Formalizing Mathematical Knowledge as a Biform Theory Graph: A Case Study*

Jacques Carette and William M. Farmer

Computing and Software, McMaster University, Canada
http://www.cas.mcmaster.ca/~carette
http://imps.mcmaster.ca/wmfarmer

May 8, 2020

Abstract. A *biform theory* is a combination of an axiomatic theory and an algorithmic theory that supports the integration of reasoning and computation. These are ideal for formalizing algorithms that manipulate mathematical expressions. A *theory graph* is a network of *theories* connected by meaning-preserving *theory morphisms* that map the formulas of one theory to the formulas of another theory. Theory graphs are in turn well suited for formalizing mathematical knowledge at the most convenient level of abstraction using the most convenient vocabulary. We are interested in the problem of whether a body of mathematical knowledge can be effectively formalized as a theory graph of biform theories. As a test case, we look at the graph of theories encoding natural number arithmetic. We used two different formalisms to do this, which we describe and compare. The first is realized in CTT_{uqe}, a version of Church's type theory with quotation and evaluation, and the second is realized in Agda, a dependently typed programming language.

1 Introduction

There are many methods for encoding mathematical knowledge. The two most prevalent are the *axiomatic* and the *algorithmic*. The axiomatic method, famously employed by Euclid in his *Elements* circa 300 BCE, encodes a body of knowledge as an *axiomatic theory* composed of a language and a set of *axioms* expressed in that language. The axioms are assumptions about the *concepts* of the language and the logical consequences of the axioms are the *facts* about the concepts. The algorithmic method in contrast uses an *algorithmic theory*, composed of a language and a set of *algorithms* that perform symbolic computations over the expressions of the language. Each algorithm procedurally encodes its input/output relation. For example, an algorithm that symbolically

* Published without appendices in: H. Geuvers et al., eds, *Intelligent Computer Mathematics (CICM 2017)*, *Lecture Notes in Computer Science*, Vol. 10383, pp. 9–24, Springer, 2017. The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-62075-6_2. This research was supported by NSERC.

adds expressions that represent rational numbers encodes the addition function $+ : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$ over the rational numbers.

A complex body of mathematical knowledge comprises many different theories; these can be captured by the *little theories method* [**?**] as a *theory graph* [**?**] consisting of theories as nodes and theory morphisms as directed edges. A *theory morphism* is a meaning-preserving mapping from the formulas of one theory to the formulas of another. The theories serve as abstract mathematical models and the morphisms serve as information conduits that enable definitions and theorems to be transported from one theory to another [**?**]. A theory graph enables mathematical knowledge to be formalized at the most convenient level of abstraction using the most convenient vocabulary. Moreover, the structure of a theory graph provides the means to access relevant concepts and facts (c&f), reduce the duplication of c&f, and enable c&f to be interpreted in multiple ways.

The axiomatic method is the basis for formalizing mathematical knowledge in proof assistants and logical frameworks. Although many proof assistants support the little theories method to some extent, very few provide the means to explicitly build theory graphs. Notable exceptions are the IMPS theorem proving system [**?**] and the MMT module system for mathematical theories [**?**].

Computer algebra systems on the other hand are based on algorithmic theories, which are not usually organized as a graph. An exception is the Axiom system [**?**] in which a network of abstract and concrete algorithmic theories are represented by Axiom categories and domains, respectively. Algorithmic theories are challenging to fully formalize because a specification of a symbolic algorithm that encodes a mathematical function requires the ability to talk about the relationship between syntax and semantics.

Axiomatic and algorithmic knowledge complement each other, and both are needed. A *biform theory* [**?**,**?**,**?**] combines both, and furthermore supports the integration of reasoning and computation. We argue in [**?**] that biform theories are needed to build *high-level theories* analogous to high-level programming languages. Biform theories are challenging to formalize for the same reasons that algorithmic theories are challenging to formalize.

We are interested in the problem of whether the little theories method can be applied to biform theories. That is, can a body of mathematical knowledge be effectively formalized as a theory graph of biform theories? We use a graph (of biform theories) encoding natural number arithmetic as a test case. We describe two different formalizations, and compare the results. The first formalization is realized using the global approach in CTT_{uqe} [**?**], a variant of CTT_{qe} [**?**,**?**], a version of Church's type theory with quotation and evaluation, while the second is realized using the local approach in Agda [**?**,**?**], a dependently typed programming language. This dual formalization, contrasting the two approaches, forms the core of our contribution; each formalization has some smaller contributions, some of which may be of independent interest.

The rest of the paper is organized as follows. The notion of a biform theory is defined and discussed in section 2. The theories that encode natural number arithmetic are presented in section 3. The CTT_{uqe} formalization is discussed in

section 4, and the Agda version in section 5. These two are presented in full in appendices A and B. Section 6 compares the two formalizations. The paper ends with conclusions and future work in section 7.

The authors are grateful to the reviewers for their comments and suggestions.

2 Biform Theories

Let $\mathcal{E}$ be a set of expressions and $f : \mathcal{E}^n \to \mathcal{E}$ be an n-ary function where $n \geq 1$. A *transformer for* f is an algorithm that implements f. Transformers manipulate expressions in various ways. Simple transformers, for example, build bigger expressions from pieces, select components of expressions, or check whether a given expression satisfies some syntactic property. More sophisticated transformers manipulate expressions in mathematically meaningful ways. We call these kinds of transformers *syntax-based mathematical algorithms (SBMAs)* [**?**]. Examples include algorithms that apply arithmetic operations to numerals, factor polynomials, transpose matrices, and symbolically differentiate expressions with variables. The *computational behavior* of a transformer is the relationship between its input and output expressions. When the transformer is an SBMA, its *mathematical meaning* is the relationship between the mathematical meanings of its input and output expressions.

A *biform theory* T is a triple (L, Π, Γ) where L is a language of some underlying logic, Π is a set of transformers for functions over expressions of L, and Γ is a set of formulas of L. L is generated from a set of symbols that include, e.g., types and constants. Each symbol is the name for a concept of T. The transformers in Π are for functions represented by symbols of L. The members of Γ are the *axioms* of T. They specify the concepts of T including the computational behaviors of transformers and the mathematical meanings of SBMAs. The underlying logic provides the semantic foundation for T. We say T is an *axiomatic theory* if Π is empty and an *algorithmic theory* if Γ is empty.

Expressing a biform theory in the underlying logic requires infrastructure for reasoning about expressions manipulated by the transformers as syntactic entities. The infrastructure provides a basis for *metareasoning with reflection* [**?**]. There are two main approaches for obtaining this infrastructure [**?**]. The *local approach* is to produce a deep embedding of a sublanguage L' of L that include all the expressions manipulated by the transformers of Π. The deep embedding consists of (1) an inductive type of *syntactic values* that represent the syntactic structures of the expressions in L', (2) an *informal quotation operator* that maps the expressions in L' to syntactic values, and (3) a *formal evaluation operator* that maps syntactic values to the values of the expressions in L' that they represent. The *global approach* is to replace the underlying logic of L with a logic such as that of [**?**] that has (1) an inductive type of *syntactic values* for all the expressions in L, (2) a *global formal quotation operator*, and (3) a *global formal evaluation operator*.

There are several ways, in a proof assistant, to construct a transformer π for $f : \mathcal{E}^n \to \mathcal{E}$. The simplest is to define f as a lambda abstraction A_f, and

then π computes the value $f(e_1, \ldots, e_n)$ by reducing $A_f(e_1, \ldots, e_n)$ using β-reduction (and possibly other transformations such as δ-reduction, etc). Another method is to specify the computational behavior of f by axioms, and then π can be implemented as a tactic that applies the axioms to $f(e_1, \ldots, e_n)$ as, e.g., rewrite rules or conditional rewrite rules. Finally, the computational behavior or mathematical meaning of f can be specified by axioms, and then π can be a program which satisfies these axioms; this program can operate on either internal or external data structures representing the expressions $e_1, \ldots, e_n$.

3 Natural Number Arithmetic: A Test Case

Figure 1 shows a theory graph composed of biform theories encoding natural number arithmetic. We start with eight axiomatic theories (seven in first-order logic (FOL) and one in simple type theory (STT)) and then add a variety of useful transformers in the appropriate theories. These eight are chosen because they fit together closely and have simple axiomatizations. Of the first-order theories, BT1 and BT5 are theories of 0 and S (which denotes the successor function); BT2 and BT6 are theories of 0, S, and $+$; and BT3, BT4, and BT7 are theories of 0, S, $+$, and $*$. Several other biform theories could be added to this graph, most notably Skolem arithmetic, the complete theory of 0, S, and $*$, which has a very complicated axiomatization [**?**]. The details of each theory is given below.

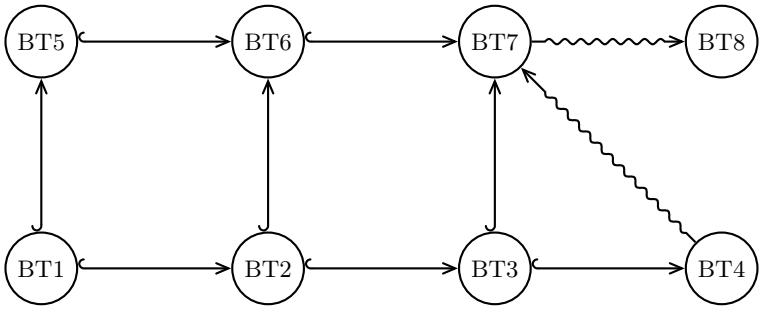

Fig. 1. Biform Theory Graph Test Case

Figure 1 shows the morphisms that connect these theories. The $\hookrightarrow$ arrows denote strict theory inclusions. The morphism from BT4 to BT7 is the identity mapping. It is meaning-preserving since each axiom of BT4 is a theorem of BT7. In particular, A7 follows from the induction schema A10. The theory morphism from BT7 to BT8 is interlogical since their logics are different. It is defined by the mapping of $0, S, +, *$ to $0_\iota, S_{\iota\to\iota}, +_{\iota\to\iota\to\iota}, *_{\iota\to\iota\to\iota}$, respectively, where $+_{\iota\to\iota\to\iota}$ and $*_{\iota\to\iota\to\iota}$ are defined constants in BT8. It is meaning-preserving since A1–A6 and the instances of the induction schema A10 map to theorems of BT8.

We have formalized this biform theory graph in two ways: the first in CTT_{uqe} using the global approach and the second in Agda using the local approach.

These are discussed in the next two sections, while the full details are given in appendices A and B. A "conventional" mathematical presentation of the theories would be as follows.

Biform Theory 1 (BT1: Simple Theory of 0 and S)
Logic: FOL. *Constants*: 0 (0-ary), S (unary).
Axioms:
A1. $S(x) \neq 0$.
A2. $S(x) = S(y) \supset x = y$.
Properties: Incomplete, undecidable.
Transformers: Recognizer for the formulas of the theory.

Biform Theory 2 (BT2: Simple Theory of 0, S, and $+$)
Extends BT1.
Logic: FOL. *Constants*: $+$ (binary, infix).
Axioms:
A3. $x + 0 = x$.
A4. $x + S(y) = S(x + y)$.
Properties: Incomplete, undecidable.
Transformers: Recognizer for the formulas of the theory and algorithm for adding natural numbers as binary numerals.

Biform Theory 3 (BT3: Simple Theory of 0, S, $+$, and $*$)
Extends BT2.
Logic: FOL. *Constants*: $*$ (binary, infix).
Axioms:
A5. $x * 0 = 0$.
A6. $x * S(y) = (x * y) + x$.
Properties: Incomplete, undecidable.
Transformers: Recognizer for the formulas of the theory and algorithm for multiplying natural numbers as binary numerals.

Biform Theory 4 (BT4: Robinson Arithmetic (Q))
Extends BT3.
Logic: FOL.
Axioms:
A7. $x = 0 \vee \exists\, y\ .\ S(y) = x$.
Properties: Essentially incomplete, essentially undecidable.

Biform Theory 5 (BT5: Complete Theory of 0 and S)
Extends BT1.
Logic: FOL.
Axioms:
A8. $(A(0) \wedge \forall\, x\ .\ (A(x) \supset A(S(x)))) \supset \forall\, x\ .\ A(x)$
where A is any formula of BT5 in which x is not bound and $A(t)$ is the result of replacing each free occurrence of x in A with the term t.
Properties: Complete, decidable.

Transformers: Generator for instances of the theory's induction schema and decision procedure for the theory.

Biform Theory 6 (BT6: Presburger Arithmetic)
Extends BT2 and BT5.
Logic: FOL.
Axioms:
A9. $(A(0) \wedge \forall\, x\ .\ (A(x) \supset A(S(x)))) \supset \forall\, x\ .\ A(x)$
where A is any formula of BT6 in which x is not bound and $A(t)$ is the result of replacing each free occurrence of x in A with the term t.
Properties: Complete, decidable.
Transformers: Generator for instances of the theory's induction schema and decision procedure for the theory.

Biform Theory 7 (BT7: First-Order Peano Arithmetic)
Extends BT3 and BT6.
Logic: FOL.
Axioms:
A10. $(A(0) \wedge \forall\, x\ .\ (A(x) \supset A(S(x)))) \supset \forall\, x\ .\ A(x)$
where A is any formula of BT7 in which x is not bound and $A(t)$ is the result of replacing each free occurrence of x in A with the term t.
Properties: Essentially incomplete, essentially undecidable.
Transformers: Generator for instances of the theory's induction schema.

Biform Theory 8 (BT8: Higher-Order Peano Arithmetic)
Logic: STT. *Types*: ι. *Constants*: 0_ι, $S_{\iota\to\iota}$.
Axioms:
A11. $S_{\iota\to\iota}(x_\iota) \neq 0_\iota$.
A12. $S_{\iota\to\iota}(x_\iota) = S_{\iota\to\iota}(y_\iota) \supset x_\iota = y_\iota$.
A13. $(p_{\iota\to o}(0) \wedge \forall\, x_\iota\ .\ (p_{\iota\to o}(x_\iota) \supset p_{\iota\to o}(S(x_\iota)))) \supset \forall\, x_\iota\ .\ p_{\iota\to o}(x_\iota)$.
Properties: Essentially incomplete, essentially undecidable, categorical for standard models.

It is important to note that axioms A8, A9 and A10 are all different since they are over different languages; in particular, **BT6** adds $+$ to the language of **BT5**, and **BT7** adds $*$ to the language of **BT6**.

