Math. Struct. in Comp. Science (2011), *vol.* 21, *pp.* 913–942. © Cambridge University Press 2011
doi:10.1017/S0960129511000156

A pluralist approach to the formalisation of mathematics

ROBIN ADAMS and ZHAOHUI LUO

Department of Computer Science,
Royal Holloway, University of London,
Egham, Surrey TW20 0EX, United Kingdom
Email: {robin,zhaohui}@cs.rhul.ac.uk

Received 10 July 2010; revised 15 December 2010

We present a programme of research for *pluralist formalisations*, that is, formalisations that involve proving results in more than one foundation.
A foundation consists of two parts: a logical part, which provides a notion of inference, and a non-logical part, which provides the entities to be reasoned about. An LTT is a formal system composed of two such separate parts. We show how LTTs may be used as the basis for a pluralist formalisation.
We show how different foundations may be formalised as LTTs, and also describe a new method for proof reuse. If we know that a translation Φ exists between two logic-enriched type theories (LTTs) S and T, and we have formalised a proof of a theorem α in S, we may wish to make use of the fact that $\Phi(\alpha)$ is a theorem of T. We show how this is sometimes possible by writing a proof script M_Φ. For any proof script M_α that proves a theorem α in S, if we change M_α so it first imports M_Φ, the resulting proof script will still parse, and will be a proof of $\Phi(\alpha)$ in T.
In this paper, we focus on the logical part of an LTT-framework and show how the above method of proof reuse is done for four cases of Φ: inclusion, the double negation translation, the A-translation and the Russell–Prawitz modality. This work has been carried out using the proof assistant Plastic.

1. Introduction

When formalising a piece of mathematics, we must first choose a *foundation*, that is, a formal language in which the mathematical entities can be defined, and theorems and proofs about these entities can be written. Usually, such a foundation consists of two parts: a non-logical part for defining the mathematical entities to be reasoned about, and a logical part that formalises the underlying logical inference (for example, a system of logic with a set of axioms and rules of deduction that determines which proofs are valid).

Much mathematical work involves working with more than one foundation: comparing the theorems that are provable in each, defining translations between foundations, comparing the class of models of each foundation, and so on.

For example, work in set theory often involves comparing several different set theories and the theorems that can be proved in each. Similar work compares the theorems provable in different fragments of first-order arithmetic (Hájek and Pudlák 1998). The large research project known as Reverse Mathematics uses several systems of second-order arithmetic (Simpson 1999), and similar work has been done in higher-order arithmetic (Kohlenbach 2005).

In this paper, we propose a programme of research for conducting *pluralist formalisations*, that is, formalisations of pieces of mathematics that involve more than one foundation.

The prevailing paradigm in the formalisation of mathematics so far has been to choose *one* foundation, implement a proof assistant that constructs formal proofs in that foundation, and proceed to build up a large library of formalised results in that foundation. Some proof assistants offer the choice of a small number of different foundations: for example, LEGO (Pollack 1994) implements four, and Coq (Coq Development Team 2004) offers the user the choice of a predicative or impredicative type theory.

There are also proof assistants that implement *logical frameworks*, such as Isabelle (Paulson 1994) and the Edinburgh LF (Harper *et al.* 1987; Harper *et al.* 1993) as implemented in Twelf (Pfenning and Schürmann 1999). These allow more than one foundation to be represented, and often provide support for representing and reasoning about relations and translations between foundations.

If we wish to formalise a large piece of mathematics that involves proving results in several different foundations, it will be essential that we can *reuse* proofs carried out in one foundation within another. We shall therefore investigate the following questions:

— What must a logical framework provide in order to be suitable for a pluralist formalisation?
— How should we represent the different foundations within this logical framework?
— How can we then reuse a proof script written in one foundation when working in another?

Our answer to the second question is that the foundations should be represented as *logic-enriched type theories* (LTTs). We shall argue that LTTs possess some advantages over other systems of logic for the purposes of a pluralist formalisation. Our method of proof reuse relies on the two systems in question being declared in a fairly similar way. Thus, when choosing a family of systems with which to conduct a pluralist formalisation, we require one that will allow for a uniform presentation and treatment of a large number of different foundations. LTTs provide such a uniform framework.

We shall present a logical framework suitable for representing LTTs, discuss several issues in the construction of LTTs and present a method for proof reuse between LTTs. The method of proof reuse that we present is quite general: we shall show with several quite varied examples how, given a translation from one foundation S to another foundation T, we are able to take proof scripts in S and reuse them when working in T.

We work with LTTs in this paper, but our method is also usable with type theories, systems of first-order logic, and so on. It should therefore be useful to people working in many different areas of the formalisation of mathematics.

It is important to note that the LTT-approach to formalisation involves formalisation of the non-logical entities as well as that of the underlying logic. Traditionally, the studies of a logical framework, such as Edinburgh LF or the system Twelf, have mainly focused on the formal representations of logical systems and usually pay less attention to the non-logical parts of a mathematical system. The LTT-approach is different: it takes the formalisation of the non-logical entities seriously, and this is also a key part of the pluralist approach to formalisation and proof reuse. We shall discuss this issue, although the focus of the current paper is mainly on the logical part of the LTT-approach.

There has not been much work on pluralist formalisations in the literature, but there has been quite a lot of research into the related problem of sharing results produced using different proof assistants. We discuss this work in Section 6.1.

1.1. *Outline*

In Section 2, we shall discuss some general issues around the formalisation of mathematics using more than one foundation. We introduce the type-theoretic framework of LTTs in Section 3, and in Section 4 we describe our method for proof reuse in more detail. In Section 5, we apply the method to four examples and show how they have been formalised using the proof assistant Plastic.

2. A pluralist approach to the formalisation of mathematics

2.1. *Mathematics with different foundations*

When building a foundational system for mathematics, one faces various choices. For example, two of the decisions that must be made are:

— whether the logic is *classical* or *intuitionistic*;
— whether *impredicative* definitions are allowed, or only *predicative*.

Each of the four possible combinations of these options has been advocated as a foundation for mathematics at some point in history:

— **Impredicative classical mathematics**. This is arguably the way in which the vast majority of practising mathematicians work. Zermelo-Fraenkel Set Theory (ZF) is one such foundation. The proof checker Mizar (Muzalewski 1993) has been used to formalise a very large body of impredicative classical mathematics. The foundation HOL, as implemented in the proof assistants Isabelle (Nipkow *et al.* 2002) and HOL-Light (Harrison 1996), is another.
— **Impredicative constructive mathematics**. Impredicative types theories such as ECC/UTT (Luo 1994) and CIC (Bertot and Castéran 2004) are examples of such foundations. These have been implemented by the proof checkers LEGO (Luo and Pollack 1992) and Coq (Coq Development Team 2004). There are also impredicative constructive set theories, such as Intuitionistic Zermelo-Fraenkel (IZF).
— **Predicative classical mathematics**. This was the approach taken by Weyl in his influential monograph *Das Kontinuum* (Weyl 1918). Stronger predicative classical systems have been investigated by Feferman (Feferman 2005) and Schütte (Schütte 1965).
— **Predicative constructive mathematics**. Its foundations are provided, for example, by Martin-Löf's type theory (Nordström *et al.* 1990; Martin-Löf 1984), whose variants are implemented in the proof assistants Agda (Agda 2008) and NuPRL (Constable

et al. 1986). There are also predicative constructive set theories, such as Constructive Zermelo-Fraenkel set theory (CZF).

One foundation may sometimes be an *extension* of another. For example, ZF is an extension of IZF; that is, everything provable in IZF is provable in ZF. There can also be *translations* between these systems. For example, the double negation translation (Gödel 1933) is a translation from the classical system of Peano Arithmetic to the intuitionistic system of Heyting Arithmetic.

The two choices listed above are by no means the only ones that must be considered when designing a foundation. We must also consider: whether equality should be intensional or extensional; which choice principles should be allowed; and so on. A wide variety of mathematical foundations are in use today.

When beginning a *pluralist* formalisation, we must consider how the several different formalisations involved can be captured by a family of formal systems in a manner that is uniform enough for proof reuse to be practicable. As we have argued elsewhere (Adams and Luo 2010), logic-enriched type theories are able to capture a remarkably wide range of foundations very faithfully, and in a very uniform manner.

2.2. *Proof reuse in logic-enriched type theories*

In this paper, we shall present a method for *proof reuse*. Suppose we have two foundations S and T, and a translation from S to T. When we are working in T, we want to be able to *reuse* proof scripts formalising results in S.

Furthermore, we shall be greedy. We do not want to prove a lemma relating S and T, and then have to apply that lemma many times. We do not want to write a program that will automatically translate an S-proof script into a T-proof script. We want to be able to take an S-proof script and reuse it, immediately and without any modification, as a T-proof script.

There are two particular situations that we wish to consider:

(1) We have two foundations S and T, and S is a *subsystem* of T. If we have shown that $S \vdash \alpha$, then we can immediately make use of the fact that $T \vdash \alpha$. When formalising a piece of mathematics that makes use of this sort of step, we wish to take a proof script that formalises a proof of α in S, and use this script to provide us with a proof of α when working in T.

(2) More generally, we have two foundations S and T, and a *translation* $\Phi : S \to T$; that is, a mapping from the language of S to the language of T such that if $S \vdash \alpha$, then $T \vdash \Phi(\alpha)$. When formalising a piece of mathematics that makes use of this sort of step, we wish to take a proof script that formalises a proof of α in S, and use this script to provide us with a proof of $\Phi(\alpha)$ when working in T.

Situation (1) is a special case of situation (2), where the translation Φ is the inclusion from S to T.

We shall be working in this paper with *logic-enriched type theories* (LTTs). These are systems of logic that consist of a type theory, which defines the mathematical objects we

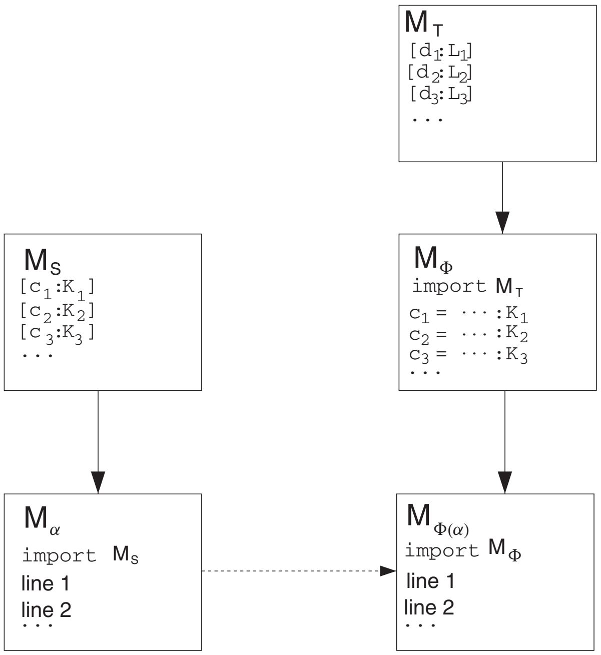

Fig. 1. Proof reuse between two LTTs

will be dealing with, and a separate *logical component*, for stating and proving propositions about those objects.

