Formalization of Biform Theories in Isabelle

FORMALIZATION OF BIFORM THEORIES IN ISABELLE

BY

LEKHANI RAY, B.Eng.

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING AND SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTERS OF SCIENCE

Masters of Science (2022)	McMaster University
(Computing and Software)	Hamilton, Ontario, Canada

TITLE:	Formalization of Biform Theories in Isabelle
AUTHOR:	Lekhani Ray
	B.Eng. (Computer Science and Engineering)
	Vellore Institute of Technology, Vellore, India
SUPERVISOR:	Dr. William M. Farmer
NUMBER OF PAGES:	xi, 103

To my parents and sister

Abstract

A biform theory is a combination of an axiomatic theory and an algorithmic theory. It is used to integrate reasoning and computation in a common theory and can include algorithms with precisely specified input-output relationships. Isabelle is one of the leading interactive theorem provers. Isabelle includes locales, a module system that uses theory morphisms to manage theory hierarchies, and that has a rich and extensive library with multiple useful proof and formalization techniques. A case study of eight biform theories of natural number arithmetic is described in the paper "Formalizing Mathematical Knowledge as a Biform Theory Graph" by J. Carette and W. M. Farmer. The biform theories form a graph linked by theory morphisms. Seven of the biform theories are in first-order logic and one is in simple type theory. The purpose of this thesis is to test how a theory graph of biform theories can be formalized in Isabelle by attempting to formalize this case study. We work with locales and sublocales in Isabelle to formalize the test case. The eight biform theories are defined as regular axiomatic theories, while the algorithms are functions defined on inductive types representing the syntax of the theories.

Acknowledgements

I want to express my deepest gratitude to my supervisor and mentor, Dr. William M. Farmer, who gave me the knowledge and motivation to pursue the research that went behind the thesis. His expertise and past work have been a massive inspiration in this journey.

Many thanks to members of my supervisory committee, Dr. Wolfram Kahl, Dr. Christopher Anand, and Dr. Ridha Khedri. Dr. Kahl's comments and constructive feedback over the years on the research topic has been instrumental in shaping the direction of the research topic.

I am grateful to the Department of Computing and Software for providing me with an opportunity to pursue my research at McMaster University and my peers who shaped my journey with their valuable input and experiences.

Notation and abbreviations

Abbreviation	**Full-form**
ATP	Automated theorem proving
BT	Biform theory
$\mathrm{CTT}_{\mathrm{uqe}}$	Church's type theory with undefinedness, quotation, and evaluation
FFMM	Formal framework for managing mathematics
FOL	First-order-logic
HOL	Higher-order-logic
IMPS	Interactive Mathematical Proof System
MMS	Mechanised mathematical system
MMT	Meta-meta-theory
PVS	Prototype Verification System
RQ	Research question
SBMA	Syntax-based mathematical algorithms
SMT	Satisfiability modulo theories
STT	Simple type theory
IsaFol	Isabelle Formalization of Logic
IsaFoR/CeTA	Isabelle/HOL Formalization of Rewriting for Certified Tool Assertions

Contents

List of Figures

Chapter 1

Introduction and problem statement

1.1 Formal mathematics

"The development of mathematics toward greater precision has led, as is well known, to the formalization of large tracts of it, so that one can prove any theorem using nothing but a few mechanical rules."

-QED manifesto Anonymous (1994)

Natural number arithmetic is likely to be a part of the core of any formal mathematics library, and the formalization of mathematical knowledge has been a long drawn effort. We define formalization as a means of expressing mathematics, both statements and verifications, in a formal language with rigid rules of grammar and precise semantics. Mathematical theorems and proofs, drawn up in a formal manner, constitute formalized mathematics. A proof checker can automatically verify whether the steps and details in the language are correct.

There are different ways of formalizing or specifying natural number arithmetic, the most notable ones being the axiomatic and algorithmic methods. The axiomatic method expresses knowledge as an axiomatic theory consisting of a language and set of axioms. The assumptions about the theory are expressed by the axioms. Logical consequences of the axioms are the facts of the theory. The axiomatic method is well suited to provide results that are both exact and systematic. The algorithmic method in comparison uses an algorithmic theory. The algorithmic theory consists of a language and a group of algorithms, which perform symbolic computations over the expressions of the language. The algorithmic method is well suited to encoding the relation between the input and output of mathematical operations.

1.2 Research motivation

Software development has increasingly become a significant and vital activity in our society. It controls all significant parts and activities of our lives including communication, power generation, and travel. One major characteristic of high-quality software is its correctness. Formal software correctness is a notion that has become highly prevalent. In software engineering, the correctness of a program or system is achieved if it behaves exactly as intended for all of its use-cases.

Proof assistants based on type theory (simple type theory or dependent type theory) or set theory, have shown their ability to systematically prove the correctness of critical software. Isabelle is one of the most widely used proof assistants. Its expressive syntax and readability make it a natural choice for someone with limited knowledge of formal methods. Yushkovskiy and Tripakis (2018) explain how Isabelle was built in a modular manner and can be extended by numerous basic theories making it expressive and relatively concise.

The motivation behind my thesis is to explore the workings of a proof assistant like Isabelle and attempt to formalize a case study consisting of eight seemingly simple biform theories (BT), linked by theory morphisms. Very few proof assistants provide the means to directly build theory graphs in a straightforward manner. A theory graph is a directed graph whose nodes are theories and edges are theory morphisms. This thesis aims to explore whether a body of mathematical knowledge can be effectively formalized as a theory graph of biform theories in a proof assistant.

1.3 Case study

The topic of this thesis has been based on and inspired by Carette and Farmer (2017). The aim of the thesis is to express and formalize this case study in Isabelle/HOL. The biform theories are theories of natural number arithmetic. Seven of the theories are in first-order logic and one is in simple type theory. BT1 and BT5 are theories of 0 and S (the successor function). BT2 and BT6 are theories of 0, S, and $+$. BT3, BT4, and BT7 are theories of 0, S, $+$, and $*$. BT8 is a theory of higher-order Peano arithmetic with the type ι of individuals and constants 0_ι and $S_{\iota\rightarrow\iota}$. The eight theories are connected by theory morphisms. Figure 1.1 gives a graphical representation of how the eight theories are connected with strict and non-strict theory inclusions and an interlogical theory morphism (see page 16 for

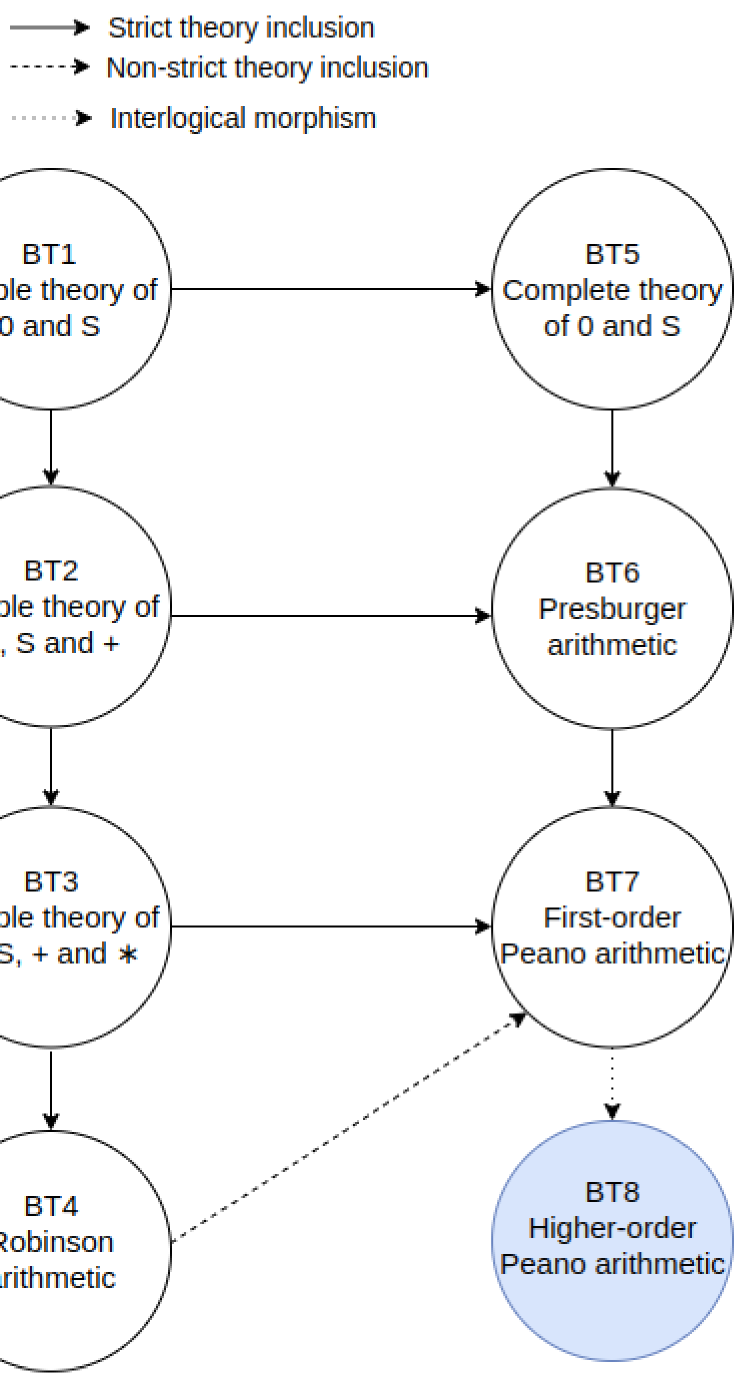

Strict theory inclusion
Non-strict theory inclusion
Interlogical morphism
BT1 Simple theory of 0 and S
BT5 Complete theory of 0 and S
BT2 Simple theory of 0, S and +
BT6 Presburger arithmetic
BT3 Simple theory of 0, S, + and ∗
BT7 First-order Peano arithmetic
BT4 Robinson arithmetic
BT8 Higher-order Peano arithmetic

Figure 1.1: Biform theory graph test case

precise definitions). These eight are chosen because they naturally fit together and have simple axiomatizations.

The case study includes a number of algorithms that manipulate expressions. We have added a variety of useful algorithms to the theories as functions on expressions. For example, we use the transformer `subTerm` that substitutes a term for the free occurences of a variable in a given term. We have defined another transformer, `subForm` that substitutes a term for the free occurences of a variable in a given formula. `subTerm`and `subForm` manipulate the syntax of terms and formulas, respectively.

The term *first-order logic*, abbreviated as FOL, is used to indicate first-order predicate logic, which is the most popular formal logic. Simple type theory is a version of classical higher-order predicate logic. It is a highly expressive natural extension of first-order-logic. Farmer (2008) argues that "simple type theory is an attractive alternative to first-order logic for practical-minded scientists, engineers, and mathematicians." The eight theories have been formalized using locales, a module system in Isabelle used for managing theory hierarchies through interpretations. The theory morphisms between the theories have been formalized using sublocales.

Details about the case study biform theory graph are found later in Chapter 3.

1.4 Challenge problem

The challenge problem for the thesis is to express the theory graph of the case study consisting of eight biform theories of natural number arithmetic (see Figure 1.1 above) in Isabelle. The case study is particularly interesting because it consists of a variety of theories. We have theories that are simple and very weak (e.g., the simple theory of 0 and S) as well as theories that are highly expressive (e.g., higher-order Peano arithmetic).

Chapter 2

Background

2.1 Traditional predicate logic

There are many definitions of logic in the literature. I would like to understand logic as a system that aims to draw a conclusion from a set of given assumptions in a sound way. Logic uses data to make inferences. This thesis deals with logic in a formal or mathematical manner. FOL and STT can be distinguished based on what they quantify over. In FOL, we are allowed to quantify only over individuals. In STT, we are allowed to quantify over individuals as well as other objects. For instance, we can also quantify over higher-order objects such as functions and predicates. The following is an example of an FOL formula:

$$\forall x \ (\exists y \ R(x, y) \to R(x, f(y))).$$

x and y are variables which denote individual objects of a chosen domain. f is a function symbol which denotes a (unary) function from the domain to the domain. R is a predicate symbol representing a (binary) relation of objects of the domain.

$\forall x$ and $\exists y$ are quantifiers which range over individuals, i.e., the members of the domain. First-order logic is restricted to this form of quantification. STT has a type of individuals as well as more complicated types such as:

$$(\iota \to \iota) \to (\iota \to o)$$

where ι is the type of individuals and o is the type of truth values.

The following expression, `compose`, is an example of an STT expression:

$$\lambda f : (\iota \to \iota) \;.\; \lambda g : (\iota \to \iota) \;.\; \lambda x : \iota \;.\; f(g(x)).$$

If f and g are variables of the type $(\iota \to \iota)$, then $\texttt{compose}(f)(g)$ is an expression in STT that denotes the composition of the functions denoted by f and g.

2.1.1 First-order-logic

First-order-logic is the leading form of predicate logic. It quantifies over individuals such as natural numbers. It cannot quantify over higher-order objects such as functions and predicates. First-order logic adds terms, predicates, and quantifiers to propositional logic. A *first-order language* (or *signature*) can be formally defined as a triple $L = (\mathcal{C}, \mathcal{F}, \mathcal{P})$, where:

- $\mathcal{C}$ is a set of *constant symbols* that denote individuals.
- $\mathcal{F}$ is a set of *function symbols* with assigned arities that denote functions on individuals.
- $\mathcal{P}$ is a set of *predicate symbols* with assigned arities that denote predicates on individuals.

BT2 of the case study is the simple theory of 0, S and $+$. The axiomatic part of BT2 is in FOL and can be represented in the language $L = (\mathcal{C}, \mathcal{F}, \mathcal{P})$ where:

- $\mathcal{C} = \{0\}$
- $\mathcal{F} = \{S, +\}$, where S is a unary operator and $+$ is a binary operator.
- $\mathcal{P} = \{=\}$, where $=$ is binary.

2.1.2 Simple type theory

Simple type theory is a prevalent form of type theory. It is higher-order because it quantifies over higher-order objects such as functions and predicates. Simple type theory is based on the same principles as in first-order logic. It includes n^{th}-order logic for all $n \geqslant 1$. Various proof assistants are based on Church's type theory, a version of simple type theory that is based on functions and uses lambda-notation and lambda-conversion to build and apply functions. These include IMPS, that can handle undefined expressions (Farmer *et al.*, 1993); Isabelle (Wenzel *et al.*, 2008), the proof assistant we use in this thesis; ProofPower (Oliveira *et al.*, 2006), a group of tools that support specification and proof in Higher Order Logic (HOL) and the Z notation; PVS (Owre *et al.*, 1992); HOL Light (Harrison, 2009); HOL4 (Slind and Norrish, 2008), and TPS, an automatic theorem proving system for Church's type theory (Andrews *et al.*, 1996). BT8 of the case study is a theory of simple type theory.

2.1.3 Transformers

A *transformer* is a program whose input and output are expressions. Transformers represent syntax-manipulating operations. Algorithms that apply arithmetic operations to numerals or transpose matrices are examples of a transformer. The *computational behavior* of a transformer is the relationship between its input and output expressions. A

2.1.4 Syntax-based mathematical algorithms

A number of the algorithms used in mathematics work by manipulating the syntactic structure of mathematical expressions. A *syntax-based mathematical algorithm* (SBMA) is a transformer that manipulates the expressions of a formal language in a mathematically meaningful way. When a transformer is an SBMA, its *mathematical meaning* is the relationship between the mathematical meanings of its input and output expressions. A *meaning formula* for an SBMA is a statement that expresses the meaning of the SBMA.