4 Study 1: Test Case Formalized in $\mathrm{CTT}_{\mathbf{uqe}}$

$\mathrm{CTT}_{\mathrm{uqe}}$ supports the global approach for metareasoning with reflection. $\mathrm{CTT}_{\mathrm{uqe}}$ contains (1) a logical base type ϵ that is an inductive type of syntactic values called *constructions* which are expressions of type ϵ, (2) a global quotation operator $\ulcorner\cdot\urcorner$ that maps each expression $\mathbf{A}_\alpha$ of $\mathrm{CTT}_{\mathrm{uqe}}$ to a construction that represents the syntactic structure of $\mathbf{A}_\alpha$, and (3) a typed global evaluation operator $[\![\cdot]\!]_\alpha$ that maps each construction $\mathbf{B}_\epsilon$ of $\mathrm{CTT}_{\mathrm{uqe}}$ representing an expression $\mathbf{A}_\alpha$ of type α to an expression whose value is the same as $\mathbf{A}_\alpha$. See [**?**] for details.

A *biform theory* of CTT_{uqe} is a triple (L, Π, Γ) where L is a language generated by a set of base types and constants of CTT_{uqe}, Π is a set of transformers over expressions of L, and Γ is a set of formulas of L. Each transformer is for a constant in L whose type has the form $\epsilon \to \cdots \to \epsilon$. We present biform theories in CTT_{uqe} as a set of base types, constants, axioms, transformers, and theorems. The base types are divided into primitive and defined base types. A defined base type is declared by a formula that equates the base type to a nonempty subset of some type of L. Similarly, the constants are divided into primitive and defined constants. A defined constant $\mathbf{c}_\alpha$ is declared by an equation $\mathbf{c}_\alpha = \mathbf{A}_\alpha$ where $\mathbf{A}_\alpha$ is a defined expression.

The biform theory graph test case given in section 3 is formalized in CTT_{uqe} as a theory graph of eight CTT_{uqe} theories as shown in appendix A. Since CTT_{uqe} is not currently implemented, it is not possible to give the transformers as implemented algorithms. Instead we described their intended behavior.

We will concentrate our discussion on BT6 (given below). We have not included the following components of BT6 (that should be in BT6 according to its definition in section 3) that are redundant or are subsumed by other components: Constants $\mathsf{BT5\text{-}DEC\text{-}PROC}_{\epsilon\to\epsilon}$, $\mathsf{IS\text{-}FO\text{-}BT1}_{\epsilon\to\epsilon}$, and $\mathsf{IS\text{-}FO\text{-}BT1\text{-}ABS}_{\epsilon\to\epsilon}$; axioms 27 and 28; and transformers π_1, π_2, π_{11}, π_{12}, and π_{13}. See [**?**] for details. Expressions of type ϵ, i.e., expressions that denote constructions, are colored red to identify where reasoning about syntax occurs.

BT6 has the usual constants (0_ι, $S_{\iota\to\iota}$, and $+_{\iota\to\iota\to\iota}$) and axioms (axioms 1–4 and 29) of Presburger arithmetic. Axiom 29 is the direct formalization of A9, the induction schema for Presburger arithmetic, stated in section 3. It is expressed as a single universal formula in CTT_{uqe} that ranges over constructions representing function abstractions of the form $\lambda\, \mathbf{x}_\iota\ .\ \mathbf{A}_o$. These constructions are identified by the transformers π_{15} and π_{16} for the defined constant $\mathsf{IS\text{-}FO\text{-}BT2\text{-}ABS}_{\epsilon\to\epsilon}$. π_{15} works by accessing data about variables, constants, and other subexpressions stored in the data structure for an expression, while π_{16} works by expanding the definition of $\mathsf{IS\text{-}FO\text{-}BT2\text{-}ABS}_{\epsilon\to\epsilon}$. π_{15} is sound if the definition expansion mechanism is sound. Showing the soundness of π_{14} would require a formal verification of the implementation of the data structure for expressions. Of course, the results of π_{14} could be checked using π_{16}.

This biform theory has a defined constant $\mathsf{bnat}_{\iota\to\iota\to\iota}$ with the usual base 2 notation for expressing natural numbers in a binary form. There is a constant $\mathsf{BPLUS}_{\epsilon\to\epsilon\to\epsilon}$ specified by axioms 5–15 for adding quotations of these natural numbers in binary form. $\mathsf{BPLUS}_{\epsilon\to\epsilon\to\epsilon}$ is implemented by transformers π_3 and π_4. π_3 is some efficient algorithm implemented outside of CTT_{uqe}, and π_4 is an algorithm that uses axioms 5–15 as conditional rewrite rules. π_4 is sound if the rewriting mechanism is sound. Showing the soundness of π_3 would require a formal verification of its program. The meaning formula for $\mathsf{BPLUS}_{\epsilon\to\epsilon\to\epsilon}$, theorem 3, follows from axioms 5–15.

This biform theory also has a transformer π_{14} for $\mathsf{BT6\text{-}DEC\text{-}PROC}_{\epsilon\to\epsilon}$ that implements an efficient decision procedure for the first-order formulas of the theory that is specified by axiom 30. The first-order formulas of the theory are

identified by the transformers π_5 and π_6 for the defined constant $\mathsf{IS\text{-}FO\text{-}BT2}_{\epsilon\to\epsilon}$ that are analogous to the transformers π_{15} and π_{16} for $\mathsf{IS\text{-}FO\text{-}BT2\text{-}ABS}_{\epsilon\to\epsilon}$.

Biform Theory 6 (BT6: Presburger Arithmetic)

Primitive Base Types

1. ι (type of natural numbers).

Primitive Constants

1. 0_ι.
2. $S_{\iota\to\iota}$.
3. $+_{\iota\to\iota\to\iota}$ (infix).
4. $\mathsf{BPLUS}_{\epsilon\to\epsilon\to\epsilon}$ (infix).
6. $\mathsf{BT6\text{-}DEC\text{-}PROC}_{\epsilon\to\epsilon}$.

Defined Constants (selected)

1. $1_\iota = S\,0_\iota$.
3. $\mathsf{bnat}_{\iota\to\iota\to\iota} = \lambda\, x_\iota \,.\, \lambda\, y_\iota \,.\, ((x_\iota + x_\iota) + y_\iota)$.
   Notational definition:
   $(0)_2 = \mathsf{bnat}\,0_\iota\,0_\iota$.
   $(1)_2 = \mathsf{bnat}\,0_\iota\,1_\iota$.
   $(a_1 \cdots a_n 0)_2 = \mathsf{bnat}\,(a_1 \cdots a_n)_2\,0_\iota$ where each a_i is 0 or 1.
   $(a_1 \cdots a_n 1)_2 = \mathsf{bnat}\,(a_1 \cdots a_n)_2\,1_\iota$ where each a_i is 0 or 1.
4. $\mathsf{is\text{-}bnum}_{\epsilon\to o} = \mathrm{I}\, f_{\epsilon\to o} \,.\, \forall\, u_\epsilon \,.\, (f_{\epsilon\to\epsilon}\, u_\epsilon \equiv$
   $\exists\, v_\epsilon \,.\, \exists\, w_\epsilon \,.\, (u_\epsilon = \ulcorner \mathsf{bnat}\,\lfloor v_\epsilon \rfloor\,\lfloor w_\epsilon \rfloor \urcorner \wedge$
   $(v_\epsilon = \ulcorner 0_\iota \urcorner \vee f_{\epsilon\to\epsilon}\, v_\epsilon) \wedge (w_\epsilon = \ulcorner 0_\iota \urcorner \vee w_\epsilon = \ulcorner 1_\iota \urcorner)))$.[1]
5. $\mathsf{IS\text{-}FO\text{-}BT2}_{\epsilon\to\epsilon} = \lambda\, x_\epsilon \,.\, \mathbf{B}_\epsilon$ where $\mathbf{B}_\epsilon$ is a complex expression such that $(\lambda\, x_\epsilon \,.\, \mathbf{B}_\epsilon)\ulcorner \mathbf{A}_\alpha \urcorner$ equals $\ulcorner T_o \urcorner$ [$\ulcorner F_o \urcorner$] if $\mathbf{A}_\alpha$ is [not] a term or formula of first-order logic with equality whose variables are of type ι and whose nonlogical constants are members of $\{0_\iota, S_{\iota\to\iota}, +_{\iota\to\iota\to\iota}\}$.
7. $\mathsf{IS\text{-}FO\text{-}BT2\text{-}ABS}_{\epsilon\to\epsilon} =$
   $\lambda\, x_\epsilon \,.\, (\mathsf{if}\ (\mathsf{is\text{-}abs}_{\epsilon\to o}\, x_\epsilon)\ (\mathsf{IS\text{-}FO\text{-}BT2}_{\epsilon\to\epsilon}\,(\mathsf{abs\text{-}body}_{\epsilon\to\epsilon}\, x_\epsilon))\ \ulcorner F_o \urcorner)$.

Axioms

1. $S\, x_\iota \neq 0_\iota$.
2. $S\, x_\iota = S\, y_\iota \supset x_\iota = y_\iota$.
3. $x_\iota + 0_\iota = x_\iota$.
4. $x_\iota + S\, y_\iota = S\,(x_\iota + y_\iota)$.
5. $\mathsf{is\text{-}bnum}\, u_\epsilon \supset u_\epsilon\ \mathsf{BPLUS}\ \ulcorner (0)_2 \urcorner = u_\epsilon$.

   ⋮

15. $(\mathsf{is\text{-}bnum}\, u_\epsilon \wedge \mathsf{is\text{-}bnum}\, v_\epsilon) \supset$
    $\ulcorner \mathsf{bnat}\,\lfloor u_\epsilon \rfloor\, 1_\iota \urcorner\ \mathsf{BPLUS}\ \ulcorner \mathsf{bnat}\,\lfloor v_\epsilon \rfloor\, 1_\iota \urcorner =$
    $\ulcorner \mathsf{bnat}\,\lfloor (u_\epsilon\ \mathsf{BPLUS}\ v_\epsilon)\ \mathsf{BPLUS}\ \ulcorner (1)_2 \urcorner \rfloor\, 0_\iota \urcorner$.
29. Induction Schema for S and $+$
    $\forall\, f_\epsilon \,.\, ((\mathsf{is\text{-}expr}^{\iota\to o}_{\epsilon\to o}\, f_\epsilon \wedge [\![\mathsf{IS\text{-}FO\text{-}BT2\text{-}ABS}_{\epsilon\to\epsilon}\, f_\epsilon]\!]_o) \supset$
    $(([\![f_\epsilon]\!]_{\iota\to o}\, 0_\iota \wedge (\forall\, x_\iota \,.\, [\![f_\epsilon]\!]_{\iota\to o}\, x_\iota \supset [\![f_\epsilon]\!]_{\iota\to o}\,(\mathsf{S}\, x_\iota))) \supset \forall\, x_\iota \,.\, [\![f_\epsilon]\!]_{\iota\to o}\, x_\iota))$.

[1] Notation of the form $\ulcorner \cdots \lfloor \cdot \rfloor \cdots \urcorner$ represents a quasiquotation; see [**?**] for details.

30. Meaning formula for $\mathsf{BT6\text{-}DEC\text{-}PROC}_{\epsilon\to\epsilon}$.
$\forall\, u_\epsilon\ .\ ((\mathsf{is\text{-}expr}^{o}_{\epsilon\to o}\, u_\epsilon \wedge \mathsf{is\text{-}closed}_{\epsilon\to o}\, u_\epsilon \wedge [\![\mathsf{IS\text{-}FO\text{-}BT2}_{\epsilon\to\epsilon}\, u_\epsilon]\!]_o) \supset$
$((\mathsf{BT6\text{-}DEC\text{-}PROC}_{\epsilon\to\epsilon}\, u_\epsilon = \ulcorner T_o \urcorner \vee \mathsf{BT6\text{-}DEC\text{-}PROC}_{\epsilon\to\epsilon}\, u_\epsilon = \ulcorner F_o \urcorner) \wedge$
$[\![\mathsf{BT6\text{-}DEC\text{-}PROC}_{\epsilon\to\epsilon}\, u_\epsilon]\!]_o = [\![u_\epsilon]\!]_o))$.

Transformers

3. π_3 for $\mathsf{BPLUS}_{\epsilon\to\epsilon\to\epsilon}$ is an efficient program that satisfies Axioms 5–15.
4. π_4 for $\mathsf{BPLUS}_{\epsilon\to\epsilon\to\epsilon}$ uses Axioms 5–15 as conditional rewrite rules.
5. π_5 for $\mathsf{IS\text{-}FO\text{-}BT2}_{\epsilon\to\epsilon}$ is an efficient program that accesses the data stored in the data structures that represent expressions.
6. π_6 for $\mathsf{IS\text{-}FO\text{-}BT2}_{\epsilon\to\epsilon}$ uses the definition of $\mathsf{IS\text{-}FO\text{-}BT2}_{\epsilon\to\epsilon}$.
14. π_{14} for $\mathsf{BT6\text{-}DEC\text{-}PROC}_{\epsilon\to\epsilon\to\epsilon}$ is an efficient decision procedure that satisfies Axiom 30.
15. π_{15} for $\mathsf{IS\text{-}FO\text{-}BT2\text{-}ABS}_{\epsilon\to\epsilon}$ is an efficient program that accesses the data stored in the data structures that represent expressions.
16. π_{16} for $\mathsf{IS\text{-}FO\text{-}BT2\text{-}ABS}_{\epsilon\to\epsilon}$ uses the definition of $\mathsf{IS\text{-}FO\text{-}BT2\text{-}ABS}_{\epsilon\to\epsilon}$.

Theorems (selected)

3. Meaning formula for $\mathsf{BPLUS}_{\epsilon\to\epsilon\to\epsilon}$
$\forall\, u_\epsilon\ .\ \forall\, v_\epsilon\ .\ ((\mathsf{is\text{-}bnum}\, u_\epsilon \wedge \mathsf{is\text{-}bnum}\, v_\epsilon) \supset$
$(\mathsf{is\text{-}bnum}\, (u_\epsilon\ \mathsf{BPLUS}\ v_\epsilon) \wedge ([\![u_\epsilon\ \mathsf{BPLUS}\ v_\epsilon]\!]_\iota = [\![u_\epsilon]\!]_\iota + [\![v_\epsilon]\!]_\iota)))$.