In situation (1) above, we can sometimes arrange it so that LTT$_S$, which represents S, is a subsystem of the LTT$_T$, which represents T. Suppose we have proof scripts M_S and M_T that define LTT$_S$ and LTT$_T$, respectively. If a proof script imports M_S and proves a theorem α, the proof script will still parse if we change it to import M_T instead. This idea was made use of in Adams and Luo (2010), where we proved several results in the predicative LTT$_W$, and were able to immediately reuse those proof scripts in an impredicative LTT that extends LTT$_W$.

However, such an approach is quite fragile as it depends on us using the same names for constants in M_S and M_T. It is also not certain that we can always define LTT$_S$ and LTT$_T$ in such a way that LTT$_S$ is a subsystem of LTT$_T$. Moreover, this approach cannot handle the more general situation (2) above.

The approach we present in this paper is as follows. Suppose we have a translation Φ from LTT$_S$ to LTT$_T$. Given proof scripts M_S and M_T that define LTT$_S$ and LTT$_T$, we shall construct a proof script M_Φ that imports M_T, and then *defines* every constant that was *declared* in M_S (see Figure 1).

A proof script M_α that imports M_S will still parse if we change it to import M_Φ instead. Furthermore, we can write M_Φ in such a way that if M_α provides a proof of α under M_S, then it provides a proof of $\Phi(\alpha)$ under M_Φ.

We shall show in this paper how this can be done when LTT$_S$ is a subsystem of LTT$_T$, and in three other cases: the double-negation translation from classical to intuitionistic logic; the *A*-translation from intuitionistic logic to itself; and the Russell–Prawitz modality from first-order logic (classical or intuitionistic) to second-order logic.

3. A type-theoretic framework for pluralist formalisations

Our approach to pluralist formalisations is based on a uniform framework in which mathematics with different foundations can be formalised, and the *type-theoretic framework* of *Logic-enriched Type Theories* (LTTs) is particularly appropriate for this.

Logic-enriched type theories were first studied by Aczel and Gambino to investigate type-theoretic interpretations of constructive set theory (Aczel and Gambino 2002; Gambino and Aczel 2006). An LTT is a formal system consisting of a *type-theoretic component* that provides *types* and *terms*, and a *logical component* that provides *propositions* and *proofs*. The intention is that the types and terms describe the collection of mathematical objects we are concerned with, and the logical component is used to reason about those objects.

We shall present three logical frameworks in this section: LF, a Church-typed version of Martin-Löf's Logical Framework[†]; LF′, an extension of LF intended for representing an LTT; and $\mathrm{LF}_{\mathrm{LTT}}$, an extension of LF intended for representing *several* LTTs simultaneously.

The *type-theoretic framework* is a method of specifying LTTs within a logical framework. The LTTs specifiable this way are capable of expressing a wide spectrum of foundations for mathematics in a uniform way. It was first proposed in Luo (2006)[‡], and in Adams and Luo (2007; 2010) we studied one of the systems in this framework (*viz.* a logic-enriched type theory LTT_W that gives a modern type-theoretic version of Weyl's system for predicative mathematics) and used it to formalise Weyl's predicative mathematics (Weyl 1918) in the proof assistant Plastic.

In this section, we review the logical framework LF and its extension LF′. We will then introduce the extension $\mathrm{LF}_{\mathrm{LTT}}$, and describe how $\mathrm{LF}_{\mathrm{LTT}}$ may be used to specify LTTs. We then present the type-theoretic framework, and give examples of logic-enriched type theories that can be specified in the framework.

3.1. *The logical framework LF*

A *logical framework*, such as Martin-Löf's logical framework (Nordström *et al.* 1990) and its Church-typed version LF (Luo 1994), is a dependent type system, together with a method for representing other formal systems within that type system.

[†] The framework LF should not be confused with the Edinburgh Logical Framework (Harper *et al.* 1987; Harper *et al.* 1993), which, unfortunately, is also called LF. One of the main differences between LF and the Edinburgh LF is that the former system is intended to be used to specify *type theories*, and hence allows *computation rules* to be declared.

[‡] LF has a variant called PAL^+ (Luo 2003), where applications are fully applied or saturated. Luo (2006) adopted the notations of PAL^+, but this is not essential, and we shall use LF and the associated notations in this paper.

In the following, we introduce LF briefly and fix our notation – the full details of LF, including its rules, can be found in Luo (1994, Chapter 9).

3.1.1. *Basic constructions.* The system LF deals with *kinds* and *objects*. The kinds are:

— $Type$, the kind of types.
— $El(A)$, the kind of objects of type A.
— $(x{:}K)K'$, the kind of dependent functional operations f, which can be applied to any object k of kind K to form the application fk of kind $[k/x]K'$.

We often omit El and write $El(A)$ simply as A. We write $K \to K'$ for $(x{:}K)K'$ when x does not occur free in K'.

When writing objects that are in the form of an application, we shall sometimes write $f(a_1, ..., a_n)$ for $f a_1 ... a_n$, and often use the infix form of binary operators: for instance, we write $A \times B$ and $P \wedge Q$ for $\times(A, B)$ and $\wedge(P, Q)$, respectively.

Two objects are definitionally equal in LF if they are $\beta\eta$-convertible.

The system LF is intended for specifying type theories that deal with *types* and *terms*. The intention is that:

— The types are represented by the objects of kind $Type$.
— The terms of type A are represented by the objects of kind $El(A)$.
— The objects of kind $(x{:}K)K'$ represent meta-functions on the type theory's syntax.

3.1.2. *Specification of type theories.* A type theory is specified in LF by declaring *constants*, each with a kind, and *computation rules*. These declarations have the effect of extending LF with additional rules (see Luo (1994) for the details).

Typically, a type in a type theory comes with its rules of formation, introduction, elimination and computation. We represent this type in LF by declaring constants corresponding to the formation, introduction and elimination rules, and declaring equality rules corresponding to the computation rules.

For example, the type N of natural numbers can be specified as in Figure 2 where, for instance, the introduction rule

$$\frac{\Gamma \vdash n : N}{\Gamma \vdash succ(n) : N}$$

is specified by means of the declaration of the constant *succ*.

In this way, LF can specify type theories that contain inductive and co-inductive types, predicative and impredicative universes, inductive-recursive types, and other more exotic types.

3.2. *Logic-enriched type theories*

The system LF is a suitable language for specifying *type theories* that deal with just two kinds of entity: types and terms. There is a single kind **Type** of all the types in the type theory.

If we wish to use a system to state and prove mathematical theorems, we must have some way of introducing logical propositions. In a type theory, one may do this by identifying

The formation rule for N

$$N \quad : \quad Type$$

The introduction rules for N (constructors)

$$\begin{array}{rcl} 0 & : & N \\ succ & : & (N)N \end{array}$$

The elimination rule over types for N

$$\begin{array}{rcl} \mathcal{E}_T^N & : & (\ C\colon (N)Type\ ) \\ & & (\ c\colon C(0)\ )\ (\ f\colon (n{:}N)(x{:}C(n))C(succ(n))\ ) \\ & & (\ z\colon N\ )\ C(z) \end{array}$$

The computation rules for N

$$\begin{array}{rcl} \mathcal{E}_T^N(C, c, f, 0) & = & c \quad : \quad C(0) \\ \mathcal{E}_T^N(C, c, f, succ(n)) & = & f(n, \mathcal{E}_T^N(C, c, f, n)) \ : \quad C(succ(n)) \end{array}$$

Fig. 2. The type of natural numbers.

propositions with types (for example, in Martin-Löf's type theory, every proposition is a type, and *vice versa*) or by taking propositions as types, but not *vice versa* (for example, in ECC/UTT (Luo 1994), every proposition is a type, but not every type is a proposition).

However, if one takes the view that logical propositions and data types should be completely separate, then one will wish to work in a different kind of system, and LF will not be adequate for specifying this different kind of system.

Logic-enriched type theories (LTTs) (Gambino and Aczel 2006) are formal systems in which there is a complete separation between (logical) propositions and (data) types. The syntax of an LTT consists of four categories of expression: *types*, *terms*, *propositions* and *proofs* (or *derivations*).

So an LTT falls naturally into two components, or 'worlds': the type-theoretic component and the logical component. This allows a lot of flexibility in the design of an LTT, as we can change one component without affecting the other (for example, we can add excluded middle to the logical component without changing the type-theoretic component). This makes LTTs suitable for capturing a wide range of different mathematical foundations.

The two components do, however, interact. The logical world may depend on the type-theoretic world: for example, given an inductive type such as N or $List(A)$, we may choose to introduce a rule of deduction allowing propositions to be proved by induction over N.

In order to specify an LTT adequately, an extension of LF is called for. This extended logical framework is obtained by extending LF by adding a new kind $Prop$, which stands for the world of logical propositions, and a new kind constructor Prf:

$$\frac{\Gamma\ valid}{\Gamma \vdash \mathbf{Prop\ kind}} \qquad \frac{\Gamma \vdash P : \mathbf{Prop}}{\Gamma \vdash \mathbf{Prf}(P)\ \mathbf{kind}}$$

This extended framework was first proposed in Luo (2006) and further studied in Adams and Luo (2010) and, in the latter paper, we called it LF'.

Conjunction

$$\begin{array}{rcl}
\wedge & : & (Prop)(Prop)Prop \\
\wedge_I & : & (P\colon Prop)(Q\colon Prop)\ (Prf(P))(Prf(Q))Prf(P \wedge Q) \\
\wedge_{E_1} & : & (P\colon Prop)(Q\colon Prop)\ (Prf(P \wedge Q))Prf(P) \\
\wedge_{E_2} & : & (P\colon Prop)(Q\colon Prop)\ (Prf(P \wedge Q))Prf(Q)
\end{array}$$

Universal quantifier

$$\begin{array}{rcl}
\forall & : & (A\colon Type)(P\colon (A)Prop)Prop \\
\forall_I & : & (A\colon Type)(P\colon (A)Prop)\ ((x{:}A)Prf(P(x)))\ Prf(\forall(A,P)) \\
\forall_E & : & (A\colon Type)(P\colon (A)Prop)\ (Prf(\forall(A,P)))\ (a\colon A)Prf(P(a))
\end{array}$$

Negation

$$\begin{array}{rcl}
\neg & : & (Prop)Prop \\
DN & : & (P\colon Prop)\ (Prf(\neg\neg P))Prf(P) \\
\dots & \dots & \dots
\end{array}$$

Fig. 3. Logical operators and direct proofs in the classical FOL.

The intention is that:

— the types are represented by the objects of kind **Type**;
— the terms of type A are represented by the objects of kind El (A);
— the propositions are represented by the objects of kind **Prop**;
— the proofs of a proposition P are represented by the objects of kind **Prf**(P).

An LTT is specified in LF' by declaring constants and computation rules. Each declaration has the effect of extending LF' with new rules – see Adams and Luo (2010) for the details.

3.2.1. *Logics.* The logic in an LTT is specified by declaring constants for the logical operators and the associated rules.

For example, say we wish an LTT's logical component to consist of classical first-order logic. This can be introduced by declaring the constants that stand for the logical operators, and constants that stand for the associated inference rules. The logical operators $\wedge$, $\forall$ and $\neg$ and some of their associated rules of inference are specified in Figure 3. Other logical operators can be introduced in a similar way.