2.1.5 Limitations of logic

Traditional logic, such as first-order logic or simple type theory, can make it challenging to express statements that capture the association between the input and output of transformers. This is usually because there is no direct way to refer to the syntactic structure of the expressions in the logic. Farmer (2014) argues that to express the meaning of transformers adequately, a logic is needed that has (1) an *inductive type* of syntactic values that represent the syntactic structures of the expressions in the logic, (2) *a quotation operator* in the logic that map expressions to syntactic values, and (3) an *evaluation operator* in the logic that maps syntactic values to the values of the expression they denote.

Another notable limitation for expressing ideas about syntax in logic is that syntactic notions, such as a “free variable”, may depend on the semantics of the expression as well as its syntax. This makes operations like substitution a lot more complicated to express than in traditional logic.

2.2 Axiomatic theories

An *axiom* is a statement that serves as a starting point from which other statements are logically derived. An *axiomatic theory*, contains a formal language and a group of axioms expressed in the language. It specifies a set of mathematical structures: the language provides names for the objects in the structures, and the axioms define the properties of the objects. In formal mathematics, it can be represented as (L, Γ), where L is the language of the theory and Γ is the set of axioms of the theory.

2.3 Algorithmic theories

An algorithm is defined as a finite sequence of instructions that may take input and produce output. An algorithmic theory uses a set of algorithms to represent mathematical knowledge. In formal mathematics, an algorithmic theory is defined as (L, Π), where L is a language, and Π is the set of transformers (algorithms) that manipulate the expressions in L. Algorithmic theories alone are laborious to formalize in traditional logic without machinery to reason about syntax.

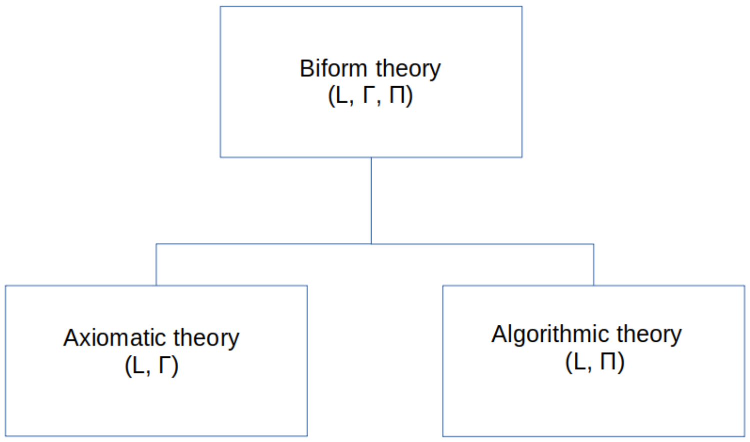


Figure 2.1: Representation of biform theory

2.4 Biform theories

A biform theory combines an axiomatic theory and an algorithmic theory to support the integration of reasoning and computation. Formalizing biform theories is difficult as it requires a way to express statements about algorithms: What they do and what their actions mean mathematically. There are various transformers for natural number arithmetic. They manipulate expressions to build bigger expressions or check whether the expression satisfies certain syntactic properties. A biform theory can be represented as the tuple (L, Γ, Π), where L is a language of the underlying logic, Π is the set of transformers (algorithms) that implement functions on expressions of L, and Γ is a set of axioms of L. Then (L, Γ) is an axiomatic theory, and (L, Π) is an algorithmic theory. Figure 2.1 shows the diagramatic representation of biform theories.

2.4.1 An example: BT2

BT2 of the case study is the simple theory of 0, S, and +. The biform theory is expressed using four axioms and includes an algorithm `add` that performs the addition of binary numerals.

We will show how BT2 could possibly be expressed in a partial manner in many-sorted first-order logic. Many-sorted first-order logic is a version of first-order logic that works with multiple domains of individuals (referred to as sorts). This makes it more practical than standard first-order logic.

The language of the MSFOL is a tuple

$\Sigma = (\mathcal{B}, \mathcal{C}, \mathcal{F}, \mathcal{P}, \tau)$

where:

$\mathcal{B}$ is the nonempty set {`Nat, BinNum`} of base types.

$\mathcal{C}$ is the set {`0, Zero, One`} of constant symbols.

$\mathcal{F}$ is the set {S`, +, JoinZero, JoinOne, val, add`} of function symbols.

$\mathcal{P}$ is the set {=} of predicate symbols.

τ is a function that maps the constant, function and predicate symbols to base types and function types constructed from the base types.

τ`(0) = Nat`

τ`(Zero) = BinNum`

τ`(One) = BinNum`

$\tau(S)$` = Nat` $\to$ `Nat`

τ`(JoinZero) : BinNum` $\to$ `BinNum`

τ`(JoinOne) : BinNum` $\to$ `BinNum`

τ`(val) = BinNum` $\to$ `Nat`

τ(add) = BinNum x BinNum $\rightarrow$ BinNum

The "no-confusion" axioms for Nat are:

A1 : $S(x) \neq 0$

A2 : $S(x) = S(y) \implies x = y$

The axioms for + are:

A3 : $x + 0 = x$

A4 : $x + S(y) = S(x + y)$

The "no confusion" axioms for BinNum are:

A5 : Zero $\neq$ One

A6 : Zero $\neq$ JoinZero(u)

A7 : Zero $\neq$ JoinOne(u)

A8 : One $\neq$ JoinZero(u)

A9 : One $\neq$ JoinOne(u)

A10 : JoinZero(u) $\neq$ JoinOne(u)

A11 : JoinZero(u) $=$ JoinZero(v) $\implies u = v$

A12 : JoinOne(u) $=$ JoinOne(v) $\implies u = v$

The function val is defined by:

A29 : val(Zero) = 0

A30 : val(One) = $S(0)$

A31 : val(JoinZero(u)) = val(u) + val(u)

A32 : val(JoinOne(u)) = val(u) + val(u) + $S(0)$

The function add is defined by:

```
A13 : add(Zero, u) = u

A14 : add(u, Zero) = u

A15 : add(One, One) = JoinZero(One)

A16 : add(One, JoinZero(u)) = JoinOne(u)

A17 : add(JoinZero(u), One) = JoinOne(u)

A18 : add(One, JoinOne(u)) = JoinZero(add(One, u))

A19 : add(JoinOne(u), One) = JoinZero(add(One, u))

A20 : add(JoinZero(u), JoinZero(v)) = JoinZero(add(u, v))

A21 : add(JoinZero(u), JoinOne(v)) = JoinOne(add(u, v))

A22 : add(JoinOne(u), JoinZero(v)) = JoinOne(add(u, v))

A23 : add(JoinOne(u), JoinOne(v)) =

                JoinZero(add(add(u, v), One))
```

2.5 Local vs. global reflection

In order to formalize biform theories, an infrastructure is required to reason with the expressions manipulated by an SBMA. There are two main approaches to building the infrastructure: local and global. In local reflection, the infrastructure is for L, the language of the SBMA's inputs and outputs. The local reflection infrastructure consists of: (1) An *inductive type* of syntactic values L' that represents the expressions in the language L. (2) a *quotation operator* $\ulcorner\cdot\urcorner$ that maps the expressions in L to the syntactic values of L'. (3) An *evaluation operator* $[\![\cdot]\!]$ that maps syntactic values of L' to the expressions in L. The quotation operator is the inverse of the evaluation operator. The local approach does not scale up since each SBMA may require a separate infrastructure (Carette and Farmer, 2017).

The global approach has been proposed as the alternative approach to the local approach in (Farmer, 2013). The global reflection consists of a single infrastructure for all SBMAs, which consists of: (1) An *inductive type* representing the entire set of expressions. (2) A *global quotation operator* $\ulcorner\cdot\urcorner$. (3) A *global evaluation operator* $[\![\cdot]\!]$. The biform theory graph of the test case of natural number has been formalized in CTT_{uqe} using the global approach (Carette and Farmer, 2017). The quotation and evaluation operators enable the formalization of syntax-based mathematical algorithms (Farmer, 2016). The biform theory graph of the test case of natural number has been formalized in Agda using the local approach (Carette and Farmer, 2017). Both methods are discussed in more detail later in chapter 3.

2.6 Theory morphisms

A *theory morphism* is a mapping of formulas of one theory to the formulas of another such that valid formulas are always mapped to valid formulas (Farmer, 2017). The theories can be considered abstract mathematical models, and the morphisms enable definitions and theorems to be transported from one theory to another without losing meaning. A morphism $\Sigma \to \Sigma'$ consists of a list of assignments, a source theory Σ, and a target theory Σ'. A *theory inclusion* from a source theory to a target theory is a theory morphism whose mapping is the identity function. A theory inclusion could be strict, or non-strict. A theory inclusion is *strict* when all the axioms of the source theory are mapped to axioms in the target theory. A theory inclusion is *non-strict* when some of the axioms of the source theory are mapped to theorems that are not axioms. If there is a theory morphism going both ways, i.e, from theory A to theory B, and from theory B to theory A, the two theories are said to be*equivalent*.

Fig 1.1 in chapter 1 shows the biform theories in the case study connected by theory morphisms. The solid arrows show strict theory inclusions. The morphism from BT4 to BT7 is a non-strict mapping. The theory morphism from BT7 to BT8 is interlogical since their corresponding logics are different. The morphism maps 0 to 0_ι, S to $S_{\iota\to\iota}$, $+$ to $+_{\iota\to\iota\to\iota}$ and $*$ to $*_{\iota\to\iota\to\iota}$ where $+_{\iota\to\iota\to\iota}$ and $*_{\iota\to\iota\to\iota}$ are defined constants in BT8.

2.7 Theory graphs

A *theory graph* is a directed graph whose nodes are theories and edges are theory morphisms. Kohlhase (2014) says that the main idea of the theory graph with respect to mathematical knowledge management is to use morphisms to structure mathematical knowledge in a modular manner. Theory graphs are well suited for formalizing mathematical knowledge because the knowledge can be expressed at "the most convenient level of abstraction using the most convenient vocabulary" (Carette and Farmer, 2017). A theory graph does away with any duplication of concepts and facts. Figure 1.1 is the theory graph for the case study of biform theories of natural numbers. It shows the theory morphisms between the biform theories.

2.8 Isabelle

Isabelle is a popular proof assistant that was built around a relatively small core. Numerous fundamental theories are utilised to extend the core. HOL is a theorem prover for higher-order logic. The theory of higher-order logic is implemented as Isabelle/HOL, and is commonly used because of its powerful specification tools. In proofs, Isabelle combines HOL as a functional programming language and Isar as the language for describing procedures in order to manipulate the proof. Isabelle/jEdit is the official graphical interface to interact with Isabelle. While carrying out a proof, Isabelle saves the proof state and a list of goals that need to be proved. The proof assistant often acquires the features of an automated theorem prover and can automatically solve or prove complex statements with `auto`.

In the past 20 years, Isabelle has been used by numerous researchers and students of computer science and mathematics worldwide. Isabelle has an extensive library with many pre-proven lemmas and theorems.

2.8.1 Isabelle theories

An Isabelle theory is the basic structure used in an Isabelle file (all Isabelle files have the extension .thy). It is a collection of types, functions, and theorems. The general format of the theory is as follows:

```
theory T
imports B₁ ··· Bₙ


begin
```

```
    ...declarations, definitions, and proofs
    end
```

where:

- $B_1 \cdots B_n$ are the names of existing theories that T is based on. $B_1 \cdots B_n$ are the direct parent theories of T. Everything defined in the parent theories (and their parents, recursively) is automatically visible in T.
- Declarations represent the newly introduced types.
- Definitions refer to the functions, locales and theorems.
- Proofs prove new theorems.

2.8.2 Overview of Isar

Isar stands for Intelligible Semi-Automated Reasoning. Isar (Wenzel and Paulson, 2006) is a textual proof format inspired by the pioneering Mizar system. The Mizar project started around 1973 as an attempt to reconstruct a mathematical vocabulary in a computer-oriented environment. It makes it possible to write structured, readable proofs. The Isar subsystem is an extension of Isabelle. It hides the implementation language almost wholly. Isar supports a calculational style of reasoning and allows us to provide structured proofs which are presented like traditional mathematical proofs and are understandable for both humans and machines.

A proof can be written in two different ways:

- Compound manner (formal method using `proof - qed`).

- Atomic manner using `by`.

A typical proof skeleton has an assumption, intermediate results, and a conclusion.

2.8.3 Isar commands

The following are some of the main commands used in the Isar proof language which appear regularly in the code in the thesis:

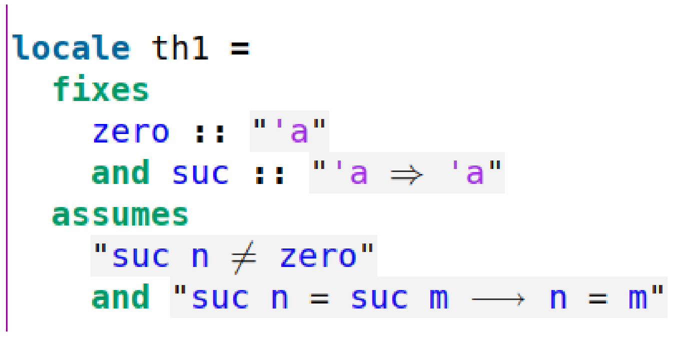
- Primitive commands: `locale, sublocale, lemma, proof, fun, function, interpretation`.
- Automatic commands: `simp, auto, blast, standard`.

The primitive commands build the basic structure of our biform theories. The individual theories are built by `locales`. Supporting functions and transformers are built by `functions`. Morphisms of the theory are built by `sublocales` and `interpretations`.

The above automatic commands are very useful in our implementations. `simp` uses the simplifier, which applies theorems with the `simp` attribute automatically. `auto` uses the simplifier and classical reasoning. Isabelle allows users to tell these reasoners to add or delete specific rules. These automatic reasoners help users prove theorems easier and make proofs shorter.

2.9 Locales

Locales is a modular system in Isabelle (Ballarin, 2014) that is used to represent complex interdependencies between structures. They are helpful for expressing hierarchies of concepts and reducing the number of parameters and assumptions that must be treated through formal development. Due to their easy extension and importation, they are the key to representing our case study of eight biform theories and their morphisms. A locale declaration consists of a sequence of parameters that are indicated with the keyword `fixes` and assumptions indicated with the keyword `assumes`.



```
locale th1 =
  fixes
    zero :: "'a"
    and suc :: "'a ⇒ 'a"
  assumes
    "suc n ≠ zero"
    and "suc n = suc m ⟶ n = m"
```

Figure 2.2: Example of a locale

Here in Figure 2.2 the locale `th1` has parameters `zero` and `suc`, along with its assumptions. Locales are a mechanism to support abstract theories. `import` and `interpretation` enable parameters and assumptions to be transported to other contexts for reuse.

2.9.1 Sublocales

Sublocales are a form of interpretation of locales and is useful for conveying the logical relations between locales. The command `sublocale` $l_2 \subseteq l_1$ causes l_1 to be interpreted in the context of l_2 and all the theorems of l_1 are made available in l_2. Hence the sublocale effectively establishes a theory morphism from l_1 to l_2. The morphism is an inclusion if its mapping is the identity mapping. An inclusion is strict if each axiom of the source theory is mapped to an axiom of the target theory. If there is a strict inclusion from l_1 to l_2, then l_1 is a subtheory of l_2.