5 Study 2: Test Case Formalized in Agda

As our goal is to, in part, compare the global approach and the local approach, the formalization in Agda [**?**,**?**] eschews the use of its reflection capabilities[2]. Thus this formalization replaces the global type ϵ (of $\mathrm{CTT}_{\mathrm{uqe}}$) by a *set* of inductive types, one for each of the biform theories. This is still reflection, just hand-rolled. We also need to express formulas in FOL (as syntax), so we need a type for that as well. To be more modular, this is done as a type for first-order logic (with equality) over any ground language. We display some illustrative samples here; the full code is available in appendix B.

An abstract theory is modeled as a *record*. For example, we have BT_1:

```
record BT₁ : Set₁ where
  field
    nat : Set₀
    Z : nat
    S : nat → nat
    S≠Z : ∀ x → ¬ (S x ≡ Z)
    inj : ∀ x y → S x ≡ S y → x ≡ y

  One : nat
  One = S Z
```

[2] As of early 2017, there is no official publication describing these features outside of the Agda documentation, but see [**?**,**?**]

where we see a field nat for the new type (pronounced Set in Agda), the two constants, and the two axioms. The host logic is dependently typed, and so the axioms refer to the constants just defined. One is not a field, but a defined constant.

For simplicity, we will take the built-in type $\mathbb{N}$, defined as an inductive type, as the *syntax* for natural numbers, which is also the syntax associated to the theory BT_1. Whereas in $\mathrm{CTT}_{\mathrm{uqe}}$ there is a global evaluation, here we also need to define evaluation explicitly (a subscript is used to indicate which theory it belongs to).

```
⟦_⟧₁ : ℕ → nat
⟦ 0 ⟧₁ = Z
⟦ suc x ⟧₁ = S ⟦ x ⟧₁
```

The accompanying code furthermore proves some basic coherence theorems which are elided here. We make two further definitions (GroundLanguage describing some language features, and FOL as our definition of first order logic) which will be explained in more detail on the next page.

```
nat-lang : GroundLanguage nat
nat-lang = record { Lang = λ X → ℕX (Carrier X)
                  ; value = λ {V} → val {V} }
where
  val : {V : DT} → ℕX (Carrier V) → (Carrier V → nat) → nat
  val  z   env = Z
  val {V} (s e) env = S (val {V} e env)
  val  (v x) env = env x

module fo₁ = FOL nat-lang
```

We can also demonstrate that the natural numbers are a model:

```
ℕ-is-T1 : BT₁
ℕ-is-T1 = record { nat = ℕ ; Z = 0 ; S = suc
  ; S≠Z = λ x → λ () ; inj = λ { x .x refl → refl } }
```

One of the languages needed is an extension of the naturals which allows variables:

```
data ℕX (Var : Set₀) : Set₀ where
  z : ℕX Var
  s : ℕX Var → ℕX Var
  v : Var → ℕX Var
```

But where the informal description in section 3 can get away with saying "Logic: FOL" and "Transformers: Recognizers for the formulas of the theory", here we need to be very explicit. To do so, we need to define some language infrastructure.

One of the important concepts is that of a *language with variables*, in other words a language with a reasonable definition of substitution. This requires *vari-*

ables to come from a type which has the structure of a decidable setoid (from the Agda library DecSetoid, and denoted DT below).

A language, expressed as an inductive type, is closed, i.e., cannot be extended. If a language does not have variables, we cannot add them. One solution is to deal with *contexts* as first-class citizens. While that is likely the best long-term solution, here we have gone with something simpler: create another language which does, and show that its variable-free fragment is equivalent to the original. As that aspect of our development is straightforward, albeit tedious, we elide it.

As we are concerned with statements in first-order logic over a variety of languages, it makes sense to modularize this aspect somewhat. Note that, as we are building syntax via inductive types, we can either build these as functors and then use a fixpoint combinator to tie the knot, or we can just bite the bullet and make one large definition. For now, we chose the latter. We do parametrize over a *ground language with variables*. In turn, this is defined as a type parametrized by a decidable setoid along with an evaluation function into some type $\top$.

```
record GroundLanguage (T : Set₀) : Set₁ where
  open DecSetoid using (Carrier)
  field
    Lang : DT → Set₀
    value : {V : DT} → Lang V → (Carrier V → T) → T
```

A logic over a language (with variables), is then also a parametrized type as well as a parametrized interpretation into types. The definition is almost the same, except that a ground language interprets into $\top$ and a logic into Set_0.

```
record LogicOverL (T : Set₀) (L : GroundLanguage T) : Set₁ where
  open DecSetoid using (Carrier)
  field
    Logic : DT → Set₀
    ⟦_⟧_ : ∀ {V} → Logic V → (Carrier V → T) → Set₀
```

The definition of first-order logic is then straightforward.

```
module FOL {T : Set₀} (L : GroundLanguage T) where
  open DecSetoid using (Carrier)
  open GroundLanguage L

  data FOL (V : DT) : Set where
    tt : FOL V
    ff : FOL V
    _and_ : FOL V → FOL V → FOL V
    _or_ : FOL V → FOL V → FOL V
    not : FOL V → FOL V
    _⊃_ : FOL V → FOL V → FOL V
    _==_ : Lang V → Lang V → FOL V
    all : Carrier V → FOL V → FOL V
    exist : Carrier V → FOL V → FOL V

  override : {V : DT} → (Carrier V → T) → Carrier V → T → (Carrier V → T)
```

```
  override { V } f x t y with DecSetoid._≟_ V y x
  ... | yes _ = t
  ... | no _  = f y
```

We can also prove that FOL is a logic over L by providing an interpretation. Of course, as we are modeling classical logic into a constructive logic, we have to use a double-negation embedding. We also choose to interpret the logic's equality _==_ as *propositional equality*, but we could make that choice a parameter as well.

```
  LoL-FOL : LogicOverL T L
  LoL-FOL = record { Logic = FOL ; ⟦_⟧_ = interp }
    where
      interp : {Var : DT} → FOL Var → (Carrier Var → T) → Set₀
      interp tt env = ⊤
      interp ff env = ⊥
      interp (e and f) env = interp e env × interp f env
      interp (e or f) env = ¬ ¬ (interp e env ⊎ interp f env)
      interp (not e) env = ¬ (interp e env)
      interp (e ⊃ f) env = (interp e env) → (interp f env)
      interp (x == y) env = value x env ≡ value y env
      interp {V} (all x p) env  = ∀ z → interp p (override {V} env x z)
      interp {V} (exist x p) env = ¬ ¬ (Σ T (λ t → interp p (override {V} env x t)))
```

With the appropriate infrastructure in place, it is now possible to define BT_6 from the theories it extends.

```
record BT₆ {t₁ : BT₁} (t₂ : BT₂ t₁) (t₅ : BT₅ t₁) : Set₁ where
  open VarLangs using (XV; x)
  open DecSetoid using (Carrier)
  open BT₂ t₂ public
  open fo₂ using (FOL; tt; ff; LoL-FOL; _and_; all)
  open LogicOverL LoL-FOL

  field
    induct : (e : FOL XV) →
      ⟦ e ⟧ (λ { x → ⟦ 0 ⟧₁ }) →
      (∀ y → ⟦ e ⟧ (λ {x → y}) → ⟦ e ⟧ (λ {x → S y})) →
      ∀ y → ⟦ e ⟧ (λ {x → y})
  postulate
    decide : ∀ {W} → (Carrier W → nat) → FOL W → FOL NoVars
    meaning-decide : {W : DT} (env : Carrier W → nat) → (env′ : ⊥ → nat) →
      (e : FOL W) →
      let res = decide env e in
      (res ≡ tt ⊎ res ≡ ff) × (⟦ e ⟧ env) ≃ (⟦ res ⟧ env′)
```

While section 4 presents the *flattened* theory, here we need only define what is new over the extended theory, namely an induction schema, a decision procedure and its meaning formula.

Here is a guide to understanding the above definition: (1) XV is a (decidable) type with a single inhabitant, x. (2) All fields of BT_2 are made publicly visible for BT_6. (3) The language of first-order logic FOL over t_2 (and some of its constructors) is also made visible. (4) (λ {x → y}) denotes a substitution for the single variable x. (5) $\simeq$ denotes *type equivalence*.

We represent numerals as vectors (of length at least 1) of binary digits.

```
data BinDigit : Set where zero one : BinDigit
data BNum : Set where
  bn : {n : ℕ} → Vec BinDigit (suc n) → BNum
```

This then allows a straightforward implementation of bplus to add numerals. It is then possible to *prove* that the meaning function for bplus is a theorem.

```
bplus-is-+ : ∀ x y → ⟦ bplus x y ⟧₂ ≡ ⟦ x ⟧₂ + ⟦ y ⟧₂
```

6 Comparison of the Two Formalizations

As expected, we were able to formalize this network of theories using both methods. Neither are fully complete; both are missing the actual decision procedures (which would be large undertakings). In particular,

- The CTT_{uqe} formalization is missing the definition of the language recognizers, as well as the full assurance of being mechanically checked. It has no "implementation" of any transformers.
- The Agda version implements evaluation but not substitution — which means that the induction statement in BT5–BT7 are not quite the same as in CTT_{uqe}; the models will be the same however. It also does not implement any theory morphisms, as record definitions are not first-class in Agda.

More importantly, because of our (explicit) choice to contrast the global and local approaches, each version uses different infrastructure to reason about syntax.

- CTT_{uqe} has a built-in inductive type of "all syntax", along with quotation and evaluation operators for the entire language of expressions.
- In the local approach, a new inductive type for each new "language" (the numerals, the numerals with plus, the numerals with plus and times, all three of these augmented with variables, first-order logic, binary digits, binary numerals) has to be created. For many of these, a variety of traversals (folds) have to be implemented "by hand" even though the recursion patterns are obvious, at least to humans. Some of these are evaluation operators (one per language). There is no formal quotation operator.

The Agda version has a number of extra features: some transformers (such as for bplus and btimes) are implemented. Furthermore, the *meaning formula* for bplus is shown to be a theorem. A variety of coherence theorems are also shown, to gain confidence that the theories really are the ones we want.

It is worth remarking that defining the language of first-order formulas is complicated in *both* versions. This has been noticed before by people doing programming language meta-theory with proof assistants: encoding languages, especially languages with binders (such as FOL) along with traversals and basic reasoning can be very tedious [**?**].

The most notable differences in the two formalizations are:

1. Because FOL is classical, but Agda's host logic is constructive, a double-negation embedding was needed.
2. The use of *type equivalence* instead of boolean equality for verifying that the interpretation of a formula of FOL and the results of the decision procedure are "the same",
3. Borrowing the notion of *contractibility* from HoTT [**?**], to encode *definite description*.
4. Extending the decision procedure to *closeable* terms (by providing an explicit, total valuation) instead of restricting to closed terms.

The first is basically forced upon us by Agda: it has no Prop type (unlike Coq), and so we do not know a priori that all interpretations of first-order formulas are actually 0-types. The second is an active design decision: the infrastructure required to define the meaning of *closed* which is useful in a constructive setting is quite complex[3]. We believe the third is novel. The fourth point requires deeper investigating.

7 Conclusion

We have proposed a biform theory graph test case composed of eight theories that encode natural number arithmetic and include a variety of useful transformers. We have formalized this test case (as a set of biform theories and theory morphisms) in CTT_{uqe} using the global approach (for metareasoning with reflection) and in Agda using the local approach. In both cases, we have produced substantial partial formalizations that indicate that full formalizations could be obtained with additional work.

Our results show that, by providing a built-in global infrastructure, the global approach has a significant advantage over the local approach. The local approach is burdened by the necessity to define an infrastructure — consisting of an inductive type and an evaluation operator for the type — for every set of expressions manipulated by a transformer. In general, new local infrastructures must be created each time a new theory is added to the theory graph. On the other hand, the global approach employs an infrastructure — consisting of an inductive type, a quotation operator, and an evaluation operator — for the entire set of expressions in the logic. This single infrastructure is used for every theory in the theory graph.

[3] It would require us to define *paths* in terms, bound and free variables along paths, quantification over paths, etc.

We recommend that future research is directed toward making the global approach for metareasoning with reflection into a practical method for formalizing biform theories. This can be done by developing and implementing logics such as CTT_{qe} [?,?] and CTT_{uqe} [?] and by adding global infrastructures to proof systems such as Agda and Coq (see [?,?] for work in this direction).

A CTT_{uqe} Formalization

CTT_{qe} [?] is a version of simple type theory with quotation and evaluation. CTT_{uqe} [?] is a variant of CTT_{qe} that admits undefined expressions, partial functions, and multiple base types of individuals. CTT_{uqe} also includes a definite description operator and conditional expression operator. This appendix presents a formalization of the biform theory graph test case in CTT_{uqe} (instead of CTT_{qe}) since CTT_{uqe} contains a notion of theory morphism. The reader is expected to be familiar with the notation of CTT_{uqe}. The following are additional notes to the reader:

1. $\{o, \epsilon, \iota\}$ is the set of base types for all eight biform theories given below. The single nonlogical base type ι is used to represent the natural numbers.
2. All constants that are not introduced as components of one of the biform theories listed below are logical constants of CTT_{uqe}, either primitive or defined. $\mathsf{is\text{-}abs}_{\epsilon\to o}$, $\mathsf{abs\text{-}body}_{\epsilon\to\epsilon}$, and $\mathsf{is\text{-}closed}_{\epsilon\to o}$ are defined logical constants not in [?]. $\mathsf{is\text{-}abs}_{\epsilon\to o}\,\mathbf{A}_\epsilon$ holds iff $\mathbf{A}_\epsilon$ represents an abstraction. If $\mathbf{A}_\epsilon$ represents an abstraction, then $\mathsf{abs\text{-}body}_{\epsilon\to\epsilon}\,\mathbf{A}_\epsilon$ represents the body of the abstraction. $\mathsf{is\text{-}closed}_{\epsilon\to o}\,\mathbf{A}_\epsilon$ holds iff $\mathbf{A}_\epsilon$ represents an expression that is closed (and eval-free).
3. The type attached to a constant may be dropped when there is no loss of meaning.
4. A constant of a type of the form $\epsilon \to \cdots \to \epsilon$ that is intended to be implemented by a transformer has a name in upper case letters.
5. Expressions of type ϵ, i.e., expressions that denote constructions, are colored red to identify where reasoning about syntax occurs.

The following are the eight biform theories and the two noninclusive theory morphisms in the test case:

Biform Theory 1 (BT1: Simple Theory of 0 and S)

Primitive Base Types

1. ι (type of natural numbers).

Primitive Constants

1. 0_ι.
2. $S_{\iota\to\iota}$.

Defined Constants (selected)

1. $1_\iota = S\,0_\iota$.