3.2.2. *Remarks.*

(1) The quantifier $\forall$ declared here can only be used to quantify over a type; that is, for a formula $\forall(A, P)$, or $\forall x{:}A, P(x)$ in the usual notation, A must be a type.
In particular, since $Prop$ is not a type (it is a kind), one cannot form a proposition by quantifying over $Prop$. Higher-order logical quantification such as $\forall X{:}Prop.X$, as found in impredicative type theories such as System F (Girard 1986) and the Calculus of Constructions (Coquand and Huet 1988), is not possible with this constant.
Similarly, since propositions are not types ($Prf(P)$ is a kind, not a type), one cannot quantify over the proofs of a proposition either.

When designing an LTT, we can thus choose whether to allow first- or higher-order quantification. This situation is in contrast with a type theory such as ECC or UTT (Luo 1994), where it would not be possible to restrict quantification to the datatypes, since $Prop$ is a type and every proposition is a type.

(2) The negation operator is classical in that the classical double-negation rule holds, as declared by the constant DN.

(3) An LTT is specified in the framework by not only specifying a collection of constants, but also *computation rules*. Computation rules are needed in the type-theoretic component for specifying inductive data types, universes, and so on. Computation rules in the logical component are also sometimes needed – for example, they were used in Adams and Luo (2010) in the specification of typed sets.
The ability to specify computation rules is the most important difference between the Martin-Löf family of logical frameworks, including LF and LF', and the Edinburgh LF (Harper *et al.* 1993).

(4) If we have introduced a universe that contains the empty type and the type of natural numbers, we can then prove, internally in the type-theoretic framework, that Peano's fourth axiom for natural numbers holds (that is, the proposition $\forall x{:}N.(s[x] \neq_N 0)$ holds). This is similar to Martin-Löf's type theory, where, in the absence of a type universe, one cannot prove Peano's fourth axiom internally (Smith 1988).

3.3. *The type-theoretic framework*

The *type-theoretic framework* is a method for specifying LTTs using a type system such as LF′, and was introduced in Luo (2006). The LTTs specifiable in the type-theoretic framework are all defined and specified in a uniform way, but should be capable of expressing a wide range of different mathematical foundations. The type-theoretic framework is thus especially suitable as the basis for a pluralist formalisation.

An LTT is specified within the type-theoretic framework by:

(1) Declaring a number of *inductive types* and *inductive families of types*.
Besides N, other examples of inductive types include lists, vectors, trees, ordinals, dependent functions and dependent pairs. In general, inductive types can be generated by inductive schemata as studied in, for example, Dybjer (1991), Coquard and Paulin-Mohring (1990) and Luo (1994).

(2) Declaring a number of *type universes*, that is, types whose objects are (names of) types.
For example, a universe U of 'small types' can be introduced as

$$U : Type \text{ and } T : U \to Type;$$

An inductive type may have names in U – for example, we can have nat as a name of N in U:

$$nat : U \text{ and } T(nat) = N : Type.$$

Notice that such a universe is *predicative* in the sense that it only contains types that do not involve U itself. The general way of introducing predicative type universes can be found in Martin-Löf (1984). Impredicative universes, such as that of propositions in UTT (Luo 1994, page 175), can also be specified in the type-theoretic framework.

(3) Declaring a number of *logical connectives* and their associated *rules of deduction*.
We may introduce some or all of the propositional connectives, first- or higher-order quantifiers, and other logical connectives (such as equality). The rules of deduction may be those of classical logic, constructive logic, minimal logic, and so on.

(4) Declaring one or more *propositional universes* (Adams and Luo 2010).

(5) Declaring the *induction rules* for each inductive data type.
An LTT may contain some data types, which are usually inductively defined, in exactly the same way as inductive types are specified in LF (see Section 3.1). For each inductive type in the LTT, there is an associated *induction rule* for proving properties of the objects of that type.
For example, an LTT may contain the type N of natural numbers as specified in Figure 2 in Section 3.1. Associated with N, there is an associated induction rule given by the constant

$$\begin{aligned}\mathscr{E}_P^N \ : \ & (P : (N)Prop) \\ & (c : P(0))(f : (x{:}N)(P(x))P(succ(x))) \\ & (n{:}N)P(n),\end{aligned}$$

which expresses the following rule:

$$\frac{\Gamma \vdash P : (N)Prop \quad \Gamma \vdash c : P(0) \quad \Gamma \vdash f : (x{:}N)(P(x))P(succ(x)) \quad \Gamma \vdash n : N}{\Gamma \vdash \mathscr{E}_P^N(P, c, f, n) : P(n)}$$

Note that when read as a proof rule, this is just the rule of induction over natural numbers.
Therefore, associated with each inductive type, there are two elimination operators: $\mathscr{E}_T$ and $\mathscr{E}_P$ (for N, they are $\mathscr{E}_T^N$ and $\mathscr{E}_P^N$). Note that the elimination operator over types, $\mathscr{E}_T$, has associated computation rules (for example, the computation rules for $\mathscr{E}_T^N$ in Figure 2), while the elimination operator over propositions, $\mathscr{E}_P$, does not[†].
We may introduce rules for induction over the whole of $Prop$, as above, or over just one propositional universe.
The induction rules connect the world of logical propositions (formally represented by $Prop$) and that of the data types (formally represented by $Type$). Quantifications over types allow one to form propositions to express logical properties of data, and the induction rules to prove those properties.

(6) Introducing types of *typed sets*.
For each type A, we can introduce a type $Set(A)$ of all *sets* of objects of type A. This type's canonical objects have the form $\{x{:}A \mid \phi\}$, where ϕ is a proposition. This allows us to introduce sets impredicatively (if ϕ can range over the whole of $Prop$),

[†] Whether the elimination operators over propositions have associated computation rules similar to those for the elimination operators over types is optional. Including or excluding these computation rules will not affect the type-theoretic component, since types and terms cannot depend on proofs, or which propositions are provable.

as in ordinary mathematics, or predicatively (if ϕ ranges only over a small universe of propositions, as in predicative mathematics – see, for example, Feferman (2005)). For further and formal details, see Luo (2006) and Adams and Luo (2010).

Such a notion of a typed set, together with the possibility of representing classical first-order logic, allows us to formalise classical predicative mathematics in the type-theoretic framework (Weyl 1918; Adams and Luo 2010), as well as impredicative mathematics (*cf.* the discussion on mathematical pluralism in Section 2).

3.3.1. *Remarks.*

(1) **Separation of propositions and types.** The type-theoretic framework has an important and salient feature: there is a clear separation between logical propositions and data types. In Martin-Löf's type theory, for example, types and propositions are identified. The second author has argued, for instance in the development of ECC/UTT (Luo 1994), that it is unnatural to identify logical propositions with data types, and there should be a clear distinction between the two. This is part of the philosophy behind the development of the type theories ECC and UTT, where data types are not propositions, although logical propositions are types.

Logic-enriched type theories, and hence our framework as presented in this paper, have gone one step further (in comparison with ECC/UTT), since there is a complete separation between propositions and types. Logical propositions or their totality $Prop$ are not regarded as types. This has led to a more flexible treatment of logics in the framework.

(2) **Consistency and adequacy.** The consistency of an LTT formulated along the lines suggested above can be shown either by a direct proof (Goguen 1994) or by an indirect mapping between the LTT concerned and a known consistent type system. For example, in Luo (2006), we map an LTT called LTT_1 (classical FOL plus inductive types) to $MLTT_e$, an extension of Martin-Löf's type theory with excluded middle. We show that LTT_1 is consistent relative to $MLTT_e$.

Such a relative consistency proof raises an interesting question: if Martin-Löf's type theory extended with excluded middle is consistent, why use an LTT at all? Why not just use $MLTT_e$? One reason is that the *meaning theory* of type theory relies on the property of *canonicity*: that every object reduces to a canonical object. This allows us to provide a meaning theory in which an inductive type is understood as consisting of its canonical objects (for example, the type of natural numbers consists of zero and its successors).

The LTTs in the type-theoretic framework possess the property of canonicity, thanks to the clear distinction between logical propositions and data types. The system $MLTT_e$ does not, since in $MLTT_e$, every inductive type contains infinitely many non-canonical objects. Hence the type-theoretic framework provides an adequate treatment of classical reasoning on the one hand, and a clean meaning-theoretic understanding of inductive types on the other.

3.4. *Working with more than one LTT*

We are concerned in this paper with formalisations that involve more than one LTT. We must therefore extend the logical framework yet again.

If we wish to declare two LTTs simultaneously, called LTT$_1$ and LTT$_2$, say, we need the logical framework to possess the following kinds, in addition to the dependent product kinds of the form $(x{:}K)K'$:

$$\mathbf{Type}_1,\ \mathrm{El}_1\,(k),\ \mathbf{Prop}_1,\ \mathbf{Prf}_1(k)$$
$$\mathbf{Type}_2,\ \mathrm{El}_2\,(k),\ \mathbf{Prop}_2,\ \mathbf{Prf}_2(k)$$

with the following rules of deduction:

$$\frac{\Gamma\ \text{valid}}{\Gamma\vdash\mathbf{Type}_1\mathbf{kind}}\qquad\frac{\Gamma\vdash A{:}\mathbf{Type}_1}{\Gamma\vdash\mathrm{El}_1\,(A)\,\mathbf{kind}}\qquad\frac{\Gamma\ \text{valid}}{\Gamma\vdash\mathbf{Prop}_1\mathbf{kind}}\qquad\frac{\Gamma\vdash P{:}\mathbf{Prop}_1}{\Gamma\vdash\mathbf{Prf}_1(P)\mathbf{kind}}$$

$$\frac{\Gamma\ \text{valid}}{\Gamma\vdash\mathbf{Type}_2\mathbf{kind}}\qquad\frac{\Gamma\vdash A{:}\mathbf{Type}_2}{\Gamma\vdash\mathrm{El}_2\,(A)\,\mathbf{kind}}\qquad\frac{\Gamma\ \text{valid}}{\Gamma\vdash\mathbf{Prop}_2\mathbf{kind}}\qquad\frac{\Gamma\vdash P{:}\mathbf{Prop}_2}{\Gamma\vdash\mathbf{Prf}_2(P)\mathbf{kind}}$$

In the framework augmented with these kinds:

— The objects of kind $\mathbf{Type}_i$ represent the types of LTT$_i$.
— The objects of kind $\mathrm{El}_i\,(A)$ represent the terms of LTT$_i$.
— The objects of kind $\mathbf{Prop}_i$ represent the propositions of LTT$_i$.
— The objects of kind $\mathbf{Prf}_i(A)$ represent the proofs of LTT$_i$.

The constants and computation rules in the declaration of LTT$_1$ will therefore involve only the kinds $\mathbf{Type}_1$, $\mathrm{El}_1\,(A)$, $\mathbf{Prop}_1$ and $\mathbf{Prf}_1(A)$, and the product kinds built up from them. When we are working in LTT$_1$, we use only these kinds. Likewise, when declaring or working in LTT$_2$, we use only the kinds $\mathbf{Type}_2$, $\mathrm{El}_2\,(A)$, $\mathbf{Prop}_2$, $\mathbf{Prf}_2(A)$, and the product kinds built from them.

3.4.1. *Logical framework* LF$_{\mathrm{LTT}}$.

We wish to give the user the ability to declare arbitrarily many *pairs* (K, C) consisting of a *topkind* K and a *kind constructor* C.