2.9.2 Locale interpretations

An interpretation is effectively a morphism from the locale to the background theory. They are used to instantiate, combine, and modify locales. They are also a robust way to verify the satisfiability and consistency of the respective locale. The command `interpretation` is for the interpretation of a locale in theories.

Chapter 3

Objective

The purpose of the thesis is to illustrate how a theory graph of biform theories can be formalized in Isabelle. For this, we have used the case study presented in Carette and Farmer (2017) as a test case. The case study is a theory graph consisting of eight biform theories of natural number arithmetic linked by theory morphisms.

3.1 Case study

The eight biform theories of the case study start with a very simple theory of the successor function and end with full higher-order Peano arithmetic. Seven of the biform theories are in FOL, and one is in STT. The case study is a good test of a proof system's capability to formalize biform theories. The following are the theories in the case study:

- BT1 is the simple theory of 0 and S (the successor function). The theory is written in FOL. It contains the following constants, axioms, and transformers:

Constants:

$0 :$ nullary

$S :$ unary

Axioms:

A1 : $S(x) \neq 0$

A2 : $S(x) = S(y) \implies x = y$

Transformers:

- Recognizer of the language of BT1.

- BT2 is the simple theory of 0, S, and $+$. It is written in FOL and extends BT1 by adding the *plus* function, which is written in infix notation. It contains the following additional constants, axioms, and transformers:

  Additional constants:

  $+ :$ binary

  Additional Axioms:

  A3 : $x + 0 = x$

  A4 : $x + S(y) = S(x + y)$

  Additional transformers:

  - Recognizer of the language of BT2.
  - Addition algorithm for binary numerals.

- BT3 is the simple theory of 0, S, $+$, and $*$. It is written in FOL and extends BT2 by adding the *times* function, which is written in infix notation. It contains the following additional constants, axioms, and transformers:

Additional constants:

$*$: binary

Additional axioms:

A5 : $x * 0 = x$

A6 : $x * S(y) = (x * y) + x$

Additional transformers:

– Recognizer of the language of BT3.

– Multiplication algorithm for binary numerals.

- BT4 is the Robinson Arithmetic. It is written in FOL and extends BT3. It contains one additional axiom:

  A7 : $x = 0 \vee \exists y \,.\, S(y) = x$

- BT5 is the complete theory of 0 and S. It is written in FOL and extends BT1. It contains the following infinite set of axioms given by the induction schema

  A8 : $(A(0) \wedge \forall x \,.\, (A(x) \rightarrow A(S(x)))) \rightarrow \forall x \,.\, A(x)$

  where A is a formula of BT1 and $A(t)$ is the result of replacing each free occurence of x in A with t.
  Additional transformers:

  – Generator for the instances of the induction schema of BT5.

  – A decision procedure for BT5.

- BT6 is the Presburger Arithmetic. It is written in FOL and extends BT2 and BT5. It contains the following infinite set of axioms given by the induction schema

$$\text{A9}: (A(0) \wedge \forall x \,.\, (A(x) \rightarrow A(S(x)))) \rightarrow \forall x \,.\, A(x)$$

where A is a formula of BT2. This is different from the axiom schema in BT5 because the language of BT2 includes $+$ unlike the language of BT1.
Additional transformers:

  - Generator for the instances of the induction schema of BT6.
  - A decision procedure for BT6.

- BT7 is the theory of first order Peano Arithmetic. It is written in FOL and extends BT3 and BT6. BT7 contains the following infinite set of axioms given by the induction schema

  $$\text{A10}: (A(0) \wedge \forall x \,.\, (A(x) \rightarrow A(S(x)))) \rightarrow \forall x \,.\, A(x)$$

  where A is a formula of BT3. This is different from the axiom schema of BT6 because the language of BT3 includes $*$ unlike the language of BT2.
  Additional transformers:

  - Generator for instances of the induction schema of BT7.

- BT8 is a theory of higher-order Peano arithmetic. The theory is written in STT. It uses the following two constants where ι is the base type of individuals.
  Constants:

  0_ι : nullary

  $S_{\iota\rightarrow\iota}$: unary

  Axioms:

$$\mathrm{A11}: S_{\iota\to\iota}(x_\iota) \neq 0_\iota$$

$$\mathrm{A12}: S_{\iota\to\iota}(x_\iota) = S_{\iota\to\iota}(y_\iota) \implies x_\iota = y_\iota$$

$$\mathrm{A13}: (p_{\iota\to o}(0) \wedge \forall x_\iota \,.\, (p_{\iota\to o}(x) \implies p_{\iota\to o}(S(x_\iota)))) \implies \forall x_\iota \,.\, p_{\iota\to o}(x_\iota)$$

Here, A13 is the induction principle for the natural numbers.

Figure 1.1 shows the morphisms that connect the eight theories. The solid arrows ($\rightarrow$) are strict theory inclusions. The morphism from BT4 to BT7 is a non-strict theory inclusion. Each axiom of BT4 is mapped to a theorem of BT7. The theory morphism from BT7 to BT8 is interlogical since their logics are different. It is defined by the mapping of 0 to 0_ι, S to $S_{\iota\to\iota}$, $+$ to $+_{\iota\to\iota\to\iota}$ and $*$ to $*_{\iota\to\iota\to\iota}$ where $+_{\iota\to\iota\to\iota}$ and $*_{\iota\to\iota\to\iota}$ are defined constants in BT8.

3.2 Challenges of the case study

The formalization of the case study of eight biform theories in Isabelle presents the following challenges.

- *Creating the individual theories.*

  The theories in the case study need to be formulated from scratch without using any pre-defined structures. This cannot be done simply by adding axioms to Isabelle, since the result would be one large theory with the properties of BT1 to BT8. We need to extend the background theory while keeping our individual theories separate. A number of additional functions and lemmas will need to be defined along the way to support the biform theories and their transformers.

- *Formalizing the transformers for the biform theory test case in Isabelle.*

The transformers are the algorithms that, along with the axioms, make up the test case biform. Formalizing them requires being able to refer to the syntax of expressions. We cannot do this directly in standard logic, and hence, this may be a challenge.

- *Reasoning about syntax-based mathematical algorithms.*
  SBMAs are transformers that manipulate syntax in a mathematically meaningful way. In order to reason about and prove facts about SBMAs, we need to reason about the interplay of syntax and semantics. This is hard to do since we cannot directly refer to the syntax in the logic of Isabelle.

- *Formalizing the axiom schemas.*
  Each of BT5, BT6, and BT7 have an infinite set of axioms given by axiom schemas. It is difficult to directly formalize axiom schemas in Isabelle and dealing with an infinite set of axioms is a challenge.

- *Finding ways to express theory morphisms in Isabelle.*
  The case study includes theory morphisms. Expressing the theory morphisms requires us to find a way in which the axioms of one theory can be imported or expressed in another theory.

3.3 Research questions

To reiterate the research problem, the aim of the thesis is to express and formalize the eight theories of the case study in Carette and Farmer (2017) in Isabelle/HOL. The case study has provided us with a good insight into Isabelle's capabilities. While formalizing the case study in Isabelle has been challenging, we seek to answer several questions.

- RQ1: Can abstract axiomatic theories be formalized in Isabelle?
- RQ2: Can SBMAs be formalized in Isabelle?
- RQ3: Can the decision procedures for BT5 and BT6 be formalized in Isabelle?
- RQ4: Can theory morphisms be formalized in Isabelle?

Isabelle provides an interactive environment for undertaking proof in an axiomatic manner. The eight theories can be formalized using locales. Locales are a module system in Isabelle used for managing theory hierarchies through interpretations. A variety of valuable transformers are added to the theories through functions and interpretations. The theory morphisms between the theories can be formalized using sublocales.

3.4 Previous solutions

In Carette and Farmer (2017), the case study of the biform theory graph has been formalized in two ways. The first method is to use global reflection in $\mathrm{CTT}_{\mathrm{uqe}}$ (Farmer, 2016), a version of Church-type theory with undefinedness, quotation, and evaluation. The second method is to use local reflection in Agda, a dependently typed programming language.

3.4.1 $\mathbf{CTT}_{\mathrm{uqe}}$

$\mathrm{CTT}_{\mathrm{qe}}$ is a version of Church's type theory with global quotation and evaluation operators. It can reason about the interplay of syntax and semantics. $\mathrm{CTT}_{\mathrm{uqe}}$ is a variant of $\mathrm{CTT}_{\mathrm{qe}}$ that accepts and works with undefined expressions, partial functions, and multiple base types of individuals. It is preferred over $\mathrm{CTT}_{\mathrm{qe}}$ as a logic for building networks of theories connected by theory morphisms.

A biform theory in $\mathrm{CTT}_{\mathrm{uqe}}$ is a triple (L, Π, Γ) where

- L is a language of $\mathrm{CTT}_{\mathrm{uqe}}$. It is generated by a set of base types and constants of $\mathrm{CTT}_{\mathrm{uqe}}$.
- Π is a set of transformers over the expressions of L.
- Γ is a set of formulas of L.

$\mathrm{CTT}_{\mathrm{uqe}}$ follows the global approach for reflection. It contains a logical base type ϵ, a global quotation operator $\ulcorner\cdot\urcorner$ and a typed global evaluation operator $[\![\cdot]\!]$. The type ϵ is a built-in inductive type of "syntactic values" that represent the expressions of the languages. A single reflection framework is used for all the languages in the theory graph, which makes the global approach scalable, unlike the local approach in Farmer (2017). The formalization in $\mathrm{CTT}_{\mathrm{uqe}}$ works very well and shows that the global reflection is a viable approach and an effective mechanism for integrating formal deduction and symbolic computation.

$\mathrm{CTT}_{\mathrm{uqe}}$ has not been implemented, but $\mathrm{CTT}_{\mathrm{qe}}$ (Church's type theory with quotation and evaluation) has been implemented in HOL Light (Carette *et al.*, 2018b).

3.4.2 Agda

Unlike $\mathrm{CTT}_{\mathrm{uqe}}$, that uses the global approach for reflection, Agda uses local reflection. Agda (Carette and Farmer, 2017) uses a set of inductive types to formalize the biform theories. The abstract theories are modeled as records. The built-in type `N`, defined as an inductive type, is used as the syntax for natural numbers.

For each theory that consists of an inductive type, evaluation needs to be defined separately. The inductive type consists of an

1. Inductive type of syntactic values that represent the expressions in L' (which is a subset of the language of the underlying logic).
2. An informal quotation operator which maps the expressions of L' to syntactic values.
3. A formal evaluation operator that maps syntactic values to the values of the expressions in L' that they represent.

In general, Agda requires that new local infrastructures must be created each time a new theory is added to the theory graph. This means we may have more than one infrastructure for our theory graph. The Agda version also does not implement any theory morphisms as the record definitions are not first-class in Agda. The Agda version has some additional features compared to the implementation in $\mathrm{CTT}_{\mathrm{uqe}}$. The transformers *bplus* and *btimes* are implemented to carry out addition and multiplication among binary numerals, respectively. Recognizers are used in this version to check that the theories used are the ones that are wanted. There is an actual implementation for the local approach in Agda; however, it does not support biform theories well.

Chapter 4

Approach

4.1 The different approaches

With most proof assistants having limited support for biform theories, the project's aim is two-fold:

1. To explore Isabelle/Isar as a proof assistant.
2. Test Isabelle's ability to formalize biform theories and link them using theory morphisms.

There are several ways in which axiomatic theories could be expressed in Isabelle.

- *Introducing new axioms*: Isabelle has many structures that allow it to define axioms with or without recursive definitions. This allows us to provide it with a certain number of axioms as assumptions. *Axiomatizations* allow us to introduce several constants simultaneously while stating axiomatic properties. However, this is not a viable approach due to the inability to separate theories from one another. Adding axioms to the theory would result in no difference among the eight biform theories.

- *Using locales and sublocales*: Locales are used in Isabelle to deal with the theorem proving context. Abstract algebra has complex interdependencies between structures. The module system of locales helps considerably with the representation of these structures. Locales are an uncomplicated yet robust extension of the Isar proof language. The notion of locales represent abstract theories in a theory graph, with a flexible form of extension and reusability. Sublocales allow interpretations of locales in other contexts, creating a network of import and interpretation relations.

- *Using type classes*: A third possibility that is similar to locales is that of type classes. Isabelle supports type classes (like those in Haskell, albeit more restrictive), and so one could create a type class and then instantiate it for concrete types. Type classes allow one to specify abstract parameters together with corresponding specifications. The abstract parameters can be instantiated using a particular type. Type classes have a direct link to the Isabelle module system of locales.

- *Using an inductive type of theories:* A powerful method of building the theory graph would be to create an inductive type of theories. This would involve creating new theories and members of the theories using the datatype command. While this would be a strong way to approach the research problem, it would also be an extremely involved and complicated process.

4.2 Our approach

Formalizing our test case of eight biform theories is a two-step process:

1. Formalizing the axiomatic theories:

   The eight biform theories are defined and formalized as *regular axiomatic theories* with the ability to import and extend axioms. In Isabelle/HOL, definitional extensions are favored over axiomatic extensions because axiomatization can create inconsistency in the proof systems and destroy Isabelle's guarantee of soundness. However, axiomatic reasoning is an integral part of mathematics and needs to be carried out safely in Isabelle. Fortunately, we can reason from axioms locally in a sound way and instantiate the axioms later using locales. Locales usually have:

   - Constants of a theory declared using `fixes.`
   - Axioms of a theory declared using `assumes.`

   Inside a locale, definitions can be made and theorems proved based on the constants (parameters) and axioms (assumptions). A locale can import/extend other locales and may also be interpreted in the context of another locale. This creates *morphisms* between the two locales (or theories, in this case). Our case study with eight theories has many morphisms and dependencies. This makes the usage of locales a natural approach to expressing the theories. There is no easy way to formalize the *axiom schemas* for BT5, BT6 and BT7. We allow instances of the induction schema to be added to the theories as needed. The axiomatic formalizations of the eight biform theories, along with their interpretations and theory morphisms have been formalized in Isabelle in the theory file `theory_graph.thy`.

2. Formalizing the algorithmic component of the theories:
   The algorithms are *transformers* that manipulate syntax to make the theories biform theories. We define *inductive types* which represent the language of the theories. A *quotation operator* is required that maps the expressions of the language to its syntactic values. We also need to formalize an *evaluation operator*, which maps the syntactic values to the values of the expressions in the language of the theory. We also reason about *syntax-based mathematical algorithms* through an evaluation mechanism that states and proves the meaning formulas for our algorithms through a lemma. The evaluation function has not been implemented for the entire language of theories, instead it has currently been implemented for sequences of 0s and 1s. All inductive types and transformers which manipulate the syntax of these types has been formalized in `syntax_operations.thy`. `BinNum.thy` contains the transformers which specifically manipulate binary numerals in our theory graphs.

Chapter 5

Axiomatic theory graph

5.1 The formalization of the biform theories

The main theory file, `theory_graph.thy`, formalizes the axiomatic parts of the eight biform theories in the case study. In the next chapter, we will extend our work to include the transformers of the theories, i.e, the algorithmic parts of the biform theories. The eight theories are represented by locales. Each locale has its own set of parameters (which are introduced by `fixes`) and assumptions (which are introduced by `assumes`). The parameters and assumptions of the locale represent the constants and axioms of the theory that the locale represents. When a locale B extends another locale A, the constants and axioms of A are imported into B.