2. $\mathsf{IS\text{-}FO\text{-}BT1}_{\epsilon\to\epsilon} = \lambda\, x_\epsilon \,.\, \mathbf{B}_\epsilon$ where $\mathbf{B}_\epsilon$ is a complex expression such that $(\lambda\, x_\epsilon \,.\, \mathbf{B}_\epsilon)\ulcorner \mathbf{A}_\alpha \urcorner$ equals $\ulcorner T_o \urcorner$ [$\ulcorner F_o \urcorner$] if $\mathbf{A}_\alpha$ is [not] a term or formula of first-order logic with equality whose variables are of type ι and whose nonlogical constants are members of $\{0_\iota, S_{\iota\to\iota}\}$.

Axioms

1. $S\, x_\iota \neq 0_\iota$.
2. $S\, x_\iota = S\, y_\iota \supset x_\iota = y_\iota$.

Transformers

1. π_1 for $\mathsf{IS\text{-}FO\text{-}BT1}_{\epsilon\to\epsilon}$ is an efficient program that accesses the data stored in the data structures that represent expressions.
2. π_2 for $\mathsf{IS\text{-}FO\text{-}BT1}_{\epsilon\to\epsilon}$ uses the definition of $\mathsf{IS\text{-}FO\text{-}BT1}_{\epsilon\to\epsilon}$.

Biform Theory 2 (BT2: Simple Theory of 0, S, and $+$)

Extended Theories

1. BT1.

Primitive Constants

3. $+_{\iota\to\iota\to\iota}$ (infix).
4. $\mathsf{BPLUS}_{\epsilon\to\epsilon\to\epsilon}$ (infix).

Defined Constants (selected)

3. $\mathsf{bnat}_{\iota\to\iota\to\iota} = \lambda\, x_\iota \,.\, \lambda\, y_\iota \,.\, ((x_\iota + x_\iota) + y_\iota)$.
   Notational definition:
   $(0)_2 = \mathsf{bnat}\, 0_\iota\, 0_\iota$.
   $(1)_2 = \mathsf{bnat}\, 0_\iota\, 1_\iota$.
   $(a_1 \cdots a_n 0)_2 = \mathsf{bnat}\, (a_1 \cdots a_n)_2\, 0_\iota$ where each a_i is 0 or 1.
   $(a_1 \cdots a_n 1)_2 = \mathsf{bnat}\, (a_1 \cdots a_n)_2\, 1_\iota$ where each a_i is 0 or 1.
4. $\mathsf{is\text{-}bnum}_{\epsilon\to o} = \mathrm{I}\, f_{\epsilon\to o} \,.\, \forall\, u_\epsilon \,.\, (f_{\epsilon\to\epsilon}\, u_\epsilon \equiv$
   $\exists\, v_\epsilon \,.\, \exists\, w_\epsilon \,.\, (u_\epsilon = \ulcorner \mathsf{bnat}\, \lfloor v_\epsilon \rfloor\, \lfloor w_\epsilon \rfloor \urcorner \wedge$
   $(v_\epsilon = \ulcorner 0_\iota \urcorner \vee f_{\epsilon\to\epsilon}\, v_\epsilon) \wedge (w_\epsilon = \ulcorner 0_\iota \urcorner \vee w_\epsilon = \ulcorner 1_\iota \urcorner)))$.
5. $\mathsf{IS\text{-}FO\text{-}BT2}_{\epsilon\to\epsilon} = \lambda\, x_\epsilon \,.\, \mathbf{B}_\epsilon$ where $\mathbf{B}_\epsilon$ is a complex expression such that $(\lambda\, x_\epsilon \,.\, \mathbf{B}_\epsilon)\ulcorner \mathbf{A}_\alpha \urcorner$ equals $\ulcorner T_o \urcorner$ [$\ulcorner F_o \urcorner$] if $\mathbf{A}_\alpha$ is [not] a term or formula of first-order logic with equality whose variables are of type ι and whose nonlogical constants are members of $\{0_\iota, S_{\iota\to\iota}, +_{\iota\to\iota\to\iota}\}$.

Axioms

3. $x_\iota + 0_\iota = x_\iota$.
4. $x_\iota + S\, y_\iota = S\, (x_\iota + y_\iota)$.
5. $\mathsf{is\text{-}bnum}\, u_\epsilon \supset u_\epsilon\ \mathsf{BPLUS}\ \ulcorner (0)_2 \urcorner = u_\epsilon$.
6. $\mathsf{is\text{-}bnum}\, u_\epsilon \supset \ulcorner (0)_2 \urcorner\ \mathsf{BPLUS}\ u_\epsilon = u_\epsilon$.
7. $\ulcorner (1)_2 \urcorner\ \mathsf{BPLUS}\ \ulcorner (1)_2 \urcorner = \ulcorner (10)_2 \urcorner$.
8. $\mathsf{is\text{-}bnum}\, u_\epsilon \supset$
   $\ulcorner \mathsf{bnat}\, \lfloor u_\epsilon \rfloor\, 0_\iota \urcorner\ \mathsf{BPLUS}\ \ulcorner (1)_2 \urcorner = \ulcorner \mathsf{bnat}\, \lfloor u_\epsilon \rfloor\, 1_\iota \urcorner$.
9. $\mathsf{is\text{-}bnum}\, u_\epsilon \supset$
   $\ulcorner \mathsf{bnat}\, \lfloor u_\epsilon \rfloor\, 1_\iota \urcorner\ \mathsf{BPLUS}\ \ulcorner (1)_2 \urcorner = \ulcorner \mathsf{bnat}\, \lfloor u_\epsilon\ \mathsf{BPLUS}\ \ulcorner (1)_2 \urcorner \rfloor\, 0_\iota \urcorner$.
10. $\mathsf{is\text{-}bnum}\, u_\epsilon \supset$
    $\ulcorner (1)_2 \urcorner\ \mathsf{BPLUS}\ \ulcorner \mathsf{bnat}\, \lfloor u_\epsilon \rfloor\, 0_\iota \urcorner = \ulcorner \mathsf{bnat}\, \lfloor u_\epsilon \rfloor\, 1_\iota \urcorner$.

11. $\mathsf{is\text{-}bnum}\ u_\epsilon \supset$
    $\ulcorner(1)_2\urcorner\ \mathsf{BPLUS}\ \ulcorner\mathsf{bnat}\ \lfloor u_\epsilon\rfloor\ 0_\iota\urcorner = \ulcorner\mathsf{bnat}\ \lfloor u_\epsilon\ \mathsf{BPLUS}\ \ulcorner(1)_2\urcorner\rfloor\ 0_\iota\urcorner$.
12. $(\mathsf{is\text{-}bnum}\ u_\epsilon \wedge \mathsf{is\text{-}bnum}\ v_\epsilon) \supset$
    $\ulcorner\mathsf{bnat}\ \lfloor u_\epsilon\rfloor\ 0_\iota\urcorner\ \mathsf{BPLUS}\ \ulcorner\mathsf{bnat}\ \lfloor v_\epsilon\rfloor 0_\iota\urcorner = \ulcorner\mathsf{bnat}\ \lfloor u_\epsilon\ \mathsf{BPLUS}\ v_\epsilon\rfloor\ 0_\iota\urcorner$.
13. $(\mathsf{is\text{-}bnum}\ u_\epsilon \wedge \mathsf{is\text{-}bnum}\ v_\epsilon) \supset$
    $\ulcorner\mathsf{bnat}\ \lfloor u_\epsilon\rfloor\ 0_\iota\urcorner\ \mathsf{BPLUS}\ \ulcorner\mathsf{bnat}\ \lfloor v_\epsilon\rfloor\ 1_\iota\urcorner = \ulcorner\mathsf{bnat}\ \lfloor u_\epsilon\ \mathsf{BPLUS}\ v_\epsilon\rfloor\ 1_\iota\urcorner$.
14. $(\mathsf{is\text{-}bnum}\ u_\epsilon \wedge \mathsf{is\text{-}bnum}\ v_\epsilon) \supset$
    $\ulcorner\mathsf{bnat}\ \lfloor u_\epsilon\rfloor\ 1_\iota\urcorner\ \mathsf{BPLUS}\ \ulcorner\mathsf{bnat}\ \lfloor v_\epsilon\rfloor\ 0_\iota\urcorner = \ulcorner\mathsf{bnat}\ \lfloor u_\epsilon\ \mathsf{BPLUS}\ v_\epsilon\rfloor\ 1_\iota\urcorner$.
15. $(\mathsf{is\text{-}bnum}\ u_\epsilon \wedge \mathsf{is\text{-}bnum}\ v_\epsilon) \supset$
    $\ulcorner\mathsf{bnat}\ \lfloor u_\epsilon\rfloor\ 1_\iota\urcorner\ \mathsf{BPLUS}\ \ulcorner\mathsf{bnat}\ \lfloor v_\epsilon\rfloor\ 1_\iota\urcorner =$
    $\ulcorner\mathsf{bnat}\ \lfloor(u_\epsilon\ \mathsf{BPLUS}\ v_\epsilon)\ \mathsf{BPLUS}\ \ulcorner(1)_2\urcorner\rfloor\ 0_\iota\urcorner$.

Transformers

3. π_3 for $\mathsf{BPLUS}_{\epsilon\to\epsilon\to\epsilon}$ is an efficient program that satisfies Axioms 5–15.
4. π_4 for $\mathsf{BPLUS}_{\epsilon\to\epsilon\to\epsilon}$ uses Axioms 5–15 as conditional rewrite rules.
5. π_5 for $\mathsf{IS\text{-}FO\text{-}BT2}_{\epsilon\to\epsilon}$ is an efficient program that accesses the data stored in the data structures that represent expressions.
6. π_6 for $\mathsf{IS\text{-}FO\text{-}BT2}_{\epsilon\to\epsilon}$ uses the definition of $\mathsf{IS\text{-}FO\text{-}BT2}_{\epsilon\to\epsilon}$.

Theorems (selected)

1. Meaning formula schema for $\mathsf{BPLUS}_{\epsilon\to\epsilon\to\epsilon}$
   $((\mathsf{is\text{-}bnum}\ \mathbf{A}_\epsilon \wedge \mathsf{is\text{-}bnum}\ \mathbf{B}_\epsilon) \supset$
   $(\mathsf{is\text{-}bnum}\ (\mathbf{A}_\epsilon\ \mathsf{BPLUS}\ \mathbf{B}_\epsilon) \wedge$
   $([\![\mathbf{A}_\epsilon\ \mathsf{BPLUS}\ \mathbf{B}_\epsilon]\!]_\iota = [\![\mathbf{A}_\epsilon]\!]_\iota + [\![\mathbf{B}_\epsilon]\!]_\iota)))$.

Biform Theory 3 (BT3: Simple Theory of 0, S, $+$, and $*$)

Extended Theories

2. BT2.

Primitive Constants

5. $*_{\iota\to\iota\to\iota}$ (infix).
6. $\mathsf{BTIMES}_{\epsilon\to\epsilon\to\epsilon}$ (infix).

Defined Constants (selected)

4. $\mathsf{IS\text{-}FO\text{-}BT3}_{\epsilon\to\epsilon} = \lambda\, x_\epsilon\ .\ \mathbf{B}_\epsilon$ where $\mathbf{B}_\epsilon$ is a complex expression such that $(\lambda\, x_\epsilon\ .\ \mathbf{B}_\epsilon)\ulcorner\mathbf{A}_\alpha\urcorner$ equals $\ulcorner T_o\urcorner$ [$\ulcorner F_o\urcorner$] if $\mathbf{A}_\alpha$ is [not] a term or formula of first-order logic with equality whose variables are of type ι and whose nonlogical constants are members of $\{0_\iota, S_{\iota\to\iota}, +_{\iota\to\iota\to\iota}, *_{\iota\to\iota\to\iota}\}$.

Axioms

16. $x_\iota * 0_\iota = 0_\iota$.
17. $x_\iota * S\, y_\iota = (x_\iota * y_\iota) + x_\iota$.
18. $\mathsf{is\text{-}bnum}\ u_\epsilon \supset u_\epsilon\ \mathsf{BTIMES}\ \ulcorner(0)_2\urcorner = \ulcorner(0)_2\urcorner$.
19. $\mathsf{is\text{-}bnum}\ u_\epsilon \supset \ulcorner(0)_2\urcorner\ \mathsf{BTIMES}\ u_\epsilon = \ulcorner(0)_2\urcorner$.
20. $\mathsf{is\text{-}bnum}\ u_\epsilon \supset u_\epsilon\ \mathsf{BTIMES}\ \ulcorner(1)_2\urcorner = u_\epsilon$.
21. $\mathsf{is\text{-}bnum}\ u_\epsilon \supset \ulcorner(1)_2\urcorner\ \mathsf{BTIMES}\ u_\epsilon = u_\epsilon$.
22. $(\mathsf{is\text{-}bnum}\ u_\epsilon \wedge \mathsf{is\text{-}bnum}\ v_\epsilon) \supset$
    $\ulcorner\mathsf{bnat}\ \lfloor u_\epsilon\rfloor\ 0\urcorner\ \mathsf{BTIMES}\ v_\epsilon = \ulcorner\mathsf{bnat}\ \lfloor u_\epsilon\ \mathsf{BTIMES}\ v_\epsilon\rfloor\ 0_\iota\urcorner$.
23. $(\mathsf{is\text{-}bnum}\ u_\epsilon \wedge \mathsf{is\text{-}bnum}\ v_\epsilon) \supset$
    $\ulcorner\mathsf{bnat}\ \lfloor u_\epsilon\rfloor\ 1_\iota\urcorner\ \mathsf{BTIMES}\ v_\epsilon = \ulcorner\mathsf{bnat}\ \lfloor u_\epsilon\ \mathsf{BTIMES}\ v_\epsilon\rfloor\ 0_\iota\urcorner\ \mathsf{BPLUS}\ v_\epsilon$.

24. $(\mathsf{is\text{-}bnum}\ u_\epsilon \wedge \mathsf{is\text{-}bnum}\ v_\epsilon) \supset$
$v_\epsilon\ \mathsf{BTIMES}\ \ulcorner \mathsf{bnat}\ \lfloor u_\epsilon \rfloor\ 0_\iota \urcorner = \ulcorner \mathsf{bnat}\ \lfloor u_\epsilon\ \mathsf{BTIMES}\ v_\epsilon \rfloor\ 0_\iota \urcorner$.
25. $(\mathsf{is\text{-}bnum}\ u_\epsilon \wedge \mathsf{is\text{-}bnum}\ v_\epsilon) \supset$
$v_\epsilon\ \mathsf{BTIMES}\ \ulcorner \mathsf{bnat}\ \lfloor u_\epsilon \rfloor\ 1_\iota \urcorner = \ulcorner \mathsf{bnat}\ \lfloor u_\epsilon\ \mathsf{BTIMES}\ v_\epsilon \rfloor\ 0_\iota \urcorner\ \mathsf{BPLUS}\ v_\epsilon$.