The effect of declaring the pair (K, C) is to extend the logical framework with the following rules of deduction:

$$\frac{\Gamma\ \text{valid}}{\Gamma\vdash K\,\mathbf{kind}}\qquad\frac{\Gamma\vdash k{:}K}{\Gamma\vdash C(k)\mathbf{kind}}\qquad\frac{\Gamma\vdash k=k'{:}K}{\Gamma\vdash C(k)=C(k')}$$

In the above example, we would declare the pairs

$$(Type_1, El_1), (Type_2, El_2), (Prop_1, Prf_1), (Prop_2, Prf_2).$$

Note that we no longer need the kinds **Type**, $\mathrm{El}\,(A)$, **Prop** and $\mathbf{Prf}(P)$ as part of the primitive syntax, so we remove them, though the user can reintroduce them if needed by declaring the topkind pairs $(\mathbf{Type}, \mathrm{El})$ and $(\mathbf{Prop}, \mathrm{Prf})$.

We call this new framework LF$_{\mathrm{LTT}}$. In summary: LF$_{\mathrm{LTT}}$ is the framework LF, with the kinds $Type$ and $El(k)$ removed, and with the ability to declare pairs (K, C) added.

3.5. *Implementation*

The proof assistant Plastic was first implement by Callaghan as an implementation of LF (Callaghan and Luo 2001). It was extended to an implementation of LF', and used in the work to formalise Weyl's predicative mathematics using the type-theoretic framework (Adams and Luo 2007; 2010).

Plastic allows the user:

— to declare constants with commands such as $c{:}K$ by $[c{:}K]$;
— to define constants with commands such as $[c{=}\cdots{:}K]$;
— to construct objects using tactics such as `Intros` and `Refine`.

The user may also declare computation rules of a certain form. Plastic automatically generates the constants and computation rules for inductive types, but logical connectives, induction rules, universes and computation rules in the logical component must be entered by hand.

For the work described in this paper, the first author is extending Plastic further to include an implementation of $\mathrm{LF}_{\mathrm{LTT}}$. The user may now declare a top-kind and constructor (K, C) by entering the command `Topkind` $K\ C$;

We shall describe the Plastic implementations of several LTTs in Section 5, and then show how they are used in studying pluralist formalisations and proof reuse.

4. Our approach to pluralist formalisations

Our approach to pluralist formalisations is based on the concept of a *translation*.

Definition 4.1 (Translation). A *translation* Φ from an LTT S to an LTT T is a mapping from the expressions of S to the expressions of T such that:

— If A is a type of S, then $\Phi(A)$ is a type of T.
— If M is a term of type A in S, then $\Phi(M)$ is a term of type $\Phi(A)$ in T.
— If P is a proposition of S, then $\Phi(P)$ is a proposition of T.
— If H is a proof of P in S, then $\Phi(H)$ is a proof of $\Phi(P)$ in T.

Suppose we have a translation Φ from S to T. If α is a theorem of S, then $\Phi(\alpha)$ is a theorem of T. We wish to find a way to take a formalisation of a proof of α in S, and use that proof script, without any modification, as a proof of $\Phi(\alpha)$ in T.

Our approach is as follows (see Figure 1). Let M_S and M_T be two proof scripts that declare the constants and rules of deduction of S and T, respectively. Let M_α be a proof script that imports M_S, and proves the theorem α.

We construct a proof script M_ϕ that imports M_T, and then *defines* every symbol that was *declared* as a constant in M_S. If M_S contains a constant declaration $c{:}K$, then M_T must define c to be an object of kind K. If M_S declares the computation rule $M = N$, then we must ensure that M and N are convertible under the definitions in M_T.

If both these conditions are met, we know that the proof script M_α will parse if we import M_ϕ instead of M_S. We give the name $M_{\phi(\alpha)}$ to the proof script with this small change made (see Figure 1).

LTT$_1$		LTT$_2$	
`Topkind` $\mathbf{Prop}_1$ Prf$_1$		`Topkind` $\mathbf{Prop}_2$ Prf$_2$	
`not`	$: \mathbf{Prop}_1 \to \mathbf{Prop}_1$	`neg`	$: \mathbf{Prop}_2 \to \mathbf{Prop}_2$
`notI`	$: (p, q{:}\mathbf{Prop}_1)(\mathbf{Prf}_1(p) \to \mathbf{Prf}_1(q))$	`negI`	$: (p, q{:}\mathbf{Prop}_2)(\mathbf{Prf}_2(p) \to \mathbf{Prf}_2(q))$
	$\to (\mathbf{Prf}_1(p) \to \mathbf{Prf}_1(\sim q))$		$\to (\mathbf{Prf}_2(p) \to \mathbf{Prf}_2(\neg q))$
	$\to \mathbf{Prf}_1(\sim p)$		$\to \mathbf{Prf}_2(\neg q)$
`notnotE`	$: (p{:}\mathbf{Prop}_1)\mathbf{Prf}_1(\sim\sim p) \to \mathbf{Prf}_1(p)$	`negnegE`	$: (p{:}\mathbf{Prop}_2)\mathbf{Prf}_2(\neg\neg p) \to \mathbf{Prf}_2(p)$
`imp`	$: \mathbf{Prop}_1 \to \mathbf{Prop}_1 \to \mathbf{Prop}_1$	`or`	$: \mathbf{Prop}_2 \to \mathbf{Prop}_2 \to \mathbf{Prop}_2$
`impI`	$: (p, q{:}\mathbf{Prop}_1)(\mathbf{Prf}_1(p) \to \mathbf{Prf}_1(q))$	`orIl`	$: (p, q{:}\mathbf{Prop}_2)\mathbf{Prf}_2(p) \to \mathbf{Prf}_2(p \vee q)$
	$\to \mathbf{Prf}_1 p \supset q$	`orIr`	$: (p, q{:}\mathbf{Prop}_2)\mathbf{Prf}_2(q) \to \mathbf{Prf}_2(p \vee q)$
`impE`	$: (p, q{:}\mathbf{Prop}_1)\mathbf{Prf}_1(p \supset q)$	`orE`	$: (p, q, r{:}\mathbf{Prop}_2)(\mathbf{Prf}_2(p) \to \mathbf{Prf}_2(r))$
	$\to \mathbf{Prf}_1(p) \to \mathbf{Prf}_1(q)$		$\to (\mathbf{Prf}_2(q) \to \mathbf{Prf}_2(r))$
			$\to \mathbf{Prf}_2(p \vee q) \to \mathbf{Prf}_2(r)$

Fig. 4. The logical components of LTT$_1$ and LTT$_2$

Example 4.2. Let LTT$_1$ be an LTT whose logical component consists of classical propositional logic with negation `not` and implication `imp`. Let LTT$_2$ be an LTT whose logical component consists of classical propositional logic with negation `neg` and disjunction `or` (see Figure 4). We shall write

$$\begin{array}{rcl} \sim \phi & \text{for} & \texttt{not}\ \phi \\ \phi \supset \psi & \text{for} & \texttt{imp}\ \phi\ \psi \\ \neg\phi & \text{for} & \texttt{neg}\ \phi \\ \phi \vee \psi & \text{for} & \texttt{or}\ \phi\ \psi. \end{array}$$

It is known that implication can be defined in terms of disjunction and negation in classical logic. This fact can be used to define a translation from LTT$_1$ to LTT$_2$:

$$[\![\sim \phi]\!] \equiv \neg\, [\![\phi]\!] \qquad [\![\phi \supset \psi]\!] \equiv \neg\, [\![\phi]\!] \vee [\![\psi]\!]\,.$$

Let M_1 and M_2 be two proof scripts that consist of the constant declarations given in Figure 4 for LTT$_1$ and LTT$_2$, respectively. We now wish to write a proof script M_Φ that imports M_2, and then *defines* every constant that was *declared* in M_1.

The proof script M_ϕ begins as follows:

`import` M_2;

$[\mathbf{Prop}_1 = \mathbf{Prop}_2]$;
$[\mathrm{Prf}_1 = \mathrm{Prf}_2]$;

$[\texttt{not} = \texttt{neg}]$;
$[\texttt{notI} = \texttt{negI}]$;
$[\texttt{notnotE} = \texttt{negnegE}]$;

We must now define `imp`. The definition is guided by (1) above:

$[\texttt{imp} = [\mathrm{p}, \mathrm{q}:\mathbf{Prop}_1] \sim \mathrm{p} \vee \mathrm{q}];$

And we must define objects `impI` and `impE` that have kinds

$$\begin{aligned}\texttt{impI} &: (p, q:\mathbf{Prop}_1)(\mathbf{Prf}_1(p) \to \mathbf{Prf}_1(q)) \to \mathbf{Prf}_1(p \supset q)\\ \texttt{impE} &: (p, q:\mathbf{Prop}_1)\mathbf{Prf}_1(p \supset q) \to \mathbf{Prf}_1(p) \to \mathbf{Prf}_1(q),\end{aligned}$$

that is,

$$\begin{aligned}\texttt{impI} &: (p, q:\mathbf{Prop}_2)(\mathbf{Prf}_2(p) \to \mathbf{Prf}_2(q)) \to \mathbf{Prf}_2(\sim p \vee q)\\ \texttt{impE} &: (p, q:\mathbf{Prop}_2)\mathbf{Prf}_2(\sim p \vee q) \to \mathbf{Prf}_2(p) \to \mathbf{Prf}_2(q).\end{aligned}$$

It is straightforward to construct these objects using a proof assistant like Plastic.

Now, suppose we have formalised a proof of the proposition α in LTT$_2$. That is, suppose we have a proof script that imports $\mathbf{M}_2$, and then constructs an object of kind $\mathbf{Prf}_2(\alpha)$. If we change the script so that it imports M_Φ instead, we know that the script will still parse, and that the script will now be a proof of $[\![\alpha]\!]$ in LTT$_2$.

For example, suppose a script imports M_2, which constructs an object of kind

$$(p:\mathbf{Prop}_2)\mathbf{Prf}_2(p \supset p).$$

We now change the script to import M_Φ instead. Under the definitions in M_Φ, we have

$$(p:\mathbf{Prop}_2)\mathbf{Prf}_2(p \supset p) = (p:\mathbf{Prop}_1)\mathbf{Prf}_1(\sim p \vee p),$$

so the script now constructs an object of kind $(p:\mathbf{Prop}_1)\mathbf{Prf}_1(\sim p \vee p)$. This is exactly as required, since $[\![\phi \supset \phi]\!] \equiv \sim [\![\phi]\!] \vee [\![\phi]\!]$.

4.1. *Remarks*

(1) Note that the construction of the module M_Φ can be seen in one sense as a formalisation of the metatheorem that $[\![\ ]\!]$ is sound, that is, it maps theorems to theorems.

(2) Note that a translation between two LTTs may involve changing the logical world (as in the examples in Sections 5.2–5.4 below), the datatype world (for example, the inclusion from LTT_W to LTT_I considered in Adams and Luo (2010)), or both.

5. Case studies in formalisation

In this section we describe several case studies in the use of this method of proof reuse that we have carried out using the proof assistant Plastic. The source code for these examples is available at `http://www.cs.rhul.ac.uk/~robin/pluralism`.

5.1. *Classical and intuitionistic LTTs*

For these examples, we shall assume that we have two LTTs: a classical LTT, called LTT$_{\text{class}}$, and an intuitionistic LTT, called LTT$_{\text{int}}$. We assume that these two LTTs have the same type-theoretic component. The logical component of $\text{LTT}_{\text{class}}$ is first-order classical logic with the connectives `eqC`, `notC`, `andC`, `orC`, `impC`, `allC` and `exC`. The logical component of LTT_{int} is first-order intuitionistic logic with the connectives `eqI`, `notI`, `andI`, `orI`, `impI`, `allI` and `exI`.

