Dependently Sorted Theorem-Proving for Mathematical Foundations

Yiming Xu

A thesis submitted for the degree of Doctor of Philosophy

Mathematical Sciences Institutes

28 February 2024

Declaration

Yiming Xu
28 February 2024

Acknowledgements

Firstly, I give my big thanks to my primary supervisor Michael Norrish, and my panel chair James Borger. I feel I am the luckiest person in the world for having you teaching me through all these years, to an extent that I think I must have saved the world in some previous life. Moreover, thanks to my associate supervisor Ranald Clauston, for his valuable feedback on my thesis. I will prove I deserve all your time, effort and expectations by continuing to do good work.

I express my deep thanks and best wishes to the most helpful senior student in the world, namely Bolin (Ameilia) Han. Thank you so much for always being considerate and kind to me for seven years. My thanks extend to my landlords Xilin Lu and Yile Liu. It is so much pleasure that I can live with you and all the three cats, Molly, Jesse, and Fufu. It is so much fun watching and listening to the cats. I wish all of you to be a happy across your life!

To my mother Shoufeng Zhang and my father Xiammin Xu, thank you for taking me to this brilliant world where I can carry out all this interesting research and learn about all these mathematics. Despite the fact of not very rich, you still always decide to do your best to make my life as happy as possible, and support my study all these years. I will do my best to pay you back. You made it possible for me to travel abroad, but you have not been traveling abroad yourself. I will take you to many countries after your retirement.

Abstract

In this thesis, we start with a picture of current existing theorem-proving systems. Then we describe our new meta-logical system (named DiaToM) for mechanizing foundations of mathematics. Using dependent sorts and first-order logic, our system (implemented as an LCF-style theorem-prover) improves on the state-of-the-art by providing efficient type-checking, convenient automatic rewriting, and interactive proof support. By formalizing the proof theory of the DiaToM logic, we verified that the most suspicious design, namely the "formula variable", does not break the first-orderness of our logic. We assess our implementation by axiomatizing Lawvere's Elementary Theory of the Category of Sets(ETCS), Shulman's Sets, Elements and Relations(SEAR), and McLarty's Categories of Categories as a Foundation of mathematics(CCAF). With a suitable choice of axioms, our system is sufficient to perform some basic mathematical constructions such as quotients, induction, and coinduction by constructing integers, lists, and co-lists. More interestingly, we can provide concrete examples of some constructions that are not possible to be carried out in simpler systems as HOL can be carried out with SEAR in our system. This demonstrates that our system is in a sweet spot of simplicity and expressiveness.

Contents

Figures

Tables

Chapter 1

Introduction

We present the problem we want to address, introduce the current candidate solutions to it, point out how they are lacking given our particular aims, and outline the rest of the thesis, which presents our proposed solution.

1.1 Overview

When most mathematicians describe their own work, they do not explicitly ground their work on any particular foundation. Similarly, there is often no explicit record of a piece of work's foundations when it appears as a publication. As a subject that admits the ultimate level of precision, a significant feature of mathematics is that it is possible to do it formally. However, to formalize a mathematical argument, we do require a primitive foundation, from which each step of our reasoning can be derived.

Then there is the question of which foundation we should use to formalize all this mathematics. The answer in short is "it depends on what we care about". Most foundations are designed with a particular aim in mind. Thus, it is important to know what the foundation is designed for, before choosing to reason with it, because the things it can capture and prove are different. The most obvious example is that some theorems can only be proved with the axiom of choice. One important difference is also evident by comparing ZF [28] and Morse-Kelley set theory [26], where the former does not allow expressing statements about proper classes at all, but the latter can. As another interesting aspect, the "size issues" [8] are a matter of concern around the topic of cardinality, which motivates the question of whether we want a universal set. In case we do, Quine's new foundation [43] gives us one. But it is actually not the only choice: Tarski–Grothendieck set theory [48] also provides such a set by adding an extra axiom to ZF, but in a different sense than NF. Therefore, the choice of foundation does make some difference.

Even if two foundations have the same power, the choice is still flexible based on our aesthetic preferences. One construction capturing the same intended idea may look neat in one

foundation, and look messy in another. The beauty of ZF in terms of being both powerful and simple is appreciated by many. But many find it inelegant that ZF can prove $1 \in 2$. To address this issue, people have developed structural set theories. The idea is to have two sorts, sets and elements. Instead of having everything being a set, we have each "thing" either a set or an element, but not both. With the restriction that the "membership" relation can only hold between a set and an element, and given that 1 and 2 are both elements of the same set, it does not make sense to write $1 \in 2$ anymore. The development of structural set theories was pioneered by William Lawvere, who wrote down the axioms of The Elementary Theory of the Category of Sets (ETCS) [29]. More recently, a more powerful theory, Sets, Elements and Relations (SEAR) was developed by Michael Shulman [7]. As a direct solution to the problem occurring in ZF, we also have an option to present ZF structurally, yielding structural ZF [9]. We will see that our system DiaToM models these structural theories particularly well.

Category theory can also serve as a foundation in a certain sense. Two well-known attempts are McLarty's Categories of Categories As a Foundation of Mathematics(CCAF) [33] and Lawvere's Elementary Theory of the Category of Categories(ETCC) [30], with the former having been shown capable of capturing many non-trivial results. And, though ETCC is known to be flawed, people have never lost interest in fixing it, and are continuing to work on similar systems.

All the examples above pertain to the traditional approach to foundations that appear in mathematical publications. Nowadays, to make it easier to present mathematics in computer programs, there comes the option of using a type theory. As pointed out in the HoTT book [49, page17], all the above foundations are clearly separated into two layers: a deductive system and a set of axioms. In contrast, a type theory is its own deductive system and can be axiom-free. Nevertheless, there remains the option of imposing some axioms on top of a type theory to get different desired effects. For example, in simple type theory, by creating a type for the ZF sets and adding relevant axioms, we can obtain the system HOLZF from HOL. With dependent type theory, in its extension HoTT, many interesting features are due to its univalence axiom. In the theorem prover Lean [21], which implements a version of dependent type theory, we can (and often do) add extra axioms stating excluded middle, and one for proof irrelevance, making the system closer to usual mathematical conventions. As a summary, we list the foundations we have mentioned with their deductive system in Table 1.1.

There are some systems implementing what we might describe as typeless logics. The space in the middle, however, has yet to be investigated. Our thesis investigates the space in the middle by establishing a meta-logical framework to serve as the deductive system for the foundations ETCS, SEAR and CCAF. By pursuing this, we investigate the mathematical expressiveness of the logics, as well as their practical utility when realized as a theorem-proving system. We

Foundation	Level of complexity	Logic layer	Implementation
HoTT	high	DTT	Arend
(variants of) CIC	high	DTT	Lean, Coq
HOLZF	medium	HOL	Isabelle/HOLZF
SEAR	medium	sorted FOL	None
ETCS	medium	sorted FOL	None
CCAF	medium	sorted FOL	None
ZF (+C)	low	FOL	Isabelle/ZF
NBG	low	FOL	Mizar

Table 1.1: Comparing Foundations

System	Foundational	Intrinsic dependent types	Generic
HOL	×	×	×
Coq/Lean/Agda	×	✓	×
Mizar	✓	✓	×
Isabelle	✓	×	✓
Metamath	✓	×	✓
DFOL	✓	✓	✓
FOLDS	✓	✓	✓
HOLZF	✓	×	×

Table 1.2: Theorem-Proving Systems

believe this demonstrates the existence of a sweet spot.

1.2 Related Work

In this section, we introduce some popular existing logic systems that are candidates for formalizing mathematics. While they all have their own advantages and work nicely for their design purpose, it is clear that none is perfect. Our discussion covers generic and non-generic theorem-proving systems, as well as theoretical systems. See Table 1.2.

By "foundational" we mean addressing the issue of foundations of mathematics, and by "generic", we mean it is possible to use different frameworks of reasoning to write proofs. We see below that a critical fact is what matters is not only "true or false" in all these dimensions, but also "in which manner".

1.2.1 Non-Generic Systems

A non-generic system implements a particular deductive framework. Users are expected to use the rules that are primitively implemented. They do not provide any generic feature to enable the use of an alternative foundation. For such a theorem prover, its design addresses the foundational issue only if it chooses to implement a mathematical foundation. Only a few theorem provers adopt this choice. The most famous one among them is Mizar. It is a system

with a 50-year history, distinguished by its rich mathematical library, and still has an active community.

For such systems, obviously, they have the deficiency that the user is already tied to their particular choice. But actually, it is not impossible to mechanize proofs using other foundations in them. There still exists the possibility of constructing the foundation as another layer above such a framework and using only this layer to do further proof. Let us, however, by taking a closer look at some possible candidates, explain why it is not a decent approach for this kind of formalization.

HOL

The system HOL, also known as *simple type theory*, is indeed very simple and minimal. It does not have any primitive notion of dependent types: a HOL type cannot involve any HOL term in it. Certainly, HOL is not as expressive as many other mainstream type theories, namely variants of the dependent type theory. Nevertheless, the existing mathematical libraries in HOL already demonstrate that HOL is still capable of many mathematical formalizations. Due to its simplicity, type-checking in HOL is very easy, and HOL is fast to process proofs.

HOL has a long history and is suitable for mechanizing many aspects of mathematics. For example, HOL is used to formalize Kepler's conjecture, as described in Hales et al. [23]. More recently, work by Bordg et al. [18] has shown that despite its simplicity, it is still possible to formalize graduate mathematics such as Grothendieck schemes in HOL.

Isabelle/ZF and Isabelle/HOLZF

Isabelle/ZF is an object logic of Isabelle declared using Isabelle's meta-logic. It is obtained by creating a type of ZF sets and adding the required axioms. Existing work in Isabelle/ZF includes the formalization of forcing by Emmanuel et al. [22], as well as the relative consistency of the Axiom of Choice by Paulson [39].

Isabelle/HOLZF is built on the top of HOL by adding an additional type of ZF sets to the HOL system. Whereas we can only work with sets in Isabelle/ZF, all the HOL types are available in Isabelle/HOLZF. It is indeed more powerful than HOL. As a witness, whereas it is not possible to formalize Partizan Games using the implemented type package in FOL, it is formalized in HOLZF by Obua [36].

Dependently Typed Systems

Dependent type theory and its extended variants are all very powerful. Implemented as theorem provers, a dependent type system can handle very sophisticated statements, as witnessed by theorem provers Lean, and Coq. They are starting to attract a number of mathematicians.

With their involvement in Lean's Xena project, Lean has successfully captured the definition of perfectoid space, as described by work by Buzzard et al. [19], which is considered to be one of the most complicated definitions ever.

However, such power comes at the cost of complexity. The type-checking machinery in dependent type theory is delicate. Reflecting on user experience, it takes longer to execute a proof step. The kernels of these theorem provers are large, and so, more human effort is required to maintain them.

It is certainly possible to formalize mathematics in these dependently typed provers. As indicated in [11], Lean implements a version of Calculus of Inductive Constructions(CIC). In its standard mathlib, there "defines an additional axiom, propositional extensionality, and a quotient construction which in turn implies the principle of function extensionality".

For formalizing traditional mathematical foundations, the fact that the system is more powerful than necessarily required counts as a disadvantage. This makes it harder to keep track of the involvement of ingredients in the system when performing a proof. Proofs in traditional mathematical foundations only rely on the relevant first-order rules, but the primitive rules of the DTT system are more than these. Some of these primitive rules construct some objects that have a counterpart that can also be constructed from axioms. A user should take care not to mix the two notions. For instance, whereas the axioms in ZF already allow a user to construct quotient sets, the ambient systems of Coq and Lean already assert the existence of quotient types as primitive. During proofs, when only intending to use the mathematical foundation encoded, a user also has to carefully choose the proof tools so they do not invoke the rules which is specific to DTT. It is possible to treat such a problem by making some restrictions. We hence need to develop some tools to restrict ourselves from unconsciously appealing to the meta-theory instead of the foundation itself.

Mizar

Mizar addresses the foundation issue by employing a mathematical foundation as its kernel. It implements the Tarski-Grothendieck set theory, a non-conservative extension of ZF. Terms in Mizar are classes in the usual set-theoretic sense. Classes that belong to another class are called a set. Reasoning in Mizar is according to the axioms and the FOL rules. In particular, users prove formulas instead of constructing proof objects. While Mizar has the notion of function and predicate symbols, it also supports statements with free second-order variables (e.g. the induction scheme for natural numbers) [35].

On top of the set theory foundation, Mizar implements a "soft type system". It allows users to have part of the convenience of types or even dependent types without a primitive notion of "type". A type is constructed as a non-empty set by the comprehension axiom schema.

Non-emptiness means quantification rules are simple, but as we will see in Chapter 2, we do want the possibility of empty sorts for some applications. In contrast with DiaToM, a Mizar term can have more than one type. There are no specific "function types" or "higher order types", as they are captured by function and predicate symbols of FOL, respectively. Accordingly, the "lambda abstraction" and "beta reduction" rule, which are primitive notions in a type theory, do not have a counterpart in Mizar. This makes type-checking much easier in Mizar because we never encounter the type-checking of a term involving a λ-abstraction.

1.2.2 Generic Systems

There are not many generic theorem provers[1], two famous ones are Isabelle and Metamath. Isabelle provides a meta-logic, enabling the declaration of logical operations in a rather uniform way without getting to the bottom level. In contrast, everything in Metamath starts from scratch. For instance, in Isabelle, to declare a new framework, a user can specify how would the terms look like, and Isabelle can treat the terms according to what the user has written in meta-logic, whereas Metamath does not even have a primitive notion of terms. Users can write things like $+ : \mathit{term} + \mathit{term} \longrightarrow \mathit{term}$ in Isabelle. It will simultaneously give that $+$ is a valid symbol acceptable by the syntax and what it produces is a term. In Metamath, users have to declare manually that there is going to be something called terms, which can be constructed as the combination of some symbols. Such symbols are initially declared with their names. In later steps, we can regulate the construction by writing expressions that mean, for example, "for the symbol $+$, we have $t + r$ is a term for any possible term t and r".

1.2.1 Example. Consider the MU puzzle, as described in Hofstadter [25]. We formalize its signature and axioms as Isabelle/MU using Isabelle's meta logic and formalize it in Metamath. The symbols M, I and U are letters that combine into strings. We identify these letters with the corresponding singleton string and let "^" denote the concatenation symbol. Strings are converted into atomic formulas by putting "[]" around them. We use "$\longrightarrow$" to denote the implication between formulas. The resulting deductive system is untyped and has only one inference rule, namely the modus ponens.

In Isabelle, there are meta-types o, t and i, whose terms play the roles of formulas, types and terms respectively. We follow the format of examples as in Paulson [38, page187]. The symbols are given in Table 1.3 and the axioms are given in Table 1.4.

The only inference rule, modus ponens, is presented as:

$$[|P \longrightarrow Q; P|] ==> Q$$

[1] Unless Coq is generic.

name	meta type	description
M	i	letter M
I	i	letter I
U	i	letter U
$\hat{}$	$i \to i \to i$	concatenation
$[-]$	$i \to o$	converting a string into an atomic formula
$\to$	$o \to o \to o$	implication

Table 1.3: Signature of MU Puzzle in Isabelle

axiom	description
$[x\hat{}I] \to [x\hat{}I\hat{}U]$	Add a U to the end of any string ending in I
$[M\hat{}x] \to [M\hat{}x\hat{}x]$	Double the string after the M
$[x\hat{}I\hat{}I\hat{}I\hat{}y] \to [x\hat{}U\hat{}y]$	Replace any III with a U
$[x\hat{}U\hat{}U\hat{}y] \to [x\hat{}y]$	Remove any UU

Table 1.4: Axioms of MU Puzzle in Isabelle

On the other hand, in Metamath, we do the following as imitating Megill and Wheeler [34, page42]:

```
$( Declare the constant symbols we will use $)
    $M I U [ ] ^ -> str wff |-$.
$( Declare the metavariables we will use $)
    $x y p q$.
$( Specify properties of the metavariables $)
    $f str x $.
    $f str y $.
    $f wff p $.
    $f wff q $.
$( Define "wff" $)
    $a str M $.
    $a str I $.
    $a str U $.
    $a str ( x ^ y ) $.
    $a wff ( p -> q ) $.
    $a wff [x] $.
$( State the axioms $)
    a1 $a |- [x^I] -> [x^I^U] $.
    a2 $a |- [M^x] -> [M^x^x] $.
    a2 $a |- [x^I^I^I^y] -> [x^U^y] $.
    a4 $a |- [x^U^U^y] -> [x^y] $.
    a5 $a |- [M^I] $.
$( Define the modus ponens inference rule $)
  ${
    min $e |- p $.
```

```
26      maj $e |- ( p -> q ) $.
27      mp  $a |- q $.
28    $}
```

It is then clear that an implementation of a foundation in Metamath requires a lot of work. As this system is not designed to be "programmable", it does not allow the development of a general framework to be instantiated by various kinds of foundations at the user level. The only thing we can do is to write an implementation of one foundation and change it slightly to work for another one.

Isabelle provides more support for high-level specifications. It can treat both formula-style reasoning, as in HOL, and proof as objects, as in Lean. Indeed, Isabelle has both HOL and MLTT as implemented object logic. In particular, the presentation of MLTT in Isabelle looks neat and reasonable. However, whereas Isabelle can handle both judgments and formulas nicely, the picture becomes much less elegant when we require both of them in a single system. Whereas everything, including proofs, is a judgment in MLTT, the sort judgment and the proof statement do not live in the same layer in a dependently sorted foundation. It is hard to have both layers in Isabelle because it has only one notion in its meta-logic that corresponds to "truth". The only obvious Isabelle approach is to capture sort judgments as assumptions, but then we encounter a pain point when there are dependent sorts, i.e., when sorts are allowed to depend on terms. Isabelle's metalogic gives a notion of "type" and "term". However, we cannot create "types" that involve "terms", there is no way to encode a dependent sort judgement as a type judgement. The only thing we can do is to use predicates instead, so a sort judgement for, say, a function from set A to set B will be a predicate applied on a term of a non-dependent sort "function", declaring its domain is A and its codomain is B. This will induce two problems: Firstly, a dependent sorted setting is usually applied for a structural set theory, where we usually do not want equality on sets, but the predicate stating sort judgment will have to involve equality such as $\mathsf{dom}(f) = A$, forces us to use equality. Even worse, as for the composition of functions, the sort-checking, which should ideally happen on the processing of syntax, becomes a check of this predicate on the level of logical deduction, i.e., something that we should prove. If the composition is iterated, we need to appeal to a sort-checking axiom for the composition function symbol to prove in each level such a predicate. For instance, regarding the composition $h \circ g \circ f : A \to B \to C \to D$, we need to get its sort from the axiom:

$$\begin{aligned}&\forall(A\ B\ C : \mathsf{set})\ (f\ g : \mathsf{fun}).\\ &\quad \mathsf{dom}(f) = A \ \wedge\ \mathsf{cod}(f) = B \ \wedge\ \mathsf{dom}(g) = B \ \wedge\ \mathsf{cod}(g) = C \implies\\ &\qquad \mathsf{dom}(g \circ f) = A \ \wedge\ \mathsf{cod}(g \circ f) = C\end{aligned}$$

Then we need to conclude $g \circ f$ first, and then apply it with h, to prove the composition has sort $A \to D$.

Moreover, since Isabelle's metalogic does not allow partial functions, ill-formed terms (i.e., with meta-type t such as the composition of arrows $f \circ g$ such that $\mathsf{cod}(g) \neq \mathsf{dom}(f)$) and predicate application can be built.

There are also a range of *logical frameworks* that allows implementing various of type theories. For instance, the theory DFOL, as we discuss below, is encoded in the logical framework LF [41], which has Twelf as an implementation. Another notable one is Dowek's Dedukti logical framework [14], which has different implementation in Dkcheck, Lambdapi, Kocheck, etc. Since our goal is to give a uniform treatment to a generalization of first-order foundations only, and it is straightforward to implement a logic directly in ML, we do not need to implement it in a logical framework as an intermediate step.

1.2.3 Theoretical Systems

1.2.4 FOLDS

First-Order Logic with Dependent Sorts(FOLDS) by Makkai [31] is an extension of first-order logic designed for "studying the science of establishing foundations" [32, page159], with the outstanding feature that the notions are carefully chosen in FOLDS so that truth automatically agrees on isomorphic models of the theory. On the other hand, as also admitted in [32, page159], FOLDS is not designed to be used for theorem proving. FOLDS has a very interesting feature: it imposes some checks on the syntax level that ensures only "meaningful" expressions can be stated. More specifically, the design of FOLDS intends only to capture the expressions that are invariant under isomorphisms. However, to a larger extent, as there are many foundations with many statements that are not preserved by isomorphisms, this is not generally a desired property.

FOLDS does not support expressing axiom schemata and does not include function symbols at all. Instead, one is restricted to use predicates only, and functions are binary predicates that admit only one witness given the first argument. In practice, function symbols can make a presentation of definitions and proofs much neater. An explicit example is provided in section 5.1. The absence of function symbols also means not being able to get much benefit from rewriting using equalities, which are intensively used in theorem provers.

1.2.5 DFOL

DFOL, by Rabe [44], is designed only for the use of the first-order fragment of DTT. One significant aspect to note is that it separates the layer of "first order statements" out of the Prop type, where users are required to construct a proof term in a DTT prover. Instead, it creates a type for statements and introduces rules to prove such statements. This allows us to separate terms from formulas, which indeed makes the embedding of FOL more faithful.

Signatures in DFOL must be implemented in a dependently-typed general framework, and hence cannot be directly extracted as an implementation on their own. The signature of, for example, standard category theory, where an arrow intrinsically depends on two objects, can be presented naturally in DFOL. As DFOL is not designed to be used on its own, it does not come with a rule allowing us to create new functions and relation symbols. Moreover, it does not provide extra support to express axiom schemata.

1.2.6 Generalized Algebraic Theories

Generalized Algebraic Theories(GAT), as introduced in Cartmell [20], is developed as a universal framework for expressing algebraic theories with dependent types. For instance, it can axiomatize the theory of categories by equalities such as $1_A \circ f = f : \mathrm{Hom}(A, A)$ expressing the left identity law, etc. For using this as a foundation, one obvious barrier is that there is not a common interest to axiomatize a foundation in an algebraic way. Most of foundations are not designed in an algebraic way, and do not obviously have an algebraic reformulation either. In fact, all the foundations we will discuss in this thesis only have models with at least two distinct objects of interest.

1.3 Our Work

We want a system to define mathematical foundations and to prove theorems on top of that foundation. Our work designs such a system. We name our system DiaToM, abbreviating the title of this thesis. It sounds ambitious to make the system capture all these foundations in general. We need to therefore appeal to a significant common feature of these foundations. All of them are proposed in first-order logic, with a signature that may involve dependent sorts. It is not surprising, because first-order logic is *the* natural language for mathematical expressions. We are quite happy to live with this constraint: a richer logic can conceal foundational decisions that we would prefer to make apparent in our axioms. We design a system and then examine our design. On the theoretical side, we formalize our logic in the existing system HOL to make sure it is not broken. For the practical aspect, we implement it as a theorem prover and carry out experiments with three existing mathematical foundations.

Our theorem prover only focuses on faithfully and neatly presenting mathematical foundations and experiments with their real power. Within each foundation, we can have a theorem prover that can be compared to main-stream theorem provers, where the foundation is chosen by the designer. It gives us opportunities to discover very useful first-order systems. For instance, whereas SEAR is discovered to be powerful for use for theorem proving, it is still far from being as complicated as DTT. Therefore, it provides an ideal workspace in which to do rather

complicated proofs with a fast and simple kernel and looks closer to the form of mathematics that might be written in a textbook. We briefly summarize our contribution as follows:

- In Chapter 2, we develop a formal logical system that is able to express various first-order axiomatic systems, where sorts can depend on terms.
- We implement our system as a theorem prover and make it usable by designing a type-inference algorithm and providing interactive tools. Implementation details are in Chapter 3
- We formalize our system in the theorem prover HOL4. We prove basic operations preserve well-formed syntax, and that our complicated formula variable instantiation rule does not break the first-orderness of our system, in Chapter 4.
- In the theorem prover implemented, we specialize this ambient logical system to capture the foundational systems ETCS, SEAR, and CCAF, as in Chapters 5 and 6. We believe we are the first to mechanize mathematics in structural and categorical foundations in any theorem prover.
- On the foundations ETCS and SEAR, which play the same foundational role as traditional material set theory, we demonstrate that our system can handle common mathematical constructions such as the development of the algebraic and co-algebraic lists.
- We provide a proof-of-concept implementation that makes logical developments practical through the development of a number of important, though basic tools. For example, in ETCS, where proofs greatly rely on internal logic, we build a tool to automatically construct the internal logic predicates corresponding to "external" predicates. In both ETCS and SEAR, we automate inductive definitions and provide tools to help with the construction of quotients.
- In SEAR, we provide evidence that it is possible to use DiaToM's simple sorts and appropriate axioms to go beyond the expressiveness of HOL. In particular, we provide an explicit example we published that cannot be even stated in HOL, but can be formalized in SEAR smoothly.
- At the end of the thesis, we propose possible future directions from the current work.
- Finally, we provide an appendix with notes on installing and using our computer implementation of DiaToM.

Chapter 2

Logic

In this chapter, we present a logical framework, called DiaToM, as indicated in the introduction, that strikes a delicate balance between expressiveness and simplicity. Our approach focuses on achieving uniformity in handling partial functions while ensuring the termination of sort-checking. To elucidate our logical system, we adopt a 'three-layered' structure encompassing sorts, terms, and formulas. Subsequently, we delve into our proof theory, which provides a natural mechanism for conducting first-order reasoning in our syntax. We will illustrate key concepts with relevant examples.

2.1 Sorts and Terms

In our system, we establish a mutually recursive relationship between sorts and terms. Each term is associated with a unique sort, which takes the form $\mathsf{st}(s, \vec{t})$, with $\vec{t}$ maintaining a record of terms $(t_1, \cdots, t_n)$ that the sort with name s depends on. To initiate this framework, we must first declare a *signature*, which is central and provides an interface to our work. The first ingredient of a signature is a list consisting of pairs $(s, [s_1, \cdots, s_n])$, where $s, s_1, \cdots, s_n$ are names of sorts. Such a list expresses that the sort s depends on terms of sorts $s_1, \cdots s_n$, where $s_1, \cdots, s_n$ are all already declared. In particular, it must start with an entry of the form $(s, [])$. Secondly, we need a record of primitive function and predicate symbols. Each record of a function symbol keeps a list of acceptable inputs, declared as a list of variables, and a sort of its output depends on terms that are built from all these input variables. A record of predicate only keeps a list of acceptable inputs. A term is either a variable or a function application:

$$t := \mathsf{Var}(n, s) \mid \mathsf{Fun}(f, s, \vec{t})$$

A variable comprises both a name and a sort, while a function term includes the function symbol's name, its sort, and a list of arguments, which are terms. A constant term is a nullary function symbol applied to an empty argument list. The sort of each term is an intrinsic property, carried in its constructor. A sort that doesn't depend on any term is called a *ground*

sort. Likewise, a term with a ground sort is termed a *ground term*. Although according to the constructors, a function term can accept any list of inputs, we only consider well-formed terms in practice. A function term is well-formed if any only if its list of inputs matches the signature, from where we can deduce the sort of the function term. Therefore, for a given signature, it is possible to recover the sort of every term from the sorts of variables that occur in it. However, in practice, as a function term is often complicated, we do not want to rebuild all their sorts from the bottom level, hence we also carry the sort of function symbols as a field of the constructor in our logic, and later, for both our formalization and our implementation.

In the following discussions, the word "variable" may refer to either a variable term or a pair consisting of a name and a sort, depending on the context.

Since terms and sorts are mutually recursive, we collect the free variables in terms and sort using mutual recursive functions Vars and Vars^s respectively, defined as:

$$\mathsf{Vars}(\mathsf{Var}(n, s)) = \{(n, s)\} \cup \mathsf{Vars}^s(s)$$
$$\mathsf{Vars}(\mathsf{Fun}(f, s, [t_1, \cdots, t_n])) = \bigcup_{1 \le k \le n} \mathsf{Vars}(t_k)$$
$$\mathsf{Vars}^s(\mathsf{st}(n, [t_1, \cdots, t_n])) = \bigcup_{1 \le k \le n} \mathsf{Vars}(t_k)$$

We give some examples for capturing the sort signature of some foundations. This reveals the importance of the role of signature-once the signature is understood, our logic can capture a large variety of foundations.

2.1.1 Example. Many material set theories only have one sort. The classic example is ZF, where the only sort is "set". Also note that even if a set theory does have both the notion of class and set within the system, it is not necessarily distinguished by the sorts. For instance, in Morse-Kelley set theory [26], Pocket set theory(also presented in [26]) and Vopenka's alternative set theory [50], the notion of proper class is captured via *definition*. In particular, it is defined to be a class that is not a member of any class. In all of such set theories, there is only one sort with the name "set", the sort signature is $[(\text{set}, [])]$.