Transformers

7. π_7 for $\mathsf{BTIMES}_{\epsilon\to\epsilon\to\epsilon}$ is an efficient program that satisfies Axioms 18–25.
8. π_8 for $\mathsf{BTIMES}_{\epsilon\to\epsilon\to\epsilon}$ uses Axioms 18–25 as conditional rewrite rules.
9. π_9 for $\mathsf{IS\text{-}FO\text{-}BT3}_{\epsilon\to\epsilon}$ is an efficient program that accesses the data stored in the data structures that represent expressions.
10. π_{10} for $\mathsf{IS\text{-}FO\text{-}BT3}_{\epsilon\to\epsilon}$ uses the definition of $\mathsf{IS\text{-}FO\text{-}BT3}_{\epsilon\to\epsilon}$.

Theorems (selected)

2. Meaning formula schema for $\mathsf{BTIMES}_{\epsilon\to\epsilon\to\epsilon}$
$((\mathsf{is\text{-}bnum}\ \mathbf{A}_\epsilon \wedge \mathsf{is\text{-}bnum}\ \mathbf{B}_\epsilon) \supset$
$(\mathsf{is\text{-}bnum}\ (\mathbf{A}_\epsilon\ \mathsf{BTIMES}\ \mathbf{B}_\epsilon) \wedge$
$([\![\mathbf{A}_\epsilon\ \mathsf{BTIMES}\ \mathbf{B}_\epsilon]\!]_\iota = [\![\mathbf{A}_\epsilon]\!]_\iota * [\![\mathbf{B}_\epsilon]\!]_\iota))))$.

Biform Theory 4 (BT4: Robinson Arithmetic (Q))

Extended Theories

3. BT3.

Axioms

26. $x_\iota = 0_\iota \vee \exists\, y_\iota\ .\ S\, y_\iota = x_\iota$.

Biform Theory 5 (BT5: Complete Theory of 0 and S)

Extended Theories

1. BT1.

Primitive Constants

7. $\mathsf{BT5\text{-}DEC\text{-}PROC}_{\epsilon\to\epsilon}$.

Defined Constants (selected)

6. $\mathsf{IS\text{-}FO\text{-}BT1\text{-}ABS}_{\epsilon\to\epsilon} =$
$\lambda\, x_\epsilon\ .\ (\mathsf{if}\ (\mathsf{is\text{-}abs}_{\epsilon\to o}\ x_\epsilon)\ (\mathsf{IS\text{-}FO\text{-}BT1}_{\epsilon\to\epsilon}\ (\mathsf{abs\text{-}body}_{\epsilon\to\epsilon}\ x_\epsilon))\ \ulcorner F_o \urcorner)$.

Axioms

27. Induction Schema for S
$\forall\, f_\epsilon\ .\ ((\mathsf{is\text{-}expr}^{\iota\to o}_{\epsilon\to o}\ f_\epsilon \wedge [\![\mathsf{IS\text{-}FO\text{-}BT1\text{-}ABS}_{\epsilon\to\epsilon}\ f_\epsilon]\!]_o) \supset$
$(([\![f_\epsilon]\!]_{\iota\to o}\ 0_\iota \wedge (\forall\, x_\iota\ .\ [\![f_\epsilon]\!]_{\iota\to o}\ x_\iota \supset [\![f_\epsilon]\!]_{\iota\to o}\ (\mathsf{S}\, x_\iota))) \supset \forall\, x_\iota\ .\ [\![f_\epsilon]\!]_{\iota\to o}\ x_\iota))$.
28. Meaning Formula for $\mathsf{BT5\text{-}DEC\text{-}PROC}_{\epsilon\to\epsilon}$
$\forall\, u_\epsilon\ .\ ((\mathsf{is\text{-}expr}^{o}_{\epsilon\to o}\ u_\epsilon \wedge \mathsf{is\text{-}closed}_{\epsilon\to o}\ u_\epsilon \wedge [\![\mathsf{IS\text{-}FO\text{-}BT1}_{\epsilon\to\epsilon}\ u_\epsilon]\!]_o) \supset$
$((\mathsf{BT5\text{-}DEC\text{-}PROC}_{\epsilon\to\epsilon}\ u_\epsilon = \ulcorner T_o \urcorner \vee \mathsf{BT5\text{-}DEC\text{-}PROC}_{\epsilon\to\epsilon}\ u_\epsilon = \ulcorner F_o \urcorner) \wedge$
$[\![\mathsf{BT5\text{-}DEC\text{-}PROC}_{\epsilon\to\epsilon}\ u_\epsilon]\!]_o = [\![u_\epsilon]\!]_o))$.

Transformers

11. π_{11} for $\mathsf{BT5\text{-}DEC\text{-}PROC}_{\epsilon\to\epsilon\to\epsilon}$ is an efficient decision procedure that satisfies Axiom 28.
12. π_{12} for $\mathsf{IS\text{-}FO\text{-}BT1\text{-}ABS}_{\epsilon\to\epsilon}$ is an efficient program that accesses the data stored in the data structures that represent expressions.
13. π_{13} for $\mathsf{IS\text{-}FO\text{-}BT1\text{-}ABS}_{\epsilon\to\epsilon}$ uses the definition of $\mathsf{IS\text{-}FO\text{-}BT1\text{-}ABS}_{\epsilon\to\epsilon}$.

Biform Theory 6 (BT6: Presburger Arithmetic)

Extended Theories

1. BT2.
5. BT5.

Primitive Constants

8. $\mathsf{BT6\text{-}DEC\text{-}PROC}_{\epsilon\to\epsilon}$.

Defined Constants (selected)

7. $\mathsf{IS\text{-}FO\text{-}BT2\text{-}ABS}_{\epsilon\to\epsilon} =$
   $\lambda\, x_\epsilon \,.\, (\mathsf{if}\ (\mathsf{is\text{-}abs}_{\epsilon\to o}\, x_\epsilon)\ (\mathsf{IS\text{-}FO\text{-}BT2}_{\epsilon\to\epsilon}\, (\mathsf{abs\text{-}body}_{\epsilon\to\epsilon}\, x_\epsilon))\ \ulcorner F_o \urcorner)$.

Axioms

29. Induction Schema for S and $+$
    $\forall\, f_\epsilon \,.\, ((\mathsf{is\text{-}expr}^{\iota\to o}_{\epsilon\to o}\, f_\epsilon \wedge [\![\mathsf{IS\text{-}FO\text{-}BT2\text{-}ABS}_{\epsilon\to\epsilon}\, f_\epsilon]\!]_o) \supset$
    $((([\![f_\epsilon]\!]_{\iota\to o}\, 0_\iota \wedge (\forall\, x_\iota \,.\, [\![f_\epsilon]\!]_{\iota\to o}\, x_\iota \supset [\![f_\epsilon]\!]_{\iota\to o}\, (\mathsf{S}\, x_\iota))) \supset \forall\, x_\iota \,.\, [\![f_\epsilon]\!]_{\iota\to o}\, x_\iota))$.
30. Meaning formula for $\mathsf{BT6\text{-}DEC\text{-}PROC}_{\epsilon\to\epsilon}$.
    $\forall\, u_\epsilon \,.\, ((\mathsf{is\text{-}expr}^{o}_{\epsilon\to o}\, u_\epsilon \wedge \mathsf{is\text{-}closed}_{\epsilon\to o}\, u_\epsilon \wedge [\![\mathsf{IS\text{-}FO\text{-}BT2}_{\epsilon\to\epsilon}\, u_\epsilon]\!]_o) \supset$
    $((\mathsf{BT6\text{-}DEC\text{-}PROC}_{\epsilon\to\epsilon}\, u_\epsilon = \ulcorner T_o \urcorner \vee \mathsf{BT6\text{-}DEC\text{-}PROC}_{\epsilon\to\epsilon}\, u_\epsilon = \ulcorner F_o \urcorner) \wedge$
    $[\![\mathsf{BT6\text{-}DEC\text{-}PROC}_{\epsilon\to\epsilon}\, u_\epsilon]\!]_o = [\![u_\epsilon]\!]_o))$.

Transformers

14. π_{14} for $\mathsf{BT6\text{-}DEC\text{-}PROC}_{\epsilon\to\epsilon\to\epsilon}$ is an efficient decision procedure that satisfies Axiom 30.
15. π_{15} for $\mathsf{IS\text{-}FO\text{-}BT2\text{-}ABS}_{\epsilon\to\epsilon}$ is an efficient program that accesses the data stored in the data structures that represent expressions.
16. π_{16} for $\mathsf{IS\text{-}FO\text{-}BT2\text{-}ABS}_{\epsilon\to\epsilon}$ uses the definition of $\mathsf{IS\text{-}FO\text{-}BT2\text{-}ABS}_{\epsilon\to\epsilon}$.

Theorems (selected)

3. Meaning formula for $\mathsf{BPLUS}_{\epsilon\to\epsilon\to\epsilon}$
   $\forall\, u_\epsilon \,.\, \forall\, v_\epsilon \,.\, ((\mathsf{is\text{-}bnum}\, u_\epsilon \wedge \mathsf{is\text{-}bnum}\, v_\epsilon) \supset$
   $(\mathsf{is\text{-}bnum}\, (u_\epsilon\ \mathsf{BPLUS}\ v_\epsilon) \wedge$
   $([\![u_\epsilon\ \mathsf{BPLUS}\ v_\epsilon]\!]_\iota = [\![u_\epsilon]\!]_\iota + [\![v_\epsilon]\!]_\iota)))$.

Biform Theory 7 (BT7: First-Order Peano Arithmetic)

Extended Theories

3. BT3.
6. BT6.

Defined Constants (selected)

8. $\mathsf{IS\text{-}FO\text{-}BT3\text{-}ABS}_{\epsilon\to\epsilon} =$
   $\lambda\, x_\epsilon \,.\, (\mathsf{if}\ (\mathsf{is\text{-}abs}_{\epsilon\to o}\, x_\epsilon)\ (\mathsf{IS\text{-}FO\text{-}BT3}_{\epsilon\to\epsilon}\, (\mathsf{abs\text{-}body}_{\epsilon\to\epsilon}\, x_\epsilon))\ \ulcorner F_o \urcorner)$.

Axioms

31. Induction Schema for S, $+$, and $*$
    $\forall\, f_\epsilon \,.\, ((\mathsf{is\text{-}expr}^{\iota\to o}_{\epsilon\to o}\, f_\epsilon \wedge [\![\mathsf{IS\text{-}FO\text{-}BT3\text{-}ABS}_{\epsilon\to\epsilon}\, f_\epsilon]\!]_o) \supset$
    $((([\![f_\epsilon]\!]_{\iota\to o}\, 0_\iota \wedge (\forall\, x_\iota \,.\, [\![f_\epsilon]\!]_{\iota\to o}\, x_\iota \supset [\![f_\epsilon]\!]_{\iota\to o}\, (\mathsf{S}\, x_\iota))) \supset \forall\, x_\iota \,.\, [\![f_\epsilon]\!]_{\iota\to o}\, x_\iota))$.

Transformers

17. π_{17} for $\mathsf{IS\text{-}FO\text{-}BT3\text{-}ABS}_{\epsilon\to\epsilon}$ is an efficient program that accesses the data stored in the data structures that represent expressions.
18. π_{18} for $\mathsf{IS\text{-}FO\text{-}BT3\text{-}ABS}_{\epsilon\to\epsilon}$ uses the definition of $\mathsf{IS\text{-}FO\text{-}BT3\text{-}ABS}_{\epsilon\to\epsilon}$.

Theorems (selected)

4. Axiom 26.
5. Meaning formula $\mathsf{BTIMES}_{\epsilon\to\epsilon\to\epsilon}$
$\forall\, u_\epsilon \,.\, \forall\, v_\epsilon \,.\, ((\mathsf{is\text{-}bnum}\, u_\epsilon \wedge \mathsf{is\text{-}bnum}\, v_\epsilon) \supset$
$(\mathsf{is\text{-}bnum}\, (u_\epsilon\ \mathsf{BTIMES}\ v_\epsilon) \wedge$
$([\![u_\epsilon\ \mathsf{BTIMES}\ v_\epsilon]\!]_\iota = [\![u_\epsilon]\!]_\iota * [\![v_\epsilon]\!]_\iota))).$

Biform Theory 8 (BT8: Higher-Order Peano Arithmetic)

Extended Theories

1. BT1.

Defined Constants (selected)

9. $+_{\iota\to\iota\to\iota} = \mathrm{I}\, f_{\iota\to\iota\to\iota} \,.\, \forall\, x_\iota \,.\, \forall\, y_\iota \,.$
$(f_{\iota\to\iota\to\iota}\, x_\iota\, 0_\iota = x_\iota \wedge$
$f_{\iota\to\iota\to\iota}\, x_\iota\, (S\, y_\iota) = S\, (f_{\iota\to\iota\to\iota}\, x_\iota\, y_\iota)).$
10. $*_{\iota\to\iota\to\iota} = \mathrm{I}\, f_{\iota\to\iota\to\iota} \,.\, \forall\, x_\iota \,.\, \forall\, y_\iota \,.$
$(f_{\iota\to\iota\to\iota}\, x_\iota\, 0_\iota = 0_\iota \wedge$
$f_{\iota\to\iota\to\iota}\, x_\iota\, (S\, y_\iota) = (f_{\iota\to\iota\to\iota}\, x_\iota\, y_\iota) + x_\iota).$

Axioms

32. Induction Axiom for the Natural Numbers
$\forall\, p_{\iota\to o} \,.\, ((p_{\iota\to o}\, 0_\iota \wedge (\forall\, x_\iota \,.\, (p_{\iota\to o}\, x_\iota \supset p_{\iota\to o}\, (S\, x_\iota)))) \supset$
$\forall\, x_\iota \,.\, p_{\iota\to o}\, x_\iota).$

Theorems (selected)

6. Axiom 27 (Induction Schema for S).
7. Axiom 39 (Induction Schema for S and $+$).
8. Axiom 31 (Induction Schema for S, $+$, and $*$).

Theory Morphism 1 (BT4-to-BT7)

Source Theory BT4.

Target Theory BT7.