We write:

$$\begin{array}{rll} M =_C N & \text{for} & \texttt{eqC}\ A\ M\ N \\ \neg_C \phi & \text{for} & \texttt{notC}\ \phi \\ \phi \wedge_C \psi & \text{for} & \texttt{andC}\ \phi\ \psi \\ \phi \vee_C \psi & \text{for} & \texttt{orC}\ \phi\ \psi \\ \phi \supset_C \psi & \text{for} & \texttt{impC}\ \phi\ \psi \\ \forall_C x{:}A.\phi & \text{for} & \texttt{allC}\ A\ [x{:}A]\phi \\ \exists_C x{:}A.\phi & \text{for} & \texttt{exC}\ A\ [x{:}A]\phi. \end{array}$$

Similarly, we write $M =_I N$ for `eqI` $A\ M\ N$, and so on.

We have two proof scripts: M_C, which declares the constants and computation rules of $\text{LTT}_{\text{class}}$, and M_I, which declares the constants and computation rules of LTT_{int}. Some of the declarations are given in Figure 5. Note that, apart from the different names for the constants, the only difference between the two scripts is the inclusion of $\neg_C\neg_C E$ in M_C.

We shall omit the subscripts $_C$ and $_I$ when there is no risk of confusion.

5.2. *Inclusion*

If an LTT L_1 is a subsystem of the LTT L_2, we can easily reuse proof scripts from L_1 as proof scripts in L_2. The mapping Φ here is the inclusion mapping.

For example, LTT_{int} is a subsystem of $\text{LTT}_{\text{class}}$. We can easily write a module M_{ItoC} that describes the inclusion mapping:

`import` M_C

$[\mathbf{Type}_{\mathtt{I}} = \mathbf{Type}_{\mathtt{C}}]$;
$[\text{El}_{\mathtt{I}} = \text{El}_{\mathtt{C}}]$;

$[\mathbf{Prop}_{\mathtt{I}} = \mathbf{Prop}_{\mathtt{C}}]$;
$[\mathbf{Prf}_{\mathtt{I}} = \mathbf{Prf}_{\mathtt{C}}]$;
[`notI` = `notC`];
[`notII` = `notCI`];
[`orI` = `orC`];
⋮

Any code that parses under M_C will also parse under M_{ItoC}.

	M_C		M_I
	`Topkind` $\mathbf{Type}_C$ El_C		`Topkind` $\mathbf{Type}_I$ El_I
	`Topkind` $\mathbf{Prop}_C$ Prf_C		`Topkind` $\mathbf{Prop}_I$ Prf_I
`eqC` :	$(A{:}\mathbf{Type}_C)A \to A \to \mathbf{Prop}_C$	`eqI` :	$(A{:}\mathbf{Type}_I)A \to A \to \mathbf{Prop}_I$
`eqCI` :	$(A{:}\mathbf{Type}_C)(a{:}A)\mathbf{Prf}_C(a =_C a)$	`eqII` :	$(A{:}\mathbf{Type}_I)(a{:}A)\mathbf{Prf}_C(a =_I a)$
`eqCE` :	$(A{:}\mathbf{Type}_C)(P{:}A \to \mathbf{Prop}_C)$ $(a, b{:}A)$ $\mathbf{Prf}_C(a =_C b) \to \mathbf{Prf}_C(Pa) \to$ $\mathbf{Prf}_C(Pb)$	`eqIE` :	$(A{:}\mathbf{Type}_I)(P{:}A \to \mathbf{Prop}_I)$ $(a, b{:}A)$ $\mathbf{Prf}_I(a =_I b) \to \mathbf{Prf}_I(Pa) \to$ $\mathbf{Prf}_I(Pb)$
`notC` :	$\mathbf{Prop}_C \to \mathbf{Prop}_C$	`notI` :	$\mathbf{Prop}_I \to \mathbf{Prop}_I$
`notCI` :	$(p, q{:}\mathbf{Prop}_C)$ $(\mathbf{Prf}_C(p) \to \mathbf{Prf}_C(q)) \to$ $(\mathbf{Prf}_C(p) \to \mathbf{Prf}_C(\neg_C q)) \to$ $\mathbf{Prf}_C(\neg_C p)$	`notII` :	$(p, q{:}\mathbf{Prop}_I)$ $(\mathbf{Prf}_I(p) \to \mathbf{Prf}_I(q)) \to$ $(\mathbf{Prf}_I(p) \to \mathbf{Prf}_I(\neg_I q)) \to$ $\mathbf{Prf}_I(\neg_I q)$
`notnotCE` :	$(p{:}\mathbf{Prop}_C)\mathbf{Prf}_C(\neg_C\neg_C p) \to$ $\mathbf{Prf}_C(p)$		
`orC` :	$\mathbf{Prop}_C \to \mathbf{Prop}_C \to \mathbf{Prop}_C$	`orI` :	$\mathbf{Prop}_I \to \mathbf{Prop}_I \to \mathbf{Prop}_I$
`orCIl` :	$(p, q{:}\mathbf{Prop}_C)\mathbf{Prf}_C(p) \to$ $\mathbf{Prf}_C(p \vee_C q)$	`orIIl` :	$(p, q{:}\mathbf{Prop}_I)\mathbf{Prf}_I(p) \to$ $\mathbf{Prf}_I(p \vee_C q)$
`orCIr` :	$(p, q{:}\mathbf{Prop}_C)\mathbf{Prf}_C(q) \to$ $\mathbf{Prf}_C(p \vee_C q)$	`orIIr` :	$(p, q{:}\mathbf{Prop}_I)\mathbf{Prf}_I(q) \to$ $\mathbf{Prf}_I(p \vee_I q)$
`orCE` :	$(p, q, r{:}\mathbf{Prop}_C)$ $(\mathbf{Prf}_C(p) \to \mathbf{Prf}_C(r)) \to$ $(\mathbf{Prf}_C(q) \to \mathbf{Prf}_C(r)) \to$ $\mathbf{Prf}_C(p \vee_C q) \to \mathbf{Prf}_C(r)$	`orIE` :	$(p, q, r{:}\mathbf{Prop}_I)$ $(\mathbf{Prf}_I(p) \to \mathbf{Prf}_I(r)) \to$ $(\mathbf{Prf}_I(q) \to \mathbf{Prf}_I(r)) \to$ $\mathbf{Prf}_I(p \vee_I q) \to \mathbf{Prf}_I(r)$

Fig. 5. The partial scripts that declare $\mathrm{LTT}_{\mathrm{class}}$ and $\mathrm{LTT}_{\mathrm{int}}$

This form of proof reuse was used in the formalisation of Weyl's predicative foundation of mathematics (Adams and Luo 2010). That formalisation involved two LTTs: the predicative $\mathrm{LTT}_{\mathrm{W}}$ and an impredicative extension. We defined the real numbers in $\mathrm{LTT}_{\mathrm{W}}$, then proved in $\mathrm{LTT}_{\mathrm{W}}$ the theorem that every set of rationals bounded above has a (real) least upper bound, and then reused that proof to prove in the impredicative LTT that every set of *reals* bounded above has a least upper bound.

5.3. *The double negation translation*

The *double negation translation*, or *Gödel–Gentzen negative translation* (Gödel 1933), is a mapping from classical logic to intuitionistic logic. That is, it is a mapping that transforms a first-order formula ϕ into a first-order formula $\phi^{\neg\neg}$, such that if ϕ is a theorem of classical logic then $\phi^{\neg\neg}$ is a theorem of intuitionistic logic.

The mapping is defined as follows:

$$\begin{aligned}
\alpha^{\neg\neg} &\equiv \neg\neg\alpha \qquad (\alpha \text{ atomic})\\
(\neg\phi)^{\neg\neg} &\equiv \neg\phi^{\neg\neg}\\
(\phi \wedge \psi)^{\neg\neg} &\equiv \phi^{\neg\neg} \wedge \psi^{\neg\neg}\\
(\phi \vee \psi)^{\neg\neg} &\equiv \neg(\neg\phi^{\neg\neg} \wedge \neg\psi^{\neg\neg})\\
(\phi \to \psi)^{\neg\neg} &\equiv \phi^{\neg\neg} \to \psi^{\neg\neg}\\
(\forall x\phi)^{\neg\neg} &\equiv \forall x\phi^{\neg\neg}\\
(\exists x\phi)^{\neg\neg} &\equiv \neg\forall x\neg\phi^{\neg\neg}.
\end{aligned}$$

To prove the soundness of the double negation translation, the most important step is the following lemma, which is proved by induction on ϕ.

Lemma 5.1. For any formula ϕ, the formula $\phi^{\neg\neg}$ is *stable*, that is, $\neg\neg\phi^{\neg\neg} \supset \phi^{\neg\neg}$ is provable in intuitionistic logic.

It is then quite straightforward to prove, by induction on the derivation of ϕ, that if ϕ is provable in classical logic, then $\phi^{\neg\neg}$ is provable in intuitionistic logic. We wish to write a proof script M_{DN} that imports M_I and then defines every constant that was declared in M_C.

5.3.1. *First attempt.* For our first attempt, we simply define

[$\mathbf{Type}_{\mathtt{C}} = \mathbf{Type}_{\mathtt{I}}$];
[$\mathrm{El}_{\mathtt{C}} = \mathrm{El}_{\mathtt{I}}$];
[$\mathbf{Prop}_{\mathtt{C}} = \mathbf{Prop}_{\mathtt{I}}$];
[$\mathbf{Prf}_{\mathtt{C}} = \mathbf{Prf}_{\mathtt{I}}$];

and then define `eqC`, `notC`, and so on, as follows:

[$\mathtt{eqC} = [\mathtt{A}{:}\mathbf{Type}_{\mathtt{C}}]\ [\mathtt{a},\mathtt{b}{:}\mathtt{A}]\ \neg_{\mathtt{I}}\neg_{\mathtt{I}}(\mathtt{a} =_{\mathtt{I}} \mathtt{b})$];
[$\mathtt{notC} = \mathtt{notI}$];
[$\mathtt{orC} = [\mathtt{p},\mathtt{q}{:}\mathbf{Prop}_{\mathtt{C}}]\ \neg_{\mathtt{I}}(\neg_{\mathtt{I}}\mathtt{p} \wedge_{\mathtt{I}} \neg_{\mathtt{I}}\mathtt{q})$];
$\vdots$

This module 'maps' formulas of $\mathrm{LTT}_{\mathrm{class}}$ to their double-negation translation, in the sense that an expression that denotes an proposition ϕ in $\mathrm{LTT}_{\mathrm{class}}$ will expand, under these definitions, to an expression that denotes $\phi^{\neg\neg}$ in $\mathrm{LTT}_{\mathrm{int}}$. For example, the expression $P \vee_C \neg_C P$ expands under the above definitions to $\neg_I(\neg_I P \wedge_I \neg_I\neg_I P)$.

However, this script will not work as the kind of `orCE` is then (omitting the $\mathbf{Prf}_I$s)

$$(P, Q, R{:}\mathbf{Prop}_I)(P \to R) \to (Q \to R) \to \neg(\neg P \wedge \neg Q) \to R,$$

and this kind is uninhabited in $\mathrm{LTT}_{\mathrm{int}}$.