2.1.2 Example. Some set theories have the notion of urelements. Such set theories are often two sorted, where the other sort is "set" (See Scott-Potter set theory [42] and Kripke-Platek Set Theory with urelements [15]). The difference between the two sorts is that: a set is a collection, whereas an urelement is not. A urelement is an element, but unlike in ETCS, where an element can only belong to one set, a urelement can serve as an element for many sets. This indicates that a urelement does not store the set in its sort. Hence the sort signature is simply $[(\text{set}, []), (\text{umem}, [])]$, consisting of two ground sorts.

2.1.3 Example. Another way that multiple ground sorts can appear is due to the usage of *stages*, indicating in which stage a set is formed. The canonical example is the set theory

called "S"[17]. Its signature is $[(\text{set}, []), (\text{stage}, [])]$.

2.1.4 Example. The sort-dependency appears naturally in many of structural set theories. For example, set theories in a local language is structural. An explicit example, the presentation of Local Intuitionistic Zermelo (LIZ) set theory is provided by Aczel [13]. To avoid clashing with our logical notions, we rename the "term" and "sort" in [13] into "member" and "set". The sort signature of LIZ is $[(\text{set}, []), (\text{mem}, [\text{set}])]$.

2.1.5 Example. In ETCS, there are two sorts, with names "object" and "arrow". An arrow sort depends on two object terms. The sort signature list of ETCS hence can be recorded as $[(\text{ob}, []), (\text{ar}, [\text{ob}, \text{ob}])]$. We use different object variables A and B to indicate the two terms that an arrow sort depends on are not required to be the same. It has a variant called "ETCS with elements" [3], which can be obtained with slight modification.

2.1.6 Example. It is possible to re-present material set theories in a structural manner without changing their strength. One option for doing this for ZFC, presented as structural ZFC [9], is to work with four sorts: set, member, function, and relation. The sort of sets is the only ground sort, a member sort depends on a set, and a function or a relation depends on two sets, representing its domain and codomain. Then $[(\text{set}, []), (\text{mem}, [\text{set}]), (\text{fun}, [\text{set}, \text{set}]), (\text{rel}, [\text{set}, \text{set}])]$ is the signature sort list. Such a signature is the same as the other system SEAR, which we will discuss in much more detail in later chapters.

2.1.1 Formulas

We are working only with classical logic and can afford to be minimal with our syntax. A formula Φ is either falsity, a predicate, an implication, a universally quantified formula, or a formula variable.

$$\Phi \ ::= \ \bot \ \mid \ \mathsf{Pred}(\mathcal{P}, \vec{t}) \ \mid \ \Phi_1 \implies \Phi_2 \ \mid \ \forall n : s. \Phi \ \mid \ \mathsf{fVar}(\mathcal{F}, \vec{v}, \vec{t})$$

In the above, $\mathcal{P}$ is a predicate name, and $\mathcal{F}$ is the name of a formula variable. Boolean operators $\wedge, \vee, \neg$ can hence be built from the implication and falsity. We define $(\exists x.\ \phi) = \neg(\forall x.\ \neg\phi)$ and the unique existence $\exists!$ is defined as in convention. We write the truth $\top$ as an abbreviation $\bot \implies \bot$.

For predicate symbol applications as atomic formulas, the symbol must either be primitive, as defined within the axiomatic framework or by the user. The only inherent primitive predicate in the system is equality, which is defined between terms of the same sort. However, we do not automatically allow equality between terms solely based on having the same sort. For instance, the designs of ETCS and SEAR do not allow us to equate objects in them. Each foundation must maintain a record (alongside function symbols, predicate symbols, and axioms) of the

sorts that support equality. Consequently, while every variable can be quantified, only those variables whose sort supports equality can be subjected to an $\exists!$-quantification.

Among the constructors mentioned earlier, the most intriguing one is that of formula variables. While term variables can be instantiated with terms, formula variables are meant to be instantiated with formulas. A formula constructed by a formula variable, denoted as $\mathsf{fVar}(\mathcal{F}, [(n_1, s_1), \cdots, (n_k, s_k)], [t_1, \cdots, t_k])$, is essentially a nested universally quantified formula of the form $\forall n_1 : s_1 \, \forall \cdots \forall n_k : s_k . \phi$, specialized by the terms $t_1, \cdots, t_k$. This specialization is only allowed if the terms $t_1, \cdots, t_k$ have the correct sorts and sort dependencies. If the dependencies are correct, then the application of the formula variable on this term list is considered well-formed. The variable list $[(n_1, s_1), \cdots, (n_k, s_k)]$ serves only to record the pattern of acceptable "inputs". It can be thought of as registering the formula variable with the name $\mathcal{F}$ temporarily as a predicate symbol in the signature, associated with this variable list. Therefore, for each variable (n_m, s_m) appear as an entity, none of itself or any variables that appear in s_m for $1 \leq m \leq k$ can be regarded to actually appear in the formula $\mathsf{fVar}(\mathcal{F}, [(n_1, s_1), \cdots, (n_k, s_k)], [t_1, \cdots, t_k])$. However, we do count the variables in $t_1, \cdots, t_k$ as actually occurring. Formula variables are distinguished by their name and the associated variable list. If a formula ϕ has $\mathsf{fVar}(\mathcal{F}, \vec{v}, \vec{t})$ as a subformula, then we say the formula variable $(\mathcal{F}, \vec{v})$ appears in ϕ.

The concept of formula variables is designed to facilitate the representation of axiom schemata with a single axiom within the theory. Here, we provide the simplest example of its usage.

2.1.7 Example. The comprehension schema in ZF theory says for each set A and each formula ϕ, there exists a set consisting of elements of A satisfying ϕ. Such a set is effectively $\{x \in A \mid \phi(x)\}$. For each formula ϕ, the instance of this axiom schema is presented in ZF as:

$$\forall A. \, \exists B. \, \forall x. \, x \in B \Leftrightarrow x \in A \ \wedge \ \phi(x)$$

In our system, once the signature of ZF is declared, the axiom above can be captured using a single formula with a formula variable in the position of ϕ as:

$$\forall A. \, \exists B. \, \forall x. \, x \in B \Leftrightarrow x \in A \ \wedge \ \mathsf{fVar}(\mathcal{F}, [\mathsf{Var}(x_0, \mathsf{set})], [x])$$

Every instance of the comprehension schema will be obtained by instantiating the formula variable in the above, using the instantiation rule we will provide in the next section.

An apparent alternative is to individually state the axiom schema for each of its instances. However, our approach offers significant advantages, particularly in terms of convenience and readability. This convenience becomes evident when we seek to derive theorem schemata from axiom schemata while maintaining a clear, mathematical presentation. In terms of

implementation, implementing each instance of an axiom schema requires us to create a function that takes a formula and produces a theorem. Manipulating these functions would be necessary to capture the proof of a theorem schema, forcing the users to operate at the implementation level. Furthermore, the derived theorems lack a formula representation and cannot be expressed within the system, and hence are not readable without a concrete instance. By using our approach, we address this problem by allowing the straightforward notation of an axiom schema and all its consequences at a practical working level, without altering the theory's strength.

Readers may observe that formula variables behave semantically as higher-order variables. For instance, in the example above, a formula variable behaves like a term of type $\mathsf{set} \to \mathsf{bool}$. This might appear to challenge the first-orderness of our system. To resolve this concern, we will provide in Chapter 4 a formalized proof demonstrating that formula variables do not introduce any additional strength to our logic.

In the following, we will write a predicate formula as $\mathcal{P}(\vec{t})$. For a formula variable with the name $\mathcal{F}$ on arguments of sorts $\vec{v}$ applied on $\vec{t}$, we write $\mathcal{F}[\vec{v}](\vec{t})$.

In the case of universal quantification, the n and s carry the name and sort of the quantified variable. To ensure that the universal quantification of the variable (n, s) in a formula ϕ is well-formed, we require that (n, s) does not appear in the sort of any variable in ϕ. Additionally, if a formula variable $(\mathcal{F}, [(n_1, s_1), \cdots, (n_k, s_k)])$ appears in ϕ, we need (n, s) to not be present in any of s_m for $1 \leq m \leq k$. This requirement is crucial for the forthcoming proof of the elimination of formula variables, which will be detailed also in Chapter 4.

We collect free variables in a formula by a function $\mathsf{Vars}^{\mathsf{f}}$ by a simple induction:

$$
\begin{aligned}
&\mathsf{Vars}^{\mathsf{f}}(\bot) = \{\} \\
&\mathsf{Vars}^{\mathsf{f}}(\mathsf{Pred}(\mathcal{P}, [t_1, \cdots, t_n])) = \textstyle\bigcup_{1 \leq k \leq n} \mathsf{Vars}(t_k) \\
&\mathsf{Vars}^{\mathsf{f}}(\phi_1 \implies \phi_2) = \mathsf{Vars}^{\mathsf{f}}(\phi_1) \cup \mathsf{Vars}^{\mathsf{f}}(\phi_2) \\
&\mathsf{Vars}^{\mathsf{f}}(\forall n : s.\ \phi) = \mathsf{Vars}^{\mathsf{f}}(\phi) \cup \mathsf{Vars}^{\mathsf{s}}(s) \setminus \{(n, s)\} \\
&\mathsf{Vars}^{\mathsf{f}}(\mathsf{fVar}(\mathcal{F}, \vec{v}, [t_1, \cdots, t_n])) = \textstyle\bigcup_{1 \leq k \leq n} \mathsf{Vars}(t_k)
\end{aligned}
$$

2.2 Theorems

A theorem consists of a set of variables Γ, called the context, a finite set A of formulas (the assumptions), and a formula ϕ as the conclusion. A theorem $\Gamma, A \vdash \phi$ reads "for all assignments σ of variables in Γ to terms respecting their sorts, if all the formulas in $\sigma(A)$ hold, then we can conclude $\sigma(\phi)$".

The context encompasses the set of variables required for the conclusion to be true, given the assumptions. This set includes at least all the free variables present in the assumptions or the conclusion. However, it may also contain variables that do not appear at all in these statements. These extra variables can be introduced through various means, such as removing universal quantifiers, instantiating formula variables, or applying previously proven theorems. Presenting a variable in the context can be seen as a specialized form of an assumption, asserting the existence of that variable. The context plays a crucial role in our system and cannot be disregarded.

2.2.1 Example. In ETCS, the object 1 plays the role of the singleton and the object 0 plays the role of the empty set. We have $\{f : 1 \to 0\} \vdash \exists f : 1 \to 0.\ \top$ as a theorem, but $\vdash \exists f : 1 \to 0.\ \top$ is easily proved to be false.

As illustrated by this example, the context is essential because not every sort necessarily has a term. The context serves as a mechanism to prevent us from using variables that do not exist.

As already indicated by this example, we need the context because it is not always the case that every sort has a term. The context can successfully prevent us from using variables that do not exist. Before working with a variable with a certain sort, we need to either prove or assume its existence. As a further minimal example: after proving $\{\} \vdash \forall f : 1 \to 0.\ \bot$, once we can construct a variable $f : 1 \to 0$, we can specialize the quantification to get $\bot$. Such a theorem is useful for proofs by contradiction. Without the restriction that available variables must come from the context, we could derive $\bot$ by specializing with a variable that does not exist, thereby breaking consistency.

2.3 Proof System

Theorems are obtained by performing proof steps using our primitive rules, as we are introducing now. The inference rules that are standard are presented in Figure 2.1, a well-formed map refers to a function sending a variable to a well-formed term.

The rules for the universal quantifier take some extra care of the sort information. Specializing the quantification on a variable n of sort s by a term t requires checking t has sort s. Once this is satisfied, we need to add all the free variables of t into the context. The formal rule is:

$$\forall\text{-E, } t \text{ is of sort } s\ \frac{\Gamma, A \vdash \forall x : s.\phi(x)}{\Gamma \cup \mathsf{Vars}(t), A \vdash \phi(t)}$$

To apply generalisation ($\forall$-I) with a variable $a : s$, the first requirement is to make sure the conclusion after abstraction is well-formed, as discussed in 2.1.1. In addition, we require that (i) a does not occur in the assumption set; (ii) there is no term in the context depending on a.

$$\text{Assume}\ \frac{}{\mathsf{Vars}(\phi), \{\phi\} \vdash \phi} \qquad \text{Ax}\ \frac{}{\mathsf{Vars}(\phi) \vdash \phi}\ \phi \text{ is an axiom}$$

$$\text{CContr}\ \frac{\Gamma, A \cup \{\neg\phi\} \vdash \bot}{\Gamma, A \vdash \phi} \qquad \text{ExF}\ \frac{}{\mathsf{Vars}(A \cup \{\phi\}), A \cup \{\bot\} \vdash \phi}$$

$$\text{Disch}\ \frac{\Gamma, A \vdash \phi}{\Gamma \cup \mathsf{Vars}(\psi), A \setminus \{\psi\} \vdash \psi \implies \phi} \qquad \text{MP}\ \frac{\Gamma_1, A_1 \vdash \phi \implies \psi \qquad \Gamma_2, A_2 \vdash \phi}{\Gamma_1 \cup \Gamma_2, A_1 \cup A_2 \vdash \psi}$$

$$\text{Sym}\ \frac{\Gamma, A \vdash a = b}{\Gamma, A \vdash b = a} \qquad \text{Trans}\ \frac{\Gamma_1, A_1 \vdash a = b \qquad \Gamma_2, A_2 \vdash b = c}{\Gamma_1 \cup \Gamma_2, A_1 \cup A_2 \vdash a = c}$$

$$\text{Refl}\ \frac{}{\mathsf{Vars}(a) \vdash a = a}\ \text{the sort of } a \text{ has equality}$$

$$\text{InstTM}\ \frac{\Gamma, A \vdash \phi}{\sigma(\Gamma), \sigma(A) \vdash \sigma(\phi)}\ \sigma \text{ is a well-formed map}$$

$$\text{FVCong}\ \frac{\Gamma_1, A_1 \vdash t_1 = t_1', \cdots, \Gamma_n, A_n \vdash t_n = t_n'}{\bigcup_{i=1}^{n} \Gamma_i, \bigcup_{i=1}^{n} A_i \vdash \mathcal{F}[\vec{s}](\vec{t}) \Leftrightarrow \mathcal{F}[\vec{s}](\vec{t'})}$$

Figure 2.1: Natural Deduction style presentation of our sorted FOL

Once all these conditions are met, we can proceed with the application of:

$$\forall\text{-I}\ \frac{\Gamma, A \vdash \phi(x)}{\Gamma \setminus \{x : s\}, A \vdash \forall x : s.\ \phi(x)}$$

2.3.1 Formula Variable Instantiation

The instantiation rule for formula variables is given as:

$$\text{Form-Inst}\ \frac{\Gamma, A(\mathcal{F}[\vec{v}]) \vdash \phi(\mathcal{F}[\vec{v}])}{\Gamma \cup \mathsf{Vars}(\psi), A[\mathcal{F}[\vec{v}] \mapsto \psi] \vdash \phi[\mathcal{F}[\vec{v}] \mapsto \psi]}$$

where we use $\mathcal{F}[\vec{v}] \mapsto \psi$ to denote the instantiation of the formula variable $\mathcal{F}[\vec{v}]$ with the formula ψ. It replaces each occurrence of $\mathcal{F}[\vec{v}](\vec{t})$ on an argument list $\vec{t}$ in the assumption or conclusion into $\psi[\vec{t}/\vec{v}]$. The formula ψ may or may not contain more formula variables, to be considered as a body of the quantification on the variable list $\vec{v}$, such that $\forall v_1, \cdots, v_n.\ \phi$ adheres to the requirements of being a well-formed formula. A formula instantiation cannot make a change to the variable list of formula variables. When the situation demands, we rely on the term instantiation rule to modify the sort list as needed before the formula variable instantiation.

A notable distinction between term instantiation and formula variable instantiation lies in their impact on contexts. While term instantiation replaces variables within contexts, formula variable instantiation primarily extends the context by introducing additional variables, without causing the removal of existing ones.

2.3.1 Example. Consider the theorem

$$\{A,\ B\} \vdash \mathcal{F}[(a_0, \mathsf{mem}(A)); (b_0, \mathsf{mem}(B))](a, b) \Leftrightarrow \mathcal{F}[(a_0, \mathsf{mem}(A)); (b_0, \mathsf{mem}(B))](a, b)$$

and the instantiation $[A \mapsto C;\ B \mapsto D]$. After the instantiation, we have

$$\{C,\ D\} \vdash \mathcal{F}[(a_0, \mathsf{mem}(C)); (b_0, \mathsf{mem}(D))](a, b) \Leftrightarrow \mathcal{F}[(a_0, \mathsf{mem}(C)); (b_0, \mathsf{mem}(D))](a, b)$$

. That is, both A and B are replaced.

But even if we instantiate $(\mathcal{F}, [(a_0, \mathsf{mem}(A)); (b_0, \mathsf{mem}(B))])$ with some formula without any variable at all, say $\top$, we still obtain:

$$\{A, B\} \vdash T \Leftrightarrow T$$

In other words, all the variables stay. This is semantically reasonable because this formula variable is regarded as a predicate on elements of A and of B, so the variables are still assumed to exist.

On the other hand, term variable instantiation is designed to capture a possibly iterated generalization succeeded by a specialization. Generalization can indeed lead to the removal of variables from the context. Notably, once formula variables have been eliminated, the need for the term instantiation rule as a primitive operation diminishes.

It should be noted that we have the liberty to insert $Q[f_0 : A \to B] \mapsto P[A, B, f_0 : A \to B]$, but the reverse is not permissible. This is evident from the well-formedness of the instantiation map. Such a map is recorded as $\forall f : A \to B.\ P[A, B, f_0 : A \to B](f)$, whereas the other way will be recorded as $\forall A\ B\ f : A \to B.\ Q[f_0 : A \to B](f)$, which abstracts the variables in the sort list.

2.3.2 Congruence

The only primitive congruence rule is exclusively provided for formula variables, as it inherently encompasses rules for both function and predicate symbols. The rule for predicate symbol is canonically represented as:

$$\text{Pred-Cong}\ \frac{\Gamma_1, A_1 \vdash t_1 = t_1',\ \cdots,\ \Gamma_n, A_n \vdash t_n = t_n'}{\bigcup_{i=1}^n \Gamma_i, \bigcup_{i=1}^n A_i \vdash P(t_1, \cdots t_n) \Leftrightarrow P(t_1', \cdots t_n')}$$

With dependent sorts, the above congruence may seem not to be sufficient. A valid concern arises when dealing with predicates and function symbols in situations where only one argument features equality while the other arguments lack equality but are identical. For instance,

consider predicates in the form of $P(A, f : A \to B)$ and $f = g$. We certainly want to conclude $P(A, f : A \to B) \Leftrightarrow P(A, g : A \to B)$, but the rule above does not directly allow it. Consider a formula variable $Q[f_0 : A \to B]$, the rule above readily gives $Q[f_0 : A \to B](f) \Leftrightarrow Q[f_0 : A \to B](g)$. Subsequently, we instantiate the formula variable $Q[f_0 : A \to B]$ by mapping it to $P(A, f_0 : A \to B)$, which, as discussed above, is permissible. This results in the desired equivalence. Such scenarios naturally arise, as can be observed in Chapter 6. This feature simplifies our implementation, as it eliminates the need for an option type when implementing such a rule.

2.3.3 Specification

Predicate Specification Rule We can extend the signature of a foundation with new predicate symbols using the predicate specification rule.

$$\text{Pred-spec}\ \frac{}{\mathsf{Vars}(\vec{t}) \vdash \mathcal{P}(\vec{t}) \Leftrightarrow \phi(\vec{t})}\ \mathcal{P} \text{ does not occur in } \phi$$

Applying such a rule will define a new predicate with the name $\mathcal{P}$. The defined predicate will be polymorphic, where each tuple whose sort is matchable with the list $\vec{t}$ can be taken as the arguments. Here the argument of the new predicate symbol is not required to be all of $\mathsf{Vars}(\phi)$, we only require the whole set of free variables involved can be recovered from the arguments. For instance, if $\{a_1 : s_1, a_2 : s_2(a_1)\}$ exhausts the free variables involved, then the predicate can just take the single argument a_2 instead of both a_1 and a_2.

Function Specification Rule The specification rule for introducing new function symbols is notably intricate. Given a theorem $\Gamma, A \vdash \exists a_1 : s_1, \cdots a_n : s_n.Q(a_1, \cdots, a_n)$ where Q is any formula, if the uniqueness of the tuple $(a_1, \cdots, a_n)$ holds in a sense acceptable to the foundation, we proceed to define function symbols $f_1, \cdots, f_n$ in such a way that their output tuple satisfies the property Q. The process of defining a new function symbol entails presenting a theorem affirming the unique existence of certain terms up to a specified relation, another theorem asserting that the relation is indeed an equivalence relation, and a further theorem ensuring the non-emptiness of the relevant sorts, where we intend to generate function terms.

In a general sense, an equivalence relation must be captured by a predicate that operates on two lists of variables, representing the two entities under consideration. These lists are referred to as "meta-tuples". Given that the built-in logic lacks a concept of tuples, we cannot directly define an equivalence relation on real "tuples". Instead, we require theorems in the form as

follows:

$$\vdash R(\langle a_1 : s_1, ..., a_n : s_n\rangle, \langle a_1 : s_1, ..., a_n : s_n\rangle)$$

$$\vdash R(\langle a_1 : s_1, ..., a_n : s_n\rangle, \langle a'_1 : s'_1, ..., a'_n : s'_n\rangle) \implies R(\langle a'_1 : s'_1, ..., a'_n : s'_n\rangle, \langle a_1 : s_1, ..., a_n : s_n\rangle)$$

$$\vdash R(\langle a^1_1 : s^1_1, ..., a^1_n : s^1_n\rangle, \langle a^2_1 : s^2_1, ..., a^2_n : s^2_n\rangle) \wedge R(\langle a^2_1 : s^2_1, ..., a^2_n : s^2_n\rangle, \langle a^3_1 : s^3_1, ..., a^3_n : s^3_n\rangle) \implies R(\langle a^1_1 : s^1_1, ..., a^1_n : s^1_n\rangle, \langle a^3_1 : s^3_1, ..., a^3_n : s^3_n\rangle)$$

If the three theorems all hold for a concrete property R, then R is an equivalence relation (abbreviated as $\mathsf{er}(R)$ in the rest of the discussion). If R is used as the equivalence relation above, the corresponding existential theorem is required to be of the form:

$$\exists a_i : s_i.\ Q(\langle a_1 : s_1, ..., a_n : s_n\rangle) \wedge \forall a'_i : s'_i.\ Q(\langle a'_1 : s'_1, ..., a'_n : s'_n\rangle) \implies R(\langle a_1 : s_1, ..., a_n : s_n\rangle, \langle a'_1 : s'_1, ..., a'_n : s'_n\rangle)$$

We abbreviate the formula above as $\exists!_R a_i : s_i.\ Q(\langle a_1 : s_1, ..., a_n : s_n\rangle)$. The sorts of the two argument lists are not required to be equal, and they are generally not equal because the sorts of the latter variables often depend on the previous ones. The rule is expressed as:

$$\frac{\Gamma_0, \emptyset \vdash \exists a_i : s_i.\top \qquad \Gamma', A' \vdash \mathsf{er}(R) \qquad \Gamma, A \vdash \exists!_R a_i : s_i.Q(\langle a_1 : s_1, ..., a_n : s_n\rangle)}{\Gamma, A \vdash Q(\langle f_1(\Gamma'), ..., f_n(\Gamma')\rangle)}$$

where

- Q and R do not contain any formula variables; and
- $\Gamma_0 \subseteq \Gamma$, $\Gamma' \subseteq \Gamma$, and $A' \subseteq A$.

Our rule's leftmost premise requires the existence of terms of the required (output) sorts, given the existence of variables in the context corresponding to the sorts of the arguments. In this way, the rule guarantees that terms built using the new function symbol will always denote values in the output sort. For the equivalence relation, we can take R to be equality, meaning we are specifying new function symbols according to unique existence. If we take R to be the $\top$, we have imported the Axiom of Choice into our system. The choice of which R's to allow is up to the designer of the object logic.

Treatment of Equalities Our logic allows the users to enable equalities on terms of every sort. However, it is not always helpful to allow such equalities. This is because, in contrast to DTT, where equalities between terms always result in equalities between types that depend on equal terms, our logic does not have a notion of equality of sorts at all. This becomes

evident when we consider our formalization of dual categories in CCAF, as we will discuss in Chapter 6.

2.3.4 Semantics *via* Translation to Sorted FOL

As formula variables and their proof rules represent a conservative extension and can be eliminated, the term-instantiation rule (InstTM in Figure 2.1) can be derived from ∀-I and ∀-E and can also be removed from the list of primitive rules. This is because once there are no formula variables, for each theorem, we can order its context into a correct dependency and generalize all the variables. Then we specialize all of them with the desired terms to capture the instantiation.

2.3.2 Example. Considering structural ZF and a theorem in it:

$$\{A,\ B,\ f : A \to B,\ g : A \to B\} \vdash (\forall a \in A.\ f \circ a = g \circ a) \implies f = g$$

where f and g are functions from A to B. To recover the instantiation $[A \mapsto X,\ B \mapsto Y,\ f \mapsto h : X \to Y,\ g \mapsto k : X \to Y]$, we generalize the theorem so it becomes

$$\vdash \forall A\ (f : A \to B)\ (g : A \to B).\ (\forall a \in A.\ f \circ a = g \circ a) \implies f = g$$

then we instantiate with $[X, Y, h, g]$, so it becomes:

$$\{X,\ Y,\ h : X \to Y,\ k : X \to Y\} \vdash (\forall a \in X.\ h \circ a = k \circ a) \implies h = k$$

as desired.

Subsequently, our semantics below ignore formula variables (meaning that our formulas come in just four forms: $\bot$, implications, the universal quantifier and predicate symbols). Our logic can be translated into non-dependent sorted FOL, which is equivalent to FOL. Given a list of sorts $s_1, \cdots, s_n$, such that s_k only depends on terms with sorts occurring earlier in the list for each $1 \le k \le n$, we create non-dependent sorts $\overline{s}_1, \cdots, \overline{s}_n$. These sorts are thought of as the non-dependent versions of $s_1, \cdots s_n$. We can think of the set of terms of sort $\overline{s_i}$ as the union of all terms of sort $s_i(\vec{t})$ for all possible tuples $\vec{t}$ of terms.

$$\{a \mid a : \overline{s_i}\} = \bigcup_{\vec{t_k}} \{a \mid a : s_i(\vec{t_k})\}$$

For example, the ETCS terms $f : A \to B$ and $g : C \to D$ are of different arrow sorts, but their translation both have sort $\overline{ar}$. For a function symbol f taking a list of terms $[t_1 : s_1, \cdots, t_n : s_n]$, we create a non-dependent sorted function symbol $\overline{f}$, such that its argument term list has the corresponding sort list $\overline{s_1}, \cdots, \overline{s_n}$. We do the same for predicate

symbols. Translation from terms of s_k into those of FOL sort $\overline{s}_k$ is done by forgetting sort dependency:

$$\begin{aligned}[\![\mathsf{Var}(x, s_k(t_1, \cdots t_m))]\!]_{\mathsf{t}} &= \mathsf{Var}(x, \overline{s}_k) \\ [\![\mathsf{Fun}(f, s_k(t_1, \cdots, t_m), (a_1, \cdots, a_n))]\!]_{\mathsf{t}} &= \mathsf{Fun}(f, \overline{s}_k, ([\![a_1]\!]_{\mathsf{t}}, \cdots, [\![a_n]\!]))\end{aligned}$$

For sorts s depending on terms $t_1 : s_1, \cdots, t_m : s_m$, we create function constants $d_{s,1}, \cdots, d_{s,m}$. For $1 \leq i \leq m$, $d_{s,i}$ takes an argument of sort $\overline{s}$ and outputs a term of sort $\overline{s_i}$. If a function symbol f takes arguments $(t_1 : s_1, \cdots, t_n : s_n)$, and outputs a non-ground sort s, where s depends on terms $r_1, \cdots, r_n$, and each s_k depends on terms $q_{k,i}$, then we add an axiom to regulate the dependency information of its sort when translated into non-dependent-sort FOL:

$$\bigwedge_k \bigwedge_i d_{s_k,i}([\![v_k]\!]_{\mathsf{t}}) = [\![q_{k,i}]\!]_{\mathsf{t}} \implies \bigwedge_j d_{s,j}([\![f(v_1, \cdots, v_n)]\!]_{\mathsf{t}}) = [\![r_j]\!]_{\mathsf{t}}$$

As an example, the composition function symbol in ETCS takes $g : B \to C$ and $f : A \to B$, and outputs $g \circ f : A \to C$. The corresponding axiom is:

$$\begin{aligned}&\forall (A : \overline{\mathsf{ob}})\ (B : \overline{\mathsf{ob}})\ (C : \overline{\mathsf{ob}})\ (f : \overline{\mathsf{ar}})\ (g : \overline{\mathsf{ar}}). \\ &\quad d_{\mathsf{ar},1}(f) = A \wedge d_{\mathsf{ar},2}(f) = B \wedge d_{\mathsf{ar},1}(g) = B \wedge d_{\mathsf{ar},2}(g) = C \implies \\ &\qquad d_{\mathsf{ar},1}(g \circ f) = A \wedge d_{\mathsf{ar},2}(g \circ f) = C\end{aligned}$$

For an arbitrary function symbol f, although its arguments can include ground terms, the axiom only needs to state information about the dependently sorted argument, where the functions d_k, as shown above, exist. If the output of a function symbol is a ground sort, we do not need such an axiom for it.

