) \approx \mathsf{rev}(((x \mathbin{+\!\!+} x) \mathbin{+\!\!+} x))$	`revAppend-1`
$\mathbf{N} + \mathbf{L} + \mathbf{App} + \mathbf{Rev} + \mathbf{Rev}'$	$\forall x.\mathsf{rev}(x) \approx \mathsf{rev}'(x)$	`revsEqual`

Table 5.3: Conjectures and theories used for the benchmark set **Ind**

Since Vampire in many cases uses incomplete heuristics, per default it was run with a complete strategy forced as well. This configuration if referred to as VampireComplete. Zipperposition supports replacing equalities by dedicated rewrite rules, which comes at the cost of the theoretical loss of some provable problems, but yields a significant gain of performance in practice. Zipperposition with these rewrite rules enabled will be referred to as ZipRewrite. As mentioned in Section 3.0.4, Cvc4 allows for theory exploration. Cvc4 with this heuristic enabled is referred to as Cvc4Gen. Table 5.4 gives an overview of all solvers used, and lists their input format and the command line options enabled for running them.

All solvers were run with a timeout of 10 seconds per problem.

5.3 Implementation

As there is no well-established standardized input format for inductive theorem provers, the translations to the input formats listed in Table 5.4, needed to be applied. Therefore the problems are written in an domain-specific language, that is syntactically very close to the mathematical notation of first-order formulas. This language is parsed by a Haskell9 program, which performs the translation $\lambda x.\ulcorner x \urcorner$ (translating to the reflective version of a formula) , $\lambda x.\dot{x}$ (translating to the reflective extension of a theory), and $\lambda x.\ddot{x}$ (translating to reflective inductive extension of a theory), as well as serialization

Solver	Induction	Input format	Commandline Options
Cvc4	✓	Smtlib2	`-quant-ind`
Cvc4Gen	✓	Smtlib2	`-conjecture-gen -quant-ind`
Z3	–	Smtlib2	default mode
Vampire	✓	Smtlib2	`-schedule casc -induction struct`
VampireComplete	✓	Smtlib2	`-induction struct -s 1`
ZipperPosition	✓	Zf	default mode
ZipRewrite	✓	Zf	default mode
Zeno	✓	ZenoHaskell	default mode

Table 5.4: An overview over solvers used for the experiments.

benchmark	Cvc4	Cvc4Gen	Z3	Vampire	VampireComplete	ZipperPosition	ZipRewrite
`N+Leq+Add+Mul-ax0`	✓	✓	✓	✓	✓	✓	✓
`N+Leq+Add+Mul-ax1`	✓	✓	✓	–	–	–	✓
`N+Leq+Add+Mul-ax2`	✓	✓	✓	✓	✓	–	✓
`N+Leq+Add+Mul-ax3`	✓	✓	✓	–	–	–	✓
`N+Leq+Add+Mul-ax4`	✓	✓	✓	✓	✓	–	✓
`N+Leq+Add+Mul-ax5`	✓	✓	✓	–	–	–	✓
`N+L+Pref+App-ax0`	✓	✓	✓	✓	✓	✓	✓
`N+L+Pref+App-ax1`	✓	✓	✓	✓	✓	–	✓
`N+L+Pref+App-ax2`	✓	✓	✓	–	–	–	–
`N+L+Pref+App-ax3`	✓	✓	✓	✓	✓	–	✓
`N+L+Pref+App-ax4`	✓	✓	✓	–	–	–	✓

Table 5.5: Results of the benchmark set **Refl$_0$**. Each benchmark's id has the format `<thry>-ax<n>` where `n` is the index of the axiom of the theory `thry` of which the reflective version should be proven.

of this domain-specific language to different output formats. The source code of this program (including a command line interface to the previously mentioned translations, and serializations) is publicly available on GitHub[1].

5.4 Results

In Table 5.5 we can see the results of solvers proving reflective versions of conjectures. What is striking is that the SMT solvers Cvc4, and Z3, can solver all benchmarks of this category, while the problem seems to be harder for the saturation based theorem