This is because the corresponding deduction rule

$$\dfrac{\neg(\neg\phi \wedge \neg\psi) \qquad \begin{array}{c}[\phi]\\ \vdots\\ \chi\end{array} \qquad \begin{array}{c}[\psi]\\ \vdots\\ \chi\end{array}}{\chi}$$

is not admissible in intuitionistic logic.

Looking at the proof of soundness, we see that we somehow need to use the fact that $\phi^{\neg\neg}$ is always stable (Lemma 5.1). This gives us the idea for our second and successful attempt.

5.3.2. *Second attempt.* We must not map $\mathbf{Prop}_C$ to the kind of all propositions, but rather to the kind of all *stable* propositions. Ideally, we would like to write

$$\mathbf{Prop}_C = \Sigma p{:}\mathbf{Prop}_I.\mathbf{Prf}_I(\neg_I\neg_I p \supset_I p).$$

However, $\mathrm{LF}_{\mathrm{LTT}}$ does not have these Σ-kinds at present.

One option would be to extend $\mathrm{LF}_{\mathrm{LTT}}$ with Σ-kinds, or some similar feature. This is an option that the authors intend to explore in the future.

As an alternative, we instead declare in M_{DN} the kind $\mathbf{Prop}_C$, the constructor $\mathbf{Prf}_C$, and the following introduction, elimination and computation rules:

$[\mathtt{PropCI}{:}(\mathtt{p}{:}\mathbf{Prop}_{\mathtt{I}})\mathbf{Prf}_{\mathtt{I}}(\neg_{\mathtt{I}}\neg_{\mathtt{I}}\mathtt{p} \supset_{\mathtt{I}} \mathtt{p}) \to \mathbf{Prop}_{\mathtt{C}}];$
$[\mathtt{PI1}{:}\mathbf{Prop}_{\mathtt{C}} \to \mathbf{Prop}_{\mathtt{I}}];$
$[\mathtt{PI2}{:}(\mathtt{p}{:}\mathbf{Prop}_{\mathtt{C}})\mathbf{Prf}_{\mathtt{I}}(\neg_{\mathtt{I}}\neg_{\mathtt{I}}(\mathtt{PI1\,p}) \supset_{\mathtt{I}} \mathtt{PI1\,p})];$
$[\mathtt{PI1}(\mathtt{PropCI\,p\,f}) = \mathtt{p}];$

$[\mathtt{PrfCI}{:}(\mathtt{p}{:}\mathbf{Prop}_{\mathtt{C}})\mathbf{Prf}_{\mathtt{I}}(\mathtt{PI1\,p}) \to \mathbf{Prf}_{\mathtt{C}}(\mathtt{p})];$
$[\mathtt{PrfCE}{:}(\mathtt{p}{:}\mathbf{Prop}_{\mathtt{C}})\mathbf{Prf}_{\mathtt{C}}(\mathtt{p}) \to \mathbf{Prf}_{\mathtt{I}}(\mathtt{PI1\,p})];$

The constructor $\mathbf{Prf}_C$ is defined as follows:

$[\mathbf{Prf}_{\mathtt{C}} = [\mathtt{p}{:}\mathbf{Prop}_{\mathtt{C}}]\ \mathbf{Prf}_{\mathtt{I}}(\mathtt{PI1\,p})];$

We can now proceed to define the constants of M_C. The connective $\vee_C$, for example, must now be defined as a binary function on this 'Sigma-kind':

$$\begin{array}{rcl}\mathtt{orC} & : & \mathbf{Prop}_C \to \mathbf{Prop}_C \to \mathbf{Prop}_C\\ (p,f) \vee_C (q,g) & \equiv & (\neg_I(\neg_I p \wedge_I \neg_I q), h)\end{array}$$

where h is a proof that $\neg_I(\neg_I p \wedge_I \neg_I q)$ is stable.

Written out in full, we have

$$\mathtt{orC} = [p,q{:}\mathbf{Prop}_C]\ \mathbf{Prop}_C I\ (\neg_I(\neg_I(PI1\ p) \wedge_I \neg_I(PI1\ q))\cdots$$

where $\cdots$ elides a proof of

$$\neg\neg\neg(\neg(PI1\ p) \wedge \neg(PI1\ q)) \supset \neg(\neg(PI1\ p) \wedge \neg(PI1\ q)).$$

The kind of `orCE` expands under the above definitions to

$$(P, Q, R{:}\mathbf{Prop}_C)(\mathbf{Prf}_I(P_1) \to \mathbf{Prf}_I(R_1)) \to (\mathbf{Prf}_I(Q_1) \to \mathbf{Prf}_I(R_1)) \to \\ \mathbf{Prf}_I(\neg_I(\neg_I P_1 \wedge_I \neg_I Q_1)) \to \mathbf{Prf}_I(R_1),$$

which is inhabited. The inhabitant we construct makes use of $R_2{:}\mathbf{Prf}_I(\neg_I\neg_I R_1 \supset_I R_1)$.

It is possible to define every constant in M_C in this fashion, and any module that imports M_C will parse if it is changed to import M_{DN} instead. We have:

— If an expression denotes a proposition ϕ under M_C, then under M_{DN} it denotes a pair consisting of $\phi^{\neg\neg}$ and a proof that $\phi^{\neg\neg}$ is stable.

— If an expression denotes a proof P of ϕ under M_C (that is, $P{:}\mathbf{Prf}_C(\phi)$), then under M_{DN} it denotes a proof of $\phi^{\neg\neg}$.

5.3.3. *Application.* As an application of this work, we can show that if $\mathrm{LTT}_{\mathrm{int}}$ is consistent, then $\mathrm{LTT}_{\mathrm{class}}$ is consistent. Suppose we had a proof script that imports M_C, and then constructs an object of type $\mathbf{Prf}_C(\bot_C)$. Then the same proof script could import M_{DN} instead, in which case it would construct an object of type $\mathbf{Prf}_I(\bot_I)$.

5.4. *The A-translation*

The *A-translation* (Friedman 1978) is a mapping from intuitionistic logic to intuitionistic logic. We fix a formula A, and then define the formula ϕ^A for every formula ϕ as follows:

$$\begin{aligned} P^A &\equiv P \vee A \qquad (P \text{ atomic}) \\ (\neg\phi)^A &\equiv \phi^A \supset A \\ (\phi * \psi)^A &\equiv \phi^A * \psi^A \qquad (* \equiv \wedge, \vee, \supset) \\ (Qx\phi)^A &\equiv Qx\phi^A \qquad (Q \equiv \forall, \exists) \end{aligned}$$

This translation is sound: if ϕ is a theorem of an intuitionistic theory T, then so is ϕ^A (Friedman 1978). The following lemma is the important lemma in the proof of this theorem.

Lemma 5.2. For any formula ϕ, we have $\vdash A \to \phi^A$.

We can make use of the A-translation for proof reuse as follows (we write two copies of a script that defines $\mathrm{LTT}_{\mathrm{int}}$, say M_I and M_I'):

M_I			M_I'		
	Topkind $\mathbf{Prop}_I$ $\mathbf{Prf}_I$			Topkind $\mathbf{Prop}_I'$ $\mathbf{Prf}_I'$	
eqI	:	$(A{:}\mathbf{Type}_I)A \to A \to \mathbf{Prop}_I$	eqI′	:	$(A{:}\mathbf{Type}_I')A \to A \to \mathbf{Prop}_I'$
notI	:	$\mathbf{Prop}_I \to \mathbf{Prop}_I$	notI′	:	$\mathbf{Prop}_I' \to \mathbf{Prop}_I'$
andI	:	$\mathbf{Prop}_I \to \mathbf{Prop}_I \to \mathbf{Prop}_I$	andI′	:	$\mathbf{Prop}_I' \to \mathbf{Prop}_I' \to \mathbf{Prop}_I'$
	⋮			⋮	

We construct our module M_A defining the A-translation as follows. We assume that M_I has been imported and an object A:$\mathbf{Prop}_I$ has been defined. We now define every constant declared in M'_I. The constant $\mathbf{Prop}'_I$ is defined to be the kind of all propositions ϕ:$\mathbf{Prop}_I$ such that $A \supset \phi$. Again, we would like to write

$$\begin{aligned}
\mathbf{Prop}'_I &= \Sigma p{:}\mathbf{Prop}_I.\mathbf{Prf}_I(A \supset_I p) \\
\mathbf{Prf}'_I(\phi, f) &= \mathbf{Prf}_I(\phi) \\
a ='_I b &= (a =_I b \vee_I A, \cdots) \\
\neg'_I(\phi, f) &= (\phi \supset_I A, \cdots) \\
(\phi, f) \vee'_I (\psi, g) &= (\phi \vee_I \psi, \cdots) \\
&\vdots
\end{aligned}$$

But as LF_{LTT} does not have Σ-kinds, we instead declare the kind $\mathbf{Prop}'_I$, the constructor $\mathbf{Prf}'_I$ and the following constants:

$$\begin{aligned}
\mathbf{Prop}'_I I &: (p{:}\mathbf{Prop}_I)\mathbf{Prf}_I(A \supset_I p) \to \mathbf{Prop}'_I \\
PI1 &: \mathbf{Prop}'_I I \to \mathbf{Prop}_I \\
PI2 &: (p{:}\mathbf{Prop}'_I)\mathbf{Prf}_I(A \supset_I PI1\ p) \\
\mathbf{Prf}_I 2\mathbf{Prf}'_I &: (p{:}\mathbf{Prop}_I)(f{:}\mathbf{Prf}_I(A \supset_I p))\mathbf{Prf}'_I(\mathbf{Prop}'_I I\ p\ f) \to \mathbf{Prf}_I(p) \\
\mathbf{Prf}'_I 2\mathbf{Prf}_I &: (p{:}\mathbf{Prop}_I)(f{:}\mathbf{Prf}_I(A \supset_I p))\mathbf{Prf}_I(p) \to \mathbf{Prf}'_I(\mathbf{Prop}'_I I\ p\ f) \\
a ='_I b &= \mathbf{Prop}'_I I(a =_I b \vee_I A)(\cdots) \\
\neg'_I p &= \mathbf{Prop}'_I I(PI1\ p \supset_I A)(\cdots) \\
p \vee'_I q &= \mathbf{Prop}'_I I(PI1\ p \vee_I PI1\ q)(\cdots)
\end{aligned}$$

together with the computation rule

$$PI1(\mathbf{Prop}'_I I\ p\ f) = p{:}\mathbf{Prop}_I.$$

Now any proof script beginning

`import` M'_I;

will also parse if we replace this line with

`import` M_I;
`[A =` $\cdots$ `:`$\mathbf{Prop_I}$`];`
`import` M_A;

As an application of the A-translation, we can show that Markov's law is admissible for quantifier-free formulas.

Theorem 5.3. Let T be an intuitionistic theory. If $T \vdash \neg\neg\exists x\phi$, where ϕ is quantifier-free (possibly with free variables other than x), then $T \vdash \exists x\phi$.

Proof. Let $A \equiv \exists x\phi$. If $T \vdash \neg\neg\exists x\phi$, then, by the soundness of the A-translation, $T \vdash (\exists x\phi^A \to A) \to A$.

It is now easy to show that $\phi^A \vdash \phi \vee A$ and $\phi \vee A \vdash \phi^A$ for ϕ quantifier-free. Thus, we have $T \vdash (\exists x(\phi \vee A) \to A) \to A$. But we have $\exists x\phi \to A$ and $\exists xA \to A$, so $T \vdash \exists x(\phi \vee A) \to A$, and thus $T \vdash A$, that is, $T \vdash \exists x\phi$. $\square$

We can make use of this result as follows: given any proof of $\neg'_I \neg'_I \exists x\phi$ under M'_I, we can obtain a proof of $\exists x\phi$ under M_I.