Translation

μ is defined as follows:

$\mu(o) = \lambda\, x_o \,.\, T_o.$
$\mu(\epsilon) = \lambda\, x_\epsilon \,.\, T_o.$
$\mu(\iota) = \lambda\, x_\iota \,.\, T_o.$

ν is the identity on the constants of BT4.

Nontrivial Obligations

Axiom 26.

Theory Morphism 2 (BT7-to-BT8)

Source Theory BT7.

Target Theory BT8.

Translation

μ is defined as follows:

$\mu(o) = \lambda\, x_o \,.\, T_o.$
$\mu(\epsilon) = \lambda\, x_\epsilon \,.\, T_o.$
$\mu(\iota) = \lambda\, x_\iota \,.\, T_o.$

ν is the identity on the constants of BT7.

Nontrivial Obligations

Axioms 3–25, 27–31.

B Agda Formalization

First, some infrastructure, then the theories themselves.

B.1 Definite Description

This defines the tools needed to do the equivalent of definite description in MLTT.

```
module DefiniteDescr where
open import Relation.Binary.PropositionalEquality using (_≡_)
open import Data.Product using (Σ; _×_)
```

Normal contractability of a type

```
isContr : Set₀ → Set₀
isContr A = Σ A (λ a → ∀ b → a ≡ b)
```

We now "expand out" that definition when A is a Σ-type with the first type that of a 2-argument function, and the second is a predicate (and thus automatically contractible). We could elide the second part via proof irrelevance.

```
isContr₂ : (B : Set₀) → let bin = B → B → B in (bin → Set₀) → Set₀
isContr₂ A P =
  let bin = A → A → A in
  Σ bin (λ f → P f ×
    (∀ (g : A → A → A) → P g → ∀ a b → f a b ≡ g a b) ×
    (∀ (pf₁ pf₂ : P f) → pf₁ ≡ pf₂))
```

B.2 Equivalences of Types

```
{-# OPTIONS --without-K #-}

module Equiv where

open import Level using (_⊔_)
open import Function using (_∘_; id; flip)
open import Relation.Binary using (IsEquivalence)
open import Relation.Binary.PropositionalEquality
  using (_≡_; refl; sym; trans; cong; cong₂; module ≡-Reasoning)

infix 4 _∼_
infix 4 _≃_
infixr 5 _•_

---------------------------------------
- Extensional equivalence of (unary) functions

_∼_ : ∀ {ℓ ℓ'} → {A : Set ℓ} {B : Set ℓ'} → (f g : A → B) → Set (ℓ ⊔ ℓ')
_∼_ {A = A} f g = (x : A) → f x ≡ g x
```

```
refl∼ : ∀ {ℓ ℓ'} {A : Set ℓ} {B : Set ℓ'} {f : A → B} → (f ∼ f)
refl∼ _ = refl

sym∼ : ∀ {ℓ ℓ'} {A : Set ℓ} {B : Set ℓ'} {f g : A → B} → (f ∼ g) → (g ∼ f)
sym∼ H x = sym (H x)

trans∼ : ∀ {ℓ ℓ'} {A : Set ℓ} {B : Set ℓ'} {f g h : A → B} → (f ∼ g) → (g ∼ h) → (f ∼ h)
trans∼ H G x = trans (H x) (G x)

∘-resp-∼ : ∀ {ℓA ℓB ℓC} {A : Set ℓA} {B : Set ℓB} {C : Set ℓC} {f h : B → C} {g i : A → B} →
  (f ∼ h) → (g ∼ i) → f ∘ g ∼ h ∘ i
∘-resp-∼ {f = f} {i = i} f∼h g∼i x = trans (cong f (g∼i x)) (f∼h (i x))

isEquivalence∼ : ∀ {ℓ ℓ'} {A : Set ℓ} {B : Set ℓ'} → IsEquivalence (_∼_ {ℓ} {ℓ'} {A} {B})
isEquivalence∼ = record { refl = refl∼ ; sym = sym∼ ; trans = trans∼ }

--------------------------------------
- Quasi-equivalences, in a more useful packaging

record _≃_ {ℓ ℓ'} (A : Set ℓ) (B : Set ℓ') : Set (ℓ ⊔ ℓ') where
  constructor qeq
  field
    f : A → B
    g : B → A
    α : (f ∘ g) ∼ id
    β : (g ∘ f) ∼ id
    - to make it contractible, could add
    - τ : ∀ x → cong f (β x) P.≡ α (f x)

id≃ : ∀ {ℓ} {A : Set ℓ} → A ≃ A
id≃ = qeq id id (λ _ → refl) (λ _ → refl)

sym≃ : ∀ {ℓ ℓ'} {A : Set ℓ} {B : Set ℓ'} → (A ≃ B) → B ≃ A
sym≃ (qeq f g α β) = qeq g f β α

trans≃ : ∀ {ℓ ℓ' ℓ''} {A : Set ℓ} {B : Set ℓ'} {C : Set ℓ''} →
  A ≃ B → B ≃ C → A ≃ C
trans≃ {A = A} {B} {C} (qeq f f⁻¹ fα fβ) (qeq g g⁻¹ gα gβ) =
  qeq (g ∘ f) (f⁻¹ ∘ g⁻¹) (λ x → trans (cong g (fα (g⁻¹ x))) (gα x))
    (λ x → trans (cong f⁻¹ (gβ (f x))) (fβ x))

- more convenient infix version, flipped

_•_ : ∀ {ℓ ℓ' ℓ''} {A : Set ℓ} {B : Set ℓ'} {C : Set ℓ''} →
  B ≃ C → A ≃ B → A ≃ C
```

```
_•_ = flip trans≃

≃IsEquiv : IsEquivalence {Level.suc Level.zero} {Level.zero} {Set} _≃_
≃IsEquiv = record { refl = id≃ ; sym = sym≃ ; trans = trans≃ }

- equivalences are injective

inj≃ : ∀ {ℓ ℓ'} {A : Set ℓ} {B : Set ℓ'} →
   (eq : A ≃ B) → (x y : A) → (_≃_.f eq x ≡ _≃_.f eq y → x ≡ y)
inj≃ (qeq f g α β) x y p = trans
   (sym (β x)) (trans
   (cong g p) (
   β y))
```

B.3 Numerals

```
    module Numerals where
```

Rather that defining our own isomorphic copy, re-use ℕ and Vec.

```
    open import Data.Nat using (ℕ; suc)
    open import Data.Vec using (_::_; []; Vec)
```

plus is a *function* on ℕ, which will (of course) implement addition. Note that this is not a 'good' function, in the sense that it is extremely inefficient. it does have the advantage of being simple and facilitate proofs. Note that it is defined (on purpose) by recursion on the left argument, while the properties in T2 turn out to be "recursive" on the right.

```
    plus : ℕ → ℕ → ℕ
    plus 0 y = y
    plus (suc x) y = suc (plus x y)
```

We represent numerals as vectors (of length at least 1) of binary digits.

```
    data BinDigit : Set where zero one : BinDigit
    data BNum : Set where
      bn : {n : ℕ} → Vec BinDigit (suc n) → BNum
```

This then allows a straightforward implementation of bplus to add numerals. It is then possible to *prove* that the meaning function for bplus is a theorem.

Convenient abbreviations

```
    0b 1b 2b : BNum
    0b = bn (zero :: [])
    1b = bn (one :: [])
    2b = bn (zero :: one :: [])

    « : BNum → BNum
    « (bn l) = bn (zero :: l)
```

Note how +1 is defined by induction on BNum.

```
    +1 : BNum → BNum
    +1 (bn (zero :: l)) = bn (one :: l)
```

```
+1 (bn (one :: [])) = bn (zero :: one :: [])
+1 (bn (one :: x :: l)) = « (+1 (bn (x :: l)))
```

Now we want to define a transformer on BNum with a "meaning formula" that says that it is addition. this too is defined by induction on BNum bplus is essentially ξ_3. It is not the only such transformer, as different representations could be even more efficient. It is also kind of ξ_4 ! What happens it that all the conditions are all vacuously true. So the axioms are then just a bunch of pattern-match. Turns out that the rules below are more 'syntax directed' than the ones given in the axioms.

```
bplus : BNum → BNum → BNum
bplus (bn {0} (zero :: [])) y = y
bplus (bn {0} (one :: [])) y = +1 y
bplus (bn {suc n} (d₀ :: l₀)) (bn {ℕ.zero} (zero :: [])) = bn (d₀ :: l₀)
bplus (bn {suc n} (d₀ :: l₀)) (bn {ℕ.zero} (one :: [])) = +1 (bn (d₀ :: l₀))
bplus (bn {suc n} (zero :: l₀)) (bn {suc m} (zero :: l₁)) =
      « (bplus (bn l₀) (bn l₁))
bplus (bn {suc n} (one :: l₀)) (bn {suc m} (zero :: l₁)) =
   +1 (« (bplus (bn l₀) (bn l₁)))
bplus (bn {suc n} (zero :: l₀)) (bn {suc m} (one :: l₁)) =
   +1 (« (bplus (bn l₀) (bn l₁)))
bplus (bn {suc n} (one :: l₀)) (bn {suc m} (one :: l₁)) =
   +1 (+1 (« (bplus (bn l₀) (bn l₁))))
```

Important: because BNum is a type, there is no need for is-bnum, as it is simply true by construction. However, here BNum is an explicit representation (as a Vector of digits) whereas in CTT_{uqe} it is done as a 'recognizer of expressions' which picksout expressions which are made up of sequences of digits. The sequencing is buried (as a right-leaning tree) in a bunch of 'bnat' calls. If we're going to go through codes to represent things, may as well use codes which are specially built for the task!

Of course, we will need an interpretation of BNum in nat, but that will be done inside T2.

For btimes, rather than be axiomatic, we go directly to a transformer.

```
_btimes_ : BNum → BNum → BNum
x btimes bn (zero :: []) = 0b
x btimes bn (one :: []) = x
bn (zero :: []) btimes bn (x₂ :: x₃ :: x₄) = 0b
bn (one :: []) btimes bn (x₂ :: x₃ :: x₄) = bn (x₂ :: x₃ :: x₄)
bn (zero :: x₁ :: x₂) btimes bn (x₃ :: x₄ :: x₅) =
      « (bn (x₁ :: x₂) btimes bn (x₃ :: x₄ :: x₅))
bn (one :: x₁ :: x₂) btimes bn (x₃ :: x₄ :: x₅) =
   let y = bn (x₃ :: x₄ :: x₅) in
   bplus (« (bn (x₁ :: x₂) btimes y)) y
```

B.4 NumPlus

```
module NumPlus where

data ℕ+ : Set₀ where
  z+ : ℕ+
  s+ : ℕ+ → ℕ+
  _`+_ : ℕ+ → ℕ+ → ℕ+

data ℕ+X (V : Set₀) : Set₀ where
  z : ℕ+X V
  s : ℕ+X V → ℕ+X V
  _`+_ : ℕ+X V → ℕ+X V → ℕ+X V
  v : V → ℕ+X V
```

B.5 NumPlusTimes

```
module NumPlusTimes where

data ℕ* (V : Set₀) : Set₀ where
  z : ℕ* V
  s : ℕ* V → ℕ* V
  _`+_ : ℕ* V → ℕ* V → ℕ* V
  _`*_ : ℕ* V → ℕ* V → ℕ* V
  v : V → ℕ* V
```

B.6 Naturals as variables, and with variables

Showing that ℕ has decidable equality (and thus can be used as a set of variables). Also build ℕX which is ℕ augmented with variables.

```
module NatVar where

open import Level renaming (zero to lzero) hiding (suc)
open import Data.Nat using (ℕ; zero; suc; _≟_)
open import Equiv
open import Data.Empty using (⊥)
open import Function using (_∘_; id)
open import Relation.Binary.PropositionalEquality
  using (_≡_; refl; cong; isEquivalence)
open import Relation.Binary using (DecSetoid)

private
  DT : Set₁
  DT = DecSetoid lzero lzero

ℕS : DT
```

```
ℕS = record
  { Carrier = ℕ
  ; _≈_ = _≡_
  ; isDecEquivalence = record
    { isEquivalence = isEquivalence
    ; _≟_ = _≟_ } }
data ℕX (Var : Set₀) : Set₀ where
  z : ℕX Var
  s : ℕX Var → ℕX Var
  v : Var → ℕX Var
ℕ⊥≃ℕ : ℕX ⊥ ≃ ℕ
ℕ⊥≃ℕ = qeq f g f∘g∼id g∘f∼id
  where
    f : ℕX ⊥ → ℕ
    f z = 0
    f (s x) = suc (f x)
    f (v ())
    g : ℕ → ℕX ⊥
    g zero = z
    g (suc n) = s (g n)
    f∘g∼id : f ∘ g ∼ id
    f∘g∼id zero = refl
    f∘g∼id (suc x) = cong suc (f∘g∼id x)
    g∘f∼id : g ∘ f ∼ id
    g∘f∼id z = refl
    g∘f∼id (s x) = cong s (g∘f∼id x)
    g∘f∼id (v ())
```

B.7 Language infrastructure

Note that much of this code is in the main paper already.

```
module Language where

open import Level using (Level; zero; suc; _⊔_)
open import Relation.Binary using (DecSetoid)
open import Relation.Nullary using (Dec; yes; no; ¬_)
open import Data.Bool using (Bool; true; false; _∧_; _∨_; _xor_)
  renaming (not to bnot; _≟_ to _=𝔹_)
open import Relation.Binary.PropositionalEquality
  using (_≡_; refl; sym; trans; cong₂)
open import Data.Empty using (⊥)
open import Data.Unit using (⊤)
open import Data.Product using (Σ; _×_; proj₁; proj₂; _,_)
open import Data.Sum using (_⊎_)
open import Data.List using (List; []; _::_; [_])
```

```
open import Variables

private
  DT : Set (suc zero)
  DT = DecSetoid zero zero
```

One of the important concepts is that of a *language with variables*, in other words a language with a reasonable definition of substitution. This requires *variables* to come from a type which has the structure of a decidable setoid (from the Agda library DecSetoid, and denoted DT below).

A language, expressed as an inductive type, is closed, i.e., cannot be extended. If a language does not have variables, we cannot add them. One solution is to deal with *contexts* as first-class citizens. While that is likely the best long-term solution, here we have gone with something simpler: create another language which does, and show that its variable-free fragment is equivalent to the original. As that aspect of our development is straightforward, albeit tedious, we elide it.