Translation of formulas only makes sense under the translation of some context that contains at least all of its free variables. Defining the translation of a context amounts to translating sort judgments of variables. We translate the sort judgment of any ground sort into $\top$. As for a variable $a : s_k(t_1, \cdots, t_n)$, we write

$$[\![a : s_k(t_1, \cdots, t_n)]\!]_{\mathsf{ts}} = \bigwedge_i d_{k,i}([\![a]\!]_{\mathsf{t}}) = [\![t_n]\!]_{\mathsf{t}}$$

to denote the translation of a context element ($[\![\cdots]\!]_{\mathsf{ts}}$ calculates the denotation of a term's sorting assertion). An entire context Γ is translated into the conjunction of the translation of its elements.

As we do for function symbols, we create for each dependent sorted predicate symbol $\mathcal{P}$ a corresponding non-dependent sorted one, written as $\overline{\mathcal{P}}$. We define the translation of formulas

by induction as:

$$\begin{aligned}
[\![\mathcal{P}(t_1 : s_1, \cdots, t_n : s_n)]\!]_{\mathsf{f}} &= \overline{\mathcal{P}}([\![t_1]\!]_{\mathsf{t}}, \cdots, [\![t_n]\!]_{\mathsf{t}}) \\
[\![\phi \implies \psi]\!]_{\mathsf{f}} &= [\![\phi]\!]_{\mathsf{f}} \implies [\![\psi]\!]_{\mathsf{f}} \\
[\![\forall x : s.\ \psi]\!]_{\mathsf{f}} &= \forall x : \overline{s}.\ [\![x]\!]_{\mathsf{ts}} \implies [\![\psi]\!]_{\mathsf{f}}
\end{aligned}$$

Finally, a theorem $\Gamma, A \vdash \psi$ translates into

$$\forall v_1 \ldots v_n. \bigwedge_{(v_i:s_i)\in\Gamma} [\![v_i : s_i]\!]_{\mathsf{ts}} \wedge [\![A]\!]_{\mathsf{f}} \implies [\![\psi]\!]_{\mathsf{f}}$$

It is routine to check that the rules are valid under the translation and hence have the intended sense. As an example, consider ∀-I. Assume $\Gamma, \{a : s(t_1, \cdots, t_n)\}, A \vdash \phi(a)$ and the variable a appears in neither Γ nor A. The theorem translates into

$$[\![\Gamma]\!]_{\mathsf{ts}}, [\![a : s(t_1, \cdots, t_n)]\!]_{\mathsf{ts}}, \bigwedge [\![A]\!]_{\mathsf{f}} \vdash [\![\phi(a)]\!]_{\mathsf{f}}$$

(where we overload $[\![\cdots]\!]_{\mathsf{ts}}$ and $[\![\cdots]\!]_{\mathsf{f}}$ to include the versions mapping sets to conjunctions of translations). The fact that a does not appear in Γ translates into the corresponding variable $a : \overline{s}$ not appearing in $[\![A]\!]_{\mathsf{f}}$, and the requirement that no variable depends on a translates to the requirement that $[\![a : s(\ldots)]\!]_{\mathsf{ts}}$ does not appear in $[\![\Gamma]\!]_{\mathsf{ts}}$ either. Therefore, we can discharge $[\![a]\!]_{\mathsf{ts}}$ from the assumption and deduce from the FOL universal elimination rule that $[\![\Gamma]\!]_{\mathsf{ts}}, [\![A]\!]_{\mathsf{f}} \vdash \forall a : [\![s]\!]_{\mathsf{s}}.\ [\![a : s]\!]_{\mathsf{ts}} \implies [\![\phi(a)]\!]_{\mathsf{f}}$. This is the translation of $\Gamma, A \vdash \forall a : s(t_1, \cdots, t_n).\phi(a)$, as required.

2.4 Limitation

Our logic was crafted to serve as a minimal framework capable of capturing mathematical foundations in a generic manner. While it offers versatility and a natural approach, it does have its inherent limitations.

Cannot Capture the Meta-Theory When compared to a Dependent Type Theory (DTT) theorem prover that allows for embedding the foundational logic and using the same prover for theorems both within the foundation and the ones about the meta-theory, our theorem prover has a limitation: it can only prove theorems within the specific foundation it is specialized into. This limitation can become apparent when we need to prove a theorem that involves elements of a broader meta-theory but ultimately results in a theorem within the specific foundation.

For instance, there are some general proofs in Shulman's paper [47], for instance, the ones on Section 7 about the Axiom of Replacement, that require concepts from category theory

in the larger mathematical context. It is possible to formalize each individual case of these theorems. But a DTT theorem prover allows the proof of the most general case using the broader meta-theory, offering a more efficient approach to handling such situations.

No Variants on Versions of First-Order Logic We emphasize that our system exhibits generality primarily within the realm of dependent sorted first-order logic. In contrast to more versatile systems like Isabelle and Metamath, where users can declare logical operators and introduce new proof rules, our system is tailored for only classical first-order logic. Nevertheless, within an appropriate foundational framework, it's possible to formalize other logics on top of our system, as illustrated in the example in Section 5.2.8. Additionally, there's the potential to extend our design to accommodate variants of first-order logic, but these extensions would typically require individual implementations, lacking a generalized mechanism for users to declare logical rules without delving into the implementation details.

Sorts are Required to be Well-Founded We also note that a term-sort-tree must be well-founded. We must start with a ground sort without any dependency, so the following construction cannot be formalized in our system.

2.4.1 Example. One presentation of a theory within a category is to have only one sort, that is an arrow. In such a setting, the identity arrow is identified with the object. It is not possible to make "arrows depend on two identity arrows" intrinsic in our setting.

Absence of Sort Variables Whereas mainstream type theories have type variables, our system does not incorporate sort variables, which may look surprising to type theorists. As a consequence, we do not have the capability to assert or prove theorems that hold universally across all sorts. This limitation is a deliberate aspect of our approach and underscores the utility of sorts. While type variables in other systems enable the expression of statements valid for all types, they necessitate consideration of all cases across existing types. In contrast, through the use of sorts, we restrict our focus to theorems applicable to a specific category of entities.

The advantage of employing sorts is to maintain separation between different categories and to ensure that certain features only apply to entities within a particular collection. This separation serves to impose type-checking for predicate and function application and therefore prevents situations where it might be possible to write down a statement such as $1 \in 2$. Sorts, in essence, help us keep distinct categories distinct and prevent issues that might arise from treating everything together as a single entity.

No Notion of Restricted Formula Variable Instantiation One of the limitations that significantly affects the flexibility of our current system is the absence of syntax condition

checks before a formula instantiation. Such checks are indeed crucial for certain foundations. While it is feasible to introduce an additional parameter to our formula variable instantiation rule to accommodate a predicate related to the instantiation map, it's important to recognize that these checks are often not doable within the foundation to be declared. In practical terms, implementing a predicate for these checks typically necessitates involvement at the implementation level. This limitation becomes particularly relevant in cases involving variations of the separation schema [1], where new "collections" are constructed from existing ones.

2.4.2 Example. When formulating an unsorted set theory in our setting, we require *restricted separation* to avoid the Russell paradox. As in 2.1.7, we need the formula to be satisfied to be conjuncted with the pre-condition that x is in a fixed set A. The same check can be done in an alternative way: we omit the conjunct $x \in A$, and check the formula includes that x is in a fixed set A before instantiation. Then the axiom can look like $\forall A.\ \exists B.\ \forall x \in B \Leftrightarrow \mathcal{F}([\mathsf{Var}(x_0, \mathsf{set})], [x])$ instead.

2.4.3 Example. Whereas the condition only on the x can be ensured by adding a conjunction, we cannot use the same approach to check *boundedness*. A bounded formula is a formula such that every quantification happens only on elements of the same set. Therefore, the syntax-checking predicate we define will hold if immediately after a quantification, we see a conjunct $x \in A$ for some fixed set A.

2.4.4 Example. As a more complicated case used in Quine's new foundation, we need a formula to be *stratified* before constructing a new set out of it. This suggests the condition-checking predicate should allow take some extra input, say, a function. A formula can be stratified if there exists a function f from each variable of it, bounded or not, to a natural number, such that whenever $x \in y$ occurs in the formula, we have $f(x) + 1 = f(y)$. It is impossible to do this check within our logic since we do not even have a notion of natural numbers. However, for implementing our logic in a language such as SML, where we do have a type of natural numbers, it would not be any problem implementing it.

No Dependent Function It is no obvious way to capture dependent functions as in dependent type theory in our system. In DTT, a dependent function $f : \Pi_{a:A} B(a)$ consists of a *family of types* indexed by a type A, captured as a function $B : A \to \mathsf{Type}$ assign each term $a : A$ a type $B(a)$. The application of the function on a outputs a term of type $B(a)$. In this sense, the type of the output varies.

The critical feature of a dependent function is that given an input, it simultaneously figure out a type where the output lives in, and a term as the output. A direct translation into our system will require a function to the collection of all sorts. As we do not have a term that serves as collection of all sorts, such a translation is not possible.

As an approximation using a foundation, say, the structural ZF, we can create a two function symbols that outputs a set and a member of it at the same time. In more symbolic notation, given $a \in A$, it is possible to create function symbols f_1 and f_2 such that $f_1(a) \in f_2(A)$. This can effectively recover some cases of dependent functions. However, note that such approximation has to be captured by two function symbols, whereas the counterpart is captured as a term in DTT.

Chapter 3

Implementation

We discuss our approach to implementing the logical system described in the last chapter using the SML language, including examples. The implementation basically follows the LCF-style [6]. In particular, this means that we implement our theorems as an abstract data type, with the primitive rules of inference being the only way to construct theorem values. We discuss how sorts, terms, formulas and theorems are represented in SML, and how the primitive rules of inference are implemented.

On top of kernels such as this, any useful system must implement a great deal of extra machinery. Many proof tools such as tactics, tacticals and derived rules in the theorem prover HOL4 can be re-implemented in our system with only mild modification. We are also able to borrow HOL's library of proof stacks, history and parts of its pretty-printer. The simplifier is implemented following Paulson's higher-order rewriting [37]. We also apply HOL's counterpart of the term net for an effective search for the theorems with matching patterns.

On the other hand, as our desired syntax is different, we must implement a new parser. Among other features, for usability, it is critical to implement type-inference (in our case, "sort inference"). We describe our algorithm in Section 3.2.

3.1 Primitive rules

We implement our datatype of terms and formulas with the following constructors:

Bound variables (i.e., terms built with the Bound constructor) are exclusively created through abstraction and are never encountered independently; they are always associated with a quantifier. The name of the quantified variable is retained and can serve as the default name when we strip the quantification, as happens during pretty-printing, for example. A theorem is a 3-tuple that faithfully captures our design.

```
1 datatype thm = thm of ((string * sort) set * form list * form)
```

```
datatype sort = Srt of string * term list
     and term = Var of string * sort
              | Bound of int
              | Fun of string * sort * term list;

datatype form = Pred of string * term list
              | fVar of string * (string * sort) list * term list
              | Conn of string * form list
              | Quant of string * string * sort * form
```

Figure 3.1: SML declarations for terms, sorts and formulas

In the interest of efficiency, compared with our design from the last chapter, we also make the connectives $\wedge$ (conjunction), $\vee$ (disjunction), $\neg$ (negation) and the existential quantifier primitive formula constructors. The implemented inference rules for each of them are taken to be primitive rules, avoiding the need for performing expansions through derived rules every time these operations are used. The implementation is generally straightforward.

Even for the complicated function specification rule, the implementation follows a systematic procedure. We take the relation provided by the user, construct the corresponding formula that serves as the condition for it to be an equivalence, and subsequently validate that the user-provided theorem indeed has this formula as its conclusion. In the following, we only delve into the two most interesting cases: the ones for term and formula variable instantiation.

3.1.1 Term variable instantiation

The instantiation map, which records the mapping between the variables to be instantiated and the actual terms, is implemented as a dictionary using a $(string\#sort, term)$-Binarymap.dictionary. The instantiation of a theorem involves the instantiation of its context, formulas in assumption sets, and the conclusion formula. This instantiation process is accomplished using a function called finst, which takes an instantiation map and a formula as its arguments. This function is organized into three stages, with the first two stages, tinst and sinst, handling terms and sorts, respectively. Since the implementation follows the de Bruijn approach for handling bound variables, instantiating a variable does not result in any renaming. Bound variables remain unchanged by any instantiation. A variable term $\mathsf{Var}(n, s)$, is instantiated into the term t if the variable (n, s) is found in the dictionary and is mapped to t; otherwise, it remains unchanged. Instantiations on function terms and sorts are constructed inductively. The instantiation function is total, meaning it handles all possible cases, and it never raises exceptions or enters into infinite loops during its execution.

The application of the instantiation rule in our system involves a preliminary check to ensure that the instantiation map is well-formed, as specified in Section 2.3. To perform this check, we define a function that assesses the well-formedness of the map. Given a map σ, this

```
fun inst_thm env th =
    if is_wfmenv env then
        let
            val G = HOLset.listItems (cont th)
            val G' = var_bigunion (List.map (fvt o (tinst env) o mk_var)
                G)
            val A = List.map (finst env) (ant th)
            val C = finst env (concl th)
        in
            thm(G',A,C)
        end
    else raise simple_fail "bad environment"
```

Figure 3.2: Code for Term Variable Instantiation Rule

function returns true if each variable (n, s) in the map's domain is mapped to a term t with the sort $\mathsf{sinst}\ \sigma\ s$. Once the check for well-formedness has been successfully passed, we proceed to instantiate the formulas within the assumption sets and the conclusion using the $\mathsf{finst}\ \sigma$ function. Regarding the context, which comprises variables (n, s), we transform them into variable terms, map them using σ and subsequently gather the variables present in all the output terms to form the new context. The SML code is: Note that we do not require the domain of the map to cover all the variables in the context. This will not result in any ill-formedness, as we have checked by verification in HOL4.

3.1.2 Formula variable instantiation

An instantiation map for formula variables associates each formula variable to a concrete formula. Each formula variable $(\mathcal{F}, \vec{v})$ in the domain of dictionary is associated with a well-formed formula of the form $\forall \vec{v}.\ \psi$. The instantiation of a formula variable $(\mathcal{F}, \vec{v})$ only takes care of subformulas of form $\mathcal{F}[\vec{v}](\vec{t})$. To facilitate this process, an auxiliary function is employed to modify such subformulas. Once a subformula of the pattern $\mathcal{F}[\vec{v}](\vec{t})$ is detected, the auxiliary function replaces the bounded variables in ψ with the actual terms $\vec{t}$. When instantiating a formula variable with a list of terms, in the form of $\mathcal{F}[v_1, \cdots, v_n](t_1, \cdots, t_n)$, we form out of the list of terms $[t_1, \cdots, t_n]$ a map of indices to actual terms, as $0 \mapsto t_n, \cdots, n-1 \mapsto t_1$. As an example:

3.1.1 Example. Consider the instantiation of $\mathcal{F}[(A, \mathrm{ob}), (B, \mathrm{ob}), (f, A \to B)](X, Y, h : X \to Y)$ with the predicate on $(A, B, f : A \to B)$ that f is an injection. We are required to specialize the quantification $\forall A\ B\ f : A \to B.\ \mathsf{Inj}(\mathsf{Bound}(0))$ with the term list $[X, Y, h : X \to Y]$. The indices map used here is $0 \mapsto h : X \to Y,\ 1 \mapsto B,\ 2 \mapsto A$, where the indices are observed inside the list of universal quantification. In this case, only the information $0 \mapsto h : X \to Y$ is used, because 1 and 2 do not appear explicitly in the body of the formula.

The subformula can appear anywhere within a formula and may involve bound variables. Leveraging the de Bruijn index approach, instead of the more complex process of stripping quantification, applying substitution, and re-abstraction, we can directly replace bound variables with terms that involve bound variables. This can be achieved efficiently by keeping track of the binding depth. The substitution essentially involves replacing specific numerical indices (corresponding to bound variables) with specific terms. As a result, the function responsible for this task takes as input a map that associates numbers (indices) with terms, as well as the formula ϕ. This function is named fprpl. The lower-level counterparts, tprpl and sprpl, which deal with terms and sorts, respectively, are relatively straightforward to define. The only non-trivial case arises in the clause concerning quantifiers, where the index values change.

3.1.2 Example. Consider the theory ZF and the formula variable $(\mathcal{F}, [(a, \mathsf{set})])$ on a set to be instantiated as the property of being non-empty, i.e., the combination of the variable list and the output of the map is $\psi := \forall A.\ \exists x_0.\ x_0 \in A$. In terms of de Brujin index, it is $\forall A.\ \exists x_0.\ \mathsf{Bound}(0) \in \mathsf{Bound}(1)$. We now want to instantiate the formula variable in the formula $\exists x.\ \mathcal{F}[(a, \mathsf{set})](x)$. The subformula containing the formula variable appears as $\mathcal{F}[(a, \mathsf{set})](\mathsf{Bound}(0))$. This means the formula variable is applied on the term $\mathsf{Bound}(0)$. Accordingly, we form a map that assigns $0 \mapsto \mathsf{Bound}(0)$, indicating the outermost bounded variable in ψ will be instantiated with $\mathsf{Bound}(0)$. However, the outermost bounded variable in $\exists x_0.\ x_0 \in \mathsf{Bound}(1)$ has its index not 0 but 1 because this formula is itself a quantification. In this case, we shift the map $0 \mapsto \mathsf{Bound}(0)$ to assign $1 \mapsto \mathsf{Bound}(1)$ instead. We then replace according to this map the index 1 in $\exists x_0.\ x_0 \in \mathsf{Bound}(1)$ into $\mathsf{Bound}(1)$, leaving the formula unchanged. Putting back the quantification on x, the resultant formula after instantiation is $\exists x.\ \exists x_0.\ \mathsf{Bound}(0) \in \mathsf{Bound}(1)$, to be interpreted as $\exists x.\ \exists x_0.\ x_0 \in x$.

The step of modifying the map on bounded indices is done by a function called $\mathsf{mapshift}$, taking a natural number regarded as the depth to be shifted, in this case, is 1, and a map to be shifted. The first thing modified by this function is the domain: As the utmost level of $\exists x_0.\ x_0 \in \mathsf{Bound}(1)$ is a quantification, the bounded variable $\mathsf{Bound}(0)$ already has an associated quantifier, so the index 0 is not in the domain of the index map anymore. Therefore, the domain should be modified to start with 1. The term that was previously the output for the index map at the input 0 should now be the output for 1 instead, more generally, the output for n should now be the output for $n + 1$. In other words, the output for n in the modified map should be acquired by applying the original map to the index $n - 1$. However, it's important to remember that the indices in the original output terms correspond to the original input indices. To maintain this correspondence, we need to adjust the indices in the output terms accordingly. In our example, the 0 in the domain becomes 1 in the shifted map, to keep the level correspondence, we need to add 1 to every index that appears in the original

```
fun fprpl bmap f =
    case f of
        Pred(P,tl) =>
          Pred(P,List.map (tprpl bmap) tl)
      | fVar(P,vl,tl) =>
          fVar(P,vl,(List.map (tprpl bmap) tl))
      | Conn(co,fl) =>
          Conn(co,List.map (fprpl bmap) fl)
      | Quant(q,n,s,b) =>
          Quant(q,n,sprpl bmap s, fprpl (mapshift 1 bmap) b)
```

Figure 3.3: SML Code of the Index-Replacing Function

output for 0. Adding a number uniformly to all the indices is done by the function tshift.

The general effect of the map mapshift is summarized as follows. The domain of mapshift i σ is $\{n + i \mid n \in \text{domain of } \sigma\}$. When applying on n in its domain, we have

$$(\text{mapshift } i\ \sigma)(n) = \text{tshift } i\ (\sigma(n - i))$$

Such an n is at least i, so the subtraction does not cause any misbehavior.

Employing this function, we define fprpl as:

3.2 Parsing

Despite the simplicity of the logic, the parsing procedure is a bit involved. Parsing consists of the following steps:

- Tokenizing a string into keywords and identifiers.
- Building an abstract syntax tree from the tokenized list.
- Turning an abstract syntax tree into a pre-sorted-syntax-structure.
- Performing type inference using unification.
- Building a well-formed, sorted syntax structure using the information obtained by type inference.

String → Token list Tokenizing is standard.

Token list → AST (Abstract Syntax Tree) The datatype of AST is declared as follows: In the Abstract Syntax Tree (AST) stage, we aim to determine the structure in which we should parse the token list without distinguishing among terms, sorts, and formulas. The constructors used for building the AST are designed to handle a variety of cases:

```
datatype ast =
         aId of string
         | aApp of string * ast list
         | aInfix of ast * string * ast
         | afV of string * ast list * ast list
         | aBinder of string * ast * ast
```

Figure 3.4: Constructors of AST

- aId (Atomic Identifier): This constructor takes a string as its argument and is used for names that stand alone. These names can represent variable names, sort names, nullary predicates, or constants.
- aApp (Application): This constructor is versatile and is used for function symbol and predicate symbol applications. It also covers two additional cases: sort dependency and negation (¬). In the case of sort dependency, the string represents the name of the sort, and the AST list is the list of terms it depends on. In the case of negation, the AST list is not eventually turned a list of terms but rather a singleton list containing a formula.
- aInfix (Infix Operator): This constructor is used for both the function symbol and predicate symbol applications as infixes and binary logical connectives. In the former case, the two ASTs will be eventually turned into argument terms, whereas in the latter case, they will be turned into formulas
- afV (Formula Variable): This constructor is specifically reserved for formula variables. Formula variables are distinguishable by the presence of two pairs of brackets, making them easy to identify.
- aBinder (Quantification): The recognition of quantifiers is straightforward, and a separate constructor is reserved exclusively for quantification cases. In this constructor, the string and the first AST capture the name and the sort of the variable, while the last AST captures the body of the quantification.

An AST records a variable is either atomic or an infix. In these two cases, the AST has a name, which is output by name_of_ast.

AST → pre-sorted-structure The next step is to transform the Abstract Syntax Tree (AST) into a preliminary representation of a term, a sort, or a formula. These preliminary structures, known as "pre-term", "pre-sort", and "pre-formula" are raw in the sense that the sorts of terms, that should eventually be stored in the constructors, are yet to be determined. See Figure 3.5.

We have three distinct functions, each responsible for creating the corresponding structure (each of pre-sort, pre-term and pre-form). Most of the pre-constructors above have the counterpart

```
1  datatype psort = psrt of string * pterm list
2                 | psvar of int
3       and pterm = ptUVar of int
4                 | pVar of string * psort
5                 | pFun of string * psort * pterm list
6                 | pAnno of pterm * psort
7  datatype pform = pPred of string * pterm list
8                 | pConn of string * pform list
9                 | pQuant of string * string * psort * pform
10                | pfVar of string * pterm list * pterm list ;
```

Figure 3.5: Constructors of Pre-syntax

for an "full version". Here are the only three exceptions: The constructor psvar reads "pre-sort variable", it will only later be associated with a concrete sort; The constructor Uvar reads "unification variable", it is generated in the case that we need some terms for an inferred sort to depend on. Both of these are only generated during the type-inference and are never built from an AST. The constructor pAnno reads "annotation". This constructor is used when the sort is annotated by the user.

The pre-syntax building functions take an AST and an *environment* and will output the pre-syntax with such an environment as well. The pre-syntax and the environment built in this step contain all the sort information that is immediately inferable from the provided AST. An environment is a tuple containing four maps and a natural number,

- A map $int \longrightarrow$ pterm, assigns each unification variable to a pre-term.
- A map $int \longrightarrow$ pterm, assigns each pre-sort variable to a pre-sort.
- A map $string \longrightarrow int$, records the correspondence between pre-variable names and the sort variable it is associated with.
- map $int \longrightarrow$ psort, records the pre-sort of each generated unification variable.

The fourth map is indeed required because a unification variable only has a name. Its constructor does not give a field to store its sort. All of them are implemented as a dictionary. The final component of an environment records the number that can be used for a fresh name for the next generated pre-sort variable or unification variable.

The three main functions we define for this step are shown in Figures 3.6, 3.7 and 3.8. In below, the function astl2ptl simply folds the function ast2pt and sortname_of_infix looks up the database to check the name of the output of an infix function symbol. The function ps_of_const takes the pre-sort of a constant. The first four components of the environment are modified by insert_us, insert_ps, record_ps and insert_pt. The function clear_ps erases from the third component. When a pre-term is not a unification variable, it has a pre-sort,

which can be obtained by applying ps_of_pt. We call fresh_var to increase the number stored in the last component of an environment by 1.

We do not perform unification during the conversion to pre-syntax, so no unification variable is generated and we do not use the first and the fourth component of the environment. The functions ast2ps, ast2pt building pre-terms and pre-sorts are mutually recursive. When attempting to turn an AST into a pre-term, the constructor aId can be transformed into either a constant using pFun or a pre-term variable. In the first case, the sort of the constant is acquired from the signature. This sort is then converted into a pre-sort and stored within the constructor. When the identifier is determined to be a variable with the name n, we check if it has already encountered this variable when converting another part of the same AST. This check is performed by examining the third map in the input environment. If the variable name n is already associated with a pre-sort variable k, then the AST is transformed into pVar(n, psvar k). In this scenario, the output consists of this pre-term, and the input environment remains unchanged. If the variable name n is not associated with a pre-sort variable in the environment, the system uses the name m recorded in the last slot of the environment. In this case, the output consists of pVar(n, psvar m), and the environment is modified by adding an extra record to the third map, mapping n to m. Additionally, the field of the fresh name is updated to $m+1$. For an AST constructed with aApp(f, *astl*), it can only be converted into a function pre-term. The resulting pre-term has the form pFun(f, psvar m, *ptl*), where m is a fresh name obtained from the environment, and the pre-term list *ptl* is constructed inductively by calling the ast2pt function. This recursive process enables the construction of function pre-terms from ASTs containing function symbol applications.

A pre-term pAnno can only be obtained from an AST of the form aInfix(a_1, :, a_2). This means the user intends to annotate the term a_1 to be of the sort a_2. In this case, we call ast2pt on a_1 and ast2ps on a_2, obtaining t_0 and s_0. The sort encoded within the constructor of t_0 will be unified with s_0 in the later step for type-inference. Other instances of infixes are treated in the same manner as a function application, except for when the symbol is not in the signature. For such a case, the function will fail, along with the other two remaining constructors that do not capture a pre-term, namely aBinder and afV.

The two constructors aBinder and afV are left to be dealt with ast2pf, the function building pre-formulas. On an afV, the function ast2pf calls ast2pt on both of the two AST lists to turn them into lists of pre-terms, with the first list consisting of pre-term variables only. When coming across an aBinder(q, v, b), the v will be an AST recording a variable that is bounded within b, an AST recording the body of the quantification. We first attempt to build the pre-term variable v. Let it have name n. The input environment *env* may already record a pre-sort for such a name, but within the quantification, it is not relevant. However, such a

```
fun ast2pt ast (env,n) =
    case ast of
        aId(a) =>
        if is_const a then
            (pFun(a,ps_of_const a,[]),env)
        else
            let val a' = if a = "_" then a ^ Int.toString n else a
                val n' = if a = "_" then n + 1 else n
            in case ps_of env a' of
                  NONE => let val (Av,env1) = fresh_var env
                              val env2 = record_ps a' (psvar Av) env1
                          in (pVar(a',psvar Av),env2)
                          end
                | SOME ps => (pVar(a',ps),env)
            end
      | aApp(str,astl) =>
        if is_fun str then
            case astl of
                [] =>
                let val (Av,env1) = fresh_var env
                in (pFun(str,psvar Av,[]),env1)
                end
              | h :: t =>
                let val (pt0,env1) = ast2pt (aApp(str,t)) env
                    val (pt,env2) = ast2pt h env1
                in (pFun_cons pt0 pt,env2)
                end
        else raise simple_fail ("not a function symbol: " ^ str)
      | aInfix(ast1,str,ast2) =>
        if str = ":" then
            case ast1 of
                aId(name) =>
                let val (ps0,env1) = ast2ps ast2 env
                in
                    case ps_of env name of
                        NONE =>
                        let val (Av,env2) = fresh_var env1
                            val env3 = record_ps name (psvar Av) env2
                            val ps = psvar Av
                            val env4 = insert_ps Av ps0 env3
                        in
                            (pAnno(pVar(name,ps),ps0), env4)
                        end
                      | SOME ps =>
                        let val name0 = psvar_name ps
                            val env2 = insert_ps name0 ps0 env1
                        in
                        (pAnno(pVar(name,ps),ps0),env2)
                        end
                end
              | _ =>
                let val (ps0,env1) = ast2ps ast2 env
                    val (pt0,env2) = ast2pt ast1 env1
                in
                    (pAnno(pt0,ps0),env2)
                end
        else
            if is_infix str then
                let val (pt1,env1) = ast2pt ast1 env
                    val (pt2,env2) = ast2pt ast2 env1
                    val (Av,env3) = fresh_var env2
                in
                    (pFun(str,psvar Av,[pt1,pt2]),env3)
                end
            else raise simple_fail ("not an infix operator: " ^ str)
      | aBinder(str,ns,b) =>
        raise simple_fail "quantified formula parsed as a term!"
      | afV _ => raise simple_fail "ast2pt.ast is a fVar, not a sort"
```

Figure 3.6: SML function producing pre-term

```
and ast2ps ast (env,n) =
    case ast of
        aId(ns) => (psrt(ns,[]),env)
      | aApp(ns,atl) =>
        let val (ptl,env1) = astl2ptl atl env
        in (psrt(ns,ptl),env1)
        end
      |aInfix(ast1,inx,ast2) =>
       let
           val (pt1,env1) = ast2pt ast1 env
           val (pt2,env2) = ast2pt ast2 env1
           val sn = sortname_of_infix inx
       in
           (psrt(sn,[pt1,pt2]),env2)
       end
      | _ => raise PER ("ast2ps.wrong␣attempt␣of␣trying␣to␣turn␣an␣
         aBinder␣or␣afV␣into␣a␣pre-sort",[],[])
```

Figure 3.7: SML function producing pre-sort

record cannot be discarded because it is still required to build the remaining parts of a formula. Thus, this information is set aside as follows: When building the body b, we erase such a record and associate n with a fresh pre-sort variable m. Building the pre-formula for b will give us another environment env_1. We erase the record $n \mapsto m$ from the third map of env_1 and put back the previous pre-sort record on n, which gives us env_2. Then env_2 will be the output environment, together with the pre-formula built.