The eight biform theories are formalized as locales in the following manner:

- BT1, the simple theory of 0 and S, is expressed in Isabelle as:

  ```
  locale th1 =
  ```

```
  fixes
    zero :: "'a"
    and suc :: "'a  ⟹  'a"
  assumes
    "suc n ≠ zero"
    and "suc n = suc m → n = m"
```

Here `'a` is a type variable that represents an abstract type of natural numbers. Each constant in this and the other locales includes `'a` in its type.

- BT2 is BT1 plus + and the two axioms that define +. BT2 is expressed in Isabelle as:

```
locale th2 = th1 +
  fixes
    plus :: "'a  ⟹  'a  ⟹  'a"
  assumes
    plus_zero: "plus n zero = n"
    and plus_suc: "plus n (suc m) = suc ( plus n m)"
```

- BT3 is BT2 plus $*$ and the two axioms that define $*$. BT3 is expressed as:

```
locale th3 = th2 +
  fixes
    mul :: "'a  ⟹  'a  ⟹  'a"
  assumes
    times_zero: "mul n zero = zero"
    and times_suc: "mul n (suc m) = plus (mul n m) n"
```

- BT4 is BT3 plus an additional axiom for Robinson arithmetic. It is expressed in Isabelle as:

  ```
  locale th4 = th3+
    assumes
      "n = zero ∨( ∃m. suc m = n)"
  ```

- BT5 is the complete theory of 0 and S. It is formalized in the theory file as `th5` and extends `th1` with the induction schema for the language of `th1`. Since, we cannot express an axiom schema with an infinite number of instances in Isabelle, we let the user add instances of the induction schema to the locale as they want. The instances should be in the language of `th1`. `th5` keeps changing over time and will always be an approximation of the full formalization of BT5. The current version of `th5` is:

  ```
  locale th5 = th1 +
   assumes
    th5_inst1: "(λp. p zero ∧ (p x → p (suc x)) → (∀x. p x))
    (λn . (n = zero) ∨ (∃m . suc m = n))"
  ```

  with one instance of the induction schema for BT5. The instance is written as a function application `f q`, where `q` is a property of the natural numbers, that beta-reduces to a substitution instance of the induction principle.

- BT6 is the complete theory of 0, S and $+$. It is formalized in the theory file as `th6` and extends `th2` with the induction schema for the language of `th2`. As was done in `th5`, we let the user add instances of the induction schema to the locale as they want.

The current version of `th6` is:

```
locale th6 = th2 + th5
```

with no new instances of the induction schema for BT6.

- BT7 is the complete theory of 0, S and $+$ and $*$. It is formalized in the theory file as `th7` and extends `th3` with the induction schema for the language of `th3`. In `th7` as well, we allow users to add instances of the induction schema to the locale as they want.
  The current version of `th7` is:

  ```
  locale th7 = th3 + th6
  ```

  with no new instances of the induction schema for BT7.

- BT8 is the theory of higher-order Peano arithmetic. We formalize BT8 in two separate ways as two different theories: `th8a` and `th8b`.

  - `th8a`: `th8a` is a direct formalization of BT8. It extends `th1` and has one additional axiom, the induction principle. The locale also defines `plus` and `times` using definite description.

    ```
    locale th8a = th1 +
     assumes
      induction_principle:
      "((p zero ∧ (∀x. p x → p (suc x))) → (∀x. p x))"
      begin
      definition
      plus where
    ```

```
"plus = (THE f. ∀ x y. f x zero = x
∧ f x (suc y) = suc (f x y))"
definition
times where
"times = (THE g. ∀ x y. g x zero = zero
∧ g x (suc y) = plus (g x y) x)"
end
```

– `th8b`: This locale extends `th3` and has the induction principle as an additional axiom. Since `th8b` extends `th3`, it already contains axioms that define `plus` and `times`, unlike `th8a`.

```
locale th8b = th3 +
 assumes p1:
  "⋀ x. ((⋀ x . p zero ∧ ( p x → p (suc x))) → p x"
```

Hence `th8b` does not directly formalize BT8 because it contains the axioms of `plus` and `times`.

5.2 Interpretations

Each locale has an interpretation that creates an instance of it in the background theory of Isabelle. Thus an interpretation is a theory morphism from the locale to the background theory. Locales allow theorems to be proven in an abstract manner using a set of assumptions. Interpretations allow these theorems to be used in other contexts.

An interpretation of a locale includes a white-space separated list of terms, which provide a complete interpretation of the locale parameters. `'a` is interpreted as `nat` in the background theory. The parameters are referred to by order of declarations. The declarations create a list of goals that then need to be proved. A number of theorems can be applied to the interpretation to simplify and prove the goals. The interpretations show that our locales have been consistently formulated. It also enables us to use the results proved in the locales with the type `nat` and other operators over `nat`.

The Interpretations for the locales are given as below:

- Interpretation for `th1`.

```
interpretation int1: th1
"0 :: nat"
"Suc :: nat ⟹ nat"
by unfold_locales auto
```

- Interpretation for `th2`.

```
interpretation int2: th2
"0 :: nat"
"Suc :: nat ⟹ nat"
"(+) :: nat ⟹ nat ⟹ nat"
by unfold_locales auto
```

- Interpretation for `th3`.

```
interpretation inst3: th3
```

```
"0 :: nat"
"Suc :: nat ⟹ nat"
"(+) :: nat ⟹ nat ⟹ nat"
"(*) :: nat ⟹ nat ⟹ nat"
by unfold_locales auto
```

- Interpretation for `th4`.

```
interpretation inst4: th4
"0 :: nat"
"Suc :: nat ⟹ nat"
"(+) :: nat ⟹ nat ⟹ nat"
"(*) :: nat ⟹ nat ⟹ nat"
proof
show "⋀n. n = 0 ∨ (∃m. Suc m = n)"
using not0_implies_Suc by auto
qed
```

In some cases, simply unfolding our locales might not prove the subgoals of our interpretation. In this case we may have to carry out a structured formal proof, where the proof begins with the keyword `proof`, and is concluded with the keyword `qed`. `not0_implies_Suc` is a lemma defined in Theory Nat that can be found in the Isabelle/HOL library.

- Interpretation for `th5`.

```
interpretation inst5: th5
```

```
"0 :: nat"
"Suc :: nat ⟹ nat"
proof
show "⋀x. (0 = 0 ∨ (∃m. Suc m = 0))
∧ (x = 0 ∨ (∃m. Suc m = x) → Suc x = 0
∨ (∃m. Suc m = Suc x)) → (∀x. x = 0 ∨ (∃m. Suc m = x))"
using not0_implies_Suc by auto
qed
```

- Interpretation for `th6`.

```
interpretation inst6: th6
"0 :: nat"
"Suc :: nat ⟹ nat"
"(+) :: nat ⟹ nat ⟹ nat"
by unfold_locales
```

- Interpretation for `th7`.

```
interpretation inst7: th7
"0 :: nat"
"Suc :: nat ⟹ nat"
"(+) :: nat ⟹ nat ⟹ nat"
"(*) :: nat ⟹ nat ⟹ nat"
by unfold_locales
```

- Interpretation for `th8a`.

```
interpretation inst8a: th8a
"0 :: nat"
"Suc :: nat ⟹ nat"
proof
show "⋀p. p 0 ∧ (∀x. p x → p (Suc x)) → (∀x. p x)"
using nat_induct by auto
qed
```

- Interpretation for `th8b`.

```
interpretation inst8b: th8b
"0 :: nat"
"Suc :: nat ⟹ nat"
"(+) :: nat ⟹ nat ⟹ nat"
"(*) :: nat ⟹ nat ⟹ nat"
proof
show "⋀p x. (⋀x. p 0 ∧ (p x → p (Suc x))) ⟹ p x"
using nat_induct by auto
qed
```

5.3 Sublocales

Figure 1.1 shows the theory morphisms that connect the eight theories in the biform theory graph test case. The expression `sublocale B ⊆ A` asserts that there is a theory morphism from the locale `A` to the locale `B`. Thus a sublocale is equivalent to a theory morphism.

Recall that a theory inclusion is strict when the axioms of one locale are already present in another locale. These theory inclusions are proved by unfolding locales.

- `th1` to `th2` is a strict theory inclusion and is expressed in the following manner:

  ```
  sublocale th2 ⊆ th1
  proof
   unfold_locales
  qed
  ```

- `th2` to `th3` is a strict theory inclusion and is expressed in the following manner:

  ```
  sublocale th3 ⊆ th2
  proof
   unfold_locales
  qed
  ```

- `th3` to `th4` is a strict theory inclusion and is expressed in the following manner:

  ```
  sublocale th4 ⊆ th3
  proof
   unfold_locales
  qed
  ```

- `th1` to `th5` is a strict theory inclusion and is expressed in the following manner:

```
sublocale th5 ⊆ th1
proof
 unfold_locales
qed
```

- `th2` to `th6` is a strict theory inclusion and is expressed in the following manner:

  ```
  sublocale th6 ⊆ th2
  proof
   unfold_locales
  qed
  ```

- `th3` to `th7` is a strict theory inclusion and is expressed in the following manner:

  ```
  sublocale th7 ⊆ th3
  proof
   unfold_locales
  qed
  ```

- `th5` to `th6` is a strict theory inclusion and is expressed in the following manner:

  ```
  sublocale th6 ⊆ th5
  proof
   unfold_locales
  qed
  ```

- `th6` to `th7` is a strict theory inclusion and is expressed in the following manner:

  ```
  sublocale th7 ⊆ th6
  proof
   unfold_locales
  qed
  ```

- `th4` to `th7` is a non-strict inclusion which is expressed in the following manner:

  ```
  sublocale th7 ⊆ th4
  proof
   unfold_locales
   show "⋀n. n = zero ∨ (∃m. suc m = n)"
    by (meson th5.axioms(2) th5_axioms th5_axioms_def)
  qed
  ```

- BT7 to BT8, which is a non-strict theory inclusion has two separate parts.

  - There is a non-strict theory inclusion from `th7` to `th8a`. There are a number of lemmas which aid in the proof of the sublocale. `q` is a function in `th8a` which checks if a given function `f` represents +. Similarly, `r` is a function which checks if a given function `g` represents ∗.

    ```
    fun (in th8a) q :: "('a ⇒ 'a ⇒ 'a) ⇒ bool"
    where
    ```

```
"q f = (∀ x y. (f x zero = x ∧ f x (suc y) = suc (f x y)))"


fun (in th8a) r :: "('a ⇒ 'a ⇒ 'a) ⇒ bool"
where
"r g = (∀ x y. g x zero = zero
  ∧ g x (suc y) = plus (g x y) x)"
```

The lemmas `definite_plus` and `definite_times` express the functions `f` and `q` by definite description.

```
lemma (in th8a) definite_plus: "plus = (THE f. q f)"
proof(simp add: plus_def)
qed


lemma (in th8a) definite_times: "times = (THE g. r g)"
proof(simp add: times_def)
qed
```

lemmas `plus_exist_unique` and `times_exist_unique` show the existence and uniqueness of `plus` and `times`.

```
lemma (in th8a) plus_exist_unique: "∃! f. q f"
sorry


lemma (in th8a) times_exist_unique: "∃! g. r g"
sorry
```

We have not completely proved the above two lemmas; however, here is how they could be proved: we require a statement that says there is a finite approximation of `f` that satisfies `q` up to `n` by induction. With the proof of existence of finite approximations, we can define a complete function and show that the function satisfies `q`. This will help us prove the uniqueness and existence of `plus`. A similar method needs to be followed in order to prove the uniqueness and existence of `times`. The lemma `th8a_f` enforces that `f` and `plus` are equivalent.

```
lemma (in th8a) th8a_f: "∀f. q f → plus = f"
proof(simp add: definite_plus plus_def plus_exist_unique)
show "∀f. (∀x. f x zero = x
∧ (∀y. f x (suc y) = suc (f x y))) → (THE f. ∀x. f x zero =
    x
∧ (∀y. f x (suc y) = suc (f x y))) = f "
by (smt (z3) q.elims(3) plus_exist_unique theI_unique)
qed
```

The lemma `th8a_g` enforces that `g` and `times` are equivalent

```
lemma (in th8a) th8a_g: "∀g. r g → times = g"
proof(simp add: definite_times times_def times_exist_unique)
show "∀g. (∀x. g x zero = zero
∧ (∀y. g x (suc y) = local.plus (g x y) x))
→ (THE g. ∀x. g x zero = zero
∧ (∀y. g x (suc y) = local.plus (g x y) x)) = g"
by (smt (z3) Uniq_def r.simps th8a.times_exist_unique
th8a_axioms the1_equality')
```

```
qed
```

The non-strict theory inclusion from `th7` to `th8a` is expressed in the following manner:

```
sublocale th8a ⊆ th7 zero suc "local.plus" "local.times"
proof unfold_locales
show "⋀n. local.plus n zero = n"
using th8a_f plus_exist_unique by auto
show "⋀n m. local.plus n (suc m) = suc (local.plus n m)"
using th8a_f plus_exist_unique by auto
show "⋀n. local.times n zero = zero"
by (metis r.elims(2) times_exist_unique
definite_times the_equality)
show "⋀n m. local.times n (suc m)
= local.plus (local.times n m) n"
by (metis times_exist_unique definite_times th8a.r.simps
th8a_axioms theI)
show "⋀x. (zero = zero ∨ (∃m. suc m = zero))
∧ (x = zero ∨ (∃m. suc m = x) → suc x = zero
∨ (∃m. suc m = suc x)) → (∀x. x = zero ∨ (∃m. suc m = x))"
by (metis q.elims(2) r.elims(2) plus_exist_unique
   times_exist_unique
th8a_g th8a.definite_times th8a_axioms theI')
qed
```

– There is a non-strict theory inclusion from `th7` to `th8b`, which is expressed in the following manner:

```
lemmas (in th8b) th8_b = p1 [where
?p = "λn . (n = zero) ∨ (∃m . suc m = n)" ]


sublocale th8b ⊆ th7
proof unfold_locales
show "⋀x. (zero = zero ∨ (∃m. suc m = zero))
∧ (x = zero ∨ (∃m. suc m = x) → suc x = zero
∨ (∃m. suc m = suc x)) → (∀x. x = zero ∨ (∃m. suc m = x))"
using th8_b by blast
qed
```

The keyword `lemmas` declares a new theorem, `th8_b`, which defines an instance of `p1`. `th8_b` replaces the attribute `p` in `p1`, which is an existing assumption in `th8b`, as can be seen above.

In the formalization of the case study in Isabelle, `th7` does not contain all the instances of the induction schema. hence all sublocales with `th7` have to be redone every time a new instance in added to `th7`.

- There is a strict theory inclusion from `th8a` to `th8b` since all the axioms in `th8a` are included as axioms in `th8b`.

```
sublocale th8b ⊆ th8a
proof unfold_locales
show "⋀p. p zero ∧ (∀x. p x → p (suc x)) → (∀x. p x)"
```

```
using p1 by auto
qed
```

- There is a non-strict theory inclusion from `th8b` to `th8a`. $+$ and $*$ has been defined in `th8a` by definite description. By proving the sublocale, we prove that the $+$ and $*$ in `th8a` is the `plus` and `times` defined axiomatically in `th2` and `th3`, respectively.