[1] `https://github.com/joe-hauns/msc-automating-induction-via-reflection`

benchmark	Cvc4	Cvc4Gen	Z3	Vampire	VampireComplete	Zipperposition	ZipRewrite
`eqRefl`	✓	✓	✓	✓	✓	✓	✓
`eqTrans`	✓	✓	✓	–	–	–	✓
`excludedMiddle-0`	✓	✓	✓	✓	✓	✓	✓
`excludedMiddle-1`	✓	✓	✓	✓	✓	✓	✓
`universalInstance`	–	–	✓	✓	✓	✓	✓
`contraposition-0`	✓	✓	✓	✓	✓	✓	✓
`contraposition-1`	✓	✓	✓	–	–	–	✓
`currying-0`	✓	✓	✓	✓	✓	–	✓
`currying-1`	✓	✓	✓	–	–	–	✓
`addGround-0`	✓	✓	✓	✓	✓	✓	✓
`addGround-1`	✓	✓	–	–	–	✓	✓
`addExists`	–	–	–	–	–	–	✓
`existsZeroAdd`	–	–	–	–	–	–	–
`mulGround`	✓	✓	✓	–	–	–	✓
`mulExists`	–	–	–	–	–	–	✓
`existsZeroMul`	–	–	–	–	–	–	–
`appendGround-0`	✓	✓	✓	✓	✓	✓	✓
`appendGround-1`	✓	✓	✓	–	–	✓	✓
`appendExists`	–	–	–	–	–	–	✓
`existsNil`	–	–	✓	✓	✓	–	✓

Table 5.6: Lists the results of solvers on the benchmark set $\mathbf{Refl_1}$. The benchmark id is the same as the id of the conjecture of which the reflective version should be proven, listed in Table 5.2.

provers. Further ZipRewrite does pretty well in this class of benchmarks as well. A potential reason for this difference in performance between the ordinary saturation approach and ZipRewrite might have to do with the following: For ZipRewrite equalities for function definitions of the reflective extensions are translated to rewrite rules that are oriented in way that they would intuitively be oriented by a human, this means that for example the axiom $(\mathsf{Ax}_{\mathsf{eval}_f})$ can be evaluated as one would intuitively do. In contrast Vampire, using superposition with the simplification ordering Knuth-Bendix Ordering (KBO) will orient this equality in the wrong way, which means that it won't be able to evaluate it in the intuitive way, which might be the reason for the difference in performance.

In Table 5.6 we see that the performance of the SMT-solvers drops as soon as more

benchmark	Cvc4	Cvc4Gen	Vampire	VampireComplete	ZipperPosition	ZipRewrite	Zeno	Cv̈c4	Cvc4̈Gen	Z̈3	Vam̈pire	VampireC̈omplete	ZipperP̈osition	ZipR̈ewrite
addCommut	✓	✓	✓	✓	✓	✓	✓	–	–	–	–	–	–	–
mulCommut	–	–	–	–	–	–	–	–	–	–	–	–	–	–
addAssoc	✓	✓	✓	✓	✓	✓	✓	–	–	–	–	–	–	–
mulAssoc	–	–	–	–	–	–	–	–	–	–	–	–	–	–
addNeutral	✓	✓	✓	✓	✓	✓	✓	–	–	–	–	–	–	–
addNeutral-0	✓	✓	✓	✓	✓	✓	✓	–	–	–	–	–	–	–
addNeutral-1	✓	✓	✓	✓	✓	✓	✓	–	–	–	–	–	–	–
mulZero	✓	✓	✓	✓	✓	✓	✓	–	–	–	–	–	–	✓
distr-0	–	–	–	–	–	–	–	–	–	–	–	–	–	–
distr-1	–	–	–	–	–	✓	–	–	–	–	–	–	–	–
leqTrans	–	–	–	–	–	–	▨	–	–	–	–	–	–	–
zeroMin	✓	✓	✓	✓	✓	✓	▨	–	–	✓	✓	–	–	✓
addMonoton-0	–	–	–	–	–	–	▨	–	–	–	–	–	–	–
addMonoton-1	–	–	–	–	–	–	▨	–	–	–	–	–	–	–
addCommutId	–	✓	✓	✓	✓	✓	✓	–	–	–	–	–	–	–
appendAssoc	✓	✓	✓	✓	✓	✓	✓	–	–	–	–	–	–	–
appendMonoton	✓	✓	✓	✓	✓	✓	✓	–	–	–	–	–	–	–
allEqRefl	✓	✓	✓	✓	✓	✓	–	–	–	✓	–	–	–	–
allEqDefsEquality	✓	✓	–	–	✓	✓	–	–	–	–	–	–	–	–
revSelfInvers	–	–	–	–	✓	–	–	–	–	–	–	–	–	–
revAppend-0	–	–	–	–	–	✓	–	–	–	–	–	–	–	–
revAppend-1	–	–	–	–	–	✓	–	–	–	–	–	–	–	–
revsEqual	–	–	–	–	–	–	–	–	–	–	–	–	–	–

Table 5.7: Lists the results of running solvers on the benchmark set **Ind**. The benchmark ids are the ones of the conjectures of Table 5.3. For every solver Slvr that supports full first-order logic with equality as input, there is a solver SL̈vr using the reflective inductive theory as an input instead of using the solvers native handling of induction. The greyed out cells mean that the problem cannot be translated to the solvers input format.

complex reasoning is involved. Especially as soon as existential quantification is involved the SMT solvers can hardly solve any of the problems. This is not surprising since SMT solvers target at solving quantifier-free fragments of first-order logic.