3.2.1 Example. Consider the parsing of the string $\circ(g, f : A \to B)$ representing the composition of two arrows into the AST:

$$\mathsf{aApp}(\text{“}\circ\text{”}, \mathsf{aId}(\text{“}g\text{”}),\ \mathsf{aInfix}(\mathsf{aId}(\text{“}f\text{”}),\ :,\ \mathsf{aInfix}(\mathsf{aId}(\text{“}A\text{”}), \to, \mathsf{aId}(\text{“}B\text{”}))))$$

as we are working in the signature of ETCS. To build it into a pre-term, we start from an empty environment $([],[],[],[],0)$. The constructor aApp indicates that it is a function term, we descend into the branches to build its arguments. The first argument $\mathsf{aId}(g)$ will be turned into a variable since there is no constant with the name g. Taking the first fresh name from the last component of the environment, we build it into the pre-variable $\mathsf{pVar}(g, \mathsf{psvar}\ 0)$, and update the environment into $([],[],[g \mapsto 0],[],1)$. The second argument is an annotation. Processing $\mathsf{aId}(f)$ gives $\mathsf{pVar}(f, \mathsf{psvar}\ 1), ([],[],[g \mapsto 0;\ f \mapsto 1],[],2)$. The pre-sort constructed with the infix “$\to$” is built into $\mathsf{psrt}(\mathsf{ar}, [\mathsf{pVar}(A, \mathsf{psvar}\ 2),\ \mathsf{pVar}(B, \mathsf{psvar}\ 3)])$, with environment $([],[],[g \mapsto 0;\ f \mapsto 1;\ A \mapsto 2;\ B \mapsto 3],[],4)$. This environment is kept the same when we output the annotated pre-term $\mathsf{pAnno}(\mathsf{pVar}(f, \mathsf{psvar}\ 1), \mathsf{psrt}(\mathsf{ar}, [\mathsf{pVar}(A, \mathsf{psvar}\ 2),\ \mathsf{pVar}(B, \mathsf{psvar}\ 3)]))$. The remaining task is to put the two arguments together and assign the function term a pre-sort

```
fun ast2pf ast (env:env,n) =
    case ast of
        aId(a) =>
        if a = "T" then (pPred("T",[]),env) else
        if a = "F" then (pPred("F",[]),env) else
        (case (lookup_pred (!psyms) a) of
             SOME l => if l = [] then (pPred(a,[]),env)
                       else raise simple_fail ("Using multi-argument "^a^
                                               " as a pred. var")
           | _ => (pfVar(a,[],[]),env))
      | aApp("~",[ast]) =>
        let val (pf,env1) = ast2pf ast env in
            (pConn("~",[pf]),env1)
        end
      | aApp(str,astl) =>
        let val (ptl, env1) = astl2ptl avl env
        in (pPred(str,ptl),env1)
        end
      | afV(str,avl,aargl) =>
        let val (pvl,env1) = astl2ptl avl env
            val (pargl,env2)  = astl2ptl aargl env1
        in
            (pfVar (str,pvl,pargl),env2)
        end
      | aInfix(ast1,str,ast2) =>
        if mem str ["&","|","<=>","==>"] then
            let
                val (pf1,env1) = ast2pf ast1 env
                val (pf2,env2) = ast2pf ast2 env1
            in
                (pConn(str,[pf1,pf2]),env2)
            end else
        if mem str ["=","=="] then
            let
                val (pt1,env1) = ast2pt ast1 env
                val (pt2,env2) = ast2pt ast2 env1
            in
                (pPred(str,[pt1,pt2]),env2)
            end else
        raise simple_fail ("not an infix operator: " ^ str)
     | aBinder(str,ns,b) =>
        if mem str ["!","?","?!"] then
            let val name = name_of_ast ns
                val pso = ps_of env name
                val env1 = clear_ps name env
                val (Av,env2) = fresh_var env1
                val env3 = record_ps name (psvar Av) env2
                val (pt,env4) = ast2pt ns env3
                val (ps,env5) = ps_of_pt pt env4
                val (pf,env6) = ast2pf b env5
                val env7 = clear_ps name env6
                val env8 = case pso of SOME ps => record_ps name ps env7
                    | _ => env7
            in
                (pQuant(str, name, ps, pf),env8)
            end
        else raise simple_fail "not a quantifier"
```

Figure 3.8: SML function producing pre-formula

variable, obtaining the pair consisting of

$$\mathsf{pFun}(\circ,\ \mathsf{psvar}\ 4,\ [\mathsf{pVar}(g, \mathsf{psvar}\ 0), \mathsf{pAnno}(\mathsf{pVar}(f, \mathsf{psvar}\ 1), \mathsf{psrt}(\mathsf{ar}, [\mathsf{pVar}(A, \mathsf{psvar}\ 2),\ \mathsf{pVar}(B, \mathsf{psvar}\ 3)]))])$$

and

$$([], [], [g \mapsto 0;\ f \mapsto 1;\ A \mapsto 2;\ B \mapsto 3], [], 5)$$

Type-inference via unification This step infers sort information. For each pre-sort variable, we will eventually unify it with a concrete pre-sort. These pre-sorts may involve some generated unification variables. In the case that the unification variables can be inferred to be a more concrete term, it is unified with that term. When the signature requires two unification variables to agree, this is resolved by unifying the two unification variables. If multiple unification variables agree, the "chasing" functions chases and chaset, will eventually direct them all to the same pre-term. Such a term can be a more concrete term or just a single unification variable.

The primary basis for type inference comes from the signature of function and predicate symbols. The type inference function for terms takes an environment env, a pre-term t_0, and a pre-sort s_0 as inputs and produces a new environment as output. The definition of such a function is shown in Figure 3.9, where the unification functions called by it are defined in Figure 3.10. Unification is performed during the process of inferring the sort of t_0 to be s_0.

The type-inference function for predicate symbols is the same as the one for function symbols, except that there is no need to infer the sort of the output term. Quantification case $\mathsf{pQuant}(q, n, s_0, b_0)$ is handled similarly to constructing the AST. We erase the association from n to the pre-sort variable k, if any, from the provided environment, assign it a new pre-sort variable m, bind $m \mapsto s_0$ in the second map of the environment, type infer the body, and then add back the association $n \mapsto k$. This is still necessary because the type-inference of this quantified formula might be a part of a connective, in which case the type-inference accumulates on a list. The variable n can still be present in other parts, where the sort information on it remains useful.

3.2.2 Example. We continue with the previous example. For type-inferring this annotation term, we look up the function symbol $\circ$ in the signature. The signature gives that for the input list $[f : A \to B,\ g : B \to C]$, the output $g \circ f$ is of sort $A \to C$. According to the signature, we generate in total 5 unification variables: Two for the arrows, and three for the domains and codomains. We associate the generated unification variables with names from 6 to 10, where 8, 9 and 10 are "pre-object", which means that their pre-sorts stored in the 4-th map in the environment are all $\mathsf{psrt}(\mathsf{ob}, [])$. In that map, we also associate 6 to the pre-arrow from 9 to 10, and 7 to the pre-arrow from 8 to 9. The type-inference on g does not make a change. That on

```
1  fun type_infer_ptl env ptl =
2      List.foldr
3          (fn (pt,env) =>
4              let val (ps,env1) = ps_of_pt pt env
5              in type_infer env1 pt ps
6              end)
7          env ptl
8  and type_infer_pfun env t ty =
9      case t of
10         pFun(f,ps,ptl) =>
11         let
12             val env = type_infer_ptl env ptl
13         in
14         (case lookup_fun (!fsyms) f of
15              SOME (s,l) =>
16              let val (uvs,nd,env1) = npsl2ptUVarl (map ns2nps l) env
17                  val (s1,nd1,env2) = fgt_name_ps (s2ps s) nd env1
18                  val tounify = zip ptl uvs
19                  val env3 = foldr
20                                 (fn ((a,b),env) => unify_pt env a b)
21                                 env1 tounify
22              in
23                  unify_ps (unify_ps env3 ty s1) ty ps
24              end
25            | _ => env)
26         end
27       | _ => raise simple_fail ("not␣a␣function␣term" ^ (stringof_pt t
          ))
28 and type_infer env t ty =
29     case t of
30         pFun(f,ps,ptl) => type_infer_pfun env t ty
31       | pAnno (pt,ps) =>
32         let val env1 = type_infer env pt ps
33             val (ps',env1') = (ps_of_pt pt env1)
34             val env2 = type_infer env1' pt ps'
35         in unify_ps env2 ty ps
36         end
37       | pVar (name,ps) =>
38         (case ps of
39              psrt(sn,ptl) =>
40              let val env = type_infer_ptl env ptl
41              in unify_ps env ty ps
42              end
43            | _ => unify_ps env ty ps)
44       | ptUVar name =>
45         (case lookup_us env name of
46              SOME ps => unify_ps env ps ty
47            | _ => insert_us name ty env)
```

Figure 3.9: Type inference from terms

the annotation term f will result in storing $1 \mapsto \mathsf{psrt}(\mathsf{ar}, [\mathsf{pVar}(A, \mathsf{psvar}\ 2),\ \mathsf{pVar}(B, \mathsf{psvar}\ 3)])$ in the second map.

Building well-formed syntax The remaining step of constructing formal syntax from pre-syntax amounts to descending along the tree structure of the pre-syntax and acquiring the sort information exclusively from the provided environment. This means that there's no need to consult the signature of function and predicate symbols at this stage. Three functions, pt2tm, ps2st and pf2fm, are responsible for taking a pre-syntax and an environment and producing the corresponding formal syntax. All four maps stored in the environment may be consulted during this step to ensure accurate sort information is obtained for constructing the formal syntax.

To acquire the sorts, we need to know where the pre-sort variables and the unification variables eventually go. This is found out by two chasing functions chasevars and chasevart. They are declared as where the functions lookupt and lookups simply find the outputs of the first and the second map in the environment. The chasing functions do terminate. This is ensured by the occurs check on the previous unification step that creates the environment. The procedure of syntax-building is straightforward. The only thing to note is that a quantified formula from $\mathsf{pQuant}(q, n, s_0, b_0)$ is built by firstly building the body to be a well-formed formula, where the sort of the variable n is obtained from not the provided environment but s_0, then abstract on the variable n. A unification variable that is not eventually assigned a concrete term will be left with a number as its name. A pre-sort variable that is not eventually assigned a concrete sort will just be assigned the first ground sort stored in the signature.

3.2.3 Example. We finally build the composition term according to the environment obtained from the last example. The sort of the whole function term is an arrow from the unification variable 8 to 10. When building this sort, we chase in the environment where these two variables go to and get that 8 goes to A. As the 10 is not associated with anything, the whole function term will end up being of an arrow sort from A to 10. The sort information of A is obtained because its pre-sort variable is unified with the counterpart of 8, which is assigned to a pre-object from the point it is generated.

```
fun unify_ps env (ps1:psort) (ps2:psort):env =
    case (chasevars ps1 env,chasevars ps2 env) of
        (psvar n1,psvar n2) =>
        if n1 = n2 then env else insert_ps n1 (psvar n2) env
      | (psvar n,ps) =>
        if occs_ps n env ps
        then raise UNIFY
                  ("occurs␣check(ps):" ^ stringof_ps (psvar n) ^ "␣" ^
                  stringof_ps ps,[])
        else insert_ps n ps env
      | (ps,psvar n) =>
        if occs_ps n env ps
        then raise UNIFY
                  ("occurs␣check(ps):" ^ stringof_ps (psvar n) ^ "␣" ^
                  stringof_ps ps,[])
        else insert_ps n ps env
      | (psrt (sn1,ptl1),psrt (sn2,ptl2)) =>
        if sn1 = sn2 then
            List.foldr (fn ((a,b),env) => unify_pt env a b)
                       env (zip ptl1 ptl2)
        else raise UNIFY ("different␣sorts:␣" ^ sn1 ^ "␣,␣" ^ sn2,[])
and unify_pt env pt1 pt2: env=
    case (chasevart pt1 env,chasevart pt2 env) of
        (ptUVar a,ptUVar b) =>
        if a = b then env else
        let val (psa, env1) = ps_of_pt pt1 env
            val (psb, env2) = ps_of_pt pt2 env1
            val env3 = unify_ps env2 psa psb
        in insert_pt a (ptUVar b) env3
        end
      | (ptUVar a, t) =>
        if occs_pt a env t
        then raise UNIFY ("occurs␣check(pt):" ^
             stringof_pt (ptUVar a) ^ "␣" ^ stringof_pt t,[])
        else
            let val (ps1, env1) = ps_of_pt pt1 env
                val (ps2, env2) = ps_of_pt pt2 env1
                val env3 = unify_ps env2 ps1 ps2
            in
                insert_pt a t env3
            end
      | (t, ptUVar a) =>
        if occs_pt a env t
        then raise UNIFY ("occurs␣check(pt):" ^
             stringof_pt t ^ "␣" ^ stringof_pt (ptUVar a),[])
        else
            let val (ps1, env1) = ps_of_pt pt1 env
                val (ps2, env2) = ps_of_pt pt2 env1
                val env3 = unify_ps env2 ps1 ps2
            in
                insert_pt a t env3
            end
      | (pVar (a1,ps1), pVar (a2,ps2)) =>
        if a1 = a2 then unify_ps env ps1 ps2
        else raise UNIFY ("different␣variable␣name",[])
      | (pFun(f,ps1,l1),pFun(g,ps2,l2)) =>
        if f = g andalso length l1 = length l2
        then (let val env = unify_ps env ps1 ps2 in
                  case (l1,l2) of
                  ([],[]) => env
                | (h1::r1,h2::r2) =>
                  let val env1 = unify_pt env h1 h2
                  in unify_pt env1 (pFun(f,ps1,r1)) (pFun(g,ps2,r2))
                  end
                | _ => raise UNIFY ("term␣list␣cannot␣be␣unified",[])
             end)
        else raise UNIFY ("different␣functions:"^ f ^ ",␣" ^ g  ,[])
      | (pAnno(pt,ps),t) => unify_pt env pt t
      | (t,pAnno(pt,ps)) => unify_pt env pt t
      | _ => raise UNIFY ("terms␣cannot␣be␣unified",[])
```

Figure 3.10: SML code for unification

Chapter 4

Verification

Despite their convenience, formula variables obviously add a great deal of complexity to our system and we want to make sure they do not break soundness. To test that our logic works as expected, we formalize the system in the theorem prover HOL [5]. By formalizing, we make sure that:

- Well-formedness of terms, sorts and formulas are preserved under our defined operations such as substitutions.
- Formula variables are only a convenience, so our system has no more power than dependent sorted FOL.

We start with the HOL formalization of the syntax of our logic. In particular, we give formal definitions of well-formed terms, sorts, and formulas. We then prove our instantiation operation preserves well-formedness. After that, we formalize our proof system via inductive relations. By the end of this chapter, we will arrive at our conclusion which addresses our concern, which can be summarized as "formula variables can be effectively eliminated".[1]

4.1 Syntax

4.1.1 Terms and Sorts

Terms and sorts are mutually recursive with each other as a HOL datatype:

```
term = Var string sort | Fn string sort (term list) | Bound num;

sort = St string (term list)
```

[1] If we modeled still more of our implementation within the logic, one might hope that an entire implementation could be automatically derived, or "extracted", from this logical specification. For example, this approach is behind the construction of the Candle theorem prover [12].

The functions collecting free variables in terms and sorts defined in Section 2.1 are directly formalized, and output sets of free variables in a term t or a sort s as $\mathsf{tfv}\ t$ and $\mathsf{sfv}\ s$.

A set V of variables is a *well-formed context* (abbreviated as "is context" in this chapter, formalized in HOL as $\mathsf{cont}\ V$) if once $(n, s) \in V$, we have $\mathsf{sfv}\ s \subseteq V$. For a term t and a sort s, the sets $\mathsf{tfv}\ t$ and $\mathsf{sfv}\ s$ inherently constitute well-formed contexts.

Because terms and sorts are algebraic, we have $\forall s\ n.\ (n, s) \notin \mathsf{sfv}\ s$. This is proved by defining a size-measuring function on terms and sorts and appealing to the fact that the sort size of any free variable appearing in a term is strictly less than the size of the term. The size measuring function is used across our whole formalization to prove the termination of functions defined on terms and sorts.

This section serves to explain the definition of term well-formedness, which is declared as:

Definition 4.1.1.

$$
\begin{aligned}
&\mathsf{wft}\ (\Sigma_s, \Sigma_f)\ (\mathsf{Var}\ n\ s) \stackrel{\mathsf{def}}{=} \mathsf{wfs}\ (\Sigma_s, \Sigma_f)\ s \\
&\mathsf{wft}\ (\Sigma_s, \Sigma_f)\ (\mathsf{Fn}\ f\ s\ tl) \stackrel{\mathsf{def}}{=} \\
&\quad \mathsf{wfs}\ (\Sigma_s, \Sigma_f)\ s \ \wedge\ (\forall t.\ \mathsf{MEM}\ t\ tl \implies \mathsf{wft}\ (\Sigma_s, \Sigma_f)\ t) \ \wedge\ \mathsf{isfsym}\ \Sigma_f\ f \ \wedge \\
&\quad \mathsf{IS\_SOME}\ (\mathsf{tlmatch}\ \emptyset\ (\mathsf{MAP\ Var'}\ (\mathsf{fsymin}\ \Sigma_f\ f))\ tl\ \mathsf{FEMPTY}) \ \wedge \\
&\quad \mathsf{in_s}\ (\mathsf{THE}\ (\mathsf{tlmatch}\ \emptyset\ (\mathsf{MAP\ Var'}\ (\mathsf{fsymin}\ \Sigma_f\ f))\ tl\ \mathsf{FEMPTY}))\ (\mathsf{fsymout}\ \Sigma_f\ f) = s \\
&\mathsf{wft}\ (\Sigma_s, \Sigma_f)\ (\mathsf{Bound}\ i) \stackrel{\mathsf{def}}{=} \mathsf{F} \\
&\mathsf{wfs}\ (\Sigma_s, \Sigma_f)\ (\mathsf{St}\ n\ tl) \stackrel{\mathsf{def}}{=} \mathsf{EVERY}\ (\lambda a.\ \mathsf{wft}\ (\Sigma_s, \Sigma_f)\ a)\ tl \ \wedge\ \mathsf{MAP\ tsname}\ tl = \Sigma_s\ '\ n
\end{aligned}
$$

where

- The predicate $\mathsf{IS\_SOME}$ on a term of the option type means that it is not the NONE, i.e. the matching does not fail.
- The term $\mathsf{in_s}\ \theta\ s$ is obtained by instantiating the sort s by the map θ. The instantiation function takes a finite map θ of HOL type $\texttt{string} \times \texttt{sort} \mapsto \texttt{term}$ and a term t or a sort s, gives the term $\mathsf{in_t}\ \theta\ t$ or $\mathsf{in_s}\ \theta\ t$ by replacing each variable v in the domain of θ into $\theta\ '\ v$, and skipping any bound variables. We will keep using the Greek letter θ for term instantiation maps. Whereas a matching can fail, an instantiation always gives an output.
- The notation $\mathsf{Var'}$ uncurries the constructor Var as a function, i.e. $\mathsf{Var'}\ (n, s) = \mathsf{Var}\ n\ s$.
- The predicate $\mathsf{EVERY}\ P$ for a term P with type $\alpha \to \texttt{bool}$ that encodes a predicate holds for an α-list if and only if every item of the list satisfies P.
- The function THE takes out the term encoded in an option type, i.e. $\mathsf{THE}\ (\mathsf{SOME}\ a) = a$

The clause on function terms says a term Fn f s tl is well-formed when f is a function symbol, all the terms in tl are well-formed, and moreover, the sort encoded in s must be the same as the sort built from the signature according to the sort of the arguments tl. As bound variables only occur when an abstraction happens while building quantified formulas, they are never well-formed when standing alone. As the check is recursive, any subterm of a well-formed term or sort has to be well-formed.

Well-formed terms are those constructed in accordance with a signature pair (Σ_s, Σ_f) where Σ_s is a list of sort names (as strings) that records the sort dependency, and Σ_f is the sort information of function symbols. In our formalization, a signature on function symbols is a finite map `string` $\mapsto$ `sort` $\times$ (`string` $\times$ `sort`) `list`. This is denoted as Σ_f. We write isfsym Σ_f f if the string f is in the domain of Σ_f, meaning it is a function symbol. We write fsymin Σ_f f for f's input variable list, and fsymout Σ_f f for the expected sort of the output based on the inputs.

To do this check, we match the variables recorded in the signature to the terms in the argument list. The matching function is of type (`string` $\times$ `sort` $\rightarrow$ `bool`) $\times$ `term` $\times$ `term` $\times$ (`string` $\times$ `sort` $\mapsto$ `term`) $\rightarrow$
(`string` $\times$ `sort` $\mapsto$ `term`) `option`. It takes a set *lcs* of local constants, a term serving as a pattern, a concrete term to be matched with, a map storing variable assignments that are already determined, and outputs a map as the result of the matching process if the two input terms can be matched. If the terms cannot be matched, it returns the option NONE. We present its definition in Figure 4.1 As above, a variable fails to match if we try to match:

- a variable to a variable with a bound variable in it.
- a local constant with another term, except to itself.
- a variable to another term, given it is already assigned to some term by the input map.

Matching a bound variable doesn't contribute any new information to be recorded in the map. Nevertheless, it can cause the matching function to fail if an attempt to match two bound variables with different depths is detected. A function symbol application can only be matched with the application of the same function symbol. Also visible from above is that this function only augments the input map; it never removes anything from it. This is a standard example of formalizing a definition with mutual recursion.

We usually start the matching with the empty map FEMPTY. Hence the map we obtain will have no bound variables in its codomain (written as no_bound in HOL). The set of local constants is taken to be empty when performing well-formedness checks. In this case, the obtained map has its domain precisely the set tfv t. We call a map whose domain is a context to be *complete*. Starting a matching with a complete map will give a complete map. Such

```
tmatch lcs (Var n s) ct f =def
 if tbounds ct ≠ ∅ then NONE
 else if (n, s) ∈ lcs then if Var n s = ct then SOME f else NONE
 else if (n, s) ∈ FDOM f then if ct = f ' (n, s) then SOME f else NONE
 else
  case smatch lcs s (sort_of ct) f of
   NONE ⇒ NONE
  | SOME f0 ⇒ SOME (f0 |+ ((n, s), ct))
tmatch lcs (Fn f1 s1 tl1) (Fn f2 s2 tl2) f =def
 if f1 = f2 then
  case tlmatch lcs tl1 tl2 f of NONE ⇒ NONE | SOME σ0 ⇒ smatch lcs s1 s2 σ0
 else NONE
tmatch lcs (Fn v0 v1 v2) (Var v3 v4) f =def NONE
tmatch lcs (Fn v5 v6 v7) (Bound v8) f =def NONE
tmatch lcs (Bound i) (Bound j) f =def if i = j then SOME f else NONE
tmatch lcs (Bound i) (Var v9 v10) f =def NONE
tmatch lcs (Bound i) (Fn v11 v12 v13) f =def NONE
smatch lcs (St n1 tl1) (St n2 tl2) f =def
 if n1 = n2 then tlmatch lcs tl1 tl2 f else NONE
tlmatch lcs [] [] f =def SOME f
tlmatch lcs [] (h :: t) f =def NONE
tlmatch lcs (h :: t) [] f =def NONE
tlmatch lcs (h1 :: t1) (h2 :: t2) f =def
 case tmatch lcs h1 h2 f of NONE ⇒ NONE | SOME f1 ⇒ tlmatch lcs t1 t2 f1
```

Figure 4.1: Definition of term matching

theorems are all formalized in HOL using mutual induction. An important result that we have formalized states that a pattern can be matched to a concrete term if and only if the term is an *instantiation* of the pattern. This result is elegantly expressed as an equivalence, which can be formulated as follows:

Theorem 4.1.1.

$$\begin{aligned}&\vdash \mathsf{IS\_SOME}\ (\mathsf{tmatch}\ \emptyset\ t_1\ t_2\ \mathsf{FEMPTY}) \iff \\ &\quad \exists \theta.\ \mathsf{cstt}\ \theta\ \wedge\ \mathsf{no\_bound}\ \theta\ \wedge\ \mathsf{tfv}\ t_1\ =\ \mathsf{FDOM}\ \theta\ \wedge\ t_2\ =\ \mathsf{in_t}\ \theta\ t_1\end{aligned}$$

where the condition $\mathsf{cstt}\ \theta$ means θ is *consistent*, meaning for each variable $(n, s) \in \mathsf{FDOM}\ \theta$, we have $\mathsf{sort\_of}\ (\theta\ '\ (n, s)) = \mathsf{in_s}\ \theta\ s$.

The procedure of instantiation can be iterated: If θ_1 contains every variable in t and θ_2 contains every variable in $\mathsf{in_t}\ \theta_1\ t$, then we can combine the two instantiation maps into a single one by instantiating the variables in the codomain of θ_1 with θ_2, resulting in a map called $\mathsf{o\_vmap}\ \theta_2\ \theta_1$. With this combined map, we can obtain the result of two instantiations with a single instantiation, as we prove $\mathsf{in_t}\ \theta_2\ (\mathsf{in_t}\ \theta_1\ t) = \mathsf{in_t}\ (\mathsf{o\_vmap}\ \theta_2\ \theta_1)\ t$. Accordingly, matching can also be iterated. If the term t_1 can be matched to t_2 and t_2 can be matched to t_3, then t_1 can be matched to t_3. By combining these theorems, we yield transitivity of matchability:

Theorem 4.1.2.

$$\begin{aligned}&\vdash \mathsf{IS\_SOME}\ (\mathsf{tmatch}\ \emptyset\ t_1\ t_2\ \mathsf{FEMPTY})\ \wedge\ \mathsf{IS\_SOME}\ (\mathsf{tmatch}\ \emptyset\ t_2\ t_3\ \mathsf{FEMPTY}) \implies \\ &\quad \mathsf{IS\_SOME}\ (\mathsf{tmatch}\ \emptyset\ t_1\ t_3\ \mathsf{FEMPTY})\end{aligned}$$

4.1.2 Formulas

The formula type is straightforward, apart from the fVar constructor:

```
form =
    ⊥
  | Pred string (term list)
  | IMP form form
  | FALL sort form
  | fVar string (sort list) (term list)
```

There are two notable differences between the presentation in Chapter 2 and the formalization. Using de Brujin indices, the quantification constructor does not have to carry the name of the variable. As a consequence, the variable list for a formula variable, encoding a λ-abstraction, does not have to carry their names either. The sorts in that list are then not necessarily

well-formed when taken in isolation because they may contain bound variables, referring to the previous items in the list.