  The sublocale is expressed in the following manner:

  ```
  sublocale th8a ⊆ th8b zero suc "local.plus" "local.times"
  proof unfold_locales
  show "⋀p x. (⋀x. p zero ∧ (p x → p (suc x))) ⟹ p x"
  by (metis th8a.axioms(2) th8a_axioms th8a_axioms_def)
  qed
  ```

  Since there are morphisms going both ways, i.e, from `th8a` to `th8b`, and from `th8b` to `th8a`, the two theories are equivalent theories as dicussed in the background chapter.

Since the users are able to add instances of the induction schema of `th5`, `th6` and `th7` to the locales, the proofs involving these locales have to be updated as the instances are added.

So far we have discussed the axiomatic part of the biform theory case study. Considering our theories have taken a slightly different shape and all the theories have been expressed in the logic of Isabelle, we have a new biform theory graph test case in Isabelle that shows the morphisms that connect our reformed theories in the next page.

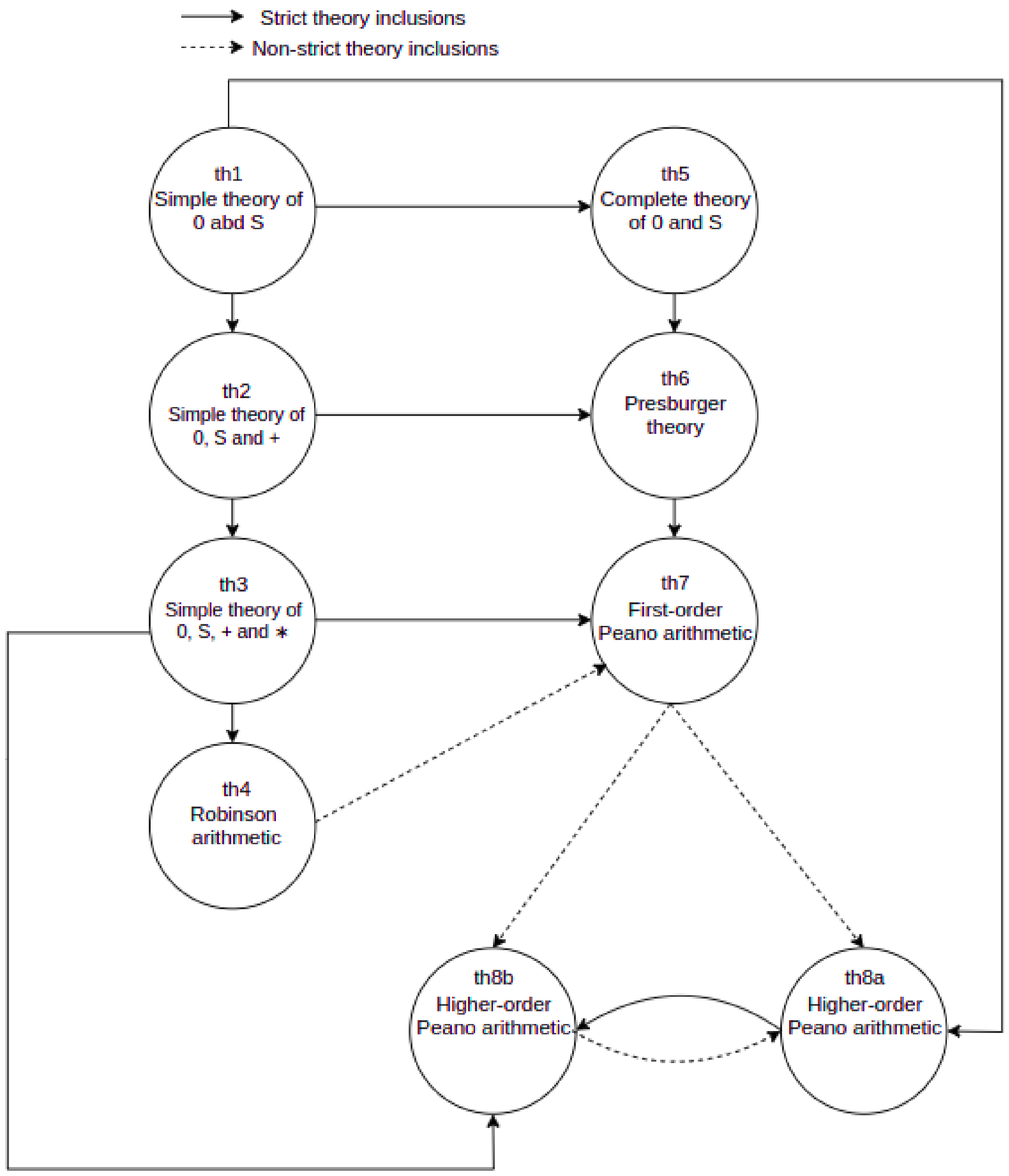


Figure 5.1: Biform theory graph test case in Isabelle

Chapter 6

Biform theory graph

This section discusses the algorithmic aspect of the biform theories, bringing to light what makes them biform. There are a number of transformers in the case study which manipulate the syntax of the biform theories.

6.1 Transformers which manipulate numerals

BT2 and BT3 have algorithms for adding and multiplying natural numbers as binary numerals respectively. We introduce the following inductive type and functions in order to assist in the formalization of these transformers:

- `BinNum` is an inductive type whose members represent binary numerals. It is defined by

  ```
  datatype BinNum = Zero | One | JoinZero BinNum | JoinOne BinNum
  ```

  An example of a binary numeral is `JoinZero(JoinOne(One))` which represents the numeral “110”.

- `len`: This function returns the length of a given binary numeral `(BinNum)`. The function is defined by recursion and pattern matching. The length of both `One` and `Zero` is one, and each application of `JoinOne` or `JoinZero` increases the length by one. For example the expression `len (JoinZero(JoineOne(One)))` would return 3. The function is defined in the following manner:

```
fun len :: "BinNum ⟹ nat"
where
"len Zero = 1"|
"len One = 1"|
"len (JoinZero x) = len x + 1"|
"len (JoinOne x) = len x + 1"
```

- `val`: This function returns a natural number equivalent of a given binary numeral (`BinNum`). The function is defined by recursion and pattern matching in the following manner:

```
fun val :: "BinNum ⟹ nat"
where
"val Zero = 0"|
"val One = 1"|
"val (JoinZero x) = 2 * (val x)"|
"val (JoinOne x) = 2 * (val x) + 1"
```

For instance, `val (JoinZero(One))` returns 2 since “10” is 2 in binary.

6.1.1 Evaluation function for binary numerals

An evaluation function has been implemented for evaluating a binary numeral as a member of the `'a` type in `th2`. It has been defined in the following manner:

```
fun (in th2) evalBinNum :: "BinNum ⟹ 'a"
where
"evalBinNum Zero = zero"|
"evalBinNum One = suc(zero)"|
"evalBinNum (JoinZero x) =
   plus (evalBinNum x) (evalBinNum x)"|
"evalBinNum (JoinOne x) =
   plus (plus (evalBinNum x) (evalBinNum x)) (suc zero)"
```

We use `plus` in the definition since `mul` has not been introduced in `th2`. While the definition of `evalBinNum` has the same form as that of `val`, `val` gives the natural number value for `BinNum` and is used in the interpretations. `evalBinNum`, on the other hand, evaluates `BinNum` as a member of our locale `th2`.

6.1.2 Addition of binary numerals

`addBinNum` is the addition algorithm for binary numerals. The function is defined in the following manner:

```
function addBinNum :: "BinNum ⟹ BinNum ⟹ BinNum"
where
"addBinNum Zero x = x"|
"addBinNum x Zero = x"|
"addBinNum One One = JoinZero One"|
"addBinNum One (JoinZero x) = JoinOne x"|
"addBinNum (JoinZero x) One = JoinOne x"|
"addBinNum One (JoinOne x) =
  JoinZero (addBinNum One x)"|
"addBinNum (JoinOne x) One =
  JoinZero (addBinNum x One)"|
"addBinNum (JoinZero x) (JoinZero y) =
  JoinZero (addBinNum x y)"|
"addBinNum (JoinZero x) (JoinOne y) =
  JoinOne (addBinNum x y)"|
"addBinNum (JoinOne x) (JoinZero y) =
  JoinOne (addBinNum x y)"|
"addBinNum (JoinOne x) (JoinOne y) =
  JoinZero (addBinNum (addBinNum x One) y)"
by pat_completeness auto

termination addBinNum by size_change
```

The termination command sets up the termination goal for a specified function. Isabelle determines that the `addBinNum` function is total using `size_change`, which is a proof method related to recursive definitions. It uses a variant of the size-change principle which has been discussed in Krauss (2009). `pat_completeness` is a proof method from the function package that automates proof of completeness for patterns of datatype constructors.

6.1.3 Multiplication of binary numerals

mulBinNum is the algorithm for the multiplication of two binary numerals. It is structured similar to addBinNum. It is defined in the following manner:

```
function mulBinNum :: "BinNum ⟹ BinNum ⟹ BinNum"
where
"mulBinNum x Zero = Zero"|
"mulBinNum Zero x = Zero"|
"mulBinNum x One = x"|
"mulBinNum One x = x"|
"mulBinNum (JoinZero x) (JoinZero y) =
JoinZero (JoinZero (mulBinNum x y))"|
"mulBinNum (JoinZero x) (JoinOne y) =
addBinNum (JoinZero (mulBinNum x y)) (JoinZero x)"|
"mulBinNum (JoinOne x) (JoinZero y) =
addBinNum (JoinZero (mulBinNum x y)) (JoinZero y)"|
"mulBinNum (JoinOne x) (JoinOne y) =
JoinOne (addBinNum (addBinNum x y) (mulBinNum x y))"
by pat_completeness auto


termination mulBinNum by size_change
```

Here, since all the subgoals express pattern compatibility, pat_completeness automates proof of completeness for mulBinNum. Isabelle determines that the function is total since it terminates using size_change.

6.1.4 Meaning formulas for addition and multiplication

The meaning formulas for `addBinNum` and `mulBinNum` state that these functions correctly implement `plus` and `times` on `BinNum`. The meaning formula for `plus` is proved by strong induction using associativity and commutativity of `plus`. It is defined as:

```
lemma (in th8b) addMeaningFormula:
"evalBinNum (addBinNum x y) = plus (evalBinNum x) (evalBinNum y)"
using plus_assoc and plus_commute
by (induction rule: addBinNum.induct, auto) (metis plus_zero)+
```

It is supported by the following lemmas that prove `plus` has the properties of its definition with the arguments reversed in `th8b`:

```
lemma (in th8b) lemma_plus_zero: "plus zero n = n"
proof(induction n rule: p1)
case (1 x)
then show ?case
by (simp add: plus_zero plus_suc)
qed


lemma (in th8b) lemma_plus_suc: "plus (suc m) n = suc (plus m n)"
proof(induction n rule: p1)
case (1 x)
then show ?case
using plus_zero plus_suc by presburger
qed
```

The following lemmas prove that `plus` is associative and commutative in `th8b`:

```
lemma (in th8b) plus_commute: "plus x y = plus y x"
proof(induction x rule: p1)
case (1 x)
then show ?case
by (simp add: plus_zero lemma_plus_suc lemma_plus_zero plus_suc)
qed


lemma (in th8b) plus_assoc: "plus (plus x y) z = plus x (plus y z)"
proof(induction x rule: p1)
case (1 x)
then show ?case
using lemma_plus_suc lemma_plus_zero by presburger
qed
```

The meaning formula for `times` is proved similarly to that of `plus`, using commutativity and associativity. It is defined in the following manner:

```
lemma (in th8b) mulMeaningFormula:
"evalBinNum (mulBinNum x y) = times (evalBinNum x) (evalBinNum y)"
using times_commute and times_assoc
by (induction rule: mulBinNum.induct)
(metis plus_zero plus_commute lemma_times_zero lemma_times_suc)
```

The meaning formula for `times` is supported by the following lemmas, which prove that `times` can be defined in a recursive manner in `th8b`:

```
lemma (in th8b) lemma_times_zero: "times zero n = zero"
```

```
proof(induct n rule: p1)

case (1 x)

then show ?case

using plus_zero times_zero times_suc by auto

qed


lemma (in th8b) lemma_times_suc:

"times (suc m) n = plus (times m n) n"

proof(induction n rule: p1)

case (1 x)

then show ?case

by (smt (verit) plus_zero plus_assoc plus_commute

plus_suc times_zero times_suc)

qed
```

The following lemmas prove that `times` is associative and commutative:

```
lemma (in th8b) times_commute: "times x y = times y x"

proof(induction x rule: p1)

case (1 x)

then show ?case

using lemma_times_suc lemma_times_zero times_zero

times_suc by presburger

qed


lemma (in th8b) times_assoc:
```

```
"times (times x y) z = times x (times y z)"
proof(induction x rule: p1)
case (1 x)
then show ?case
by(metis plus_zero plus_commute lemma_times_zero
lemma_times_suc times_zero)
qed
```

6.2 Transformers which manipulate expressions

6.2.1 Language of the theories

The file `syntax_operations.thy` contains three datatypes: `Var, Term` and `Form`. These datatypes correspond to the first-order component of the language of the theories.

`Var` is defined by:

```
datatype Var = Var string
```

`Term` is defined by:

```
datatype Term = V Var | Z | S Term |
  Plus Term Term | Times Term Term
```

`Term` is the datatype whose members represent the expressions from the theories. It contains the following constructors:

- `Z` refers to the natural number 0 and is of type `Term.`

- `S` refers to successor. It can take a `Term` and returns a type `Term`.
- `Plus` refers to addition. It can take two `Terms` and returns a `Term`.
- `Times` refers to multiplication. It takes two `Terms` and returns a `Term`.

`Form` is a datatype that enables the creation of formulas using quantifiers, boolean and connectives in propositional calculus. `Form` is defined by:

```
datatype Form = Eq Term Term | Neg Form | Imp Form Form |
                And Form Form | Or Form Form |
                Forall Var Form | Forsome Var Form
```

It contains the following constructors:

- `Eq` refers to $=$ and takes two `Terms` and returns a `Form`.
- `Neg` refers to $\neg$ and takes a `Form` and returns a `Form`.
- `Imp` refers to $\rightarrow$ and takes two `Forms` and returns a `Form`.
- `And` refers to $\wedge$ and takes two `Forms` and returns a `Form`.
- `Or` refers to $\vee$ and take two `Forms` and returns a `Form`.
- `Forall` refers to $\forall$ and takes a `Var` and a `Form` and returns a `Form`.
- `Forsome` refers to $\exists$ and takes a `Var` and a `Form` and returns a `Form`.

6.2.2 Recognizers of the formulas of the theories

BT1, BT2 and BT3 have recognizers of the formulas of their respective theories.

- `isTh1Term` : `th1` is the implementation of BT1, which is a theory of 0 and S. If the term does not contain a `plus` or `times,` then it is in `th1` and the function `isTh1Term` returns a `True`. However if a `plus` or `times` is detected in the term, then the function `isTh1Term` returns a `False`. The function `isTh1Term` is defined in the following manner:

```
fun isTh1Term :: "Term ⟹ bool"
where
"isTh1Term (V v) = True"|
"isTh1Term Z = True" |
"isTh1Term (S n) = isTh1Term n" |
"isTh1Term (Plus s t) = False" |
"isTh1Term (Times s t) = False"
```

- `isTh1Form` :

  `IsTh1Form` checks if a given formula is in `th1`. It is defined in the following manner:

```
fun isTh1Form :: "Form ⟹ bool"
where
"isTh1Form (Eq a b) = (isTh1Term a ∧ isTh1Term b)"|
"isTh1Form (Neg a) = isTh1Form a"|
"isTh1Form (Imp a b) = (isTh1Form a ∧ isTh1Form b)"|
"isTh1Form (Forall a b) = isTh1Form b"|
"isTh1Form (Forsome a b) = isTh1Form b"|
"isTh1Form (And a b) = (isTh1Form a ∧ isTh1Form b)" |
"isTh1Form (Or a b) = (isTh1Form a ∧ isTh1Form b)"
```

The function `IsTh1Form` returns True only if its individual components belong to `th1`. For example the formula `isTh1Form (Imp (Eq (V v) Z) (Eq Z Z))` returns the value `True` since each individual component of the formula, `(Eq (V v) Z)`, `(Eq Z Z)` and `(Imp (Eq (V v) Z) (Eq Z Z))` as a whole are in `th1`.