The pattern of performance within the superposition provers is the same as before. ZipRewrite does a bit better than the superposition solvers without any explicitly specified rewrite rules, which might be accounted to the same reason as before.

Table 5.7 lists the results of the final experiment. As the previous experiments have shown reasoning in the reflective theories is hard even for very simple conjectures. Therefore it is not surprising that it is even harder for problems that require inductive reasoning to solve.

Nevertheless there are some problems that can be solved using the reflective inductive extension instead of built-in induction heuristics. The most striking result is that Z3 is able to solve benchmarks that involve induction, even though it is a SMT-solver without any support for inductive reasoning.

5.5 Discussion

Our experimental results show that meta-level reasoning using a reflective extension of a theory is hard, hence replacing induction by reflective reasoning does not make the problem easier at once. Despite that, the experiments show that some solvers can solver problems with this approach to induction, which can be considered a proof of concept.

In fact this approach could become more feasible if solvers would have techniques that are more tuned to this kind of benchmarks.

One problem mentioned before might be the ordering of equalities in saturation algorithms. This could be approached using different term orderings like Lexicographical Path Ordering (LPO), transfinite KBO [LW07], or KBO with weight functions tailored for this application.

Another approach to make reasoning in these inductive extensions more feasible, would be to consider the sorts introduced when extending the signature ($\mathsf{var}_\sigma, \mathsf{term}_\Sigma, \mathsf{form}$, and env) as inductive datatypes themselves. Obviously it would not make sense to use solvers' support for inductive reasoning on them, and not on the original signature, but since many SMT solvers support reasoning about term algebras without full induction, this could yield a performance boost.

A further possibility to make reflective reasoning more feasible is to encode the reflective extension differently. For example the choice of the primary connectives for the reflective extension (in our case $\dot{\vee}$, and $\dot{\neg}$) could have effects on proof search as well.

What our last experiment also highlights is the fact that induction is hard; with or without reflective reasoning involved. Properties that seem trivial for a human like commutativity of multiplication, cannot be proven by any of the solvers we investigated.

CHAPTER 6

Conclusion

We saw different approaches to inductive reasoning, that can be categorized in what many ways. The categorization that was most important for our needs is the distinction between first-order induction, like in the theory **PA**, and second-order induction, like in **TA**.

As theories involving second-order induction are not even semi-decidable we decided to focus on developing a method for first-order induction. First-order induction involves an infinite number of axioms, namely all instances of the induction scheme. As modern theorem provers are computer programs, they need a finite input, which means this infinite number of axioms is impractical. Mathematical practice is to write down these infinite sets of axioms as schemes of formulas. Alas these schemes of axioms are not part of standard input syntax of todays theorem proves. In order to circumvent this shortcoming we developed a method to express these schematic definitions in the language of first-order logic by means of a conservative extension, which we called the reflective extension of a theory.

We showed that this reflective extension is indeed a conservative extension of the base theory, and contains a truth predicate, which gives us the means to quantify over formulas within the language of first-order logic.

Using this method we replaced the first-order induction scheme of **PA** by the axioms needed for the reflective extension, and a single additional axiom, called the reflective induction axiom, and proved that the resulting theory is indeed a conservative extension of **PA**. Further we demonstrated how to replace the induction scheme of a theory with arbitrary inductive datatypes. This kind of conservative extension is what we called the reflective inductive extension.

Additionally we sketched how this approach can be used to formalize the Hoare calculus in pure first-order logic, using the reflective extension of a theory of integer arithmetic.

Our experiments show that reasoning in the reflective extension of a theory is hard for modern theorem provers, even for very simple problems. Despite the bad performance in general, we have a positive result serving as a proof of concept of our method, namely that the SMT-solver Z3, which does not support induction natively was able to solve problems that require inductive reasoning. Further our experiments confirmed that inductive reasoning is hard for most state-of-the-art theorem provers, even for very simple problems. This confirms again the need for new methods to be developed in inductive reasoning.