4.1.1 Example. In the previous presentation as in Chapter 2, a formula variable to be instantiated with a predicate on the meta-tuple of the form $[A, B, f : A \to B]$ is recorded $\mathcal{F}[A : \mathsf{set}, B : \mathsf{set}, f : A \to B]$. With de Brujin indices, it is now recorded $\mathcal{F}[\mathsf{set}, \mathsf{set}, \mathsf{ar} : \mathsf{Bound}\ 1 \to \mathsf{Bound}\ 0]$. The $\mathsf{Bound}\ 0$ here, according to the construction of de Brujin indices, refers to the quantified term that is closest to the current level, that is, the second item in the list. The $\mathsf{Bound}\ 1$ refers to the term bound at the head of the list. Accordingly, the iterated quantification $\forall A\ B\ f : A \to B.\ \phi$ in Chapter 2 will become $\forall \mathsf{set}\ \mathsf{set}\ (\mathsf{Bound}\ 1 \to \mathsf{Bound}\ 0).\ \phi$ in the formalization.

We write $\mathsf{ffv}\ f$ for the set of free variables in a formula f, and write $\mathsf{fVars}\ f$ for the set of formula variables, consisting of pairs (P, sl). Variables that appear in the sort list of a formula variable are now interesting. Such a sort list is regarded as obtained by iterated abstraction, and the variables remaining in this list are the ones that do not become bound during the abstraction. Our well-formedness predicate on formulas will refer to this set of variables. There are also many places where we need to collect variables. To collect variables uniformly, we introduce a new syntax. We define $\mathsf{Uof}\ f\ s$ as an abbreviation of $\bigcup\{f\ x \mid x \in s\}$. The HOL term Uof has type $(\alpha \to \beta \to \mathtt{bool}) \to (\alpha \to \mathtt{bool}) \to \beta \to \mathtt{bool}$. It admits neat rewritings such as:

$$\vdash \mathsf{Uof}\ f\ (A \cup B) = \mathsf{Uof}\ f\ A \cup \mathsf{Uof}\ f\ B$$

$$\vdash \mathsf{Uof}\ f\ A \subseteq B \iff \forall a.\ a \in A \Rightarrow f\ a \subseteq B$$

Then the free variables appearing in the sort list of a formula variable in f are collected as $\mathsf{fVslfv}\ f \stackrel{\mathsf{def}}{:=} \mathsf{Uof}\ (\mathsf{slfv} \circ \mathsf{SND})\ (\mathsf{fVars}\ f)$, where slfv serves to collect the variables in a sort list.

Our predicate, wff asserting the well-formedness of a formula takes a 4-tuple $(\Sigma_s, \Sigma_f, \Sigma_p, \Sigma_e)$ and a formula. It is inductive on the formula. The first and second components record sort dependency and function symbols and the third component records predicate symbols. Equality is not stored as a predicate symbol. The last component Σ_e records the name of the sorts where we can use equality.

The complication of well-formed formulas is brought by quantification and formula variables. Beyond these two cases, other clauses are simply:

$$\vdash \mathsf{wff}\ (\Sigma_s, \Sigma_f, \Sigma_p, \Sigma_e)\ \bot$$

$$\begin{aligned}\vdash\ & \mathsf{wft}\ (\Sigma_s, \Sigma_f)\ t_1\ \wedge\ \mathsf{wft}\ (\Sigma_s, \Sigma_f)\ t_2\ \wedge\ \mathsf{sort\_of}\ t_1\ =\ \mathsf{sort\_of}\ t_2\ \wedge \\ & \mathsf{has\_eq}\ \Sigma_e\ (\mathsf{tsname}\ t_2)\ \Rightarrow \\ & \mathsf{wff}\ (\Sigma_s, \Sigma_f, \Sigma_p, \Sigma_e)\ (\mathsf{EQ}\ t_1\ t_2)\end{aligned}$$

$$\begin{aligned}\vdash\ & (\forall\, t.\ \mathsf{MEM}\ t\ tl\ \Rightarrow\ \mathsf{wft}\ (\Sigma_s, \Sigma_f)\ t)\ \wedge\ \mathsf{ispsym}\ \Sigma_p\ p\ \wedge \\ & \mathsf{IS\_SOME}\ (\mathsf{tlmatch}\ \emptyset\ (\mathsf{MAP}\ \mathsf{Var'}\ (\Sigma_p\ '\ p))\ tl\ \mathsf{FEMPTY})\ \Rightarrow \\ & \mathsf{wff}\ (\Sigma_s, \Sigma_f, \Sigma_p, \Sigma_e)\ (\mathsf{Pred}\ p\ tl)\end{aligned}$$

$$\vdash \mathsf{wff}\ (\Sigma_s, \Sigma_f, \Sigma_p, \Sigma_e)\ f_1\ \wedge\ \mathsf{wff}\ (\Sigma_s, \Sigma_f, \Sigma_p, \Sigma_e)\ f_2\ \Rightarrow\ \mathsf{wff}\ (\Sigma_s, \Sigma_f, \Sigma_p, \Sigma_e)\ (\mathsf{IMP}\ f_1\ f_2)$$

As we can readily read, an equality is well-formed if and only if both sides are well-formed terms with the same sort, where the sort is required to have equality. A predicate is well-formed if the arguments are well-formed, and have the correct sorts according to the signature.

Now let us investigate the complicated cases. According to the discussion in Chapter 2, before making quantifiers, we need to check on the variable (n, s) to be quantified that it does not appear in any formula variable and is on the top level, meaning it does not appear in any sort of a free variable in f. When the quantification is allowed, we build it via the formula $\forall_{\mathsf{mk}}\ n\ s\ f$. It in turn calls a function fabs that replaces the term $\mathsf{Var}\ n\ s$ by a bound term $\mathsf{Bound}\ i$, where the index i is initially 0, and increases as the recursion descends past other quantifiers in the formula.

The inductive rule for universal quantification is thus:

$$\begin{aligned}\vdash\ & \mathsf{wff}\ (\Sigma_s, \Sigma_f, \Sigma_p, \Sigma_e)\ f\ \wedge\ \mathsf{wfs}\ (\Sigma_s, \Sigma_f)\ s\ \wedge\ (n, s)\ \notin\ \mathsf{fVslfv}\ f\ \wedge \\ & (\forall\, n_1\ s_1.\ (n_1, s_1)\ \in\ \mathsf{ffv}\ f\ \Rightarrow\ (n, s)\ \notin\ \mathsf{sfv}\ s_1)\ \Rightarrow \\ & \mathsf{wff}\ (\Sigma_s, \Sigma_f, \Sigma_p, \Sigma_e)\ (\forall_{\mathsf{mk}}\ n\ s\ f)\end{aligned}$$

One might consider an alternative possibility of characterizing this well-formedness without inductive rules. One could substitute the variable into the body of the formula, and assert the abstracted formula to be well-formed if the formula obtained by substitution is well-formed. We can derive such a characterization of well-formed universal formulas as a consequence of

our definition, formalized as:

$$\begin{aligned}
&\vdash \mathsf{wff}\ (\Sigma_s, \Sigma_f, \Sigma_p, \Sigma_e)\ (\mathsf{FALL}\ s\ b) \iff \\
&\quad \exists f\ n. \\
&\qquad \mathsf{wff}\ (\Sigma_s, \Sigma_f, \Sigma_p, \Sigma_e)\ f\ \wedge\ \mathsf{wfs}\ (\Sigma_s, \Sigma_f)\ s\ \wedge\ (n, s)\ \notin\ \mathsf{fVslfv}\ f\ \wedge \\
&\qquad (\forall\, n_1\ s_1.\ (n_1, s_1)\ \in\ \mathsf{ffv}\ f\ \Longrightarrow\ (n, s)\ \notin\ \mathsf{sfv}\ s_1)\ \wedge\ \mathsf{FALL}\ s\ b\ =\ \forall_{\mathsf{mk}}\ n\ s\ f
\end{aligned}$$

The most complicated clause is that for formula variables. Eventually, we will define:

$$\vdash \mathsf{wffstl}\ (\Sigma_s, \Sigma_f)\ sl\ tl\ \Longrightarrow\ \mathsf{wff}\ (\Sigma_s, \Sigma_f, \Sigma_p, \Sigma_e)\ (\mathsf{fVar}\ P\ sl\ tl)$$

The rest of this section serves to explain wffstl.

As with the characterization above for universal quantification, we can see this as defining well-formed specialization. Throughout the specialization, the sort list keeps changing: the bounded variables become concrete terms. The bound-variable-replacing function, as outlined in Figure 3.3, can be readily translated into our formalization. Here our case is even simpler because we are replacing one bound variable at a time, and so we do have to take a list to map all bound variables at once. The sort $\mathsf{srpl}\ i\ t\ s$ is the result after substituting the bound variable i with the concrete term t in s. We define a specialization of a list as:

Definition 4.1.2.

$$\begin{aligned}
&\vdash (\forall\, i\ t.\ \mathsf{specsl}\ i\ t\ []\ =\ [])\ \wedge \\
&\quad \forall\, i\ t\ s\ sl.\ \mathsf{specsl}\ i\ t\ (s :: sl)\ =\ \mathsf{srpl}\ i\ t\ s :: \mathsf{specsl}\ (i\ +\ 1)\ t\ sl
\end{aligned}$$

We then define the "specializability" predicate for a sort list and a term list, in below:

Definition 4.1.3.

$$\begin{aligned}
&\mathsf{wfabsap}\ \Sigma\ []\ []\ \stackrel{\mathsf{def}}{=}\ \mathsf{T} \\
&\mathsf{wfabsap}\ \Sigma\ (s :: sl)\ (t :: tl)\ \stackrel{\mathsf{def}}{=} \\
&\quad (\forall\, n_0\ s_0\ st.\ \mathsf{MEM}\ st\ sl\ \wedge\ (n_0, s_0)\ \in\ \mathsf{sfv}\ st\ \Longrightarrow\ \mathsf{sbounds}\ s_0\ =\ \emptyset)\ \wedge\ \mathsf{wft}\ \Sigma\ t\ \wedge \\
&\quad s\ =\ \mathsf{sort_of}\ t\ \wedge\ \mathsf{wfs}\ \Sigma\ s\ \wedge\ \mathsf{wfabsap}\ \Sigma\ (\mathsf{specsl}\ 0\ t\ sl)\ tl \\
&\mathsf{wfabsap}\ \Sigma\ (s :: sl)\ []\ \stackrel{\mathsf{def}}{=}\ \mathsf{F} \\
&\mathsf{wfabsap}\ \Sigma\ []\ (t :: tl)\ \stackrel{\mathsf{def}}{=}\ \mathsf{F}
\end{aligned}$$

The first parameter it takes is the function symbol signature. This is because it has to check that all the terms for specialization are well-formed, and right before being specialized, the sort on the top level of the quantification must be checked for well-formedness as well. This function

inductively checks that at each step, we are specializing a sort list capturing a well-formed quantification with a well-formed term.

While not every item in sl is required to be well-formed, their selection is not arbitrary. It should be the sorts $s_1, \cdots, s_n$ in the quantification list of a well-formed formula $\forall\, s_1\, \forall \cdots \forall\, s_n.\, \phi'$, constructed by abstracting a variable list from a formula ϕ. As the objects we abstract away are variables, the most natural approach to obtain such a list is through iterated abstraction from a variable list. Of course, we stipulate that the variable list subject to abstraction must exhibit well-formed dependency. We established a general predicate that determines whether a list can be abstracted from a formula, characterized as:

$$\begin{aligned}
\vdash\ & (\mathsf{wfvl}\ \Sigma\ []\ f \iff \mathsf{T}) \wedge \\
& (\mathsf{wfvl}\ \Sigma\ (h :: t)\ f \iff \\
& \quad \mathsf{wfvl}\ \Sigma\ t\ f \wedge \mathsf{wfs}\ \Sigma\ (\mathsf{SND}\ h) \wedge h \notin \mathsf{fVslfv}\ f \wedge \\
& \quad \forall\, n\ s.\, (n, s) \in \mathsf{ffv}\ (\mathsf{mk_FALLL}\ t\ f) \Rightarrow h \notin \mathsf{sfv}\ s)
\end{aligned}$$

where $\mathsf{mk\_FALLL}$ simply repeats $\forall_{\mathsf{mk}}$. Then we can formally prove the statement $\mathsf{wff}\ \Sigma\ f \wedge \mathsf{wfvl}\ \Sigma_1\ vl\ f \Rightarrow \mathsf{wff}\ \Sigma\ (\mathsf{mk\_FALLL}\ vl\ f)$,

meaning abstracting a well-formed variable list, as defined above, from a well-formed formula, will give a well-formed formula.

To convert an abstractable variable list into a sort list, we employ a function that abstracts a single variable from a sort list:

Definition 4.1.4.

$$\begin{aligned}
& \mathsf{abssl}\ (n, s)\ i\ [] \stackrel{\mathsf{def}}{=} [] \\
& \mathsf{abssl}\ (n, s)\ i\ (h :: t) \stackrel{\mathsf{def}}{=} \mathsf{sabs}\ (n, s)\ i\ h :: \mathsf{abssl}\ (n, s)\ (i + 1)\ t
\end{aligned}$$

where $\mathsf{sabs}\ (n, s)\ i\ st$ replaces each occurrence of the variable (n, s) in st with $\mathsf{Bound}\ i$. The function $\mathsf{vl2sl}$ that does the conversion can be characterized as

$$\vdash \mathsf{vl2sl}\ (v :: vl) = \mathsf{SND}\ v :: \mathsf{abssl}\ v\ 0\ (\mathsf{vl2sl}\ vl)$$

Any sort list obtained this way from a well-formed variable list would have no outstanding index, i.e. indices that appear in the n-th item of the list are no more than $n - 1$.

Conversely, we can create from a list of sorts containing bound variables a list of terms by providing a list of names. This is done by the function $\mathsf{sl2vl}$, taking a string list and a sort list.

This is effectively done by specializing a sort list, calling the function specsl defined before.

$$\mathsf{sl2vl}\ []\ []\ \stackrel{\mathsf{def}}{=}\ []$$
$$\mathsf{sl2vl}\ (n :: nl)\ (s :: sl)\ \stackrel{\mathsf{def}}{=}\ (n, s) :: \mathsf{sl2vl}\ nl\ (\mathsf{specsl}\ 0\ (\mathsf{Var}\ n\ s)\ sl)$$

Of course, a successful application of $\mathsf{sl2vl}$ requires the two lists to have the same length. If the names nl are all distinct and do not overlap with any name that appears in sl, then we have $\mathsf{vl2sl}\ (\mathsf{sl2vl}\ nl\ sl)\ =\ sl$. The second condition is indeed necessary: Consider taking the name list $[A, f]$ and the sort list $[\mathsf{set}, A \to B]$, then the re-abstraction after the substitution gives $[\mathsf{set}, 0 \to B]$ instead of the original list.

We want to collect the variables remaining after the abstraction since the well-formed rule for quantification requires avoiding them. To ease the collecting process, we impose a further condition on their names:

$$\vdash \mathsf{okvnames}\ vl \iff \forall\, m\ n.\ m\ <\ n\ \wedge\ n\ <\ \mathsf{LENGTH}\ vl\ \Rightarrow\ \mathsf{EL}\ n\ vl\ \notin\ \mathsf{sfv}\ (\mathsf{SND}\ (\mathsf{EL}\ m\ vl))$$

This condition says a variable to be abstracted away never has another chance to appear again and remain as a free variable in the output list.

4.1.2 Example. The names in $[B, f : A \to B]$ are ok, but those in $[f : A \to B, A]$ are not. This is because the variable A is abstracted away but then appears again freely in the sort of f.

For the elimination proof, we want to realize formula variable renaming as formula variable instantiation. We require formulas of form

$$\forall_{\mathsf{mk}}\ n_1\ s_1\ \forall_{\mathsf{mk}}\ \cdots\ \forall_{\mathsf{mk}}\ n_k\ s_k\ \mathcal{F}[(n_1, s_1), \cdots, (n_k, s_k)](\mathsf{Var}\ n_1\ s_1, \cdots, \mathsf{Var}\ n_k\ s_k)$$

to be well-formed. During the proof, there is a step for substituting all the variables (n_i, s_i), for $1 \le i \le k$, uniformly by the bound variables at once, by using a map sending a variable to an index. We need all the variables to be distinct to make this map. This is made easy by taking all the variable names to be distinct.

We want to consider the sort list obtained by iterating abstraction of a variable list. It is not necessary to concern ourselves with the specific formula from which this list is abstracted. Such validation is conducted prior to us receiving a map and conducting the instantiation check. Therefore, our sole requirement is for this list to be abstractable from a formula without any free variables whatsoever. We can use $\bot$ as this body formula. In total, a well-formed sort list-term list application pair is defined as:

Definition 4.1.5.

$$\begin{aligned}&\mathsf{wffstl}\ \Sigma\ sl\ tl \stackrel{\mathsf{def}}{=}\\ &\quad \mathsf{wfabsap}\ \Sigma\ sl\ tl\ \wedge\\ &\quad \exists\, vl.\ \mathsf{wfvl}\ \Sigma\ vl\ \bot\ \wedge\ \mathsf{vl2sl}\ vl\ =\ sl\ \wedge\ \mathsf{ALL\_DISTINCT}\ vl\ \wedge\ \mathsf{okvnames}\ vl\end{aligned}$$

Apart from the abstractability and specializability, the two extra conditions are due to the discussion above. Certainly, once we have a well-formed variable list, we can rename them to fulfil the two extra conditions as well. The two final conjuncts of the definition (under the existential quantifier) are redundant, but make subsequent proofs a bit easier.

4.1.3 Preservation of Well-Formedness

The definition of syntax well-formedness above is intricate and susceptible to fragility. We do not want our kernel's operations to break any of these conditions. Therefore, we formalized proofs showing well-formedness is preserved by the operations we use to define our proof rules. The two most important such theorems are those on term and formula variable instantiations.

Term Variable Instantiation

The eligibility of the instantiation map is checked before carrying out the instantiation. The restrictions on a term map are completeness, consistency and terms in the codomain all being well-formed (under a given signature Σ_f). These three are combined into a single predicate $\mathsf{wfvmap}\ \Sigma_f$. The formalized theorem, expressing the preservation of well-formedness by instantiating with such a map, looks like:

$$\begin{aligned}&\vdash (\forall\, fsym.\\ &\quad \mathsf{isfsym}\ \Sigma_f\ fsym\ \Rightarrow\\ &\quad \mathsf{sfv}\ (\mathsf{fsymout}\ \Sigma_f\ fsym)\ \subseteq\ \bigcup\ \{\ \mathsf{tfv}\ (\mathsf{Var}\ n\ s)\ |\ \mathsf{MEM}\ (n, s)\ (\mathsf{fsymin}\ \Sigma_f\ fsym)\ \}\ )\ \Rightarrow\\ &\forall\, \theta.\ \mathsf{wff}\ (\Sigma_s, \Sigma_f, \Sigma_p, \Sigma_e)\ f\ \wedge\ \mathsf{wfvmap}\ (\Sigma_s, \Sigma_f)\ \theta\ \Rightarrow\ \mathsf{wff}\ (\Sigma_s, \Sigma_f, \Sigma_p, \Sigma_e)\ (\mathsf{in_f}\ \theta\ f)\end{aligned}$$

The assumption on the function signature says the output term is not of a sort that contains a variable that never appears in the argument list. For instance, we do not want function symbols to take an input of sort $A \to B$ and give a term $A \to X$. The proof is by inducting on well-formedness clauses, with three interesting cases.

Predicates Given a predicate of form $\mathsf{Pred}\ P\ l$ under the signature Σ_p storing P with the variable list l_0, the well-formedness for $\mathsf{Pred}\ P\ l$ means that l_0 is matchable to l. The preservation of well-formedness under instantiation means l_0 is still matchable to $\mathsf{MAP}\ (\mathsf{in_t}\ \theta)\ l$. By Theorem 4.1.1, we realize the l as an instantiation of l_0. Then $\mathsf{MAP}\ (\mathsf{in_t}\ \theta)\ l$ is turned

into an iterated instantiation, and hence an instantiation of l_0 by Theorem 4.1.1. Hence Theorem 4.1.1 and Theorem 4.1.2 prove this case.

Universal Quantifier Assuming f is well-formed and for every $\mathsf{wfvmap}\ \theta_0$, we have $\mathsf{in_f}\ \theta_0\ f$ is well-formed, we want $\mathsf{in_f}\ \theta\ (\forall_{\mathsf{mk}}\ n\ s\ f)$ to be well-formed for a given $\mathsf{wfvmap}\ \theta$. We show this by constructing $\mathsf{in_f}\ \theta\ (\forall_{\mathsf{mk}}\ n\ s\ f)$ as an application of $\forall_{\mathsf{mk}}\ n\ s$ with an eligible variable (n, s) from a well-formed formula and conclude with the well-formedness clause for quantification. This is a tedious but typical manipulation of de Brujin indices.

Formula Variables The most complicated case is on formula variables, requiring a proof of $\mathsf{wffstl}\ \Sigma\ (\mathsf{MAP}\ (\mathsf{in_s}\ \theta)\ sl)\ (\mathsf{MAP}\ (\mathsf{in_t}\ \theta)\ tl)$ from $\mathsf{wffstl}\ \Sigma\ sl\ tl$. There are two checks: By induction on tl, we can check $\mathsf{wfabsap}\ \Sigma\ (\mathsf{MAP}\ (\mathsf{in_s}\ \theta)\ sl)\ (\mathsf{MAP}\ (\mathsf{in_t}\ \theta)\ tl)$ from $\mathsf{wfabsap}\ \Sigma\ sl\ tl$. For the other check, given vl is a well-formed abstraction list with the extra two conditions, abstracting into sl, we build a list with the same condition such that it abstracts into $\mathsf{MAP}\ (\mathsf{in_s}\ \theta)\ sl$. This is done by applying the $\mathsf{sl2vl}$ function on $\mathsf{MAP}\ (\mathsf{in_s}\ \theta)\ sl$. Our task is hence provide a proper list of names. Such a list is chosen by avoiding all the names in vl, the domain of θ, and the variables in the instantiation $\mathsf{MAP}\ (\mathsf{in_s}\ \theta)\ sl$.

Formula Variable Instantiation

The formalized instantiation function is inductive on constructors. We use the Greek letter δ for formula variable instantiation maps.

$$
\begin{aligned}
\vdash\ & (\forall\,\delta.\ \mathsf{in_{fV}}\ \delta\ \bot\ =\ \bot)\ \wedge\ (\forall\,\delta\ p\ tl.\ \mathsf{in_{fV}}\ \delta\ (\mathsf{Pred}\ p\ tl)\ =\ \mathsf{Pred}\ p\ tl)\ \wedge \\
& (\forall\,\delta\ f_1\ f_2.\ \mathsf{in_{fV}}\ \delta\ (\mathsf{IMP}\ f_1\ f_2)\ =\ \mathsf{IMP}\ (\mathsf{in_{fV}}\ \delta\ f_1)\ (\mathsf{in_{fV}}\ \delta\ f_2))\ \wedge \\
& (\forall\,\delta\ s\ \phi.\ \mathsf{in_{fV}}\ \delta\ (\mathsf{FALL}\ s\ \phi)\ =\ \mathsf{FALL}\ s\ (\mathsf{in_{fV}}\ \delta\ \phi))\ \wedge \\
& \forall\,\delta\ P\ sl\ tl. \\
& \quad \mathsf{in_{fV}}\ \delta\ (\mathsf{fVar}\ P\ sl\ tl)\ = \\
& \quad \texttt{if}\ (P, sl)\ \in\ \mathsf{FDOM}\ \delta\ \texttt{then}\ \mathsf{fprpl}\ (\mathsf{mk\_bmap}\ (\mathsf{REVERSE}\ tl))\ (\delta\ '\ (P, sl)) \\
& \quad \texttt{else}\ \mathsf{fVar}\ P\ sl\ tl
\end{aligned}
$$

The function fprpl is the same as the one in implementation. The map δ is of type $\texttt{string} \times \texttt{sort list} \mapsto \texttt{form}$, recording where the formula variable (P, sl) goes. We require for each $(P, sl)\ \in\ \mathsf{FDOM}\ \delta$, the combined formula $\mathsf{FALLL}\ sl\ (\delta\ '\ (P, sl))$ is well-formed (under the signature Σ), written $\mathsf{wffVmap}\ \Sigma\ \delta$. When it sees a $\mathsf{fVar}\ P\ sl\ tl$ with $(P, sl)\ \in\ \mathsf{FDOM}\ \delta$, the function fprpl replaces all the bounded variables into concrete terms in parallel. This is effectively specializing $\mathsf{FALL}\ sl\ (\delta\ '\ (P, sl))$ with the term list tl. The inductive proof of the well-formedness of formula variable instantiation has only two interesting cases: that for formula variables and universal quantification.

Instantiating $\mathsf{fVar}\ P\ sl\ tl$ amounts to formally realizing it as an iterated specialization, which can be obtained by proving the preservation of well-formedness on a single specialization, and then induction on the length of the abstraction list.

The quantification case is more involved. We just give an example to illustrate the renaming procedure involved in this proof.

4.1.3 Example. Consider the formula $\forall A.\ \mathcal{F}[\mathsf{set}](A)$, and the predicate stating a map $f : A \to B$ factors through this object, then we want to instantiate with

$$\lambda X.\ \exists g : A \to X\ h : X \to B.\ h \circ g = f$$

Then after the instantiation, the variable A is no longer abstractable, since it appears in the sort of the free variable f. We thereby rename the A in the map into A' instead, so the instantiation gives:

$$\exists g : A' \to A\ h : A \to B.\ h \circ g = f$$

Abstraction will give a formula that is, via de Brujin index, equal to

$$\forall K.\ \exists g : A' \to K\ h : K \to B.\ h \circ g = f$$

Then we can rename A' back to A, yields:

$$\forall K.\ \exists g : A \to K\ h : K \to B.\ h \circ g = f$$

which is well-formed.

The whole procedure is summarized into chained equalities:

$$\begin{aligned}
&\mathsf{FALL}\ s\ (\mathsf{in}_{\mathsf{fV}}\ \delta\ (\mathsf{abst}\ (n, s)\ f)) \\
=\ &\mathsf{FALL}\ s \\
&(\mathsf{frename}\ (nn, s)\ n\ (\mathsf{abst}\ (n, s)\ (\mathsf{in}_{\mathsf{fV}}\ (\mathsf{fVmap_rename}\ (n, s)\ nn\ \delta)\ f))) \\
=\ &\mathsf{frename}\ (nn, s)\ n \\
&(\mathsf{FALL}\ s\ (\mathsf{abst}\ (n, s)\ (\mathsf{in}_{\mathsf{fV}}\ (\mathsf{fVmap_rename}\ (n, s)\ nn\ \delta)\ f)))
\end{aligned}$$

We reduce the left-hand side to the right-hand side. As f is well-formed, its instantiation is well-formed by the inductive hypothesis, the well-formedness is preserved by the abstraction and finally by the renaming. This completes the induction.

4.2 Proof Rules

Recall that a theorem in our formalization is a 3-tuple (Γ, A, ϕ) consisting of Γ, a HOL set of type `string × sort → bool`, A, a HOL set of assumptions of type `form → bool` and ϕ, a formula. We use functions cont, assum and concl to extract the three slots. The set of formula variables in a theorem is collected with the function thfVars. A proof is a list of theorems, where each step is obtainable from some previous step by applying some primitive rules. We formalize the predicate Pf Σ *axs pf* to mean the list *pf* is a proof under signature Σ from the axioms *axs*. We write PfDrv Σ *axs th* to signify that there is a proof that ends with *th*, meaning this theorem is derivable. Rules for connectives are straightforward. For instance, the rule for modens ponens says:

$$\begin{aligned}\vdash\ &\mathsf{Pf}\ \Sigma\ axs\ pf_1\ \wedge\ \mathsf{Pf}\ \Sigma\ axs\ pf_2\ \wedge\ \mathsf{MEM}\ (\Gamma_1, A_1, \mathsf{IMP}\ f_1\ f_2)\ pf_1\ \wedge\\ &\mathsf{MEM}\ (\Gamma_2, A_2, f_1)\ pf_2\ \Rightarrow\\ &\mathsf{Pf}\ \Sigma\ axs\ (pf_1\ +\!\!+\ pf_2\ +\!\!+\ [(\Gamma_1\ \cup\ \Gamma_2, A_1\ \cup\ A_2, f_2)])\end{aligned}$$

We exclude the less interesting cases and only present those that are necessary to discuss before explaining our elimination proof.

Congruence As in our design, our primitive congruence rule is only on formula variables. It considers formula variables with a non-empty list of arguments with sorts $s_1, \cdots, s_n$, recorded in the list *sl*. The congruence rule takes a list of proofs that involve an item with its conclusion an equality. These equations are where the congruence would take place. The list is recorded by a function *Pfs* of type `num → ((string × sort → bool) × (form → bool) × form) list`, such that for each n < LENGTH *sl*, the proof *Pfs* gives us a theorem proving an equation between terms of sort EL *n sl*. The theorems on equality are captured by a function *eqths* from a natural number to a theorem. The function GENLIST simply takes a natural number n and a function, and produces a list by taking the output of the function up to n. An application of this rule produces a big proof by attaching all the input proofs together, and attaching a theorem with its context and assumption set as the union of the counterpart from the equality theorems, and has conclusion IFF (fVar *P sl* tl_1) (fVar *P sl* tl_2), produced by a function fVcong.