- `isTh2Term`: If a term does not contain Times, then it is in `th2` and the function `isTh2Term` returns a `True`. If the `Term` contains `Times`, `isTh2Term` returns `False`. The function `isTh2Term` is defined in the following manner:

```
fun isTh2Term :: "Term ⟹ bool"
where
"isTh2Term (V v) = True"|
"isTh2Term Z = True"|
"isTh2Term (S n) = isTh2Term n"|
"isTh2Term (Plus m n) = (isTh2Term m ∧ isTh2Term n)"|
"isTh2Term (Times m n) = False"
```

- `isTh2Form` : This function returns `True` for a formula if its individual components are in `th2`. For example, `isTh2Form (Imp (Eq (V v) (Times Z Z)) (Eq Z Z))` is going to return the value `False`. This is because `Times Z Z` is a term in `th3`. The function `isTh2Form` is defined in the following manner:

```
fun isTh2Form :: "Form ⟹ bool"
```

```
where
"isTh2Form (Eq x y) = (isTh2Term x ∧ isTh2Term y)"|
"isTh2Form (Neg x) = isTh2Form x"|
"isTh2Form (Imp x y) = (isTh2Form x ∧ isTh2Form y)"|
"isTh2Form (Forall x y) = isTh2Form y"|
"isTh2Form (Forsome x y) = isTh2Form y"|
"isTh2Form (And x y) = (isTh2Form x ∧ isTh2Form y)" |
"isTh2Form (Or x y) = (isTh2Form x ∧ isTh2Form y)"
```

- Since the language for the theories in based on `th3`, all formulas and terms are by default in the language of `th3`, and hence we need no recognizers for this theory.

6.2.3 Induction schema generators

BT5, BT6 and BT7 contain generators for the instances of the respective theory's induction schemas. The instances of the induction schema for `th5, th6` and `th7` are generated through one base parent function.

```
fun indSchemaInst :: "Form ⟹ Var ⟹ Form"
where
"indSchemaInst a x = Imp (And (subForm Z x a)
(Imp (Forall x a) (subForm (S (V x)) x a))) (Forall x a)"
```

There are functions specific to each theory which checks if a given formula is in the form of `th5, th6` or `th7`. They have been formalized in Isabelle in the following manner:

```
fun indSchemaInstT5 :: "Form ⟹ Var ⟹ Form"
where
"indSchemaInstT5 a x = (if (isTh1Form a)
then (indSchemaInst a x) else (Eq Z Z))"

fun indSchemaInstT6 :: "Form ⟹ Var ⟹ Form"
where
"indSchemaInstT6 a x = (if (isTh2Form a)
then (indSchemaInst a x) else (Eq Z Z))"

fun indSchemaInstT7 :: "Form ⟹ Var ⟹ Form"
where
"indSchemaInstT7 a x = indSchemaInst a x"
```

We take the help of two functions, `subTerm` and `subForm` to create the induction schemas and be able to instantiate them.

- `subTerm` substitutes a `Term` for the free occurence of a variable in a `Term`. It is defined in the following recursive manner:

```
fun subTerm :: "Term ⟹ Var ⟹ Term ⟹ Term"
where
"subTerm t (Var x) (V (Var y)) =
(if (x = y) then t else V (Var y))"|
"subTerm t (Var x) Z = Z"|
"subTerm t (Var x) (S t1) =
S (subTerm t (Var x) t1)"|
"subTerm t (Var x) (Plus t1 t2) =
Plus (subTerm t (Var x) t1) (subTerm t (Var x) t2)"|
"subTerm t (Var x) (Times t1 t2) =
Times (subTerm t (Var x) t1) (subTerm t (Var x) t2)"
```

- `subForm` substitutes a `Term` for the free occurences of a variable in a `Form`. It is defined in the following recursive manner:

```
fun subForm :: "Term ⟹ Var ⟹ Form ⟹ Form"
where
"subForm t (Var x) (Eq t1 t2) =
Eq (subTerm t (Var x) t1) (subTerm t (Var x) t2)"|
"subForm t (Var x) (Neg a) =
Neg (subForm t (Var x) a)"|
"subForm t (Var x) (Imp a b) =
Imp (subForm t (Var x) a) (subForm t (Var x) b)"|
"subForm t (Var x) (Forall (Var y) a) =
(if y = x then Forall (Var y) a
else Forall (Var y) (subForm t (Var x) a))"|
"subForm t (Var x) (Forsome (Var y) a) =
(if y = x then Forsome (Var y) a
else Forsome (Var y) (subForm t (Var x) a))"|
"subForm t (Var x) (And a b) =
And (subForm t (Var x) a) (subForm t (Var x) b)"|
"subForm t (Var x) (Or a b) =
Or (subForm t (Var x) a) (subForm t (Var x) b)"
```

6.2.4 How do we connect the transformers to the locales?

The evaluation function for `BinNum` (`evalBinNum`) has been created successfully. However, this has not been done for other inductive types. The induction schemas for `th5`, `th6`, and `th7` have been created and instantiated. However, we do not have a way to take an instance and evaluate it so that it becomes an actual axiom, or a theory of the locale. Quotation and evaluation functions are required in order to carry out this operation. Similarly, the recognizers are not directly connected to the locales, it gives us a piece of data that represents a term or a formula in `th1`, `th2`, or `th3`. The transformers cannot be connected to the locales in a general way. The connection has been created for `addBinNum` and `mulBinNum` through the meaning formulas which connect these transformers to `plus` and `times`.

In summary, we have created a model of the syntax of all of the locales but have limited success in connecting the model of the locales to their transformers.

6.3 Decision procedures

The decision procedures for BT5 and BT6 are represented as transformers in BT5 and BT6. However, the formalization of the case study graph of biform theories in Isabelle does not have an implementation for these decision procedures. A decision procedure for Presburger theory, which is BT6 in our case study graph, has already been implemented in Isabelle (Chaieb (2021)). A decision procedure for BT5 could be constructed from the decision procedure for BT6 by restricting the domain. Implementing these two decision procedures would be a large undertaking and is currently out of the scope of the project.

Chapter 7

Related work

The field of formal methods has various techniques and approaches to developing libraries of algebraic theories. We have referred to and studied a lot of this work in order to understand how the process of formalization of biform theories in Isabelle should go about. We split the work related to the thesis into three sections:

- **Formalization of mathematical theories in Isabelle**: This section will primarily look into the various mathematical theories that have been previously formalized in Isabelle.
- **Formalization of biform theories in mathematical systems**: This section digs into previous attempts at the formalization of biform theories.
- **The use of locales in Isabelle**: This section looks into the various tutorials and reports written on the usage of locales. It also reviews some of the introductory material on the official Isabelle website.

7.1 Formalization in Isabelle

As a strong proof assistant, Isabelle has successfully formalized a number of proofs in higher-order logic, first-order logic, and Zermelo-Fraenkel (ZF) set theory. A complete compilation of the proofs can be found on the official Isabelle website.

While formalizing the theory graph (Carette and Farmer, 2017) we come across several common mathematical theories. The formalization of a significant number of mathematical theories has been carried out and documented in Isabelle through the *Archive of Formal proofs*. The Archive of Formal Proofs is an assemblage of proof libraries, examples, and significant scientific developments, all mechanically checked in the theorem prover Isabelle.

Robinson arithmetic, a first-order theory whose signature is that of first-order Peano arithmetic, has been represented in the theory graph as `th4`. It has been formalized in Isabelle as a separate AFP entry in Popescu and Traytel (2020).

Presburger arithmetic is the theory of natural numbers with addition in FOL. The signature of Presburger arithmetic contains the successor, the addition operation and equality. It omits the multiplication operation entirely. The axioms include a schema of induction. Presburger arithmetic is much weaker than Peano arithmetic (discussed below). It includes addition and multiplication operations. Unlike Peano arithmetic, Presburger arithmetic is a decidable theory. A theory is decidable if it is possible to decide, whether a sentence belongs to the theory. Presburger arithmetic has been formalized in Isabelle in Chaieb and Nipkow (2003). The decision algorithm is formulated as a functional program. It makes minimal assumptions and is created in an adaptable way for theorem provers.

First-order Peano arithmetic is a system that consists of a constant 0, a unary function S, and the binary function symbols $+$ and $*$. Full Peano arithmetic cannot be directly formalized in first-order logic because the induction principle involves quantification over predicates, which is not directly expressible in first-order logic.

The archive contains the following formal proofs: "Soundness and Completeness of an Axiomatic System for First-Order Logic" (From, 2021) works on the formalization of the soundness and completeness of an axiomatic system for first-order logic in Isabelle. Paulson (2021) formalizes Gödel's incompleteness theorem using the nominal package in Isabelle for dealing with bound variables (Urban and Kaliszyk, 2012).

IsaFoR/CeTA (Isabelle/HOL Formalization of Rewriting for Certified Tool Assertions) is a project that consists of two parts; a library that comprises of the formalization of concrete techniques and abstract results of rewriting literature in Isabelle/HOL, and CeTA, which is an automatically generated Haskell program for certifying proofs (Sternagel *et al.*, 2012). *IsaFoR* (Isabelle Formalization of Rewriting) supports a number of techniques, including termination of functions via rewriting (Krauss *et al.*, 2011), certified ordered completion (Sternagel and Winkler, 2018), non-termination of rewrite systems and equational reasoning. IsaFoL is a repository that contains several formalizations of logical calculi in Isabelle. The project aims to develop lemma libraries and methodology for formalizing modern research in automated reasoning.

7.2 Formalization of biform theories

The formalization of mathematics is a relatively new method where human beings explain mathematical proofs and definitions to computer systems (Massot, 2021), and it makes sense to dig into the benefits of formalized mathematics. The primary benefit of formalized mathematics is the certainty that a proof is correct. The idea of a biform theory was first introduced as a part of a *Formal Framework for Managing Mathematics* (FFMM) in Farmer and von Mohrenschildt (2003). It was developed as a part of the MathScheme project (Carette *et al.*, 2011) at McMaster University. The goal of FFMM is to combine computer algebra and computer theorem proving into a single system. Biform theories are one of the principle ideas in FFMM, providing a formal context in which deduction and computation can be combined. The formalization of biform theories has been carried out previously in Chiron (Farmer, 2007), Agda (Carette and Farmer, 2017), and CTT_{qe} (Carette and Farmer, 2017).

Chiron is designed as a practical, general-purpose logic for mechanizing mathematics. It includes elements of type theory, a method for handling undefinedness, and a way to reason about the syntax of expressions. It is an exceptionally well-suited logic for formalizing biform theories. Farmer (2007) illustrates how biform theories can be formalized in Chiron. In Ni (2009), a pre-constructed built-in operator is used to implement a data structure for biform theories. This helps biform theories translate semantic definition to code implementation. In order to formalize biform theories efficiently, we need to be able to effectively link axiomatic and algorithmic theories. Hence, if a symbol is manipulated, it should correspond to a semantic function defined axiomatically. Carette *et al.* (2018a) describes a project that can effectively generate a network of biform theories using a methodology that can express, manipulate and manage mathematical knowledge.

A major practical limitation of most frameworks is the development of large libraries of biform theories. Kohlhase *et al.* (2013) introduces *The Universal Machine* that makes it easy to write biform theories using MMT. MMT is a knowledge representation format focusing on modularity and independence of logic. A rule-based rewriting engine that consists of snippets of implementation occuring in the biform MMT theory graph is created. The machine helps convert between a biform theory's axiomatic and algorithmic realm.

Formalizing biform theories directly in Chiron can be verbose and slightly tedious due to Chiron's low-level logic. This lays the foundation for *MathScheme Language* (MSL), a high-level specification language developed on top of Chiron. MSL, which can be seen as high-level syntactic sugar for Chiron, is more convenient for specifying and relating theories when building the library of formalized mathematics. The MathScheme Library is established upon the little theories method in which a network of biform theories is created out of a part of mathematical knowledge. The biform theories are interconnected via theory morphisms. Tran (2011) explains the techniques that have been developed and used to construct the MSL.

A robust theory library goes a long way in making a mechanized mathematical system useful. Abbasi (2009) demonstrates how to generate an extensive theory library for an MMS using the module system Mei and the underlying logic of Chiron. Biform theories are used to represent the theories built in the theory library. Zhang (2009) also uses biform theories to build a library of theory types. This library is based on module systems of typed programming languages and algebraic specification languages, independent of the underlying logic.

Church's type theory is a formal logical language. It includes classical first-order and propositional logic, and is extremely expressive in a practical sense. It is used in most modern applications of type theory. Farmer (2017) discusses whether the little theories method can be used to formalize biform theories in $\mathrm{CTT}_{\mathrm{qe}}$. It is stated here that the "long-range goal is to implement a system for developing *biform theory graphs* utilizing logic equipped with quotation and evaluation". This has been carried out in Carette and Farmer (2017) utilizing $\mathrm{CTT}_{\mathrm{uqe}}$, a version of Church's type theory with undefinedness, quotation, and evaluation. $\mathrm{CTT}_{\mathrm{uqe}}$ supports global reflection. The paper also formalizes the biform theory graph in Agda, a dependently typed programming language. Agda uses the local reflection for formalizing the biform theories. Both methods have been discussed extensively in Chapter 3.

7.3 Review of available literature on locales

Isabelle is a comprehensive system for implementing logical formalisms. During the formalization of the theory graph of eight biform theories, we had to create various recursive functions and transformers. Nipkow (2013) is a thorough document that introduces various concepts that are required to work with Isabelle, including HOL as a functional language and how to write simple inductive proofs and the various proof patterns we could come across. While writing out an Isabelle/Isar document for the first time, the inner and outer syntax can be confusing. The inner syntax in Isabelle consists of types and terms of the logic. The outer syntax is that of Isabelle/Isar theory sources (specifications and proofs). *The Isabelle/Isar Reference Manual* Wenzel and Paulson (2006) is a comprehensive guide to everything Isabelle. It describes all general language elements, including locales and their interpretations.

The modular construct of locales has made them an attractive choice for the formalization of biform theories in Isabelle. Our case study has clearly defined theory hierarchies and Ballarin (2014) discusses locales as a *module system for mathematical theories*. We look at locales as a system that manages theory hierarchies in a theorem prover. Well-defined examples of theories and morphisms are generated through extensions (locales).

The earliest available report on locales is (Kammüller *et al.*, 1999) where the process of local assumptions and definitions along with sectioning in theorem provers are supported through locales. The report works as an early tutorial explaining rules, definitions, scope, and instances. The current official tutorial for locales on the Isabelle page has come a long way since both design and implementation of locales have evolved considerably since Kammüller and Wenzel released the report (Kammüller *et al.*, 1999). Ballarin (2010) covers all significant facilities of locales, including the use of locales and their interpretations in theories and proofs. Locales also need to be used in contexts where proofs need to be created with regular usage. Ballarin (2006) explains how locales may be seen as detached proof contexts and allow reuse of specifications. The paper also serves as a report to better understand locales, as acknowledged by the author.