This thesis gives theoretical foundations, and a proof of concept for reflective reasoning. Future work could go in two directions, practical and theoretical. Firstly specialized reasoning procedures could be developed in order to make this work tractable in practice. This could include rules that bypass the axioms that relate the formulas ϕ and $\ulcorner \phi \urcorner$, and instead replacing whole formulas by there reflective counterpart in one step, and vice versa. As discussed in section 5.4 there are many heuristics that could be explored in order to improve reasoning in this theory.
On the theoretical side an in-depth investigation of the ideas presented in section 4.4, needs to be conducted. Further the relation between the reflective extension of a theory with quantification over formulas, and second-order logic could be explored.

Bibliography

[Aub79] Raymond Aubin. Mechanizing structural induction part II: strategies. *Theor. Comput. Sci.*, 9:347–362, 1979.

[Bar82] Jon Barwise. *Handbook of mathematical logic*. Elsevier, 1982.

[BBCW20] Alexander Bentkamp, Jasmin Blanchette, Simon Cruanes, and Uwe Waldmann. Superposition for lambda-free higher-order logic. *CoRR*, abs/2005.02094, 2020.

[Bek05] L D Beklemishev. Reflection principles and provability algebras in formal arithmetic. *Russian Mathematical Surveys*, 60(2):197–268, apr 2005.

[BGP12] James Brotherston, Nikos Gorogiannis, and Rasmus Lerchedahl Petersen. A generic cyclic theorem prover. In Ranjit Jhala and Atsushi Igarashi, editors, *Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings*, volume 7705 of *Lecture Notes in Computer Science*, pages 350–367. Springer, 2012.

[BIS92] Siani Baker, Andrew Ireland, and Alan Smaill. On the use of the constructive omega-rule within automated deduction. In Andrei Voronkov, editor, *Logic Programming and Automated Reasoning,International Conference LPAR'92, St. Petersburg, Russia, July 15-20, 1992, Proceedings*, volume 624 of *Lecture Notes in Computer Science*, pages 214–225. Springer, 1992.

[BM75] Robert S. Boyer and J. Strother Moore. Proving theorems about LISP functions. *J. ACM*, 22(1):129–144, 1975.

[BR20a] Ahmed Bhayat and Giles Reger. A combinator-based superposition calculus for higher-order logic. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I*, volume 12166 of *Lecture Notes in Computer Science*, pages 278–296. Springer, 2020.

[BR20b] Ahmed Bhayat and Giles Reger. A polymorphic vampire - (short paper). In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France,* *July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 361–368. Springer, 2020.

[BS11] James Brotherston and Alex Simpson. Sequent calculi for induction and infinite descent. *J. Log. Comput.*, 21(6):1177–1216, 2011.

[BT19] Stefano Berardi and Makoto Tatsuta. Classical system of martin-lof's inductive definitions is not equivalent to cyclic proofs. *Logical Methods in Computer Science*, 15(3), 2019.

[BW99] Stefan Berghofer and Markus Wenzel. Inductive datatypes in HOL - lessons learned in formal-logic engineering. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin-Mohring, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*, volume 1690 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 1999.

[CJRS12] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Hipspec: Automating inductive proofs of program properties. In Jacques D. Fleuriot, Peter Höfner, Annabelle McIver, and Alan Smaill, editors, *ATx'12/WInG'12: Joint Proceedings of the Workshops on Automated Theory eXploration and on Invariant Generation, Manchester, UK, June 2012*, volume 17 of *EPiC Series in Computing*, pages 16–25. EasyChair, 2012.

[Com94] Hubert Comon. Inductionless induction, 1994.

[Cru17] Simon Cruanes. Superposition with Structural Induction. In *Proc. of FRoCoS*, pages 172–188, 2017.

[DJ07] Lucas Dixon and Moa Johansson. Isaplanner 2: A proof planner in isabelle. *DReaM Technical Report (System description)*, 2007.

[DKW08] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 27(7):1165–1178, 2008.

[dN98] Hans de Nivelle. The resolution calculus, alexander leitsch. *J. Log. Lang. Inf.*, 7(4):499–502, 1998.

[EP20] Mnacho Echenim and Nicolas Peltier. Combining induction and saturation-based theorem proving. *J. Autom. Reason.*, 64(2):253–294, 2020.

[GKR18] Bernhard Gleiss, Laura Kovács, and Simon Robillard. Loop analysis by quantification over iterations. In Gilles Barthe, Geoff Sutcliffe, and Margus Veanes, editors, *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*, volume 57 of *EPiC Series in Computing*, pages 381–399. EasyChair, 2018.

[HHK$^+$20] Márton Hajdú, Petra Hozzová, Laura Kovács, Johannes Schoisswohl, and Andrei Voronkov. Induction with generalization in superposition reasoning. In Christoph Benzmüller and Bruce R. Miller, editors, *Intelligent Computer Mathematics - 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings*, volume 12236 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 2020.