Suppose we have a proof script M_α that imports M'_I and proves $\neg\neg\exists x\phi$:

import M'_I
$\vdots$
$[\mathrm{P}' = \cdots : \mathbf{Prop}'_{\mathrm{I}}]$;
Claim $\mathrm{H}:\mathbf{Prf}_{\mathrm{I}'}(\neg'_{\mathrm{I}}\neg'_{\mathrm{I}}\exists'_{\mathrm{I}}\mathrm{x}{:}\mathrm{T}.\mathrm{P}'\mathrm{x})$;
Proof
$\vdots$
Qed

Then we can produce a proof script that imports M_I and proves $\exists x\phi$:

import M_I;
$\vdots$
$[\mathrm{P} = \cdots : \mathbf{Prop}_{\mathrm{I}}]$;
$[\mathrm{A} = \exists_{\mathrm{I}}\mathrm{x}{:}\mathrm{T}.\mathrm{Px}]$;
import M_α;

Claim $\mathrm{K}:\mathbf{Prf}_{\mathrm{I}}(\exists_{\mathrm{I}}\mathrm{x}{:}\mathrm{T}.\mathrm{Px})$;
Proof
$\vdots$
Qed

In this script, the line defining P (line 3) is the result of replacing $\wedge_{I'}$ with $\wedge_I$, and $\vee_{I'}$ with $\vee_I$, and so on, in line 6.

The proof K makes use of H, which is now a proof of $((\exists_I x{:}T.(Px \vee_I A)) \supset_I A) \supset_I A$.

5.4.1. *Remark.* It is unsatisfactory to have to work in two copies of $\mathrm{LTT}_{\mathrm{int}}$, and to have separate definitions of P and P'. We would like to be able to take a script that proves $\neg\neg\exists xPx$ in $\mathrm{LTT}_{\mathrm{int}}$ and produce a script that proves $\exists xPx$ *in* $\mathrm{LTT}_{\mathrm{int}}$. This requires a more sophisticated module mechanism than the one that currently exists in Plastic.

5.5. *The Russell–Prawitz modality*

The following mapping from a first-order language to a second-order language was first introduced in Russell (1903) and was named *the Russell–Prawitz modality* by Aczel

(Aczel 2001):

$$
\begin{aligned}
[\![P]\!] &\equiv P \qquad (P \text{ atomic}) \\
[\![\neg\phi]\!] &\equiv \forall p.\, [\![\phi]\!] \supset p \\
[\![\phi \wedge \psi]\!] &\equiv \forall p.\, [\![\phi]\!] \supset [\![\psi]\!] \supset p \\
[\![\phi \vee \psi]\!] &\equiv \forall p.([\![\phi]\!] \supset p) \supset ([\![\psi]\!] \supset p) \supset p \\
[\![\phi \supset \psi]\!] &\equiv [\![\phi]\!] \supset [\![\psi]\!] \\
[\![\forall x \phi]\!] &\equiv \forall x\, [\![\phi]\!] \\
[\![\exists x \phi]\!] &\equiv \forall p.(\forall x\, [\![\phi]\!] \supset p) \supset p
\end{aligned}
$$

It is easy to turn this mapping into a mapping between two LTTs, which can then be handled by our method.

Let LTT_1 be a first-order LTT with connectives `not1`, `and1`, `or1`, `imp1`, `all1` and `ex1`. Let LTT_2 be a second-order LTT with connective `imp2`, a first-order quantifier

$$\mathtt{all2}:(A:\mathbf{Type}_2)(A \to \mathbf{Prop}_2) \to \mathbf{Prop}_2$$

and a second-order quantifier

$$\mathtt{All2}:(\mathbf{Prop}_2 \to \mathbf{Prop}_2) \to \mathbf{Prop}_2.$$

We write

$$
\begin{aligned}
\phi \supset_2 \psi &\quad \text{for} \quad \mathtt{imp2}\ \phi\ \psi \\
\forall_2 x{:}A.\phi &\quad \text{for} \quad \mathtt{all2}\ A\ [x{:}A]\phi \\
\forall_2 p{:}\mathbf{Prop}_2.\phi &\quad \text{for} \quad \mathtt{All2}\ [p{:}\mathbf{Prop}_2]\phi.
\end{aligned}
$$

Let M_1 and M_2 be two proof scripts that declare these two LTTs.

We can write a module M_{RP} that imports M_2, and then defines every constant declared in M_1:

$$
\begin{aligned}
\mathbf{Type}_1 &= \mathbf{Type}_2 \\
\mathbf{Prop}_1 &= \mathbf{Prop}_2 \\
\mathbf{Prf}_1 &= \mathbf{Prf}_2 \\
\mathtt{not1} &= [p{:}\mathbf{Prop}_1]\forall q{:}\mathbf{Prop}_2.p \supset_2 q \\
\mathtt{and1} &= [p, q{:}\mathbf{Prop}_1]\forall r{:}\mathbf{Prop}_2.p \supset_2 q \supset_2 r \\
\mathtt{or1} &= [p, q{:}\mathbf{Prop}_1]\forall r{:}\mathbf{Prop}_2.(p \supset_2 r) \supset_2 (q \supset_2 r) \supset_2 r \\
\mathtt{imp1} &= \mathtt{imp2} \\
\mathtt{all1} &= \mathtt{all2} \\
\mathtt{ex1} &= [A{:}\mathbf{Type}_1][P{:}A \to \mathbf{Prop}_1]\forall_2 p{:}\mathbf{Prop}_2.(\forall_2 x{:}A.Px \supset_2 p) \supset_2 p.
\end{aligned}
$$

Thus, our method of proof reuse can be applied to the Russell–Prawitz modality.

5.6. *Other applications*

The formalisation examples discussed in this paper have all concentrated on translations that redefine the logical connectives, while leaving the world of data types unchanged. As we mentioned above (*cf.* the remark at the end of Section 4), the LTT-approach to reuse also allows the possibility of changing the world of data types in formalisations. An example of this would be to reuse the results in the formalisation of predicative mathematics in the formalisation of impredicative mathematics – this is studied in Adams and Luo (2010).

There could be further applications, such as:

(1) Translations that redefine the type constructors.
For example, there is a translation from System T to System F described in Girard *et al.* (1990):

$$\begin{aligned} [\![Bool]\!] &= \Pi X.X \to X \to X \\ [\![Nat]\!] &= \Pi X.X \to (X \to X) \to X \\ [\![A \to B]\!] &= [\![A]\!] \to [\![B]\!] \\ [\![A \times B]\!] &= \Pi X.([\![A]\!] \to [\![B]\!] \to X) \to X. \end{aligned}$$

Our method of proof reuse can be applied in this case. The module declaring System T will declare constants

$$\begin{aligned} Bool &: \mathbf{Type} \\ Nat &: \mathbf{Type} \to :\mathbf{Type} \to \mathbf{Type} \to \mathbf{Type} \\ \times &: \mathbf{Type} \to \mathbf{Type} \to \mathbf{Type}. \end{aligned}$$

The module M_Φ will redefine these constants.

(2) If we have two first-order systems S and T, and every axiom of S is a theorem of T, then every theorem of S is a theorem of T.
Our method of proof reuse can be applied in this case. For each axiom α of S, the module M_S will contain a constant declaration c_α:$\mathbf{Prf}(\alpha)$. The corresponding line in M_Φ will be a proof of α in T.
However, it appears that our method cannot handle *interpretations* between first-order theories. An *interpretation* (in the sense of Shoenfield (1967)) between S and T maps (for example) a unary function symbol f of S to a *formula* $\phi[x, y]$ of T such that $T \vdash \forall x \exists ! y \phi[x, y]$.
This does not fit into the pattern of the translations above, and there is no obvious way to adapt our method to this situation without changing the theories themselves (for example, by adding a unique choice operator).

6. Conclusions

We have demonstrated an original method for proof reuse when conducting formalisations that make use of more than one LTT. For some translations Φ between an LTT S and an

LTT T it is possible to write a module M_Φ such that, if a proof script imports M_S and proves α, then changing the script to import M_Φ will give a proof of $\Phi(\alpha)$ in T.

The case studies we have discussed in this paper are all about the logical components. As we have pointed out in various places in the paper, it is possible to consider translations that change the data type parts (for example, to change from predicative mathematics to impredicative mathematics). We expect further development in this respect, with further case studies investigating how such non-logical changes can be made and used in practice.

This method should be quite general, and applicable to work in type theories and other systems of logic. However, if one wishes to formalise a large piece of mathematics that involves proving results in several different foundations, then LTTs would seem to be particularly suitable, as they allow for a uniform presentation and treatment of a wide range of different foundations.

For future work, we wish to formalise one such piece of mathematics using Plastic and this method of proof reuse.

6.1. *Related work*

There has been considerable work on proof reuse in recent years. For the most part, this work has concentrated on the problem of allowing users of different proof assistants to share one another's work. This is a different problem to the one we consider in this paper, which is how to formalise a piece of mathematics that involves several different foundations. Nevertheless, the two lines of research should be able to benefit from one another.

6.1.1. *Logosphere.*

The large project Logosphere, which was initiated by Schürmann, Pfenning, Kohlhase and Owre, aims to build a large library of formalised proofs that users of many different proof assistants can all contribute to and make use of. It does this by representing the many different systems, and the translations between them, in the Edinburgh LF. In particular, Howe's translation from HOL to NuPRL (Howe 1996; 1998) has been formalised and verified to be correct using the proof assistant Twelf (Schürmann and Stehr 2006).

The representation of HOL in ELF consists of a type $\mathtt{tp{:}type}$ to represent the types of HOL, together with a function $\mathtt{tm{:}tp} \to \mathtt{type}$ to represent the terms of each type. The representation of Nuprl consists of a type $\mathtt{n-tm{:}type}$ to represent the terms of Nuprl. The translation Φ is then represented by two functions

$$\begin{aligned} \mathtt{transtp} &: \mathtt{tp} \to \mathtt{n-tm} \to \mathtt{type} \\ \mathtt{transtm} &: \mathtt{tm}\ A \to \mathtt{n-tm} \to \mathtt{type} \end{aligned}$$

with $\mathtt{transtp}\ A\ B$ being inhabited if and only if $\Phi(A) = B$. Twelf is able to verify that `transtp` and `transtm` are total functions that satisfy the desired properties.

Their approach is thus intended to be used to translate between two systems implemented in two different proof checkers, using a logical framework implemented in a third. Our approach is intended to be used by someone using just one proof checker, which implements a logical framework.

The most important difference is that they represent the translation as a pair of objects in the logical framework, and formally verify its correctness. We do not represent the translation as an object. Instead, it is effected by replacing a set of declarations with a set of definitions.

It remains to be seen how the two approaches compare in practice. However, we anticipate that our approach should be more convenient for a pluralist formalisation, as there is less overhead for the user. The translations are invisible to the user. Given a translation $\Phi : S \to T$, when the user is working in T, there is no need for them to invoke Φ in order to make use of the results proved in S. A change in the proof scripts produced in S will produce an immediate change in the theorems that are visible to the user and available for use in T.