The connective IFF is defined as in convention. It is formally presented as:

$$\begin{aligned}
&\vdash sl \neq [] \wedge \mathsf{wffVsl}\ (\mathsf{FST2}\ \Sigma)\ sl \wedge \\
&\quad (\forall n.\ n < \mathsf{LENGTH}\ sl \Rightarrow \\
&\qquad \mathsf{is_EQ}\ (\mathsf{concl}\ (eqths\ n)) \wedge \mathsf{Pf}\ \Sigma\ axs\ (Pfs\ n) \wedge \mathsf{MEM}\ (eqths\ n)\ (Pfs\ n) \wedge \\
&\qquad \mathsf{sort_of}\ (\mathsf{Leq}\ (\mathsf{concl}\ (eqths\ n))) = \mathsf{EL}\ n\ sl) \wedge \\
&\quad (\forall s.\ \mathsf{MEM}\ s\ sl \Rightarrow \mathsf{wfs}\ (\mathsf{FST2}\ \Sigma)\ s) \Rightarrow \\
&\quad \mathsf{Pf}\ \Sigma\ axs \\
&\qquad (\mathsf{FLAT}\ (\mathsf{GENLIST}\ (\mathsf{LENGTH}\ sl - 1)\ Pfs) \Cat [\mathsf{fVcong}\ (\mathsf{GENLIST}\ (\mathsf{LENGTH}\ sl - 1)\ eqths)\ P\ sl])
\end{aligned}$$

where FST2 takes the first two components of a 4-tuple. Note that all the sorts of the terms to be equated are all well-formed. In particular, the formula variable it outputs will have a sort list without any bound variable. By the discussion in Section 2.3.2, it is sufficient to consider lists of well-formed sorts only.

Instantiation In the formalization, for the term instantiation θ to be applied to the theorem (Γ, A, ϕ), we include an extra condition that the domain of the well-formed map θ to cover the whole context Γ. Such a condition shortens the proof and is clearly reasonable because any consistent map could be expanded to cover every variable we want to include. The theorem $\mathsf{in}_\mathsf{t}^\mathsf{th}\ \theta\ (\Gamma, A, f)$ is the result of the instantiation. The assumption set and conclusion are obtained by applying the formula instantiation. The context is the set $\{\ \mathsf{tfv}\ (\theta\ '\ v) \mid v \mid v \in \mathsf{FDOM}\ \theta \cap \Gamma\ \}$. The rule for term instantiation is then presented as:

$$\begin{aligned}
&\vdash \mathsf{Pf}\ \Sigma\ axs\ pf \wedge \mathsf{MEM}\ th\ pf \wedge \mathsf{wfvmap}\ (\mathsf{FST2}\ \Sigma)\ \theta \wedge \mathsf{cont}\ th \subseteq \mathsf{FDOM}\ \theta \Rightarrow \\
&\quad \mathsf{Pf}\ \Sigma\ axs\ (pf \Cat [\mathsf{in}_\mathsf{t}^\mathsf{th}\ \theta\ th])
\end{aligned}$$

The rule for formula instantiation is similar, where the condition $\mathsf{wfvmap}\ (\mathsf{FST}\ \Sigma)\ \theta$ is replaced by $\mathsf{wffVmap}\ \Sigma\ \delta$ and $\mathsf{cont}\ th \subseteq \mathsf{FDOM}\ \theta$ is replaced by $\mathsf{thfVars}\ th \subseteq \mathsf{FDOM}\ \delta$. Again, the instantiated theorem's assumptions and conclusion are the image of the instantiation map. The context is $\Gamma \cup \{\ \mathsf{ffv}\ (\delta\ '\ fv) \mid fv \in \mathsf{thfVars}\ (\Gamma, A, f)\ \}$ It is evident that instantiating formula variables can only expand the context. We will call the instantiated theorem $\mathsf{in}_\mathsf{f}^\mathsf{th}\ \delta\ (\Gamma, A, f)$.

Universal Quantifier The restriction imposed by the generalization rule is verified by ensuring that the variable (x, s) does not appear in the set of prohibited variables. These prohibited variables are compiled as a union of variables collected using our previously defined Uof. We refer to this union as $\mathsf{genavds}\ (\Gamma, A, f)$. Furthermore, generalization cannot introduce new variables. If we intend to abstract a variable whose sort includes an additional free variable, we must first apply the context expansion rule to include it. Once these conditions are met, the generalization rule removes the free variable from the context and abstracts it in the

conclusion while leaving the assumptions intact. It produces the theorem $\mathsf{gen}\ (x, s)\ (\Gamma, A, f)$.

$$\begin{aligned}&\vdash \mathsf{Pf}\ \Sigma\ axs\ pf\ \wedge\ \mathsf{MEM}\ (\Gamma, A, f)\ pf\ \wedge\ \mathsf{wfs}\ (\mathsf{FST2}\ \Sigma)\ s\ \wedge\ \mathsf{sfv}\ s\ \subseteq\ \Gamma\ \wedge\\ &\quad (x, s)\ \notin\ \mathsf{genavds}\ (\Gamma, A, f)\ \Rightarrow\\ &\quad \mathsf{Pf}\ \Sigma\ axs\ (pf\ +\!\!+\ [\mathsf{gen}\ (x, s)\ (\Gamma, A, f)])\end{aligned}$$

As for specialization, it checks the term is of the correct sort only:

$$\begin{aligned}&\vdash \mathsf{Pf}\ \Sigma\ axs\ pf\ \wedge\ \mathsf{MEM}\ (\Gamma, A, \mathsf{FALL}\ s\ f)\ pf\ \wedge\ \mathsf{wft}\ (\mathsf{FST2}\ \Sigma)\ t\ \wedge\ \mathsf{sort\_of}\ t\ =\ s\ \Rightarrow\\ &\quad \mathsf{Pf}\ \Sigma\ axs\ (pf\ +\!\!+\ [\mathsf{spec}\ t\ (\Gamma, A, \mathsf{FALL}\ s\ f)])\end{aligned}$$

4.3 Elimination of Formula Variables

While formula variables provide a convenient way to work with our system, they are essentially higher-order variables. In systems like HOL, if we were to create a datatype for "sets", a formula variable $\mathcal{F}[A : \mathsf{set}]$ would correspond to a HOL term of type $\mathsf{set} \to \mathsf{bool}$. In a dependently typed system like Lean, which has types for sets and dependent types for functions, a formula variable $\mathcal{F}[A : \mathsf{set},\ B : \mathsf{set},\ f : A \to B]$ would capture a term of type $\Pi_{A:\mathsf{set}}\Pi_{B:\mathsf{set}}\Pi_{f:A\to B}\mathsf{bool}$. Such terms do not exist in a first-order system. Therefore, it's not immediately clear why we can introduce this type of formula without compromising the first-order nature or the soundness of the system. We will now present evidence that this design is indeed safe by proving that our system is not more powerful than a dependent sorted first-order logic without any formula variables.

Every theorem provable in our system can also be derived in a simpler system without formula variables. We will write Pf_0 instead of Pf for the notion of proofs and derivability in this proof system, described as follows. In Pf_0, we use formulas coupled with contexts to specify its initial axioms. The proof system Pf_0 has the same proof rules as Pf, with only one exception: The formula variable congruence rule, which implies every congruence rule, is replaced by a rule that directly gives any case of congruence. i.e. the congruence rule as in Section 2.3.2 is replaced by the following rule on concrete formulas.

$$\frac{\Gamma_1, A_1 \vdash t_1 = t_1',\ \cdots,\ \Gamma_n, A_n \vdash t_n = t_n'}{\bigcup_{i=1}^n \Gamma_i, \bigcup_{i=1}^n A_i \vdash \phi(t_1, \cdots t_n) \Leftrightarrow \phi(t_1', \cdots t_n')}$$

In what follows, by abuse of terminology, the phrase "concrete formula" may refer to either a formula in Pf-proof system without any formula variables, or a formula in the Pf_0-proof system, where the constructor of formula variables does not exist.

We prove that every proof in Pf corresponds to an "instantiated proof" in Pf_0.

$$\begin{array}{ccc} \text{AX} & \xrightarrow{\mathsf{Pf}_0\text{-counterpart of fVar-inst with concrete formulas}} & \text{AX'} \\ {\scriptstyle\text{derivation}}\downarrow & & \downarrow{\scriptstyle\text{derivation}'} \\ \Gamma, A \vdash \phi & \xrightarrow{\mathsf{Pf}_0\text{-counterpart of fVar-inst with concrete formulas}} & \Gamma', A' \vdash \phi' \end{array}$$

The set AX on the top left corner is the original axioms, and AX' is all its instances as theorems, obtained by first applying the Ax-rule and then instantiating the formula variables. The proof in the left half of the picture happens entirely in Pf, and that on the right happens entirely in Pf_0. We will prove that if a theorem $\Gamma, A \vdash \phi$ is derivable in Pf, then every of its concrete instances is derivable in Pf_0. We obtain it by proving a strengthened result saying that under certain well-formedness assumptions, after adjusting all distinct formula variables into distinct names, instantiating variables and then all the formula variables into concrete formulas, the resultant theorem has a Pf_0-proof.

In the following description, as before, we will use θ and δ for instantiation maps for term and formula variables respectively. By abuse of notation, we will denote an instantiation on a formula or a theorem by putting the instantiation maps into a bracket in front of it. For example, we write $(\delta; \theta)\phi$ to mean instantiate the formula ϕ with variable map θ first, and then instantiate with formula variable map δ.

A concrete theorem from Pf is always obtained by taking a proved theorem, instantiating the variables, and then, if any, instantiating the formula variables all into concrete formulas. We need to notice that if we swap the order by instantiating formulas first and then the term variables, we are not able to get all of the derivable instances. This is because predicates have to be "sort-checked" against their signatures. For instance, before instantiating the B in $\mathcal{P}[\mathsf{mem}(A), \mathsf{mem}(B)](a, b)$ to be $\mathsf{Pow}(A)$, we cannot instantiate the predicate $\mathcal{F}$ into the membership predicate IN. If the formula instantiation δ covers every formula variable in a theorem $(\theta)th$, and has only concrete formulas in its codomain, then instantiating with it will produce a concrete theorem. One might imagine that to prove every instantiation $(\delta; \theta)th$ of a theorem th has a Pf_0-proof. However, it is possible for two formula variables to be identified upon term instantiation if they have the same name but take arguments of different sorts. For instance, the formula $\mathcal{F}[\mathsf{mem}(A)](a)$ and $\mathcal{F}[\mathsf{mem}(B)](b)$ will become the same if we instantiate $B \mapsto A$. To see how it obscures the proof: when proving the formula variable instantiation case, the aim is to complete the dashed arrows in a commutative square:

$$\begin{array}{ccccc} th & \overset{\theta'}{\dashrightarrow} & (\theta')th & \overset{\delta'}{\dashrightarrow} & (\delta'; \theta')th \\ \downarrow{\scriptstyle\delta_0} & & & & \vdots \\ (\delta_0)th & \xrightarrow{\theta} & (\theta; \delta_0)th & \xrightarrow{\delta} & (\delta; \theta; \delta_0)th \end{array}$$

The inductive hypothesis says: Assume $\mathsf{PfDrv}\ \Sigma\ ax\ th$, then for each variable map θ' and formula variable map δ', sending all the formula variables in $(\theta')th$ into a concrete formula, the theorem $(\delta';\theta')th$ has a Pf_0-proof. The goal is to prove for every formula instantiation $(\delta_0)th$, each eligible two-step instantiation with θ and δ is also provable in Pf_0. In the diagram above, the fixed maps are drawn in solid arrows. The proofs amount to finding out the dashed arrows. We want maps θ' and δ' to be used to specialize the inductive hypothesis, producing a theorem derivable in Pf_0 by the inductive hypothesis. Then we apply a proof step in Pf_0 to prove the desired theorem $(\delta;\theta;\delta_0)th$ in Pf_0.

In terms of formulas, we require for each formula f appearing in th, the conclusion, or a hypothesis, the commutativity of the diagram:

$$\begin{array}{ccccc}
\phi & \overset{\theta'}{\dashrightarrow} & (\theta')\phi & \overset{\delta'}{\dashrightarrow} & (\delta';\theta')\phi \\
\downarrow{\scriptstyle \delta_0} & & & & \vdots \\
(\delta_0)\phi & \xrightarrow{\theta} & (\theta;\delta_0)\phi & \xrightarrow{\delta} & (\delta;\theta;\delta_0)\phi
\end{array}$$

Ideally, the vertical dashed arrow should be the identity. If so, then we only have to modify the context by adding in some variables.

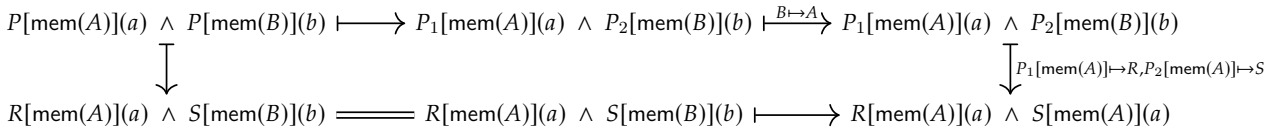
The goal can be achieved by effectively swapping θ and δ_0. This will require a modification of δ_0 according to the term instantiation θ into another map δ_0', which satisfies $(\delta_0';\theta)\phi = (\theta;\delta_0)\phi$ Once this is achieved, we end up with a term instantiation followed by two consecutive formula instantiations. We can then compose the two formula variable instantiations. This allows us to take the composition of δ_0' followed by δ as the δ' required.

Back to the aforementioned example $\mathcal{F}[\mathsf{mem}(A)](a) \wedge \mathcal{F}[\mathsf{mem}(B)](b)$, we consider the case when the two distinct formula variables on members of A and of B, with the same name $\mathcal{F}$, are renamed into R and S respectively by δ_0, where θ is $[B \mapsto A]$. There is no formula map that fits in the dash in the diagram below since $\mathcal{F}[\mathsf{mem}(A)]$ and $\mathcal{F}[\mathsf{mem}(B)]$ are identified after instantiation, and hence do not have any chance to eventually become different formula variables $R[\mathsf{mem}(B)]$ and $S[\mathsf{mem}(B)]$.

$$\begin{array}{ccc}
\mathcal{F}[\mathsf{mem}(A)](a) \wedge \mathcal{F}[\mathsf{mem}(B)](b) & \longrightarrow & \mathcal{F}[\mathsf{mem}(A)](a) \wedge \mathcal{F}[\mathsf{mem}(A)](b) \\
\downarrow & & \vdots \\
R[\mathsf{mem}(A)](a) \wedge S[\mathsf{mem}(B)](b) & \longrightarrow & R[\mathsf{mem}(A)](a) \wedge S[\mathsf{mem}(A)](a)
\end{array}$$

To avoid this problem, we ensure different formula variables are not identified by term variable instantiation by renaming all of them with distinct names. A map capturing such a renaming is encoded by a finite map of type $(\texttt{string} \times \texttt{sort})\ \texttt{list} \mapsto \texttt{string}$. We use the ones that are injections on a set s, which we call a *uniquenification*. We will always use μ for an

unique-ification map. Again, by abuse of notation, we write $(\mu)\phi$ or $(\mu)th$ for unique-ification of a formula or a theorem. Unique-ification ensures that formula variables are not accidentally identified during term instantiation, preserving the distinctness of formula variables. With the unique-ification, our problematic example can be resolved by following the flow below. As $\mathcal{F}_1$ and $\mathcal{F}_2$ now have different names, they can never be identified and can be sent to different places.



With this adjustment, our theorem for the formula variable elimination is formally proved as:

Theorem 4.3.1.

$$
\begin{aligned}
\vdash\ & \mathsf{wfsigaxs}\ \Sigma\ axs\ \wedge\ \mathsf{wfsigaths}\ \Sigma\ aths\ \wedge\ \mathsf{Pf}\ \Sigma\ axs\ pf\ \wedge \\
& \mathsf{Uof}\ (\mathsf{UCIth}\ \Sigma)\ (\mathsf{IMAGE}\ \mathsf{ax2th}\ axs)\ \subseteq\ aths\ \wedge\ \mathsf{MEM}\ th\ pf\ \wedge\ \mathsf{wfvmap}\ (\mathsf{FST}\ \Sigma)\ \theta\ \wedge \\
& \mathsf{wfcfVmap}\ \Sigma\ \delta\ \wedge\ \mathsf{thfVars}\ ((\theta;\mu)th)\ \subseteq\ \mathsf{FDOM}\ \delta\ \wedge\ \mathsf{cont}\ th\ \subseteq\ \mathsf{FDOM}\ \theta\ \wedge \\
& \mathsf{uniqifn}\ \mu\ (\mathsf{thfVars}\ th)\ \Rightarrow \\
& \mathsf{Pf}_0\mathsf{Drv}\ \Sigma\ aths\ ((\delta;\theta;\mu)th)
\end{aligned}
$$

There are four well-formedness assumptions in this theorem. Two are on the signature and two are on the instantiation maps. Proving this theorem amounts to completing the commutative squares similar to the above ones, but extended by an initial step of unique-ification. The assumption $\mathsf{thfVars}\ ((\theta;\mu)th) \subseteq \mathsf{FDOM}\ \delta$ says the domain of the final formula variable map covers all the formula variables in the theorem to be instantiated. As this map is concrete, this assumption makes sure we always end up with a concrete theorem. We use subset inclusion instead of equality because it is required to proceed the induction. In each case, we can easily restrict the domains to make it an equality. This allows us to assume the domains of the maps cover precisely what is required during the following discussion about our main proof.

The induction steps on equality rules, the context expansion and the assume rule are trivial and the ones on logical operators are very straightforward: The top arrows can just be taken to be identical to the bottom arrows. The other ones are more interesting. Throughout the proof, the manipulation mainly happens on the level of formulas. The modification of context is tedious but routine. The readability of relevant arguments highly relies on the usage of Uof, with its first argument varies among all the variable-collecting functions we have seen.

We prove the interesting inductive cases. As the modifications on contexts are rather straightforward, we will focus on explaining the story on the level of formulas.

Term Variable Instantiation

The inductive hypothesis says that every possible three-stage-instantiation of th, as in the dashed maps on the top row, has a Pf_0-proof. We then construct an Pf_0-proof of $(\delta; \theta; \mu; \theta_0)th$.

$$\begin{array}{ccccccc} th & \overset{\mu'}{\dashrightarrow} & (\mu')th & \overset{\theta'}{\dashrightarrow} & (\theta'; \mu')th & \overset{\delta'}{\dashrightarrow} & (\delta'; \theta'; \mu')th \\ \downarrow{\scriptstyle \theta_0} & & & & & & \vdots \\ (\theta_0)th & \xrightarrow{\mu} & (\mu; \theta_0)th & \xrightarrow{\theta} & (\theta; \mu; \theta_0)th & \xrightarrow{\delta} & (\delta; \theta; \mu; \theta_0)th \end{array}$$

The variable instantiation θ can indeed identify some formula variables, but it does not matter: we only require the formula variables that are not identified at the end to be kept separated in the top row instantiation. If they are identified at the bottom, no extra care is required to keep them separated on the top. But as the inductive hypothesis requires the first map μ' to be a unique-ification, actually they are kept separated along all the maps.

In this diagram, no formula instantiation happens until the instantiation δ and δ', so the formulas in $(\delta'; \theta'; \mu')th$ and $(\delta; \theta; \mu; \theta_0)th$ will be in the same pattern and their formula variables are in a one-to-one correspondence. To identify their sort lists, we take θ' to be the composition $\mathsf{o\_vmap}\ \theta\ \theta_0$. Then the formula variables in the two theorems only differ with a renaming. Therefore, the formula instantiation maps δ and δ' only differ in a renaming of their domains. In other words, the effect of instantiation δ' is a renaming followed by δ. We only have to recover this renaming, amounts to find the vertical renaming map σ in:

$$\begin{array}{ccccc} \psi & \xrightarrow{\mu'} & (\mu')\psi & \xrightarrow{\theta'} & (\theta'; \mu')\psi \\ \downarrow{\scriptstyle \theta_0} & & & & \vdots{\scriptstyle \sigma} \\ (\theta_0)\psi & \xrightarrow{\mu} & (\mu; \theta_0)\psi & \xrightarrow{\theta} & (\theta; \mu; \theta_0)\psi \end{array}$$

Again, to make it clear, we extract the change on a formula variable as the diagram:

$$\begin{array}{ccccc} (P, sl) & \xrightarrow{\mu'} & (\mu'\ '\ (P, sl), sl) & \xrightarrow{\theta'} & (\mu'\ '\ (P, sl), \mathsf{MAP}\ \theta'\ sl) \\ \downarrow{\scriptstyle \theta_0} & & & & \vdots{\scriptstyle \sigma} \\ (P, \mathsf{MAP}\ \theta_0\ sl) & \xrightarrow{\mu} & (\mu\ '\ (P, \mathsf{MAP}\ \theta_0\ sl), \mathsf{MAP}\ \theta_0\ sl) & \xrightarrow{\theta} & (\mu\ '\ (P, \mathsf{MAP}\ \theta_0\ sl), \mathsf{MAP}\ \theta\ (\mathsf{MAP}\ \theta_0\ sl)) \end{array}$$

It is clear what the map σ should do:

- For a formula variable (P', sl'), which is known to be in the form as in the upper right corner of the diagram above, use the fact that μ' is a bijection to find out the original (P, sl) in th.
- Instantiate (P, sl) with θ_0, then (P', sl') will be renamed into $\mu\ '\ (P, \mathsf{MAP}\ \theta_0\ sl)$.

The renaming σ can be recaptured as a formula variable instantiation and we compose it with δ. This gives the map δ' in the original diagram.

Modus Ponens

Upon initial inspection, modus ponens might seem as straightforward as the other rules for connectives. However, it does require additional attention due to a unique feature: While the other logical rules concerning falsity and discharge preserve the presence of proper subformulas in the input theorems, modus ponens eliminates the antecedent of the implication theorem that it takes as an input.

In the formal proof, the inductive hypothesis gives: if $\mathsf{thfVars}\ (\Gamma_1, A_1, \mathsf{IMP}\ \phi\ \psi)$ and $\mathsf{thfVars}\ (\Gamma_2, A_2, \phi)$ both have Pf-proof, and for all feasible instantiation maps μ_1, θ_1, δ_1 and μ_2 θ_2 and δ_2, on the two theorems respectively, the instantiations have a Pf_0-proof. We want to prove any feasible 3-step-instantiation of $(\Gamma_1 \cup \Gamma_2, A_1 \cup A_2, \psi)$, with μ, θ and δ, has a Pf_0-proof.

The problem is that the domain of μ and δ only contains the formula variables from $(\Gamma_1 \cup \Gamma_2, A_1 \cup A_2, \psi)$, but the inductive hypothesis is on theorems involving the ϕ, and asks for a map that includes formula variables from it. We hence adopt a modification on those two maps: The renaming μ will be extended to also include formula variables in ϕ, and to be kept as an injection. The formula map δ will be extended to also include formula variables that originally comes from ϕ, and send them all to $\bot$, so it does not bring any extra free variables. The choice of formulas does not make a difference, because all the instantiated concrete formulas, in our case, all the $\bot$ brought by this instantiation, will disappear after applying modens ponens in Pf_0. The proof from here is straightforward.

Universal Quantifier

Generalization The generalization changes the conclusion and delete the generalized variable from the context of the theorem (Γ, A, ψ). We have a commutative diagram:

$$\begin{array}{ccccccc}
\psi & \xrightarrow{\mu} & (\mu)\psi & \xrightarrow{\theta'} & (\theta';\mu)\psi & \xrightarrow{\delta} & (\delta;\theta';\mu)\psi \\
\downarrow{\scriptstyle \forall_{\mathsf{mk}}\ x\ s} & & & & & & \downarrow{\scriptstyle \forall_{\mathsf{mk}}\ nn\ ((\theta)s)} \\
\forall_{\mathsf{mk}}\ x\ s\ \phi & \xrightarrow{\mu} & \mu(\forall_{\mathsf{mk}}\ x\ s\ \phi) & \xrightarrow{\theta} & (\theta;\mu)(\forall_{\mathsf{mk}}\ x\ s\ \phi) & \xrightarrow{\delta} & (\delta;\theta;\mu)(\forall_{\mathsf{mk}}\ x\ s\ \phi)
\end{array}$$

where the map θ' sends (x, s) to $\mathsf{Var}\ nn\ ((\theta)s)$ and agrees with θ on every other variable, for a new nn with some conditions. The unique-ification and formula map on the top row is the same as the bottom one. We need a renaming process here. To keep the formula maps on the two rows the same, we take the option to rename the variable in the body of the formula, hence the generalization in Pf_0 is on the variable with the new name. We need to avoid the

names in δ and in θ. We also want to avoid the names that already exist in the context. Once the choice of the new name nn avoids these three variable sets, we can prove the equality:

$$(\delta; \theta; \mu)(\mathsf{gen}\ (x, s)\ th) = \mathsf{gen}\ (nn, (\theta)s)\ ((\delta; \theta'; \mu)th)$$

which directly implies our result.

Specialization The proof of the specialization case is quite simple. As expressed in the diagram, deferring the specialization just amounts to specializing with the instantiated term instead.

$$\begin{array}{ccccccc}
\psi & \xrightarrow{\mu} & (\mu)\psi & \xrightarrow{\theta} & (\theta;\mu)\psi & \xrightarrow{\delta} & (\delta;\theta;\mu)\psi \\
\downarrow{\scriptstyle \mathsf{specf}\ t} & & & & & & \downarrow{\scriptstyle \mathsf{specf}\ ((\theta)t)} \\
\mathsf{specf}\ t\ \psi & \xrightarrow{\mu} & (\mu)(\mathsf{specf}\ t\ \psi) & \xrightarrow{\theta} & (\theta;\mu)(\mathsf{specf}\ t\ \psi) & \xrightarrow{\delta} & (\delta;\theta;\mu)(\mathsf{specf}\ t\ \psi)
\end{array}$$

This is due to the three neat equations:

$$\begin{aligned}
(\theta)(\mathsf{spec}\ t\ th) &= \mathsf{spec}\ ((\theta)t)\ ((\theta)th) \\
(\mu)(\mathsf{spec}\ t\ th) &= \mathsf{spec}\ t\ ((\mu)th) \\
(\delta)(\mathsf{spec}\ t\ th) &= \mathsf{spec}\ t\ ((\delta)th)
\end{aligned}$$

under easy well-formedness conditions.

The highlight is that this case is the reason we need term instantiation to be primitive, and do not allow generalization over the sort list of formula variables. Semantically, it is indeed reasonable. But if we allow such instantiation, our induction will get stuck on this step. This is because there might be more formula variables that become instantiable after the specialization. However, there is no obvious way to conclude anything about such newly introduced formula variables from the induction hypothesis.

4.3.1 Example. Consider the (ill-formed) quantified formula variable $\forall B.\ \mathcal{F}[A \to B](f)$. Before we specialize the variable B, it does not make sense to instantiate such a formula. Its codomain is not specified, so we do not know which formula makes sense for such instantiation.

Congruence

Despite the simplicity of this case, we point out that we use choice in our formalization via a step of Skolemization.

The choice happens as follows: the inductive hypothesis gives a list of Pf-theorems, each with a step proving an equation, and their instantiations all admit a Pf_0-proof. We only have the

existence of such Pf_0-proofs, but there may be more than one. Choosing one particular proof is requested by the Pf_0-congruence rule since it requires an explicit list of proofs to be put together. Avoiding such a choice may be possible but is not trivial. It will demand us to strengthen our theorem to be strong enough to give a machinery of building a Pf_0-proof of an instantiation of a theorem from a Pf-proof of the theorem.

Chapter 5

Two Structural Set Theories

In this chapter, we describe the formalization of two structural set theories, namely ETCS [29] and SEAR [7], in DiaToM. With each, we work within a well-pointed boolean topos. This ensures usual mathematical constructions can be performed in both of them. Notably, they both possess fundamental mathematical constructs such as products, coproducts, exponentials, and initial and terminal objects. ETCS incorporates the existence of these constructs as primitive axioms. In contrast, SEAR constructs them manually.

In terms of power, SEAR is stronger than ETCS. ETCS stands out for its simplicity. It can be finitely axiomatized and yet possesses the mathematical strength of HOL, allowing for a significant range of mathematical operations. SEAR, on the other hand, is more complex, as it has two axiom schemata. These schemata enable the proof of highly potent theorems, thus exceeding the expressiveness limitations of HOL.

We will commence with the introduction of ETCS due to its simpler presentation. Most of our explanations and demonstrations will be carried out in SEAR. For the formal presentation of a theorem in this and the next chapter, when the context is obvious and there are no assumptions, we only present the conclusion as a formula.

5.1 Formalizing ETCS

The theory ETCS stands as a classic example of structural set theory. It was developed early by William Lawvere as a pioneering piece of work on formulating set theory with a "categorical" spirit. This happens in the sense of focusing on how objects interact with each other, in contrast to asking "What is in an object". It is achieved by omitting the primitive notion of "membership" and working entirely with the objects, here the sets, and the interaction between them, here the functions. With the presence of the terminal object, which is assumed by the axioms, we can recover a notion of "elements" via a special kind of function: those from the terminal objects. Moreover, as ETCS assumes well-pointedness, imposing two functions to be

equal if they agree on all the elements of the domain, we can mostly just think of working with more conventional versions of set theory.