As we are concerned with statements in first-order logic over a variety of languages, it makes sense to modularize this aspect somewhat. Note that, as we are building syntax via inductive types, we can either build these as functors and then use a fixpoint combinator to tie the knot, or we can just bite the bullet and make one large definition. For now, we chose the latter. We do parametrize over a *ground language with variables*. In turn, this is defined as a type parametrized by a decidable setoid along with an evaluation function into some type $\top$.

```
record GroundLanguage (T : Set₀) : Set₁ where
  open DecSetoid using (Carrier)
  field
    Lang : DT → Set₀
    value : {V : DT} → Lang V → (Carrier V → T) → T
```

A logic over a language (with variables), is then also a parametrized type as well as a parametrized interpretation into types. The definition is almost the same, except that a ground language interprets into $\top$ and a logic into Set_0.

```
record LogicOverL (T : Set₀) (L : GroundLanguage T) : Set₁ where
  open DecSetoid using (Carrier)
  field
    Logic : DT → Set₀
    ⟦_⟧_ : ∀ {V} → Logic V → (Carrier V → T) → Set₀
```

The definition of first-order logic is then straightforward.

```
module FOL {T : Set₀} (L : GroundLanguage T) where
  open DecSetoid using (Carrier)
  open GroundLanguage L

  data FOL (V : DT) : Set where
    tt : FOL V
    ff : FOL V
    _and_ : FOL V → FOL V → FOL V
```

```
    _or_ : FOL V → FOL V → FOL V
    not : FOL V → FOL V
    _⊃_ : FOL V → FOL V → FOL V
    _==_ : Lang V → Lang V → FOL V
    all : Carrier V → FOL V → FOL V
    exist : Carrier V → FOL V → FOL V

override : {V : DT} → (Carrier V → T) → Carrier V → T → (Carrier V → T)
override {V} f x t y with DecSetoid._≟_ V y x
... | yes _ = t
... | no _  = f y
```

We can also prove that FOL is a logic over L by providing an interpretation. Of course, as we are modeling classical logic into a constructive logic, we have to use a double-negation embedding. We also choose to interpret the logic's equality `_==_` as *propositional equality*, but we could make that choice a parameter as well.

```
LoL-FOL : LogicOverL T L
LoL-FOL = record { Logic = FOL ; ⟦_⟧_ = interp }
  where
    interp : {Var : DT} → FOL Var → (Carrier Var → T) → Set₀
    interp tt env = ⊤
    interp ff env = ⊥
    interp (e and f) env = interp e env × interp f env
    interp (e or f) env = ¬ ¬ (interp e env ⊎ interp f env)
    interp (not e) env = ¬ (interp e env)
    interp (e ⊃ f) env = (interp e env) → (interp f env)
    interp (x == y) env = value x env ≡ value y env
    interp {V} (all x p) env  = ∀ z → interp p (override {V} env x z)
    interp {V} (exist x p) env = ¬ ¬ (Σ T (λ t → interp p (override {V} env x t)))
```

B.8 Some languages of variables

```
module Variables where

open import Level using (Level; zero; suc)
open import Relation.Binary using (DecSetoid)
open import Relation.Nullary using (Dec; yes; no)
open import Data.Bool using (Bool) renaming (_≟_ to _=𝔹_)
open import Relation.Binary.PropositionalEquality
  using (_≡_; refl; sym; trans)
open import Data.Empty using (⊥)
open import Data.Unit using (⊤; tt)

private
  DT : Set (suc zero)
```

```
  DT = DecSetoid zero zero

NoVars : DT
NoVars = record
  { Carrier = ⊥
  ; _≈_ = λ _ _ → ⊤
  ; isDecEquivalence = record
    { isEquivalence = record { refl = tt ; sym = λ _ → tt ; trans = λ _ _ → tt }
    ; _≟_ = λ () } }

DBool : DT
DBool = record
  { Carrier = Bool
  ; _≈_ = _≡_
  ; isDecEquivalence = record
    { isEquivalence = record { refl = refl ; sym = sym ; trans = trans }
    ; _≟_ = _≟𝔹_ } }

- For convenience, some simple "languages of variables"
module VarLangs where
  data X : Set₀ where x : X
  XV : DT
  XV = record
    { Carrier = X
    ; _≈_ = λ _ _ → ⊤
    ; isDecEquivalence = record
      { isEquivalence = record
        { refl = tt
        ; sym = λ _ → tt
        ; trans = λ _ _ → tt }
      ; _≟_ = λ _ _ → yes tt } }
```

B.9 T1

```
- The encoding uses the 'local method'.
module T1 where

open import Relation.Binary using (DecSetoid)
open DecSetoid using (Carrier)
open import Level using () renaming (zero to lzero)

- we use ⊥, ¬ and ≡ from the 'meta' logic
open import Data.Empty using (⊥)
open import Relation.Nullary using (¬_)
open import Relation.Binary.PropositionalEquality
```

```
    using (_≡_; refl)
open import Data.Nat using (ℕ; suc) - instead of defining our own
    - isomorphic copy
open import Data.Product using (Σ; _,_; proj₁; proj₂)
open import Data.List using ([_])
open import Data.Bool using (false)

- we will eventually need this
open import Language
open import NatVar

private
  DT = DecSetoid lzero lzero

record BT₁ : Set₁ where
  field
    nat : Set₀
    Z : nat
    S : nat → nat
    S≠Z : ∀ x → ¬ (S x ≡ Z)
    inj : ∀ x y → S x ≡ S y → x ≡ y

  One : nat
  One = S Z
```

where we see a field nat for the new type (pronounced Set in Agda), the two constants, and the two axioms. The host logic is dependently typed, and so the axioms refer to the constants just defined. One is not a field, but a defined constant.

For simplicity, we will take the built-in type ℕ, defined as an inductive type, as the *syntax* for natural numbers, which is also the syntax associated to the theory BT_1. Whereas in $\mathrm{CTT}_{\mathrm{uqe}}$ there is a global evaluation, here we also need to define evaluation explicitly (a subscript is used to indicate which theory it belongs to).

```
  ⟦_⟧₁ : ℕ → nat
  ⟦ 0 ⟧₁ = Z
  ⟦ suc x ⟧₁ = S ⟦ x ⟧₁
  - and some coherence theorems:
  pres-S≠Z : (x : ℕ) → ¬ ⟦ suc x ⟧₁ ≡ ⟦ 0 ⟧₁
  pres-S≠Z x = S≠Z ⟦ x ⟧₁

  pres-inj : (x y : ℕ) → S ⟦ x ⟧₁ ≡ S ⟦ y ⟧₁ → ⟦ x ⟧₁ ≡ ⟦ y ⟧₁
  pres-inj x y pf = inj ⟦ x ⟧₁ ⟦ y ⟧₁ pf
```

The accompanying code furthermore proves some basic coherence theorems which are elided here. We make two further definitions (GroundLanguage describing

some language features, and FOL as our definition of first order logic) which will be explained in more detail on the next page.

```
nat-lang : GroundLanguage nat
nat-lang = record { Lang = λ X → ℕX (Carrier X)
                  ; value = λ {V} → val {V} }
where
  val : {V : DT} → ℕX (Carrier V) → (Carrier V → nat) → nat
  val  z   env = Z
  val {V} (s e) env = S (val {V} e env)
  val  (v x) env = env x

module fo₁ = FOL nat-lang
```

We can also demonstrate that the natural numbers are a model:

```
ℕ-is-T1 : BT₁
ℕ-is-T1 = record { nat = ℕ ; Z = 0 ; S = suc
  ; S≠Z = λ x → λ () ; inj = λ { x .x refl → refl } }

- inverse of the type of sub-term of ℕX
SubTermType : {V : DT} → ℕX (Carrier V) → Set₀
SubTermType {_} z = ⊥
SubTermType {V} (s x) = ℕX (Carrier V)
SubTermType {V} (v x) = Carrier V

- paths in a ℕX
data Path {V : DT} : (e : ℕX (Carrier V)) → SubTermType {V} e → Set₀ where
  sp : (x : ℕX (Carrier V)) → Path (s x) x
  vp : (x : Carrier V) → Path (v x) x
```

B.10 T2

```
module T2 where
open import T1 using (BT₁)
open import Numerals
open import NumPlus
open import NatVar using (ℕX)

open import Relation.Binary using (DecSetoid)
open DecSetoid using (Carrier)
open import Level using () renaming (zero to lzero)

open import Relation.Binary.PropositionalEquality
  using (_≡_; refl; trans; cong; sym; cong₂)
open import Data.Nat using (ℕ; suc) - instead of defining our own
  - isomorphic copy
open import Data.Vec using (_::_; []; Vec)
```

```
open import Language using (GroundLanguage; module FOL)
open GroundLanguage using (value)

private
  DT = DecSetoid lzero lzero
---------------------------------

record BT₂ (t1 : BT₁) : Set where
  open BT₁ t1 public
  field
    _+_ : nat → nat → nat
    right-0 : ∀ x → x + Z ≡ x
    x+Sy≡Sx+y : ∀ x y → x + S y ≡ S (x + y)
    - Wikipedia lists
    - y ≡ 0 ∪ Σ ℕ (λ x → S x ≡ y)
    - as an additional axiom. It allows addition
    - to be defined recursively.

  bnat : nat → nat → nat
  bnat x y = (x + x) + y

  - the following two functions are not (unfortunately)
  - private, as T2a will need to prove things about them.
  dig-to-nat : BinDigit → nat
  dig-to-nat zero = Z
  dig-to-nat one = S Z

  unroll : {n : ℕ} → Vec BinDigit n → nat
  unroll [] = Z
  unroll (x :: l) = bnat (unroll l) (dig-to-nat x)

  ⟦_⟧₂ : BNum → nat
  ⟦ bn (x :: l) ⟧₂ = bnat (unroll l) (dig-to-nat x)

  - just to make sure we've done things right.
  lemma₁ : ⟦ 0b ⟧₂ ≡ Z
  lemma₁ = trans (right-0 _) (right-0 Z)

  lemma₂ : ⟦ 1b ⟧₂ ≡ S Z
  lemma₂ = trans (cong (λ z → z + S Z) (right-0 Z)) (
    trans (x+Sy≡Sx+y Z Z)
      (cong S (right-0 Z)))

  - two coherence theorems are provable here (« x is x + x and + 1 on the right is S)
  «-is-*2 : ∀ x → ⟦ « x ⟧₂ ≡ ⟦ x ⟧₂ + ⟦ x ⟧₂
```

```
«-is-*2 (bn (x :: x₁)) = let num = ⟦ bn (x :: x₁) ⟧₂ in right-0 (num + num)

x+1 : ∀ x → S x ≡ x + ⟦ 1b ⟧₂
x+1 x = sym (trans (cong (λ z → x + z) lemma₂) (
  trans (x+Sy≡Sx+y x Z) (
  cong S (right-0 x))))

nat+-lang : GroundLanguage nat
nat+-lang = record { Lang = λ X → ℕ+X (Carrier X)
                   ; value = λ {V} → val {V} }
where
  val : {V : DT} → ℕ+X (Carrier V) → (Carrier V → nat) → nat
  val  z       env = Z
  val {V} (s n)  env = S (val {V} n env)
  val {V} (e '+ f) env = val {V} e env + val {V} f env
  val  (v x)   env = env x

module fo₂ = FOL nat+-lang

- we can inject ℕX into ℕ+X
inject1⇒2 : ∀{V} → ℕX V → ℕ+X V
inject1⇒2 ℕX.z   = z
inject1⇒2 (ℕX.s t) = s (inject1⇒2 t)
inject1⇒2 (ℕX.v x) = v x
```