Theory morphisms are often considered an alternative to axiomatic type classes to allow theorems to be reused across families of types. They have been implemented in theorem provers such as IMPS (Farmer *et al.*, 1993) and PVS (Owre *et al.*, 1992). Isabelle uses locales as a lightweight implementation of theory morphisms.

Chapter 8

Conclusion and future work

The main aim of the work done was to express and formalize the case study of eight biform theories in Carette and Farmer (2017) in Isabelle/HOL. The case study includes a variety of valuable transformers. Transformers represent syntax-manipulating operations such as inference and computation rules. The case study has been previously formalized in Agda using local reflection and in CTT_{uqe} using global reflection. This thesis partially formalizes the case study in Isabelle, an interactive theorem prover. Since transformers are algorithms that manipulate expressions, the meaning formulas of biform theory rules can only be directly formalized in a logic with support for reasoning about the syntax of expressions. While traditional logic does not offer this kind of support, Isabelle is relatively well suited to formalize biform theories. While carrying out the formalization, we also draw conclusions on whether we can express theory morphisms and induction schemas for theories in Isabelle adequately.

8.1 Conclusion of contribution

The conclusions of the contribution to the thesis have been laid out based on the research questions that have been asked in Chapter 4.

- **RQ1: Can abstract axiomatic theories be formalized in Isabelle?**
  Axiomatization is a general language element in Isabelle, it introduces several constants simultaneously. Locales have helped formalize our abstract axiomatic theories as separate individual theories in Isabelle. Our formalization starts with this step. The formalization of the axiomatic theories in locales works smoothly when the number of axioms is finite. However, with an infinite number of axioms, the formalization becomes challenging. The theory of higher-order peano arithmetic has been formalized in Isabelle in two separate ways: `th8a` and `th8b`. `th8b` is simpler to formalize, but it is not a direct formalization of BT8. `th8a`, on the other hand, is a direct formalization of BT8. It uses definite description to define `plus` and `times`. Extra work is needed to show that these definitions define the real `plus` and `times`.

- **RQ2: Can SBMAs be formalized in Isabelle?**
  SBMAs can be expressed in a straightforward manner. We have defined transformers that manipulate expressions in a mathematically meaningful way. These transformers are our SBMAs. The functions `addBinNum`and `mulBinNum` are SBMAs since they apply arithmetic operations to binary numerals. However, we have not been able to completely express the transformers for all languages of the theory graph due to the lack of a reflection infrastructure for quotation and evaluation.

- **RQ3: Can decision procedures for BT5 and BT6 be formalized in Isabelle?**
  Yes, the decision procedure for BT6 has already been implemented in Isabelle Chaieb and Nipkow (2003). Thus, the decision procedure for BT5 can be implemented as well by restricting the domain of the decision procedure for BT6.

  We have successfully defined functions that create instances of the induction schema. BT5 and BT6 are decidable theories. While BT5 and BT6 are also complete, BT7 is neither complete nor decidable. An induction schema instance has been created with the help of a number of functions and the recursive datatype `Var, Term` and `Form`. This helps us conclude that induction schemas for complete and decidable theories can be expressed adequately in Isabelle. However, the induction schemas cannot be used in BT5, BT6 and BT7

- **RQ4: Can theory morphisms be formalized in Isabelle?**
  The sublocales between the theories are a strong representation of theory morphisms. Hence the theory morphisms for the case study graph can be expressed adequately in Isabelle.

8.2 Recommended work

It is recommended that the following work be carried out for the future of the biform theory graph case study and Isabelle:

- Create a reflection infrastructure with quotation, that translates expressions of the locales to members of `Term` and `Form`, and evaluation, that translates members of `Term` and `Form` to expressions of the locale.

- Provide a way to create axiom schemas with an infinite number of axioms in locales in Isabelle. This will be useful for fully formalizing BT5, BT6 and BT7 in the biform theory graph case study.

- Add more theories to theory graph of natural number arithmetic:
  More biform theories could be added to the case study to investigate Isabelle's formalization capabilities. Skolem arithmetic, the complete theory of 0, S, and $*$, which has a complex axiomatization, can be added to the theory graph.

Appendix A

Formalization in Isabelle

Here are the theory files written in the Isabelle/HOL environment to formalize the test case of eight biform theories.

A.1 BinNum.thy

```
theory BinNum
imports Main
begin

datatype BinNum = Zero | One | JoinZero BinNum | JoinOne BinNum

fun len :: "BinNum ⟹ nat"
where
"len Zero = 1"|
"len One = 1"|
"len (JoinZero x) = len x + 1"|
"len (JoinOne x) = len x + 1"

fun val :: "BinNum ⟹ nat"
where
"val Zero = 0"|
"val One = 1"|
"val (JoinZero x) = 2*(val x)"|
"val (JoinOne x) = 2*(val x) + 1"

function addBinNum :: "BinNum ⟹ BinNum ⟹ BinNum"
where
"addBinNum Zero x = x"|
"addBinNum x Zero = x"|
"addBinNum One One = JoinZero One"|
"addBinNum One (JoinZero x) = JoinOne x"|
"addBinNum(JoinZero x) One = JoinOne x"|
"addBinNum One (JoinOne x) = JoinZero(addBinNum One x)"|
"addBinNum(JoinOne x) One = JoinZero(addBinNum x One)"|
"addBinNum(JoinZero x) (JoinZero y) = JoinZero(addBinNum x y)"|
"addBinNum(JoinZero x) (JoinOne y) = JoinOne(addBinNum x y)"|
"addBinNum(JoinOne x) (JoinZero y) = JoinOne(addBinNum x y)"|
"addBinNum(JoinOne x) (JoinOne y) =
 JoinZero(addBinNum(addBinNum x One) y)"
by pat_completeness auto

termination addBinNum
by size_change

function mulBinNum :: "BinNum ⟹ BinNum ⟹ BinNum"
where
"mulBinNum x Zero = Zero"|
"mulBinNum Zero x = Zero"|
"mulBinNum x One = x"|
"mulBinNum One x = x"|
"mulBinNum (JoinZero x) (JoinZero y) =
       JoinZero (JoinZero (mulBinNum x y))"|
"mulBinNum (JoinZero x) (JoinOne y) =
       addBinNum (JoinZero (mulBinNum x y)) (JoinZero x)"|
"mulBinNum (JoinOne x) (JoinZero y) =
       addBinNum (JoinZero (mulBinNum x y)) (JoinZero y)"|
"mulBinNum (JoinOne x) (JoinOne y) =
       JoinOne (addBinNum (addBinNum x y) (mulBinNum x y))"
by pat_completeness auto

termination mulBinNum
by size_change

end
```

A.2 Syntax operations

```
theory syntax_operations
imports Main
begin

datatype Var = Var string
datatype Term = V Var | Z | S Term |
       Plus Term Term | Times Term Term
datatype Form = Eq Term Term | Neg Form | Imp Form Form |
       And Form Form | Or Form Form | Forall Var Form |
       Forsome Var Form

fun isTh1Term :: "Term ⟹ bool"
where
"isTh1Term (V v) = True"|
"isTh1Term Z = True" |
"isTh1Term (S n) = isTh1Term n" |
"isTh1Term (Plus s t) = False" |
"isTh1Term (Times s t) = False"

fun isTh2Term :: "Term ⟹ bool"
where
"isTh2Term (V v) = True"|
"isTh2Term Z = True"|
"isTh2Term (S n) = isTh2Term n"|
"isTh2Term (Plus m n) = (isTh2Term m ∧ isTh2Term n)"|
"isTh2Term (Times m n) = False"

fun isTh1Form :: "Form ⟹ bool"
where
"isTh1Form (Eq a b) = (isTh1Term a ∧ isTh1Term b)"|
"isTh1Form (Neg a) = isTh1Form a"|
"isTh1Form (Imp a b) = (isTh1Form a ∧ isTh1Form b)"|
"isTh1Form (Forall a b) = isTh1Form b"|
"isTh1Form (Forsome a b) = isTh1Form b"|
"isTh1Form (And a b) = (isTh1Form a ∧ isTh1Form b)" |
"isTh1Form (Or a b) = (isTh1Form a ∧ isTh1Form b)"

fun isTh2Form :: "Form ⟹ bool"
where
"isTh2Form (Eq a b) = (isTh2Term a ∧ isTh2Term b)"|
"isTh2Form (Neg a) = isTh2Form a"|
"isTh2Form (Imp a b) = (isTh2Form a ∧ isTh2Form b)"|
"isTh2Form (Forall a b) = isTh2Form b"|
"isTh2Form (Forsome a b) = isTh2Form b"|
"isTh2Form (And a b) = (isTh2Form a ∧ isTh2Form b)" |
"isTh2Form (Or a b) = (isTh2Form a ∧ isTh2Form b)"

fun subTerm :: "Term ⟹ Var ⟹ Term ⟹ Term"
where
"subTerm t (Var x) (V (Var y)) =
       (if (x = y) then t else V (Var y))"|
"subTerm t (Var x) Z = Z"|
"subTerm t (Var x) (S t1) = S (subTerm t (Var x) t1)"|
"subTerm t (Var x) (Plus t1 t2) =
       Plus (subTerm t (Var x) t1) (subTerm t (Var x) t2)"|
"subTerm t (Var x) (Times t1 t2) =
       Times (subTerm t (Var x) t1) (subTerm t (Var x) t2)"

fun subForm :: "Term ⟹ Var ⟹ Form ⟹ Form"
where
"subForm t (Var x) (Eq t1 t2) =
       Eq (subTerm t (Var x) t1) (subTerm t (Var x) t2)"|
"subForm t (Var x) (Neg a) = Neg (subForm t (Var x) a)"|
"subForm t (Var x) (Imp a b) =
       Imp (subForm t (Var x) a) (subForm t (Var x) b)"|
"subForm t (Var x) (Forall (Var y) a) =
       (if y = x then Forall (Var y) a else Forall (Var y) (subForm
          t (Var x) a))"|
"subForm t (Var x) (Forsome (Var y) a) =
       (if y = x then Forsome (Var y) a
       else Forsome (Var y) (subForm t (Var x) a))"|
"subForm t (Var x) (And a b) =
       And (subForm t (Var x) a) (subForm t (Var x) b)"|
"subForm t (Var x) (Or a b) =
       Or (subForm t (Var x) a) (subForm t (Var x) b)"

fun indSchemaInst :: "Form ⟹ Var ⟹ Form"
where
       "indSchemaInst a v =
       Imp (And (subForm Z v a) (Imp (Forall v a)
       (subForm (S (V v)) v a))) (Forall v a)"

fun indSchemaInstT5 :: "Form ⟹ Var ⟹ Form"
where "indSchemaInstT5 a v =
       (if (isT1Form a) then (indSchemaInst a v) else (Eq Z Z))"
(* check if a is th5 form *)

fun indSchemaInstT6 :: "Form ⟹ Var ⟹ Form"
where "indSchemaInstT6 a v =
        (if (isT2Form a) then (indSchemaInst a v) else (Eq Z Z))"
(* check if a is th6 form*)

fun indSchemaInstT7 :: "Form ⟹ Var ⟹ Form"
where "indSchemaInstT7 a v = indSchemaInst a v"

end
```