As already seen in Example 2.1.5, ETCS has two sorts: objects (A, B, ...; a ground sort) and arrows (*e.g.,* $A \to B$), where an arrow sort depends on two object terms. Equality can only hold between arrows. An object is to be considered as a set in the usual sense: an arrow $1 \to X$ is regarded as an element of the set X.

We believe the approach we take is the original design as in Lawvere [29]. The ways to present ETCS are actually not unique. As pointed out in [10], alternatively, we can present ETCS in one sort, or two sorts without dependency as well. In these cases, information that is naturally captured by sort judgements must be captured using predicates such as "being a set" and "being an element". For such an approach, we are required to impose equality on sets as well, as it is the only obvious way to verify the domain and codomain condition when treating partial function symbols, e.g., composition. Such an approach, however, is employed in Lawvere's later paper on ETCC. Presenting ETCS with dependent sorts does not yield any obvious pain points. This is evident from the smoothness of the formalization of Lawvere's original paper, where all 6 statements labelled as "theorems" admit a direct mechanical translation.

Despite its elegance in presenting some branches of mathematics, ETCS is not considered to be perfect, or even better than the material approach of set theory. One spot where inconvenience arises is the treatment of ordinal and cardinal numbers. Whereas they admit explicit constructions in material set theories, it is hard to yield a treatment of them without always explicitly using the notion of well-ordered sets and comparing cardinalities using injections or surjections.

For the current version of ETCS, people may also not like the fact that it hard-coded the Axiom of Choice as one of its axioms. This axiom is employed when proving Theorem 5 as in Lawvere's original paper. Thankfully, as there exists a model that satisfies all the other axioms but not AC Riehl [45], the axiom of choice is independent of the others. There are many variants of ETCS of interest. For the purposes of our experimentation, we have focused on the original version of ETCS, as we will describe in the following.

5.1.1 Basic settings

The table for primitive function symbols required by the signature of ETCS is displayed below.

The only primitive predicate symbol is the equality between arrows. Unlike the other two systems SEAR and CCAF that we have experimented with, where some predicates have to be primitive, with their behavior characterized by axioms, all the predicate symbols in ETCS can

Symbol	Input	Output	Source
$\circ$	$[f : A \to B,\ g : B \to C]$	$g \circ f : A \to C$	signature
id	$[A : \mathrm{Ob}]$	$\mathsf{id}(A) : A \to A$	signature
$\times$	$[A : \mathrm{Ob},\ B : \mathrm{Ob}]$	$A \times B : \mathrm{Ob}$	Axiom 1
π_1	$[A : \mathrm{Ob},\ B : \mathrm{Ob}]$	$\pi_1(A, B) : A \times B \to A$	Axiom 1
π_2	$[A : \mathrm{Ob},\ B : \mathrm{Ob}]$	$\pi_2(A, B) : A \times B \to B$	Axiom 1
$+$	$[A : \mathrm{Ob},\ B : \mathrm{Ob}]$	$A + B : \mathrm{Ob}$	Axiom 1
i_1	$[A : \mathrm{Ob},\ B : \mathrm{Ob}]$	$i_1(A, B) : A \to A + B$	Axiom 1
i_2	$[A : \mathrm{Ob},\ B : \mathrm{Ob}]$	$i_2(A, B) : B \to A + B$	Axiom 1
$\varnothing$	$[]$	$\varnothing : \mathrm{Ob}$	Axiom 1
$\mathbf{1}$	$[]$	$\mathbf{1} : \mathrm{Ob}$	Axiom 1
Exp	$[A : \mathrm{Ob},\ B : \mathrm{Ob}]$	$B^A : \mathrm{Ob}$	Axiom 2
ev	$[A : \mathrm{Ob},\ B : \mathrm{Ob}]$	$\mathsf{ev}(A, B) : A \times B^A \to B$	Axiom 2
$\mathbb{N}$	$[]$	$\mathbb{N} : \mathrm{Ob}$	Axiom 3
z	$[]$	$z : \mathbf{1} \to \mathbb{N}$	Axiom 3
s	$[]$	$s : \mathbb{N} \to \mathbb{N}$	Axiom 3

Table 5.1: Primitive symbols required by ETCS

be obtained by definitions. In order to state the axioms, we now define required predicate symbols (in contrast to via *axiomatizing*).

- Initial object is to be regarded as the empty set, and admits a function to any set.

$$\vdash \forall X.\ \mathsf{intl}(X) \Leftrightarrow \forall A.\ \exists! f : X \to A.\ \top$$

- Terminal object is to be regarded as the singleton set, and admits a function from any set.

$$\vdash \forall X.\ \mathsf{tml}(X) \Leftrightarrow \forall A.\ \exists! f : A \to X.\ \top$$

- Product set is regarded as the set of pairs, with two projections giving the components:

$$\vdash \forall AB\ p_1 : AB \to A\ p_2 : AB \to B.$$
$$\mathsf{isPr}(p_1, p_2) \Leftrightarrow \forall X\ f : X \to A\ g : X \to B.\ \exists! fg : X \to AB.p_1 \circ fg = f\ \wedge\ p_2 \circ fg = g$$

- Coproduct set is regarded as a disjoint union, with two inclusions from the sets where the elements come from.

$$\vdash \forall AB\ i_1 : A \to AB\ i_2 : B \to AB.$$
$$\mathsf{iscoPr}(i_1, i_2) \Leftrightarrow \forall X\ f : A \to X\ g : B \to X.\ \exists! fg : AB \to X.\ fg \circ i_1 = f\ \wedge\ fg \circ i_2 = g$$

- Equalizer of two functions $f, g : A \to B$ captures the set $\{a \in A \mid f(a) = g(a)\}$. This set is the E in the definition and admits an inclusion to the set A.

$$\begin{aligned}&\vdash \forall A\ B\ f\ g : A \to B\ E\ e : E \to A.\\ &\quad \mathsf{isEq}(f, g, e) \Leftrightarrow\\ &\quad f \circ e = g \circ e \wedge \forall X\ a : X \to A.\ f \circ a = g \circ a \implies \exists! a_0 : X \to E.\ a = e \circ a_0\end{aligned}$$

- Coequalizer of two functions $f, g : A \to B$ captures the quotient set B/R, where R relates b_1 to b_2 if and only if they come from the same element in A. That is, there exists an element a such that $f(a) = b_1$ and $g(a) = b_2$. The set is the E as below and admits a surjection as the quotient map from the set B.

$$\begin{aligned}&\vdash \forall A\ B\ f\ g : A \to B\ E\ e : B \to E.\\ &\quad \mathsf{iscoEq}(f, g, e) \Leftrightarrow\\ &\quad e \circ f = e \circ g \wedge \forall X\ b : B \to X.\ b \circ f = b \circ g \implies \exists! b_0 : E \to X.\ b = b_0 \circ e\end{aligned}$$

We highlight the point that in the definition of the initial and terminal object, the assertion of a variable of a certain sort is given by applying the existential to the truth. A non-existence assertion of a variable is of the form $\forall v : s.\ \bot$, which allows us to prove falsity by specialization from each term of such a sort.

The definition of a product, stated in English, will start with "a product consists of an object and two projections", referring to three terms. As the object is encoded in the sort of projection maps, we do not have to put it into the argument. For the same reason, the definition of (co-)equalizer does not make the equalizer object.

In both SEAR and CCAF, the counterparts of all the definitions above exist. The definitions look exactly the same as above. In the relevant parts of this thesis, we use the same symbol to be the corresponding notion for these two systems as well, as does the definition below on exponential.

As the first definition that requires us to take a decision, the definition of exponential is more complicated. Since the characterizing property of the exponential refers to the notion of the product, we need to determine whether we should fix a chosen product or directly refer to all the possible products in the definition. The second option is possible. Indeed, we can write

out a definition without referring to any particular products:

$$
\begin{aligned}
&\forall A\ E\ AE\ B\ (e : AE \to B).\\
&\quad \mathsf{isExp}(A, E, e) \Leftrightarrow\\
&\quad\ \exists (p_1 : AE \to A)\ (p_2 : AE \to E).\\
&\quad\quad \mathsf{isPr}(p_1, p_2) \wedge\\
&\quad\quad\ \forall X\ AX\ (p_1' : AX \to A)\ (p_2' : AX \to A)\ (f : AX \to B).\\
&\quad\quad\quad \mathsf{isPr}(p_1', p_2') \implies\\
&\quad\quad\quad \exists!(g : X \to E).\ \forall (h : AX \to AE).\ p_1 \circ h = p_1' \wedge p_2 \circ h = g \circ p_2' \implies e \circ h = f
\end{aligned}
$$

There are two products involved in this definition: the product between A and the exponential object E written in B^A in the usual notation, and the ones between A and X, which is the AX above. For the first one, although it is possible to avoid referring to the projection maps, referring to the product object AE is unavoidable. This is because an exponential must contain the information of its evaluation map, which is a map from this product. Therefore, this object has to be present in its sort. It is also notable that since the association between the object and the product involving it (here the A and the product AE) are not evident in the syntax: instead of getting the A from sort information of e, we have to take A as an argument as well. Another option with an explicit reference to a chosen product but without the usage of any function symbol is to omit the quantification on the projections p_1 and p_2 and replace the A and E in the argument list of isExp with them. But it is already clear that none of these options is as simple as the following definition, where we directly use the chosen product symbol.

$$
\begin{aligned}
&\vdash \forall A\ B\ E\ (e : A \times E \to B).\\
&\qquad \mathsf{isExp}(e) \Leftrightarrow \forall X\ (f : A \times X \to B).\ \exists! h : X \to E. e \circ \langle \pi_1(A, X), h \circ \pi_2(A, X) \rangle = f
\end{aligned}
$$

We adopt this definition.

Now we are equipped with all the ingredients to post the full list of formal ETCS axioms. The numbering of the axioms again follows from Lawvere [29].

- Identity acts on both sides

$$\vdash \forall B\ A\ (f : B \to A).\ f \circ \mathsf{id}(B) = f \wedge \forall B\ A\ (f : B \to A).\ \mathsf{id}(A) \circ f = f$$

- Composition is associative

$$\vdash \forall A\ B\ C\ D\ (f : A \to B)\ (g : B \to C)\ (h : C \to D).\ (h \circ g) \circ f = h \circ g \circ f$$

- Axiom 1: The constant $\mathbf{1}$ is terminal and $\varnothing$ is initial. The function symbols $\times$, π_1, and π_2 produce product information and $+$, i_1 and i_2 produce coproduct information. Equalizer and coequalizer exist for any parallel pair of maps.

$$\vdash \mathsf{tml}(\mathbf{1}) \ \wedge \ \mathsf{intl}(\varnothing)$$
$$\vdash \forall A\ B.\ \mathsf{isPr}(\pi_1(A,B), \pi_2(A,B)) \ \wedge \ \mathsf{iscoPr}(i_1(A,B), i_2(A,B))$$
$$\vdash \forall A\ B\ f\ g : A \to B.\ (\exists E\ e : E \to A.\ \mathsf{isEq}(f,g,e)) \ \wedge \ (\exists E\ e : B \to E.\ \mathsf{iscoEq}(f,g,e))$$

- Axiom 2: The function symbols $\_^-$ and ev produce the exponential for each pair of objects.

$$\vdash \forall A\ B.\ \mathsf{isExp}(\mathsf{ev}(A,B) : A \times B^A \to B)$$

- Axiom 3: The constant $\mathbb{N}$ is a natural number object. It has the universal property: To specify a map from $\mathbb{N}$ to X is to give a starting point $x_0 : 1 \to X$ and a map $X \to X$, describing how to get the next output from the previous output.

$$\vdash \forall X\ x_0 : 1 \to X\ t : X \to X.\ \exists x : \mathbb{N} \to X.\ x \circ z = x_0 \ \wedge \ x \circ s = t \circ x$$

- Axiom 4 (well-pointedness): Two arrows are equal if and only if they agree on all the elements of the domain.

$$\vdash \forall A\ B\ f\ g : A \to B.\ \neg(f = g) \implies \exists a : 1 \to A.\ \neg(f \circ a = g \circ a)$$

- Axiom 5: Axiom of choice (see below for further explanation)

$$\vdash \forall A\ B\ a : \mathbf{1} \to A\ f : A \to B.\ \exists g : B \to A.\ f \circ g \circ f = f$$

- Axiom 6: Every non-initial object admits a map from the terminal object 1, i.e., has an element.

$$\forall X.\ \neg\mathsf{intl}(X) \implies \exists x : \mathbf{1} \to X.\ \top$$

- Axiom 7: Every element of a coproduct factors through one inclusion map. i.e., any element of a disjoint union comes from one of the disjunct.

$$\vdash \forall A\ B\ f : \mathbf{1} \to A{+}B.\ (\exists f_0 : \mathbf{1} \to A.\ i_1(A,B) \circ f_0 = f) \vee (\exists f_0 : \mathbf{1} \to B.\ i_2(A,B) \circ f_0 = f)$$

- Axiom 8: There exists an object with two distinct elements.

$$\vdash \exists X\ (x_1\ x_2 : \mathbf{1} \to X).\ x_1 \neq x_2$$

The original version of Axiom 1 as from Lawvere [29] asserts all finite limits exist. The category-theoretic definition of limits refers to the concept *diagram*, which is a functor from a category encoding an arrangement of arrows to the category where the required limit lives. Whereas ETCS is developed with some spirit of category theory, it itself does not capture much category theory. Instead, it is simply a model of a particular category. The notion of "functor" is between two categories, and hence lives on the meta-level when talking about the ETCS system. Such a notion does not exist within ETCS. Therefore, the original Axiom 1 cannot be stated directly. We address this issue by appealing to a theorem from category theory: the existence of finite limits is guaranteed by the existence of three particular limits (and dually, co-limits): the terminal object, the product, and the equalizer.

We might adopt another approach at the level of the implementation. As the functor required for constructing the limit encodes no more than a "shape", it is possible to create a data type representing the shape. Without any explicit usage of meta-level category theory, we could implement a rule that takes a term in this data type and outputs a theorem stating a limit of this graph exists. We do not adopt such an option. Firstly, we prefer working with the mathematical level instead of the programming level and prefer axiom formulas to more primitive rule implementation when possible. Secondly, according to experiments up to now, there is no current usage of a general (co)-limit, so our choice does not cause any problem. The special limits we take as primitive in our axioms are invoked explicitly, indicating that it would be convenient to assume their existence.

We do not create function symbols for equalizers and coequalizers. It is possible to apply the function specification rule in Section 2.3 to create them. A function symbol that creates a (co-)equalizer object will take two arrows, between which equalities can be written, and produce an object, where there is no notion of equality. This means that even if we have $f_1 = f_2 : A \to B$, we cannot write $\mathsf{Eqo}(f_1, g) = \mathsf{Eqo}(f_2, g)$ provided Eqo is such a function symbol. Nevertheless, when proving a theorem with conclusion $\phi(f_1)$ for some property ϕ, our congruence rule can be applied as a shortcut to reduce it into $\phi(f_2)$. We conclude from our rule that $\mathcal{F}([f : A \to B], [f_1]) \Leftrightarrow \mathcal{F}([f : A \to B], [f_2])$, and instantiate the formula variable to be ϕ. Note that it does require the formula ϕ not to have any free variables with f_1 appearing in their sorts. But in most cases, the interesting forms of ϕ will look like $\forall f : X \to \mathsf{Eqo}(f_1, g).\ \psi(f)$ for some object X, for $\mathsf{Eqo}(f_1, g)$ being an equalizer. We do recommend using the universal property to perform this reduction, as is always possible.

Theorem 3 in Section 5.1.2 will be an example showing that such function symbols would not bring any particular convenience.

The universal property of the natural number object only gives the existence of the function. This is actually uniquely determined, as can be proved from well-pointedness.

The approach taken by Axiom 5 to state the axiom of Choice is indeed unusual. The quantification on $a : \mathbf{1} \to A$ actually adds an additional assumption required from the g to exist, namely the non-emptyness of the domain of f. It does imply the canonical form that every epimorphism has a section. Indeed, for an epimorphism $f : A \to B$, from Axiom 6, A is either initial or has an element. If A is initial then so does B as f is epic, but then the g can be taken by the identity. Otherwise, we have $f \circ g \circ f = g$, which implies $f \circ g = \mathsf{id}(B)$, again since f is an epi.

5.1.2 Theorems in Lawvere's paper

In this section, we present the formalization of the 6 theorems in Lawvere [29]. As the translations of the proofs are mostly routine, we will omit them and focus on explaining the presentations and roles of the formal theorems.

Theorems 1 and 2 express basic properties of the natural numbers. While the axiom concerning the natural number object N initially only permits defining functions via recursion, Theorem 1 increases the flexibility by allowing definition by primitive recursion as well.

5.1.1 Theorem 1 (Primitive recursion).

$$\begin{aligned}
&\vdash \forall A\ B\ (g : A \to B)\ (h : (A \times \mathbb{N}) \times B \to B).\\
&\quad \exists f : A \times \mathbb{N} \to B.\\
&\qquad f \circ \langle \pi_1(A, \mathbf{1}), z \circ \pi_2(A, \mathbf{1}) \rangle = g \circ \pi_1(A, \mathbf{1}) \wedge\\
&\qquad h \circ \langle \mathsf{id}(A \times \mathbb{N}), f \rangle = f \circ \langle \pi_1(A, \mathbb{N}), s \circ \pi_2(A, \mathbb{N}) \rangle
\end{aligned}$$

The meaning of this theorem is obvious when we consider what the functions do to elements. Take an element $a : 1 \to A$ of A, the function $A \times N \to B$ proved to exist has value $f \circ \langle a, z \rangle = g \circ a$, because the second component is zero. Recursively, knowing the value of $f \circ \langle a, n \rangle$ for an element n of $\mathbb{N}$, we can deduce the value of $f \circ \langle a, n^+ \rangle$, where $n^+ := s \circ n$ is the successor of n, to be $h \circ \langle \langle a, n \rangle, f \circ \langle a, n \rangle \rangle$.

Primitive recursion is useful when defining functions on $\mathbb{N}$ taking multiple arguments, such as addition. To define it, we take both A and B to be $\mathbb{N}$, take g to be the identity $\mathsf{id}(\mathbb{N})$ (meaning adding by the element z is the constant function), and take h to be $s \circ \pi_2(\mathbb{N} \times \mathbb{N}, \mathbb{N})$. While calculating the sum of two elements a and b^+, the elements a and b are stored in the $\mathbb{N} \times \mathbb{N}$ part of the triple product, and the sum of them is stored in the $\mathbb{N}$ part. We take the

projection on the second component to take out the sum of a and b, then take its successor. This description is actually exactly what is in our mind when doing natural number addition. Whereas Theorem 1 captures this elegantly, its canonical expression in material set theories will be much more involved. Function applications are not presented naturally as composition to an element, because a function is a subset of the set of pairs. Moreover, Theorem 1's function is "typed", which means we make sure we are working with input that makes sense from the very beginning. As with the case for simple recursion, the function is uniquely determined by g and h.

Theorem 2 proves some of Peano's axioms. The original theorem presented in Lawvere [29] is formalized as.

5.1.2 Theorem 2.

$$
\begin{aligned}
&\vdash \forall n : \mathbf{1} \to \mathbb{N}.\ s \circ n \neq z \\
&\vdash \mathsf{Mono}(s) \\
&\vdash \forall A\ a : A \to \mathbb{N}. \\
&\quad \mathsf{Mono}(a) \wedge (\exists z_0 : \mathbf{1} \to A. z = a \circ z_0) \\
&\quad (\forall n : \mathbf{1} \to \mathbb{N}. \\
&\quad (\exists n_0 : \mathbf{1} \to A.\ n = a \circ n_0) \implies \exists n'.\ s \circ n = a \circ n') \\
&\qquad \implies \mathsf{Iso}(a)
\end{aligned}
$$

The first two parts express that zero is not a successor, and the successor function is an injection. The third part actually expresses the principle of induction. In the language of category theory, a monomorphism is to be regarded as an inclusion from a subset. An element $n : \mathbf{1} \to \mathbb{N}$ belongs to the subset $a : A \to N$ if n factors through A, i.e., if n is in the image a as an inclusion map. With the spirit that a subset on $\mathbb{N}$ corresponds to a statement on $\mathbb{N}$, the theorem reads "if a statement holds for zero and if it holds an element n, then it also holds for its successor, then the predicate holds for the whole $\mathbb{N}$".

Later, for formalizing natural number arithmetic, we will prove the "working version" of the induction principle. From now on, we write 2 to denote the disjoint union $1 + 1$. By Axiom 7, the two inclusions $i_1(1, 1)$ and $i_2(1, 1)$ are exactly *the* only two elements of 2. The object 2 can then be regarded as the two-element set for truth values. We denote them as falsity $\bot_I$ and truth $\top_I$ respectively.

Using Theorem 5, to be proved later, we prove each monomorphism uniquely corresponds to a characteristic function, to be regarded as a predicate ϕ that holds on an element x if and only

if $\phi \circ x = \top_I$:

$$\begin{aligned}
&\vdash \forall A\ X\ a : A \to X. \\
&\quad \mathsf{Mono}(a) \implies \\
&\quad \exists! \phi : X \to 2.\ (\forall x : \mathbf{1} \to X.\ (\exists x_0 : \mathbf{1} \to A.\ a \circ x_0 = x) \Leftrightarrow \phi \circ x = \top_I)
\end{aligned}$$

Then we can identify the subset a as in Theorem 3 using its characteristic function $p : \mathbb{N} \to 2$. It in turn gives the induction principle a more conventional reformulation, which we proved as:

$$\begin{aligned}
&\vdash \forall p : \mathbb{N} \to 1 + 1.\ p = \top_{\mathbb{N}} \Leftrightarrow p \circ z = \top \wedge \\
&\quad \forall n : \mathbf{1} \to \mathbb{N}. p \circ n = \top \implies p \circ s \circ n = \top_I
\end{aligned}$$

Here $\top_{\mathbb{N}} := \top_I \circ !_{\mathbb{N}}$ (where $!\_$ is the unique map to the terminal object) is the characteristic function for the whole $\mathbb{N}$, which assigns every element of $\mathbb{N}$ to $\top_I$. The mechanism to apply it might not be obvious now, because it requires the property, described as a formula, to be captured by a monomorphism, or an arrow $\mathbb{N} \to 2$. Its practical application becomes more straightforward after the development of internal logic within ETCS. Internal logic enables the transformation of properties expressed as first-order formulas into characteristic functions, thereby allowing us to apply the aforementioned principle. Moreover, the construction is not restrictive to $\mathbb{N}$ but can be used to build a predicate on any set. We leave the details of this construction to Section 5.1.3.

One standard and useful fact that holds in any topos is that any arrow can be factorized into an epimorphism, regarded as a surjection onto its image, and a monomorphism, regarded as an inclusion from its image to its codomain. This result is required for all the three remaining Theorems 4, 5, and 6. It is a corollary of Theorem 3:

5.1.3 Theorem 3.

$$\begin{aligned}
&\vdash \forall A\ B\ f : A \to B. \\
&\quad \forall\ R'\ (k' : B + B \to R')\ I\ (q' : I \to B)\ R\ (k : R \to A \times A)\ I'\ (q : A \to I'). \\
&\qquad \mathsf{iscoEq}(i_1(B,B) \circ f, i_2(B,B) \circ f, k') \wedge \mathsf{isEq}(k' \circ i_1(B,B), k' \circ i_2(B,B), q') \wedge \\
&\qquad \mathsf{isEq}(f \circ \pi_1(A,A), f \circ \pi_2(A,A), k) \wedge \mathsf{iscoEq}(\pi_1(A,A) \circ k, \pi_2(A,A) \circ k, q) \implies \\
&\qquad \exists! h : I' \to I.\ q' \circ h \circ q = f \wedge \mathsf{Iso}(h)
\end{aligned}$$

The commutative diagram for this theorem is:

$$\begin{array}{ccccccccc}
R & \xrightarrow{k} & A \times A & \underset{\pi_2(A,A)}{\overset{\pi_1(A,A)}{\rightrightarrows}} & A & \xrightarrow{f} & B & \underset{i_2(B,B)}{\overset{i_1(B,B)}{\rightrightarrows}} & B + B & \xrightarrow{k'} & R' \\
 & & & & \downarrow{\scriptstyle q} & & \uparrow{\scriptstyle q'} & & & & \\
 & & & & I' & \xrightarrow{h} & I & & & &
\end{array}$$

As for the order, the coequalizer k' of $i_1(B,B) \circ f$ and $i_2(B,B) \circ f$, as well as the equalizer k of $f \circ \pi_1(A,A)$ and $f \circ \pi_2(A,A)$, are formed in the first step. The q and q' are formed as the coequalizer and equalizer for the horizontal maps obtained by composing the inclusion/projection with k' and k, respectively. The theorem expresses that there is a unique isomorphism linking this pair of the equalizer and the coequalizer.

We have opted to exclusively employ predicates for representing equalizer and coequalizer information, abstaining from the introduction of any additional function symbols. As mentioned, here an evidence that such function symbols would not enhance the theorem in any meaningful way. If we use function symbols, the term that is simply a variable with the name R' as above will become $\mathsf{coEqo}(i_1(B,B) \circ f), i_2(B,B) \circ f)$. The construction is nested. In the next step, the name of the object I would be even more cumbersome. The term will be a bulky function symbol application, expressed as:

$$\mathsf{Eqo}(\mathsf{coEqa}(i_1(B,B) \circ f, i_2(B,B) \circ f) \circ i_1(B,B), \mathsf{coEqa}(i_1(B,B) \circ f, i_2(B,B) \circ f) \circ i_2(B,B))$$

In that case, it is important to use only variable names, in both the proof and the theorem, to keep the expressions handy to work with.

5.1.4 Theorem 4.

$$\begin{aligned}
&\vdash \forall J\ X\ (s : J \to 2^X).\\
&\quad \exists U\ (a : U \to X).\\
&\qquad \mathsf{Mono}(a) \wedge\\
&\qquad \forall x : \mathbf{1} \to X.\ (\exists x_0.\ x = a \circ x_0) \Leftrightarrow \exists j : \mathbf{1} \to J.\ \mathsf{ev}(X,2) \circ \langle x, s \circ j \rangle = \top_I
\end{aligned}$$

According to the discussion about Theorem 2, a subset of X can be identified by its characteristic function $X \to 2$. Taking its transpose, such a characteristic function is an element $\mathbf{1} \to 2^X$ of the exponential. A function from a set J to 2^X has its image as a subset in 2^X. We regard what is in this image as the family indexed by J. For each element $j : \mathbf{1} \to J$, the set indexed by it is the transpose of $s \circ j : \mathbf{1} \to 2^X$. Theorem 4 says such a family can be unioned into a subset of X. The union is captured by the monomorphism a in the theorem. Indeed, the right-hand side of the iff says an element $x : \mathbf{1} \to X$ is in a precisely if it is in some set indexed by some element of J.

It is worth emphasizing that Theorem 4 captures the union of possibly infinitely many sets. Given two subsets $a_1 : A_1 \to X$ and $a_2 : A_2 \to X$, their disjoint union admits a map into X. By taking the mono-epi factorization of this map, binary union, and hence unions of finitely many sets, are easy to construct. This theorem proves a statement with far more generality, providing us with the means to effectively handle potentially infinite disjunctions.

5.1.5 Theorem 5.

$$\vdash \forall A\ X\ a : A \to X.\ \mathsf{Mono}(a) \implies \exists A'\ a' : A' \to X.\ \mathsf{Mono}(a') \ \wedge\ \mathsf{Iso}(\binom{a}{a'})$$

Theorem 5 proves every subset has a complement, with $\binom{a}{a'}$ denoting the canonical arrow from the coproduct. It uses a corollary of the Axiom 5 (Choice). The usage is to prove a lemma that says for each element x of X which is not in a, there exists a special subset that contains x and does not intersect with a. Then the complement of a is taken to be the union of all these subsets. The usage of choice is specific to ETCS. The counterpart in SEAR is easier due to a comprehension theorem schema that we can prove, which allows us to immediately get the set constructed by negating any formula, and in particular, the formula asserting an element belongs to a certain subset.

The final non-meta theorem that appears states another well-known result in topos theory. As a slogan, it reads "any equivalence relation is the kernel pair of its coequalizer". In standard mathematical presentation, an equivalence relation on a set A is a subset of $A \times A$. Explicitly, it is the set $\{(a_1, a_2) \mid a_1 R a_2\}$. This set can be represented by the two projection maps from it to A. In ETCS, a relation on A consists of two maps $f_0, f_1 : R \to A$ with a common domain. The concept of being an equivalence relation can also be formulated in the diagrammatic manner and is expressed as $\mathsf{Equiv}(f_0, f_1)$.