[Hor11] Leon Horsten. *The Tarskian Turn: Deflationism and Axiomatic Truth*. Mit Press. MIT Press, 2011.

[HUW14] John Harrison, Josef Urban, and Freek Wiedijk. History of interactive theorem proving. In Jörg H. Siekmann, editor, *Computational Logic*, volume 9 of *Handbook of the History of Logic*, pages 135–214. Elsevier, 2014.

[HW17] Stefan Hetzl and Tin Lok Wong. Some observations on the logical foundations of inductive theorem proving. *Logical Methods in Computer Science*, 13(4), 2017.

[KKRV16] Evgenii Kotelnikov, Laura Kovács, Giles Reger, and Andrei Voronkov. The vampire and the FOOL. In Jeremy Avigad and Adam Chlipala, editors, *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, pages 37–48. ACM, 2016.

[KP13] Abdelkader Kersani and Nicolas Peltier. Combining superposition and induction: A practical realization. In Pascal Fontaine, Christophe Ringeissen, and Renate A. Schmidt, editors, *Frontiers of Combining Systems - 9th International Symposium, FroCoS 2013, Nancy, France, September 18-20, 2013. Proceedings*, volume 8152 of *Lecture Notes in Computer Science*, pages 7–22. Springer, 2013.

[KRV17a] Laura Kovács, Simon Robillard, and Andrei Voronkov. Coming to terms with quantified reasoning. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 260–270. ACM, 2017.

[KRV17b] Laura Kovács, Simon Robillard, and Andrei Voronkov. Coming to Terms with Quantified Reasoning. In *Proc. of POPL*, pages 260–270, 2017.

[Lei12] K. Rustan M. Leino. Automating induction with an SMT solver. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, volume 7148 of *Lecture Notes in Computer Science*, pages 315–331. Springer, 2012.

[LW07] Michel Ludwig and Uwe Waldmann. An extension of the knuth-bendix ordering with lpo-like properties. In Nachum Dershowitz and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings*, volume 4790 of *Lecture Notes in Computer Science*, pages 348–362. Springer, 2007.

[Moo19] J. Strother Moore. Milestones from the pure lisp theorem prover to ACL2. *Formal Aspects Comput.*, 31(6):699–732, 2019.

[PCI$^+$20] Grant Passmore, Simon Cruanes, Denis Ignatovich, Dave Aitken, Matt Bray, Elijah Kagan, Kostya Kanishev, Ewen Maclean, and Nicola Mometto. The imandra automated reasoning system (system description). In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 464–471, Cham, 2020. Springer International Publishing.

[Pfe84] Frank Pfenning. Analytic and non-analytic proofs. In Robert E. Shostak, editor, *7th International Conference on Automated Deduction, Napa, California, USA, May 14-16, 1984, Proceedings*, volume 170 of *Lecture Notes in Computer Science*, pages 394–413. Springer, 1984.

[RK15] Andrew Reynolds and Viktor Kuncak. Induction for SMT Solvers. In *Proc. of VMCAI*, pages 80–98, 2015.

[RSV18] Giles Reger, Martin Suda, and Andrei Voronkov. Unification with abstraction and theory instantiation in saturation-based reasoning. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*, volume 10805 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2018.

[RV19] Giles Reger and Andrei Voronkov. Induction in Saturation-Based Proof Search. In *Proc. of CADE*, pages 477–494, 2019.

[SDE12] William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. Zeno: An Automated Prover for Properties of Recursive Data Structures. In *Proc. of TACAS*, pages 407–421, 2012.

[Sha91] Stewart Shapiro. *Foundations without foundationalism: A case for second-order logic*, volume 17. Clarendon Press, 1991.

[Str12] Sorin Stratulat. A unified view of induction reasoning for first-order logic. In Andrei Voronkov, editor, *Turing-100 - The Alan Turing Centenary, Manchester, UK, June 22-25, 2012*, volume 10 of *EPiC Series in Computing*, pages 326–352. EasyChair, 2012.

[Tak13] Gaisi Takeuti. *Proof theory*, volume 81. Courier Corporation, 2013.

[vB06] Johan van Benthem. Modal frame correspondences and fixed-points. *Stud Logica*, 83(1-3):133–155, 2006.

[Vor14] Andrei Voronkov. Avatar: The architecture for first-order theorem provers. In *Proceedings of the 16th International Conference on Computer Aided Verification-Volume 8559*, pages 696–710. Springer-Verlag, 2014.