B.11 T2a

```
module T2a where
open import T1 using (BT₁)
open import T2 using (BT₂)
open import Numerals

open import Relation.Binary.PropositionalEquality
  using (_≡_; refl; trans; cong; sym; cong₂)
open import Data.Nat using (ℕ; suc) - instead of defining our own
  - isomorphic copy
open import Data.Vec using (_::_; []; Vec)

---------------------------------
- an extension of BT₂ that assumes commutativity and associativity
record BT₂ext {t1 : BT₁} (t2 : BT₂ t1) : Set₀ where
  open BT₂ t2 public
  field
    - commutativity is needed for some proofs
    - and is not provable; neither is associativity,
    - in general. So while this is more than Q,
```

```
  - it is still 'ok' in the sense that these are
  - equational axioms and not schemas.
  comm-+ : ∀ x y → x + y ≡ y + x
  assoc-+ : ∀ x y z → (x + y) + z ≡ x + (y + z)

- useful below.
left-0 : ∀ x → Z + x ≡ x
left-0 x = trans (comm-+ Z x) (right-0 x)

- to show that this definition is correct, we need a number
- of properties
shift-S : ∀ x y → x + S y ≡ S x + y
shift-S x y = trans (x+Sy≡Sx+y x y) (
  trans (cong S (comm-+ x y)) (
  trans (sym (x+Sy≡Sx+y y x)) (
    (comm-+ y (S x)))) )

- two different ways of writing 2x+2
2x+2 : ∀ x → S x + S x ≡ S ((x + x) + S Z)
2x+2 x = trans (x+Sy≡Sx+y (S x) x)
  (cong S (trans (comm-+ (S x) x) (
    trans (x+Sy≡Sx+y x x) (
    trans (cong S (sym (right-0 (x + x)))) (
      (sym (x+Sy≡Sx+y (x + x) Z)))))))

shuffle : ∀ x y → (x + y) + (x + y) ≡ (x + x) + (y + y)
shuffle x y = trans (assoc-+ x y (x + y))
  (trans (cong (λ z → x + z) (trans (sym (assoc-+ y x y))
    (trans (cong (λ z → z + y) (comm-+ y x))
      (assoc-+ x y y))))
  (sym (assoc-+ x x _)))

add1-is-S : ∀ x → ⟦ 1b ⟧₂ + x ≡ S x
add1-is-S x = trans (cong (λ z → z + x) (lemma₂)) (
  trans (comm-+ (S Z) x) (
  trans (x+Sy≡Sx+y x Z)
    (cong S (right-0 x))))

+1-is-S : ∀ x → ⟦ +1 x ⟧₂ ≡ S ⟦ x ⟧₂
+1-is-S (bn (zero :: l)) = x+Sy≡Sx+y (unroll l + unroll l) Z
+1-is-S (bn (one :: [])) =
  trans (right-0 _) (
  trans (cong (λ x → ⟦ 1b ⟧₂ + x) lemma₂) (
  trans (x+Sy≡Sx+y _ Z)
    (cong S (right-0 _))))
```

```
+1-is-S (bn (one :: x :: x₁)) =
  trans («-is-*2 (+1 (bn (x :: x₁)))) (
  trans (cong₂ _+_ (+1-is-S (bn (x :: x₁))) (+1-is-S (bn (x :: x₁))))
    (2x+2 _))

- because of the invariants that we keep in the types, and the
- way that bplus is defined _as a function_, we can actually
- prove the meaning function 'internally'. Only needs 8 cases
- whereas the paper proof needs 10 (?).
- Plus I have no idea if the paper spec is complete.

  bplus-is-+ : ∀ x y → ⟦ bplus x y ⟧₂ ≡ ⟦ x ⟧₂ + ⟦ y ⟧₂

bplus-is-+ (bn {0} (zero :: [])) y = trans (sym (left-0 ⟦ y ⟧₂)) (cong (λ z → z + ⟦ y ⟧₂) (sym lemma₁))
bplus-is-+ (bn {0} (one :: [])) y = trans (+1-is-S y) (sym (add1-is-S ⟦ y ⟧₂))
bplus-is-+ (bn {suc n} (d₀ :: l₀)) (bn {ℕ.zero} (zero :: [])) =
  trans (sym (right-0 _)) (cong (λ x → ⟦ bn (d₀ :: l₀) ⟧₂ + x) (sym lemma₁))
bplus-is-+ (bn {suc n} (d₀ :: l₀)) (bn {ℕ.zero} (one :: [])) =
  let num = bn (d₀ :: l₀) in
  trans (+1-is-S num) (x+1 ⟦ num ⟧₂)
bplus-is-+ (bn {suc n} (zero :: l₀)) (bn {suc n₁} (zero :: l₁)) =
  let n₁ = bn l₀
      n₂ = bn l₁
      num = bplus n₁ n₂
      v₁ = ⟦ n₁ ⟧₂
      v₂ = ⟦ n₂ ⟧₂ in
  trans («-is-*2 num) (trans (cong₂ _+_ (bplus-is-+ n₁ n₂) (bplus-is-+ n₁ n₂))
  (trans (cong (λ z → z + (v₁ + v₂)) (comm-+ v₁ v₂))
  (trans (assoc-+ v₂ v₁ _) (
    trans (cong (λ z → v₂ + z) (trans (sym (assoc-+ v₁ v₁ v₂)) (cong (λ z → z + v₂) (sym («-is-*2 n₁))))
  (trans (comm-+ v₂ _) (
  trans (assoc-+ _ v₂ v₂)
    (cong (λ z → ⟦ « n₁ ⟧₂ + z) (sym («-is-*2 n₂)))))))))
bplus-is-+ (bn {suc n} (zero :: l₀)) (bn {suc n₁} (one :: l₁)) =
  let n₁ = bn l₀
      n₂ = bn l₁
      num = bplus n₁ n₂
      v₁ = ⟦ n₁ ⟧₂
      v₂ = ⟦ n₂ ⟧₂ in
  trans (+1-is-S («  num)) (
  trans (cong S (trans («-is-*2 num)
    (trans (cong₂ _+_ (bplus-is-+ n₁ n₂) (bplus-is-+ n₁ n₂))
      (shuffle v₁ v₂))))
  (trans (sym (x+Sy≡Sx+y (v₁ + v₁) (v₂ + v₂)))
    (cong₂ _+_ (sym («-is-*2 n₁))
      (trans (x+1 _)
        (cong₂ _+_ (trans (sym («-is-*2 n₂)) (right-0 _)) lemma₂)))))
```

```
bplus-is-+ (bn {suc n} (one :: l₀)) (bn {suc n₁} (zero :: l₁)) =
  let n₁ = bn l₀
      n₂ = bn l₁
      num = bplus n₁ n₂
      v₁ = ⟦ n₁ ⟧₂
      v₂ = ⟦ n₂ ⟧₂ in
      trans (+1-is-S («_ num)) (
      trans (cong S (trans («-is-*2 num)
        (trans (cong₂ _+_ (bplus-is-+ n₁ n₂) (bplus-is-+ n₁ n₂))
        (trans (shuffle v₁ v₂)
          (comm-+ _ _)))))
      (trans (sym (x+Sy≡Sx+y _ _))
      (trans (comm-+ _ _) (cong₂ _+_ (trans (x+1 _) (trans (cong₂ _+_ (sym («-is-*2 n₁)) lemma₂) (c
        (sym («-is-*2 n₂))))))
bplus-is-+ (bn {suc n} (one :: l₀)) (bn {suc n₁} (one :: l₁)) =
  let n₁ = bn l₀
      n₂ = bn l₁
      num = bplus n₁ n₂
      v₁ = ⟦ n₁ ⟧₂
      v₂ = ⟦ n₂ ⟧₂ in
      trans (+1-is-S (+1 («_ num)))
      (trans (cong S (+1-is-S («_ num)))
      (trans (cong (λ z → S (S z)) (trans («-is-*2 num)
        (trans (cong₂ _+_ (bplus-is-+ n₁ n₂) (bplus-is-+ n₁ n₂))
        (shuffle v₁ v₂))))
      (trans (cong S (trans (sym (x+Sy≡Sx+y _ _)) (comm-+ _ _)))
      (trans (sym (x+Sy≡Sx+y _ _))
      (trans (comm-+ _ _)
        (cong₂ _+_ (trans (cong S (sym («-is-*2 n₁))) (sym (x+Sy≡Sx+y _ _)))
          (trans (cong S (sym («-is-*2 n₂))) (sym (x+Sy≡Sx+y _ _)))))))))
```

B.12 T3

```
module T3 where
open import T1 using (BT₁)
open import T2 using (BT₂)
open import Numerals using (_btimes_)
open import NumPlusTimes
open import Language using (GroundLanguage; module FOL)

open import Level renaming (zero to lzero)
open import Relation.Binary using (DecSetoid)
open DecSetoid using (Carrier)
open import Relation.Binary.PropositionalEquality using (_≡_)
private
  DT = DecSetoid lzero lzero
```

```
record BT₃ (t₁ : BT₁) (t₂ : BT₂ t₁) : Set₀ where
  open BT₂ t₂ public

  field
    _*_ : nat → nat → nat
    right-zero : ∀ x → x * Z ≡ Z
    S* : ∀ x y → x * S y ≡ (x * y) + x
    btimes-is-* : ∀ a b → ⟦ a btimes b ⟧₂ ≡ ⟦ a ⟧₂ * ⟦ b ⟧₂

  nat*-lang : GroundLanguage nat
  nat*-lang = record { Lang = λ X → ℕ* (Carrier X)
                     ; value = λ {V} → val {V} }
  where
    val : {V : DT} → ℕ* (Carrier V) → (Carrier V → nat) → nat
    val z       env = Z
    val {V} (s n)   env = S (val {V} n env)
    val {V} (e '+ f) env = val {V} e env + val {V} f env
    val {V} (e '* f) env = val {V} e env + val {V} f env
    val (v x)   env = env x

  module fo₃ = FOL nat*-lang
```

B.13 T4

```
module T4 where
open import T1 using (BT₁)
open import T2 using (BT₂)
open import T3 using (BT₃)
open import Numerals

open import Relation.Binary.PropositionalEquality using (_≡_)
open import Data.Sum using (_⊎_)
open import Data.Product using (Σ)

record BT₄ (t1 : BT₁) (t2 : BT₂ t1) (t3 : BT₃ t1 t2) : Set₀ where
  open BT₃ t3 public
  field
    no-junk : ∀ x → x ≡ Z ⊎ Σ nat (λ y → S y ≡ x)
```

B.14 T5

```
module T5 where
open import Relation.Binary using (DecSetoid)
open import Level using () renaming (zero to lzero)
```

```
DT : Set₁
DT = DecSetoid lzero lzero

open import T1 using (BT₁)

open import Relation.Binary.PropositionalEquality using (_≡_)
open import Data.Empty using (⊥)
open import Data.Sum using (_⊎_)
open import Data.Product using (Σ; _×_; _,_)
open import Data.Bool using (Bool)
open import Equiv using (_≃_)

open import Variables using (module VarLangs; NoVars; DBool)
open import Language
  using (module FOL;
    module LogicOverL)

record BT₅ (t₁ : BT₁) : Set₁ where
  open BT₁ t₁ public
  open VarLangs using (XV; x)
  open DecSetoid using (Carrier)
  open DecSetoid DBool using (_≈_)
  open LogicOverL fo₁.LoL-FOL
  open fo₁ using (FOL; tt; ff)

  field
    induct : (e : FOL XV) →
      ⟦ e ⟧ (λ { x → Z }) →
      (∀ (y : nat) → ⟦ e ⟧ (λ {x → y}) → ⟦ e ⟧ (λ {x → S y})) →
      ∀ (y : nat) → ⟦ e ⟧ (λ {x → y})
  postulate
    decide : ∀ {W} → (Carrier W → nat) → FOL W → FOL NoVars - T5-dec-proc
    meaning-decide : {W : DT} (env : Carrier W → nat) → (env′ : ⊥ → nat) →
      (e : FOL W) →
      let res = decide env e in
      (res ≡ tt ⊎ res ≡ ff) × (⟦ e ⟧ env) ≃ (⟦ res ⟧ env′)
```

B.15 T6

```
module T6 where
open import Relation.Binary using (DecSetoid)
open import Level using () renaming (zero to lzero)

DT : Set₁
DT = DecSetoid lzero lzero
```

```
open import T1 using (BT₁)
open import T2 using (BT₂)
open import T5 using (BT₅)

open import Relation.Binary.PropositionalEquality using (_≡_)
open import Data.Empty using (⊥)
open import Data.Sum using (_⊎_)
open import Data.Product using (Σ; _×_; _,_)
open import Data.Bool using (Bool)
open import Equiv using (_≃_)

open import Variables using (module VarLangs; NoVars; DBool)
open import Language
  using (module FOL;
    module LogicOverL)
```

With the appropriate infrastructure in place, it is now possible to define BT_6 from the theories it extends.

```
record BT₆ {t₁ : BT₁} (t₂ : BT₂ t₁) (t₅ : BT₅ t₁) : Set₁ where
  open VarLangs using (XV; x)
  open DecSetoid using (Carrier)
  open BT₂ t₂ public
  open fo₂ using (FOL; tt; ff; LoL-FOL; _and_; all)
  open LogicOverL LoL-FOL

  field
    induct : (e : FOL XV) →
      ⟦ e ⟧ (λ { x → ⟦ 0 ⟧₁ }) →
      (∀ y → ⟦ e ⟧ (λ {x → y}) → ⟦ e ⟧ (λ {x → S y})) →
      ∀ y → ⟦ e ⟧ (λ {x → y})
  postulate
    decide : ∀ {W} → (Carrier W → nat) → FOL W → FOL NoVars
    meaning-decide : {W : DT} (env : Carrier W → nat) → (env′ : ⊥ → nat) →
      (e : FOL W) →
      let res = decide env e in
      (res ≡ tt ⊎ res ≡ ff) × (⟦ e ⟧ env) ≃ (⟦ res ⟧ env′)
```

While section 4 presents the *flattened* theory, here we need only define what is new over the extended theory, namely an induction schema, a decision procedure and its meaning formula.

Here is a guide to understanding the above definition: (1) XV is a (decidable) type with a single inhabitant, x. (2) All fields of BT_2 are made publicly visible for BT_6. (3) The language of first-order logic FOL over t_2 (and some of its constructors) is also made visible. (4) $(\lambda \{x \to y\})$ denotes a substitution for the single variable x. (5) $\simeq$ denotes *type equivalence*.

B.16 T7

```
module T7 where
open import Relation.Binary using (DecSetoid)
open import Level using () renaming (zero to lzero)

DT : Set₁
DT = DecSetoid lzero lzero

open import T1 using (BT₁)
open import T2 using (BT₂)
open import T3 using (BT₃)
open import T5 using (BT₅)
open import T6 using (BT₆)

open import Relation.Binary.PropositionalEquality using (_≡_)
open import Data.Empty using (⊥)
open import Data.Sum using (_⊎_)
open import Data.Product using (Σ; _×_; _,_)
open import Data.Bool using (Bool)
open import Equiv using (_≃_)

open import Variables using (module VarLangs; NoVars; DBool)
open import Language
  using (module FOL;
    module LogicOverL)
record BT₇ {t₁ : BT₁} {t₂ : BT₂ t₁} (t₃ : BT₃ t₁ t₂) (t₅ : BT₅ t₁) (t₆ : BT₆ t₂ t₅) : Set₁ where
  open VarLangs using (XV; x)
  open DecSetoid using (Carrier)
  open BT₃ t₃ public
  open fo₃ using (FOL; tt; ff; LoL-FOL; _and_; all)
  open LogicOverL LoL-FOL

  field
    induct : (e : FOL XV) →
      ⟦ e ⟧ (λ { x → ⟦ 0 ⟧₁ }) →
      (∀ y → ⟦ e ⟧ (λ {x → y}) → ⟦ e ⟧ (λ {x → S y})) →
      ∀ y → ⟦ e ⟧ (λ {x → y})
  postulate
    decide : ∀ {W} → (Carrier W → nat) → FOL W → FOL NoVars
    meaning-decide : {W : DT} (env : Carrier W → nat) → (env' : ⊥ → nat) →
      (e : FOL W) →
      let res = decide env e in
      (res ≡ tt ⊎ res ≡ ff) × (⟦ e ⟧ env) ≃ (⟦ res ⟧ env')
```

B.17 T8

```
module T8 where
open import DefiniteDescr using (isContr₂)

open import Relation.Nullary using (¬_)
open import Relation.Binary.PropositionalEquality using (_≡_)
open import Data.Product using (Σ; proj₁; _×_)

record BT₈ : Set₁ where
  field
    ι : Set₀
    ze : ι
    S : ι → ι
    S≠Z : ∀ x → ¬ (S x ≡ ze)
    inj : ∀ x y → S x ≡ S y → x ≡ y
    induct : (p : ι → Set₀) → p ze → (∀ x → p x → p (S x)) → (∀ y → p y)

  bin : Set₀
  bin = ι → ι → ι

  +-pred : bin → Set₀
  +-pred f = (∀ x → f x ze ≡ x) ×
    (∀ x y → f x (S y) ≡ S (f x y))

  field
    +-uniq : isContr₂ ι +-pred

  _+_ : bin
  _+_ = proj₁ +-uniq

  *-pred : bin → Set₀
  *-pred f = (∀ x → f x ze ≡ ze) ×
    (∀ x y → f x (S y) ≡ f x y + x)

  field
    *-uniq : isContr₂ ι *-pred

  _*_ : bin
  _*_ = proj₁ *-uniq
```