6.1.2. *Little theories.* The proof assistant IMPS is designed to use the *little theories* methodology (Farmer 2000), whereby a number of different theories and translations between them are specified using a version of higher-order logic called LUTINS (Farmer *et al.* 1990). A translation from one theory T_1 to another T_2 is specified by a mapping from the constants of T_1 to the expressions of T_2 satisfying certain conditions, such that the translation of every axiom of T_1 is a theorem of T_2.

The LTTs we have been dealing with in this paper have a richer type structure than the theories that can be specified in LUTINS. Our LTTs include dependent types, inductive types and computation rules. Apart from this difference, the class of translations that can be handled by the two methods is remarkably similar.

However, IMPS required support for translations to be built into the implementation. We have shown in this paper that, when working with a logical framework, a lightweight mechanism for supporting these translations is automatically available for free.

The work that has been done in IMPS shows how useful it can be to work with a variety of theories and translations in the course of a formalisation. We hope that the programme of research proposed in this paper will prove that this is also true when we are working in LTTs.

6.1.3. *Other methods of proof reuse.* The method of proof reuse between different systems of logic presented in this paper is, to the best of the authors' knowledge, original.

Previous work on proof reuse in dependent type theories has concentrated on proof reuse within a single type theory in the following situations:

(1) Given an isomorphism between types A and B, to automatically generate proofs about B from proofs about A (Beckert and Klebanov 2004).
(2) Given an inductive type A and an inductive type B formed by extending A with new constructors, to reuse proofs about A to interactively generate proofs about B (Boite 2004).
(3) The project LATIN (Iancu and Rabe 2011) has similar aims to Logosphere.
(4) Garillot and Gonthier's method of *mathematical components* (Garillot *et al.* 2009) is a disciplined way of systematically organising the development of large formalisations involving many algebraic structures, in such a way that (for example) results about groups can be applied to rings. This is not the same problem as the one considered

in this paper, which involves translations between systems that involve changing both the type structure and the logic. Nevertheless, there are superficial similarities, and it remains to be seen if the two methods have anything to offer each other.

6.2. *Future work*

Plastic currently has a very primitive module mechanism, which, nevertheless, was sufficient for the work in this paper. However, the method we have presented relies on editing proof scripts to change the files they import, which is very inconvenient in practice.

A more sophisticated module system that allowed *parameterised* modules would make this form of proof reuse much more convenient. We would like to implement such a module mechanism in Plastic in future work.

We would also like to establish a better theoretical basis for this type of work. We would like to establish some criteria for when a translation can be represented by a module using our method. We would like to study the theory of modules, module interfaces and module functors described above. The result will likely be very similar to the theory of *institutions* (Goguen and Burstall 1984), since there are many superficial similarities. However, institutions are not exactly what we need, since institutions are a model-theoretic notion, whereas we require a syntactic notion.

Acknowledgements

We would like to thank the anonymous referees for their very helpful and detailed comments, and particularly for making us aware of related work in the area of proof reuse.

References

Aczel, P. (2001) The Russell-Prawitz modality. *Mathematical Structures in Computer Science* **11** (4) 541–554.

Aczel, P. and Gambino, N. (2002) Collection principles in dependent type theory. In: Luo, Z., McKinna, J. and Pollack, R. (eds.) Types for Proofs and Programs. *Springer-Verlag Lecture Notes in Computer Science* **2277** 1–23.

Adams, R. and Luo, Z. (2007) Weyl's predicative classical mathematics as a logic-enriched type theory. In: Altenkirch, T. and McBride, C. (eds.) Types for Proofs and Programs. *Springer-Verlag Lecture Notes in Computer Science* **4502** 1–17.

Adams, R. and Luo, Z. (2010) Weyl's predicative classical mathematics as a logic-enriched type theory. *ACM Transactions on Computational Logic* **11** (2) 1–29.

Agda (2008) The Agda proof assistant. (Available at `http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php`.)

Beckert, B. and Klebanov, V. (2004) Proof reuse for deductive program verification. In: *Proceedings of the Software Engineering and Formal Methods, Second International Conference (SEFM '04)*, IEEE Computer Society 77–86.

Bertot, Y. and Castéran, P. (2004) *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*, Texts in Theoretical Computer Science, Springer-Verlag.

Boite, O. (2004) Proof reuse with extended inductive types. In: Slind, K., Bunker, A. and Gopalakrishnan, G. (eds.) Theorem Proving in Higher Order Logics. *Springer-Verlag Lecture Notes in Computer Science* **3223** 50–65.

Callaghan, P. C. and Luo, Z. (2001) An implementation of typed LF with coercive subtyping and universes. *Journal of Automated Reasoning* **27** (1) 3–27.

Constable, R. *et al.* (1986) *Implementing Mathematics with the NuPRL Proof Development System*, Prentice-Hall.

Coq (2004) *The Coq Proof Assistant Reference Manual (Version 8.0), INRIA*, The Coq Development Team.

Coquand, T. and Huet, G. (1988) The calculus of constructions. *Information and Computation* **76** (2-3) 95–120.

Coquard, T. and Paulin-Mohring, C. (1990) Inductively defined types. In: Martin-Löf, P. and Mints, G. (eds.) International Conference in Computer Logic (COLOG-88). *Springer-Verlag Lecture Notes in Computer Science* **417** 50–66.

Dybjer, P. (1991) Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In: Huet, G. and Plotkin, G. (eds.) *Logical Frameworks*, Cambridge University Press 280–306.

Farmer, W., Guttman, J. and Thayer, F. (1990) IMPS : An interactive mathematical proof system. In: Stickel, M. (ed.) 10th International Conference on Automated Deduction. *Springer-Verlag Lecture Notes in Computer Science* **449** 653–654.

Farmer, W. M. (2000) An infrastructure for intertheory reasoning. In: McAllester, D. (ed.) Automated Deduction – CADE-17. *Springer-Verlag Lecture Notes in Computer Science* **1831** 115–131.

Feferman, S. (2005) Predicativity. In: Shapiro, S. (ed.) *The Oxford Handbook of Philosophy of Mathematics and Logic*, Oxford University Press.

François Garillot, F., Gonthier, G., Mahboubi, A. and Rideau, L. (2009) Packaging Mathematical Structures. In: Nipkow, T. and Urban, C. (eds.) Proceedings Theorem Proving in Higher Order Logics. *Springer-Verlag Lecture Notes in Computer Science* **5674** 327–342.

Friedman, H. (1978) Classically and intuitionistically provable functions. In: *Higher Set Theory* 21–28.

Gambino, N. and Aczel, P. (2006) The generalised type-theoretic interpretation of constructive set theory. *Journal of Symbolic Logic* **71** (1) 67–103.

Girard, J.-Y. (1986) The system F of variable types, fifteen years later. *Theoretical Computer Science* **45** (2) 159–192.

Girard, J.-Y., Lafont, Y. and Taylor, P. (1990) *Proofs and Types*, Cambridge University Press.

Gödel, K. (1933) On intuitionistic arithmetic and number theory. *Collected Works* 287–295.

Goguen, H. (1994) *A Typed Operational Semantics for Type Theory*, Ph.D. thesis, University of Edinburgh.

Goguen, J. and Burstall, R. (1984) Introducing institutions. In: Clarke, E. and Kozen, D. (eds.) Logics of Programs *Springer-Verlag Lecture Notes in Computer Science* **164** 221–256.

Hájek, P. and Pudlák, P. (1998) *Metamathematics of First-Order Arithmetic*, Perspectives in Mathematical Logic **3**, Springer-Verlag.

Harper, R., Honsell, F. and Plotkin, G. (1987) A framework for defining logics. *Proc. 2nd Ann. Symp. on Logic in Computer Science*, IEEE.

Harper, R., Honsell, F. and Plotkin, G. (1993) A framework for defining logics. *Journal of the Association for Computing Machinery* **40** (1) 143–184.

Harrison, J. (1996) HOL Light: A tutorial introduction. In: Srivas, M. and Camilleri, A. (eds.) Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96). *Springer-Verlag Lecture Notes in Computer Science* **1166** 265–269.

Howe, D. J. (1996) Importing mathematics from HOL into Nuprl. In: von Wright, J., Grundy, J. and Harrison, J. (eds.) Proceedings of the Ninth International Conference on Theorem Proving in Higher-Order Logics (TPHOL '96). *Springer-Verlag Lecture Notes in Computer Science* **1125** 141–156.

Howe, D. J. (1998) Toward sharing libraries of mathematics between theorem provers. In: *Proceedings Frontiers of Combining Systems, FroCoS'98*, Kluwer Academic Publishers.

Iancu, M. and Rabe, F. (2011) Formalising foundations of mathematics. *Mathematical Structures in Computer Science* **21**.

Kohlenbach, U. (2005) Higher order reverse mathematics. In: Simpson, S. (ed.) *Reverse mathematics 2001*, Lecture Notes in Logic **21**, Association for Symbolic Logic 281–295.

Luo, Z. (1994) *Computation and Reasoning: A Type Theory for Computer Science*, Oxford University Press.

Luo, Z. (2003) PAL$^+$: a lambda-free logical framework. *Journal of Functional Programming* **13** (2) 317–338.

Luo, Z. (2006) A type-theoretic framework for formal reasoning with different logical foundations. In: Okada, M. and Satoh, I. (eds.) Proceedings of the 11th Annual Asian Computing Science Conference. *Springer-Verlag Lecture Notes in Computer Science* **4435** 214–222.

Luo, Z. and Pollack, R. (1992) LEGO Proof Development System: User's Manual. LFCS Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh.

Martin-Löf, P. (1984) *Intuitionistic Type Theory*, Bibliopolis.

Muzalewski, M. (1993) An Outline of PC Mizar. Fondation Philippe le Hodey, Brussels.

Nipkow, T., Paulson, L. C. and Wenzel, M. (2002) Isabelle/HOL: a proof assistant for higher-order logic. *Springer-Verlag Lecture Notes in Computer Science* **2283**.

Nordström, B., Petersson, K. and Smith, J. (1990) *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press.

Paulson, L. C. (1994) Isabelle: A Generic Theorem Prover. *Springer-Verlag Lecture Notes in Computer Science* **828**.

Pfenning, F. and Schürmann, C. (1999) Twelf – a meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) Automated Deduction – CADE-16. *Lecture Notes in Computer Science* **1632** 202–206.

Pollack, R. (1994) *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*, Ph.D. thesis, Edinburgh University.

Russell, B. (1903) *The Principles of Mathematics*, Routledge.

Schürmann, C. and Stehr, M.-O. (2006) An executable formalization of the HOL/Nuprl connection in the metalogical framework Twelf. In: Hermann, M. and Voronkov, A. (eds.) 11th International Conference on Logic for Programming Artificial Intelligence and Reasoning. *Springer-Verlag Lecture Notes in Computer Science* **4246** 150–166.

Schütte, K. (1965) Predicative well-orderings. *Studies in Logic and the Foundations of Mathematics* **40** 280–303.

Shoenfield, J. (1967) *Mathematical logic*, Addison-Wesley.

Simpson, S. G. (1999) *Subsystems of Second-Order Arithmetic*, Springer-Verlag.

Smith, J. (1988) The independence of Peano's fourth axiom from Martin-Löf's type theory without universes. *Journal of Symbolic Logic* **53** (3).

Weyl, H. (1918) *Das Kontinuum*. (English translation – The Continuum: a critical examination of the foundation of analysis, Dover 1994.)