A.3 Theory graph

```
theory theory_graph
imports Main BinNum syntax_operations
begin

locale th1 =
fixes
zero :: "'a"
and suc :: "'a ⇒ 'a"
assumes
"suc n ≠ zero"
and "suc n = suc m ⟶ n = m"

locale th2 = th1 +
fixes
plus :: "'a ⇒ 'a ⇒ 'a"
assumes
plus_zero: "plus n zero = n"
and plus_suc: "plus n (suc m) = suc ( plus n m)"

fun (in th2) evalBinNum :: "BinNum ⇒ 'a"
where
"evalBinNum Zero = zero"|
"evalBinNum One = suc(zero)"|
"evalBinNum (JoinZero x) = plus (evalBinNum x) (evalBinNum x)"|
"evalBinNum (JoinOne x) =
       plus (plus (evalBinNum x) (evalBinNum x)) (suc zero)"
```

```
locale th3 = th2 +
fixes
times :: "'a ⇒ 'a ⇒ 'a"
assumes
times_zero: "times n zero = zero"
and times_suc: "times n (suc m) = plus (times n m) n"

locale th4 = th3 +
assumes
"n = zero ∨ (∃m . suc m = n)"

locale th5 = th1 +
assumes
th5_inst1:
       "(λp. p zero ∧ (p x ⟶ p (suc x))
       ⟶ (∀x. p x)) (λn . (n = zero) ∨ (∃m . suc m = n))"

locale th6 = th2 + th5

locale th7 = th3 + th6

locale th8a = th1 +
assumes
induction_principle:
       "((p zero ∧ (∀x. p x ⟶ p (suc x))) ⟶ (∀x. p x))"
begin
definition
plus where
       "plus = (THE f. ∀ x y. f x zero = x
       ∧ f x (suc y) = suc (f x y))"
definition
times where
       "times = (THE g. ∀ x y. g x zero = zero
       ∧ g x (suc y) = plus (g x y) x)"
end

fun (in th8a) q :: "('a ⇒ 'a ⇒ 'a) ⇒ bool"
where
"q f = (∀ x y. (f x zero = x ∧ f x (suc y) = suc (f x y)))"

fun (in th8a) r :: "('a ⇒ 'a ⇒ 'a) ⇒ bool"
where
```

```
"r g = (∀ x y. g x zero = zero ∧ g x (suc y) = plus (g x y) x)"

locale th8b = th3 +
assumes
p1: "⋀x. ((⋀x . p zero ∧ ( p x ⟶ p (suc x)))) ⟹ p x"


interpretation int1: th1
"0 :: nat"
"Suc :: nat ⇒ nat"
by unfold_locales auto

interpretation int2: th2
"0 :: nat"
"Suc :: nat ⇒ nat"
"(+) :: nat ⇒ nat ⇒ nat"
by unfold_locales auto

interpretation inst3: th3
"0 :: nat"
"Suc :: nat ⇒ nat"
"(+) :: nat ⇒ nat ⇒ nat"
"(*) :: nat ⇒ nat ⇒ nat"
by unfold_locales auto

interpretation inst4: th4
"0 :: nat"
"Suc :: nat ⇒ nat"
"(+) :: nat ⇒ nat ⇒ nat"
"(*) :: nat ⇒ nat ⇒ nat"
proof
show "∧n. n = 0 ∨ (∃m. Suc m = n)"
using not0_implies_Suc by auto
qed

interpretation inst5: th5
"0 :: nat"
"Suc :: nat ⇒ nat"
proof
show "∧x. (0 = 0 ∨ (∃m. Suc m = 0))
∧ (x = 0 ∨ (∃m. Suc m = x) ⟶ Suc x = 0
∨ (∃ m. Suc m = Suc x)) ⟶ (∀ x. x = 0 ∨ (∃m. Suc m = x))"
using not0_implies_Suc by auto
```

```
qed

interpretation inst6: th6
"0 :: nat"
"Suc :: nat ⇒ nat"
"(+) :: nat ⇒ nat ⇒ nat"
by unfold_locales auto

interpretation inst7: th7
"0 :: nat"
"Suc :: nat ⇒ nat"
"(+) :: nat ⇒ nat ⇒ nat"
"(*) :: nat ⇒ nat ⇒ nat"
by unfold_locales auto

interpretation inst8a: th8a
"0 :: nat"
"Suc :: nat ⇒ nat"
proof
show "∧p. p 0 ∧ (∀x. p x ⟶ p (Suc x)) ⟶ (∀x. p x)"
using nat_induct by auto
qed

interpretation inst8b: th8b
"0 :: nat"
"Suc :: nat ⇒ nat"
"(+) :: nat ⇒ nat ⇒ nat"
"(*) :: nat ⇒ nat ⇒ nat"
proof
show "∧p x. (∧x. p 0 ∧ (p x ⟶ p (Suc x))) ⟹ p x"
using nat_induct by auto
qed

lemma (in th8b) th8_b_ind:
  "∀p.((p zero ∧ (∀x. p x ⟶ p (suc x))) ⟶ (∀x. p x))"
using p1 by auto

lemma (in th8b) lemma_plus_zero: "plus zero n = n"
proof(induction n rule: p1)
case (1 x)
then show ?case
by (simp add: plus_zero plus_suc)
qed
```

```
lemma (in th8b) lemma_plus_suc:
 "plus (suc m) n = suc (plus m n)"
proof(induction n rule: p1)
case (1 x)
then show ?case
using plus_zero plus_suc by presburger
qed

lemma (in th8b) plus_commute: "plus x y = plus y x"
proof(induction x rule: p1)
case (1 x)
then show ?case
by (simp add: plus_zero lemma_plus_suc lemma_plus_zero plus_suc)
qed

lemma (in th8b) plus_assoc:
  "plus (plus x y) z = plus x (plus y z)"
proof(induction x rule: p1)
case (1 x)
then show ?case
using lemma_plus_suc lemma_plus_zero by presburger
qed

lemma (in th8b) addMeaningFormula:
  "evalBinNum (addBinNum x y)
    = plus (evalBinNum x) (evalBinNum y)"
using plus_assoc and plus_commute
by (induction rule: addBinNum.induct, auto) (metis plus_zero)+

lemma (in th8b) lemma_times_zero: "times zero n = zero"
proof(induct n rule: p1)
case (1 x)
then show ?case
using plus_zero times_zero times_suc by auto
qed

lemma (in th8b) lemma_times_suc:
 "times (suc m) n = plus (times m n) n"
proof(induction n rule: p1)
case (1 x)
then show ?case
by (smt (verit) plus_zero plus_assoc
 plus_commute plus_suc times_zero times_suc)
qed

lemma (in th8b) times_commute: "times x y = times y x"
proof(induction x rule: p1)
case (1 x)
then show ?case
using lemma_times_suc lemma_times_zero
  times_zero times_suc by presburger
qed

lemma (in th8b) times_assoc:
  "times (times x y) z = times x (times y z)"
proof(induction x rule: p1)
case (1 x)
then show ?case
by(metis plus_zero plus_commute
  lemma_times_zero lemma_times_suc times_zero)
qed

lemma (in th8b) mulMeaningFormula:
  "evalBinNum (mulBinNum x y)
    = times (evalBinNum x) (evalBinNum y)"
using times_commute and times_assoc
  by (induction rule: mulBinNum.induct) (metis plus_zero
  plus_commute lemma_times_zero lemma_times_suc)

sublocale th2 ⊆ th1
proof unfold_locales
qed

sublocale th5 ⊆ th1
proof unfold_locales
qed

sublocale th3 ⊆ th2
proof unfold_locales
qed

sublocale th6 ⊆ th2
proof unfold_locales
qed
```

```
sublocale th4 ⊆ th3
proof unfold_locales
qed

sublocale th7 ⊆ th3
proof unfold_locales
qed

sublocale th6 ⊆ th5
proof unfold_locales
qed

sublocale th7 ⊆ th6
proof unfold_locales
qed

sublocale th7 ⊆ th4
proof unfold_locales
show "⋀n. n = zero ∨ (∃m. suc m = n)"
by (meson th5.axioms(2) th5_axioms th5_axioms_def)
qed

lemma (in th8a) definite_plus: "plus = (THE f. q f)"
proof(simp add: plus_def)
qed

lemma (in th8a) definite_times: "times = (THE g. r g)"
proof(simp add: times_def)
qed

lemma (in th8a) plus_exist_unique: "∃! f. q f"
sorry

lemma (in th8a) times_exist_unique: "∃! g. r g"
sorry

lemma (in th8a) th8a_f: "∀f. q f ⟶ plus = f"
proof(simp add: definite_plus plus_def plus_exist_unique)
show "∀f. (∀x. f x zero = x ∧ (∀y. f x (suc y) = suc (f x y)))
⟶ (THE f. ∀x. f x zero = x
∧ (∀y. f x (suc y) = suc (f x y))) = f "
by (smt (z3) q.elims(3) plus_exist_unique theI_unique)
qed
```

```
lemma (in th8a) th8a_g: "∀g. r g ⟶ times = g"
proof(simp add: definite_times times_def times_exist_unique)
show "∀g. (∀x. g x zero = zero
∧ (∀y. g x (suc y) = local.plus (g x y) x))
⟶ (THE g. ∀x. g x zero = zero
∧ (∀y. g x (suc y) = local.plus (g x y) x)) = g"
by (smt (z3) Uniq_def r.simps th8a.times_exist_unique
th8a_axioms the1_equality')
qed

sublocale th8a ⊆ th7 zero suc "local.plus" "local.times"
proof unfold_locales
show "⋀n. local.plus n zero = n"
using th8a_f plus_exist_unique by auto
show "⋀n m. local.plus n (suc m) = suc (local.plus n m)"
using th8a_f plus_exist_unique by auto
show "⋀n. local.times n zero = zero"
by (metis r.elims(2) times_exist_unique definite_times the_equality)
show "⋀n m. local.times n (suc m)
= local.plus (local.times n m) n"
by (metis times_exist_unique definite_times th8a.r.simps
th8a_axioms theI)
show "⋀x. (zero = zero ∨ (∃m. suc m = zero))
∧ (x = zero ∨ (∃m. suc m = x) ⟶ suc x = zero
∨ (∃m. suc m = suc x)) ⟶ (∀x. x = zero ∨ (∃m. suc m = x))"
by (metis q.elims(2) r.elims(2) plus_exist_unique times_exist_unique
th8a_g th8a.definite_times th8a_axioms theI')
qed

lemmas (in th8b) th8_b = p1 [where ?p = "λn . (n = zero) ∨ (∃m .
   suc m = n)" ]

sublocale th8b ⊆ th7
proof unfold_locales
show "⋀x. (zero = zero ∨ (∃m. suc m = zero))
∧ (x = zero ∨ (∃m. suc m = x) ⟶ suc x = zero
∨ (∃m. suc m = suc x)) ⟶ (∀x. x = zero ∨ (∃m. suc m = x))"
using th8_b by blast
qed

sublocale th8a ⊆ th8b zero suc "local.plus" "local.times"
proof unfold_locales
```

```
show "∧p x. (∧x. p zero ∧ (p x ⟶ p (suc x))) ⟹ p x"
by (metis th8a.axioms(2) th8a_axioms th8a_axioms_def)
qed

sublocale th8b ⊆ th8a
proof unfold_locales
show "⋀p. p zero ∧ (∀x. p x ⟶ p (suc x)) ⟶ (∀x. p x)"
using p1 by auto
qed

end
```

Bibliography

Abbasi, M. (2009). *Development of a portion of a theory library for mechanized mathematics system*. Master's thesis, McMaster University.

Andrews, P. B., Bishop, M., Issar, S., Nesmith, D., Pfenning, F., and Xi, H. (1996). TPS: A theorem-proving system for classical type theory. *J. Autom. Reason.*, **16**(3), 321–353.

Anonymous (1994). The QED manifesto. In A. Bundy, editor, *Automated Deduction - CADE-12, 12th International Conference on Automated Deduction, Nancy, France, June 26 - July 1, 1994, Proceedings*, volume 814 of *Lecture Notes in Computer Science*, pages 238–251. Springer.

Ballarin, C. (2006). Interpretation of locales in Isabelle: Managing dependencies between locales.

Ballarin, C. (2010). Tutorial to locales and locale interpretation. pages 123–140. Universidad de La Rioja.

Ballarin, C. (2014). Locales: A module system for mathematical theories. *J. Autom. Reason.*, **52**(2), 123–153.

Carette, J. and Farmer, W. M. (2017). Formalizing Mathematical Knowledge as a Biform Theory Graph: A case study. *CoRR*, **abs/1704.02253**.

Carette, J., Farmer, W. M., and O'Connor, R. (2011). MathScheme: Project description. In J. H. Davenport, W. M. Farmer, J. Urban, and F. Rabe, editors, *Intelligent Computer Mathematics - 18th Symposium, Calculemus 2011, and 10th International Conference, MKM 2011, Bertinoro, Italy, July 18-23, 2011. Proceedings*, volume 6824 of *Lecture Notes in Computer Science*, pages 287–288. Springer.

Carette, J., Farmer, W. M., and Sharoda, Y. (2018a). Biform theories: Project description. In F. Rabe, W. M. Farmer, G. O. Passmore, and A. Youssef, editors, *Intelligent Computer Mathematics - 11th International Conference, CICM 2018, Hagenberg, Austria, August 13-17, 2018, Proceedings*, volume 11006 of *Lecture Notes in Computer Science*, pages 76–86. Springer.

Carette, J., Farmer, W. M., and Laskowski, P. (2018b). HOL light QE. volume abs/1802.00405.

Chaieb, A. (2021). Hol/presburger.thy.

Chaieb, A. and Nipkow, T. (2003). Generic proof synthesis for Presburger arithmetic. Technical report.

Farmer, W. M. (2007). Biform theories in Chiron. In M. Kauers, M. Kerber, R. Miner, and W. Windsteiger, editors, *Towards Mechanized Mathematical Assistants, 14th Symposium, Calculemus 2007, 6th International Conference,* *MKM 2007, Hagenberg, Austria, June 27-30, 2007, Proceedings*, volume 4573 of *Lecture Notes in Computer Science*, pages 66–79. Springer.

Farmer, W. M. (2008). The seven virtues of simple type theory. *Journal of Applied Logic*, **6**(3), 267–286.

Farmer, W. M. (2013). The formalization of syntax-based mathematical algorithms using quotation and evaluation. *CoRR*, **abs/1305.6052**.

Farmer, W. M. (2014). Meaning formulas for syntax-based mathematical algorithms. In T. Kutsia and A. Voronkov, editors, *6th International Symposium on Symbolic Computation in Software Science, SCSS 2014, Gammarth, La Marsa, Tunisia, December 7-8, 2014*, volume 30 of *EPiC Series in Computing*, pages 10–11. EasyChair.

Farmer, W. M. (2016). Incorporating quotation and evaluation into Church's Type Theory: Syntax and semantics. In M. Kohlhase, M. Johansson, B. R. Miller, L. de Moura, and F. W. Tompa, editors, *Intelligent Computer Mathematics - 9th International Conference, CICM 2016, Bialystok, Poland, July 25-29, 2016, Proceedings*, volume 9791 of *Lecture Notes in Computer Science*, pages 83–98. Springer.

Farmer, W. M. (2017). Theory morphisms in Church's Type Theory with Quotation and Evaluation. In H. Geuvers, M. England, O. Hasan, F. Rabe, and O. Teschke, editors, *Intelligent Computer Mathematics - 10th International Conference, CICM 2017, Edinburgh, UK, July 17-21, 2017, Proceedings*, volume 10383 of *Lecture Notes in Computer Science*, pages 147–162. Springer.

Farmer, W. M. and von Mohrenschildt, M. (2003). An overview of a formal framework for managing mathematics. *Ann. Math. Artif. Intell.*, **38**(1-3), 165–191.

Farmer, W. M., Guttman, J. D., and Thayer, F. J. (1993). IMPS: an interactive mathematical proof system. *J. Autom. Reason.*, **11**(2), 213–248.

From, A. H. (2021). Soundness and completeness of an axiomatic system for first-order logic. *Arch. Formal Proofs*, **2021**.

Harrison, J. (2009). HOL light: An overview. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66. Springer.

Kammüller, F., Wenzel, M., and Paulson, L. C. (1999). Locales - A sectioning concept for Isabelle. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin-Mohring, and L. 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 149–166. Springer.

Kohlhase, M. (2014). Mathematical knowledge management and information retrieval: Transcending the one-brain-barrier. **1226**, 8.

Kohlhase, M., Mance, F., and Rabe, F. (2013). A universal machine for biform theory graphs. In J. Carette, D. Aspinall, C. Lange, P. Sojka, and

W. Windsteiger, editors, *Intelligent Computer Mathematics - MKM, Calculemus, DML, and Systems and Projects 2013, Held as Part of CICM 2013, Bath, UK, July 8-12, 2013. Proceedings*, volume 7961 of *Lecture Notes in Computer Science*, pages 82–97. Springer.

Krauss, A. (2009). *Automating recursive definitions and termination proofs in higher-order logic*. Ph.D. thesis, Technische Universität München.

Krauss, A., Sternagel, C., Thiemann, R., Fuhs, C., and Giesl, J. (2011). Termination of Isabelle functions via termination of rewriting. In M. C. J. D. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*, volume 6898 of *Lecture Notes in Computer Science*, pages 152–167. Springer.

Massot, P. (2021). Why formalize mathematics?

Ni, H. (2009). *Chiron: Mechanizing Mathematics in OCaml*. Master's thesis, McMaster University.

Nipkow, T. (2013). Programming and proving in Isabelle/HOL. In *Technical report, University of Cambridge*.

Oliveira, M., Cavalcanti, A., and Woodcock, J. (2006). Unifying theories in proofpower-z. In *International Symposium on Unifying Theories of Programming*, pages 123–140. Springer.

Owre, S., Rushby, J. M., and Shankar, N. (1992). PVS: A prototype verification system. In D. Kapur, editor, *Automated Deduction - CADE-11, 11th*

International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer.

Paulson, L. C. (2021). A mechanised proof of gödel's incompleteness theorems using nominal Isabelle. *CoRR*, **abs/2104.13792**.

Popescu, A. and Traytel, D. (2020). Robinson arithmetic. *Arch. Formal Proofs*, **2020**.

Slind, K. and Norrish, M. (2008). A brief overview of HOL4. In O. A. Mohamed, C. A. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer.

Sternagel, C. and Winkler, S. (2018). Certified ordered completion. *CoRR*, **abs/1805.10090**.

Sternagel, C., Thiemann, R., Winkler, S., and Zankl, H. (2012). CeTA - A tool for certified termination analysis. *CoRR*, **abs/1208.1591**.

Tran, M. Q. (2011). *Algebraic Constructions Applied to Theories.* Master's thesis, McMaster University.

Urban, C. and Kaliszyk, C. (2012). General bindings and alpha-equivalence in nominal Isabelle. *Log. Methods Comput. Sci.*, **8**(2).

Wenzel, M. and Paulson, L. C. (2006). Isabelle/Isar.

Wenzel, M., Paulson, L. C., and Nipkow, T. (2008). The Isabelle framework. In O. A. Mohamed, C. A. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 33–38. Springer.

Yushkovskiy, A. and Tripakis, S. (2018). Comparison of two theorem provers: Isabelle/HOL and Coq. *CoRR*, **abs/1808.09701**.

Zhang, H. (2009). *A Language and a Library of Algebraic Theory-types.* Master's thesis, McMaster University.