5.1.6 Theorem 6.

$$\begin{aligned}\vdash \forall R\ A\ (f_0\ f_1 : R \to A)\ E\ (e : E \to A \times A)\ Q\ (q : A \to Q).\\ \mathsf{Equiv}(f_0, f_1) \ \wedge\ \mathsf{iscoEq}(f_0, f_1, q) \ \wedge\ \mathsf{isEq}(q \circ \pi_1(A,A), q \circ \pi_2(A,A), e) \implies\\ \exists h_1\ h_2.\ e \circ h_1 = \langle f_0, f_1\rangle \ \wedge\ \langle f_0, f_1\rangle \circ h_2 = e \ \wedge\ h_1 \circ h_2 = \mathsf{id}(E) \ \wedge\ h_2 \circ h_1 = \mathsf{id}(R)\end{aligned}$$

The coequalizer q of f_1 and f_2 is the quotient of the set A by the equivalence relation given by the pair. The kernel pair of q is a set consisting of pairs of elements of A that are sent to the same place by the quotient map. This set is the E as in Theorem 6. Theorem 6 then states that E has a structural-respectful isomorphism to R. This effectively expresses that the diagram:

$$\begin{array}{ccc} R & \xrightarrow{f_1} & A \\ \downarrow{\scriptstyle f_0} & & \downarrow{\scriptstyle q} \\ A & \xrightarrow{q} & Q \end{array}$$

is a pullback. In terms of elements: given two elements a_1 and a_2 of A, they are identified in the quotient if and only if they come from a pair in R. That is, in other words, if and only if they are related by the equivalence relation.

5.1.3 Comprehension via Internal logic

A comprehension schema in a system refers to the mechanism for extracting a part of an object of interest that satisfies a certain condition to form another object. Such schemata play a vital role in any foundational framework. While it may not be apparent from its axioms alone that we allow a certain form of comprehension, we are able to derive a practical comprehension schema thanks to the internal logic. We now briefly discuss the topic of internal logic. A more detailed treatment is given in [46]. Despite the restrictions of the ETCS comprehension we will derive, this schema is sufficiently powerful for many mathematical applications. In fact, it equips ETCS as a system that is at least as strong as HOL. The advantage of ETCS, in terms of simplicity, becomes evident when compared to HOL. While HOL incorporates higher-order logic into its kernel, ETCS remains a first-order system with only a 2-sort dependency.

Recall that an arrow $p : X \to 2$ corresponds to a predicate on X in the sense that if $x : 1 \to X$, then $p \circ x = \top_I$ means p is true for x. Our arrow p gives rise to a separate object as characterized by the theorem:

$$\forall A\ (p : A \to 2).\ \exists B\ (i : B \to A).\ \mathsf{Inj}(i) \wedge \forall a : 1 \to A.\ p \circ a = \top_I \Leftrightarrow \exists b.\ a = i \circ b$$

The existence of i is witnessed by the pullback of the map $\top_I$ along p.

ETCS proves that for each first-order formula expressed in the language that is *bounded*, we can capture it by such an arrow. The word *bounded* is a terminology in first-order logic. It holds for a formula precisely when all the quantifiers appear in it binds some variable that represents an element, in contrast to representing a set or an arrow in general, i.e., those ones whose domain is not $\mathbf{1}$. For example, the formula $\forall n : \mathbf{1} \to N.\exists m : \mathbf{1} \to N.\ n + m \neq n$ is bounded, whereas $\forall A\ a : \mathbf{1} \to 2^A.\mathsf{Fin}(A) \implies \exists n : \mathbf{1} \to N.\ |a| = n$, stating every finite set has a cardinality, is not bounded due to the quantification on A.

Let us call the formulas of our logic (all formulas seen so far) *external formulas*. We automatically construct a corresponding *internal formula* as a term of the logic. When the external formula is on variables with sorts $(1 \to X_1), (1 \to X_2), \ldots$, then the internal formula will be an arrow of sort $\Pi X_i \to 2$. For an external formula $\Phi[x_1 : 1 \to X_1, \ldots]$, then let $p : \Pi X_i \to 2$ be the corresponding formula. We require

$$\forall a : 1 \to \Pi X_i.\ p \circ a = \top_I \Leftrightarrow \Phi[(\pi_i \circ a)/x_i]$$

where $\Phi[t/x]$ is the substitution of term t for variable x. This could be regarded as an axiom, one rather like Separation in ZF. However, we can instead prove all results of this form automatically. This is simply by rewriting with all the theorems with relevant definitions and properties of the internal logic operators as explained below.

We have implemented an automatic translation (a "derived rule") that generates an internal logic formula given a list of variables, considered as the arguments, and the formula. The translation produces an internal logic predicate and proves that it gives the value $\top_I$ if and only if the formula is true when applied to the arguments.

5.1.7 Example. We illustrate our algorithm with an example over $\mathbb{N}$, the natural number object, the arrow $\mathsf{SUC} : \mathbb{N} \to \mathbb{N}$, and the function symbol $\_^+$, such that $n^+ \stackrel{\text{def}}{=} \mathsf{SUC} \circ n$. Then, the pair $([n], m^+ - n^+ = m - n)$ encodes a simple unary predicate on n. In this case, the output of our derived rule is an arrow term $p : \mathbb{N} \to 2$ satisfying:

$$\forall n : 1 \to \mathbb{N}.\ \ p \circ n = \top \Leftrightarrow m^+ - n^+ = m - n$$

If the list of arguments is $[m, n]$ instead, the produced arrow $p : \mathbb{N} \times \mathbb{N} \to 2$ will satisfy:

$$\forall m, n : 1 \to \mathbb{N}.\ p \circ \langle m, n \rangle = \top \Leftrightarrow m^+ - n^+ = m - n$$

Operator	Sort	Defining Property
$\wedge_I$	$2 \times 2 \to 2$	$\wedge_I \circ \langle p_1, p_2 \rangle \Leftrightarrow p_1 = \top_I \wedge p_2 = \top_I$
$\vee_I$	$2 \times 2 \to 2$	$\vee_I \circ \langle p_1, p_2 \rangle \Leftrightarrow p_1 = \top_I \vee p_2 = \top_I$
$\Rightarrow_I$	$2 \times 2 \to 2$	$\Rightarrow_I \circ \langle p_1, p_2 \rangle \Leftrightarrow p_1 = \top_I \implies p_2 = \top_I$
$\neg_I$	$2 \to 2$	$\neg_I \circ p = \top_I \Leftrightarrow p = \bot_I$
$\forall_X$	$2^X \to 2$	$\forall_X \circ \overline{p} \circ y = \top_I \Leftrightarrow \forall x. p \circ \langle x, y \rangle = \top_I$
$\exists_X$	$2^X \to 2$	$\exists_X \circ \overline{p} \circ y = \top_I \Leftrightarrow \exists x : 1 \to X. p \circ \langle x, y \rangle = \top_I$

Table 5.2: Operators of the Internal Logic

To convert formulas into internal formulas, we need to first convert terms into 'internal terms'. In particular, function symbols will map into arrows of an appropriate sort. For example, if our 'external formula' is on variables $[x : 1 \to \mathbb{N}, y : 1 \to \mathbb{N}]$, then any 'internal term' built as part of this translation will be from $\mathbb{N} \times \mathbb{N}$. In our $\mathbb{N}$-example, the arrow corresponding to y^+ will be $\mathsf{SUC} \circ \pi_2(\mathbb{N}, \mathbb{N})$. In most circumstances, the connection between the function symbol and the arrow will simply be that symbol's definition. For generality's sake, our implementation stores the external-internal correspondence of function and predicate symbols in a simple dictionary data structure.

Our formula-converting function is recursive on the structure of the formula, using the semantics of the various connectives and quantifiers given in Table 5.2. The only built-in predicate, equality, corresponds to the characteristic map of the diagonal monomorphism. For user-defined predicates, such as $<$ over natural numbers, users can store the correspondences manually. The induction steps for the connectives are straightforward. For quantifiers, for example, consider the formula $\forall a : 1 \to A.\ \ a = a_0$. Begin by converting the body $a = a_0$ into a predicate on

$[a, a_0]$; and then transpose the output and post-compose with the internal logic operator $\forall_A$. The existential case is similar.

5.2 Formalizing SEAR

We now present the system SEAR, another structural set theory designed by Michael Shulman [7]. Its standard presentation does not use any categorical terminology, aiming to demonstrate structural set theory is independent of category theory. The spirit behind SEAR is more similar to ETCS, but from its signature, its presentation is intuitively closer to structural ZF. In terms of strength, it is equivalent to ZF without choice. Adding choice to it yields a system equivalent to ZFC, stronger than ETCS.

5.2.1 Basic settings

As *per* Shulman's original construction, SEAR has three sorts: sets (A, B, ...; a ground sort); members ($\_ \in A$, depending on a set term); and relations ($A \nrightarrow B$, depending on two set terms). SEAR also adds a primitive predicate $\mathsf{Holds}(R : A \nrightarrow B, a \in A, b \in B)$, declaring that the relation R relates a and b. Equality can hold between relations with the same domain and codomain and elements of the same set.

In SEAR, a relation $R : A \nrightarrow B$ is called a function if for each member $a \in A$, there exists a unique $b \in B$ such that $\mathsf{Holds}(R, a, b)$. In practice, we want to be able to write $f(a)$ as the result of applying a function to an argument, but we cannot do this if we are restricted to just the relation sort. A first thought might be to create a function symbol Eval, that takes a relation and a member of A, so the term $\mathsf{Eval}(R : A \nrightarrow B, a \in A)$ is a member of B. However, such a function symbol breaks soundness, as the term $\mathsf{Eval}(R, a)$ can be expressed for every a of the correct sort before checking the function condition on R. In particular, we can write a term $\mathsf{Eval}(R : 1 \nrightarrow 0, \star)$, nominally producing an element of 0.

Rather, we introduce a function sort which is a "proper subsort" of the relation sort.[1] A function f from A to B is written $f : A \to B$, and we add the following axiom describing terms of function sorts:

$$\begin{aligned} &\vdash \forall R.\ \mathsf{isFunction}(R) \implies \\ &\quad \exists! f : A \to B.\ \forall (a \in A)\ (b \in B).\ \mathsf{Eval}(f, a) = b \Leftrightarrow \mathsf{Holds}(R, a, b) \end{aligned}$$

The $\mathsf{isFunction}$ predicate embodies the definition above, and we also have a new Eval function symbol that takes a function term from A to B and a member term of A, and outputs a member term of B.

[1] Shulman (personal communication) agrees that the resulting system is still effectively SEAR as he conceives it.

Symbol	Input	Output	Source	Function/Predicate
Holds	$[R : A \nrightarrow B,\ a \in A,\ b \in B]$	-	Axiom 1	Predicate
App	$[f : A \to B,\ a \in A]$	$\mathsf{App}(f, a) \in B$	Axiom 2	Function
$\mathbb{N}_0$	[]	$\mathbb{N}_0 : \mathsf{set}$	Axiom 4	Function
z_0	[]	$z_0 \in \mathbb{N}_0$	Axiom 4	Function
s_0	[]	$s_0 : \mathbb{N}_0 \to \mathbb{N}_0$	Axiom 4	Function

Table 5.3: Primitive symbols required by SEAR

We will write $\mathsf{Eval}(f, a)$ simply as $f(a)$ in the rest of the thesis. The Eval symbol is typed so that only function terms can be its first argument. It is clear that this is a conservative extension, as any theorems involving Eval can be expressed using just Holds and uses of the $\mathsf{isFunction}$ hypothesis if desired.

SEAR is a remarkably simple system with very few primitive symbols and only 6 axioms. These symbols are illustrated in Figure 5.3. Interestingly, the first five axioms are already sufficient for handling the vast majority of mathematical proofs, underscoring the simplicity of SEAR. The sole function symbol, used for function term application, was introduced due to the additional sort we added. Notably, SEAR does not even employ primitive symbols for relation composition and identity relations; these are all derived from their respective axioms. In our notation, we represent relation composition as an infix $\circ_R$, the identity function on set A as $\mathsf{Id}(A)$, and the identity relation on A as $\mathsf{id}(A)$. Furthermore, the relation term $A \nrightarrow B$, arising from a function $f : A \to B$, is constructed as $\mathsf{asR}(f)$. We write the composition of function terms as an infix $\circ$. All these function symbols are deduced from the specification rule using equality, as they are indeed unique. In the following, we present the first four axioms.

5.2.1 Axiom 0. *There exists a set with an element.*

$$\vdash \exists A\ (a \in A).\ \top$$

5.2.2 Axiom 1. *For a formula that can be regarded as a property on elements of two sets A and B, it defines a relation R from A and B .*

$$\{A,\ B\} \vdash \exists!(R : A \nrightarrow B).\ \forall a \in A\ b \in B.\ \mathsf{Holds}(R, a, b) \Leftrightarrow \mathcal{F}[a_0 \in A, b_0 \in B](a, b)$$

An instantiation of this axiom with a suitable formula variable will produce a unique relation between A and B that holds precisely between elements where the formula holds. Note that the $\mathcal{F}[a_0 \in A, b_0 \in B]$ does not have to be instantiated with a formula that is only on these two variables, it may have other free variables. For example, when A and B are both $\mathbb{N}$, we can use the formula that holds for two natural numbers n_1 and n_2 if their sum is another

natural number n. It does not require the variables a and b to appear in the formula either. Instantiating it with $\top$ and $\bot$ gives the total and empty relation.

5.2.3 Axiom 2 (Tabulation). *Any relation has a tabulation. Such a tabulation is a set together with two projections, similar to the way used in ETCS Theorem 6 for capturing a relation. A relation is uniquely determined by its tabulation.*

$$\begin{aligned}\vdash \forall A\ B\ (R : A \nrightarrow B).\ \exists T\ p : T \to A\ q : T \to B.\\ (\forall x\ y.\ \mathsf{Holds}(R, x, y) \Leftrightarrow \exists r.\ \mathsf{App}(p, r) = x\ \wedge\ \mathsf{App}(q, r) = y) \wedge\\ \forall r\ s.\ \mathsf{App}(p, r) = \mathsf{App}(p, s)\ \wedge\ \mathsf{App}(q, r) = \mathsf{App}(q, s)\ \Longrightarrow\ r = s\end{aligned}$$

The first three axioms are already sufficient to prove the existence of a singleton set and an empty set. We use the function specification with the fact that they are unique up to bijection to create function symbols 1 and 0 for them. The only element of 1 is represented $\star$. A subset of A is defined in SEAR as a relation from A to 1. The elements related to 1 are regarded as belonging to the subset. Axiom 3 asserts the existence of the power set.

5.2.4 Axiom 3. *For each set A, there exists a set P and a relation $e : A \nrightarrow P$ with the following property: for every subset of A, there exists a member of P such that for each element a of A, the relation e holds between a and s if and only if a is in such a subset.*

$$\vdash \forall A.\ \exists P\ e : A \nrightarrow P.\ \forall S_0 : \mathbf{1} \nrightarrow A.\ \exists! s \in P.\ \forall x \in A.\ \mathsf{Holds}(S_0, \star, x) \Leftrightarrow \mathsf{Holds}(e, x, s)$$

Although it asserts its existence, Axiom 3 does not choose any function symbol for it, so we cannot write a $\mathsf{Pow}(A)$ for the power set of A for this moment. This gives us an example of applying that function specification rule.

5.2.5 Example. We write $\mathsf{isPow}(P, e : A \nrightarrow P)$ as the statement "for every subset $S : 1 \nrightarrow$ of A, there exists a unique element $s \in P$ such that for every $a \in A$, we have $\mathsf{Holds}(e, a, s)$ iff a is a belongs to S". The equivalence relation between two such pairs $(P, e : A \nrightarrow P)$ and $(P', e : A \nrightarrow P')$ is the existence of isomorphism that preserves the membership relation, i.e., isomorphism consists of functions $i : P \to P'$ and $j : P' \to P$ plus the condition that $\mathsf{asR}(i) \circ_R e = e'$ and $\mathsf{asR}(j) \circ_R e' = e$. This creates function symbols $\mathsf{Pow}(A)$ and $\mathsf{In}(A) : A \nrightarrow \mathsf{Pow}(A)$.

We write $\mathsf{IN}(a, s) := \mathsf{Holds}(\mathsf{In}(A), a, s)$ for $a \in A$ and $s \in \mathsf{Pow}(A)$.

Axiom 4 asserts the existence of a "pre-natural number set".

5.2.6 Axiom 4. *There exists a set $\mathbb{N}_0$ with a distinguished element z_0 and an injective endomorphism such that z_0 is not in its image.*

$$\vdash (\forall n \in \mathbb{N}_0.\ \mathsf{App}(s_0, n) \neq z_0)) \ \wedge\ (\forall n\ m.\ \mathsf{App}(s_0, n) \neq \mathsf{App}(s_0, m) \Leftrightarrow n = m)$$

The set $\mathbb{N}_0$ is not the one we work with as the natural number object in ETCS. The axiom asserts the existence of a zero and a successor function, but many useful facts do not necessarily hold for $\mathbb{N}_0$, say, there is only one zero element, or an element of n_0 is either a successor or zero. The machinery of defining functions from $\mathbb{N}_0$ is neither stated in nor implied by Axiom 4. Later in Section 5.2.4, after we construct the automatic function for inductive rules, we will be able to take out a set of natural numbers out of $\mathbb{N}_0$.

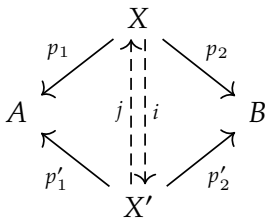
We leave the final axiom, the "collection schema", to the end of this section.

The axioms do not give any of the chosen products, equalizers, and exponentials. The existence of them are all proven from the first four axioms. Here we give a concrete example of the application of our function specification rule declared in 2.3.

5.2.7 Example. We define the predicate isPr so its definition is identical to that in ETCS, as in Section 5.1.1, and prove theorems:

$$\{A,\ B\} \vdash \exists X\ (p_1 : X \to A)\ (p_2 : X \to B).\ \mathsf{isPr}(p_1, p_2)$$
$$\{A,\ B\} \vdash \forall A\ B.\ \exists X\ (p_1 : X \to A)\ (p_2 : X \to B).\ \top$$

The equivalence relation we use here is the existence of maps i, j, such that $i \circ j = 1_X, j \circ i = 1_{X'}$, and all the triangles in the following diagram commutes.



The application of specification hence produces the theorem:

$$\{A,\ B\} \vdash \mathsf{isPr}(\pi_1(A, B) : A \times B \to A, \pi_2(A, B) : A \times B \to B)$$

with the three function symbols $\times, \pi_1, \pi_2$, stored as $(\mathsf{ob}, [A : \mathsf{ob}, B : \mathsf{ob}]), (A \times B \to A, [A : \mathsf{ob}, B : \mathsf{ob}]), (A \times B \to B, [A : \mathsf{ob}, B : \mathsf{ob}])$ in the signature, that produces the chosen product set with projections.

5.2.2 Comprehension in SEAR

Comprehension in SEAR is much easier than the counterpart in ETCS. Thanks to formula variables, we can directly instantiate formulas ϕ to a derived theorem schema to form from a set X the subset $\{x \in X \mid \phi(x)\}$. Our subset is constructed *via* a member of the power set $\mathsf{Pow}(X)$, and ultimately, we can extract it as a term of set sort with an injection to X. This construction is described by the following two theorems (following Shulman [7]). Such two theorems, used intensively in the following sections, are also consequences of the first three axioms only. First, we prove the existence of the member of the power set.

$$\exists! s \in \mathsf{Pow}(A).\ \forall a.\ \mathsf{IN}(a, s) \Leftrightarrow \mathcal{F}[a \in \mathsf{mem}(A)](a)$$

We also have the existence of a set B and an injection from it into A:

$$\exists B\ (i : B \to A).\ \mathsf{Inj}(i) \wedge \forall (a \in A).\ \mathcal{F}[a \in \mathsf{mem}(A)](a) \Leftrightarrow \exists b \in B.\ a = i(b) \tag{5.1}$$

This injection we construct from each predicate is unique up to respectful isomorphism. If there are $i : B \to A$ and $i' : B' \to A$, which are both injections and moreover, we have $\forall a.\ P(a) \Leftrightarrow \exists b \in B.\ a = i(b)$ and $\forall a.\ P(a) \Leftrightarrow \exists b \in B'.\ a = i'(b)$, then the relation between pairs $(B, i : B \to A)$ and $(B', i' : B' \to A)$ defined by

$$\begin{aligned}&\exists (f : B \to B')\ (g : B' \to B).\\ &\quad f \circ g = \mathsf{Id}(B) \wedge g \circ f = \mathsf{Id}(B') \wedge i' \circ f = i \wedge i \circ g = i'\end{aligned}$$

holds. This is clearly an equivalence relation. Moreover, for all sets A, the existence of a set B and a map $B \to A$ is witnessed by the identity isomorphism. Therefore, once we instantiate the P above into a concrete predicate without any formula variables, we have met all of the specification rule's antecedents, and we can use it to define two constants: the subset and its inclusion into the ambient set. The sets of natural numbers, integers, lists, and co-lists are all constructed in this way. More generally, given any member $s \in \mathsf{Pow}(A)$, we use the specification rule to turn it into a "real set" via the constant $\mathsf{m2s}(s)$ of the set sort. This set is injected into A by the map $\mathsf{minc}(s) : \mathsf{m2s}(s) \to A$.

The following isset predicate, connecting a member (s) to a set (B, given implicitly in i's sort) is also occasionally useful:

$$\mathsf{isset}(i : B \to A, s \in \mathsf{Pow}(A)) \overset{\text{def}}{\Leftrightarrow} \mathsf{Inj}(i) \wedge \mathsf{image}(i, B) = s$$

5.2.3 Defining functions by cases in SEAR

Despite its power, SEAR is a set theory and is not designed for usage in a theorem prover. This causes inconvenience but is mostly possible to overcome by developing automatic tools. Here we present such a problem due to the lack of type theory in SEAR, as well as our solution.

Functions defined by "if ... else ..." statement plays an important role in formalization. Such a function takes a conditional predicate and two expressions, and gives one of them according to the truth value of the predicate. With simple types, it has the type $\mathsf{bool} \to \alpha \to \alpha \to \alpha$, where the predicate has type bool and α is the type of the expression.

It is possible to recover such a statement without types if everything lives uniformly on the same level. If the condition ϕ on elements of A is identified with the set $s := \{a \in A \mid \phi(a)\}$, as it can happen in some systems such as ZF, then the function "if ϕ then t else e" can be constructed as the set $f := \{(x, y) | (s \neq \emptyset \ \wedge \ y = t) \vee s = \emptyset \ \wedge \ y = e\}$. Applying such a function on x can be defined as $\mathsf{apply}(f, x) := \mathsf{UCHOICE}\ \{y \mid (x, y) \in f\}$. Here the $\mathsf{UCHOICE}$ is the unique choice operator that only singles out the only element of the set, as the set must be singleton. No actual Axiom of Choice is used. But we cannot directly "if ... then ... else ..." statement if the statement has to be captured by a formula. In such a case, such an expression will mix formulas and terms at the same level.

However, we often want to state the conditions using formulas. Moreover, in many cases, there is more than one case we are interested in. Effectively, we often want to define functions $f : A \to B$ with the behavior:

$$
\begin{aligned}
&\forall a \in A. \\
&\quad (\phi_1(a) \implies f(a) = t_1(a)) \wedge \\
&\quad (\phi_2(a) \implies f(a) = t_2(a)) \wedge \\
&\quad \cdots \\
&\quad (\mathsf{ELSE} \implies f(a) = t_{n+1}(a))
\end{aligned}
$$

where $\phi_1, \cdots, \phi_n$ are any formula with or without the free variable a, considered as predicates on a. Terms $t_1, \cdots, t_{n+1}$ may or may not involve a, but all of them are required to be a member of the same set B.

We hence implement an automation that takes a formula ψ as above, proves that the function described by the formula exists, and outputs the existential statement as a theorem, in the

form of:

$$
\begin{aligned}
&\mathsf{Vars}(\psi)\\
&\vdash \exists! f : A \to B.\\
&\quad \forall a \in A.\\
&\qquad (\phi_1(a) \implies f(a) = t_1(a)) \wedge\\
&\qquad (\neg\phi_1(a) \wedge \phi_2(a) \implies f(a) = t_2(a)) \wedge\\
&\qquad (\neg\phi_1(a) \wedge \neg\phi_2(a) \wedge \phi_3(a) \implies f(a) = t_3(a)) \wedge\\
&\qquad (\neg\phi_1(a) \wedge \cdots \wedge \neg\phi_n(a) \implies f(a) = t_{n+1}(a))
\end{aligned}
$$

Our method uses the function-defining theorem:

$$
\begin{aligned}
\{X,\ Y\} \vdash (\forall x \in X.\ \exists! y \in Y.\ \mathcal{R}[x_0 \in X, y_0 \in Y](x, y)) \implies\\
\exists! f : X \to Y.\ \forall x \in X.\ \mathcal{R}[x_0 \in X, y_0 \in Y](x, f(x))
\end{aligned}
$$

We present the following flow of constructions.

- Instantiate the X and Y in 5.2.3 above as A and B.
- Automatically prove a family of theorems, all in the same form:

$$
\begin{aligned}
\mathsf{Vars}(\psi),\ \phi_1(a) \vdash &\quad \exists! b \in B.\ \phi_1(a) \wedge t_1(a) = b\\
\mathsf{Vars}(\psi),\ \neg\phi_1(a) \wedge \phi_2(a) \vdash &\quad \exists! b \in B.\ \neg\phi_1(a) \wedge \phi_2(a) \wedge t_2(a) = b\\
\mathsf{Vars}(\psi),\ \neg\phi_1(a) \wedge \neg\phi_2(a) \wedge \phi_3(a) \vdash &\quad \exists! b \in B.\ \neg\phi_1(a) \wedge \neg\phi_2(a) \wedge \phi_3(a) \wedge t_2(a) = b\\
&\cdots\\
\mathsf{Vars}(\psi),\ \neg\phi_1(a) \wedge \cdots \wedge \neg\phi_n(a) \vdash &\quad \exists! b \in B.\ \neg\phi_1(a) \wedge \cdots \wedge \neg\phi_n(a) \wedge t_{n+1}(a) = b
\end{aligned}
$$

  Their proofs are easy: as the terms $t_k(a)$ are all unique (explicitly, they are all specializations of $\forall a \in A.\ \exists! a' \in A.\ a' = a$), we simply move the assumptions into the "$\exists!$" quantifier.

- Taking the body of the conclusion of each theorem proved in the previous step, disjunct them, and putting back the "$\exists!$" quantifier yields the formula:

$$
\begin{aligned}
&\exists! b \in B.\\
&\ \phi_1(a) \wedge t_1(a) = b \vee\\
&\neg\phi_1(a) \wedge \phi_2(a) \wedge t_2(a) = b \vee\\
&\cdots \vee\\
&\neg\phi_1(a) \wedge \cdots \wedge \neg\phi_n(a) \wedge t_{n+1}(a) = b
\end{aligned}
\qquad (*)
$$

  Then all the above theorems are still true if all their conclusion are replaced by this big disjunction. This is easy to prove by proving the equivalence between their original

conclusion and $(*)$ under each of their assumptions. Indeed, as all but one of the disjunct in $(*)$ contains the negation of some conjunct in the original conclusion, precisely one disjunct holds, and the other ones will be automatically turned into falsity.

- Now we have a family of theorems, all with the same conclusion. We can put them together into one theorem by disjuncting their assumptions, yielding:

$$\mathsf{Vars}(\psi),\ \phi_1(a) \vee (\neg\phi_1(a) \wedge \phi_2(a)) \vee \cdots \vee (\neg\phi_1(a) \wedge \cdots \wedge \neg\phi_n(a)) \vdash (*)$$

- Prove the disjunction as the antecedent of the last step is equivalent to truth. Hence we prove the theorem $\mathsf{Vars}(\psi) \vdash (*)$. Now this theorem is in the form of the first half of the implication, which appears in the conclusion of 5.2.3. We instantiate it by taking the $\mathcal{R}[a \in A, b \in B]$ as the body of the quantification of $(*)$. Modens ponens immediately proves:

$$\mathsf{Vars}(\psi) \vdash \exists! f : A \to B. \forall a \in A.\ (*)$$

- The remaining task is to normalize the disjunction on the conclusion side of the above theorem into a conjunction of implications. This is done by realizing the equivalence

$$\begin{aligned}
(*) &\Leftrightarrow && \top \implies (*) && (5.2)\\
&\Leftrightarrow && \phi_1(a) \vee \neg\phi_1(a) \wedge \phi_2(a) \vee \cdots \vee \neg\phi_1(a) \wedge \cdots \wedge \neg\phi_n(a) \implies (*) && (5.3)\\
&\Leftrightarrow && (\phi_1(a) \implies (*)) \wedge \cdots \wedge (\neg\phi_1(a) \wedge \cdots \wedge \neg\phi_n(a) \implies (*)) && (5.4)
\end{aligned}$$

- Each conjunct in the final step of the equivalence can be rewritten into a much simpler form, where the right-hand side of each implication consists of an equation only. This is done by simplifying away all the disjuncts in $(*)$ that involve the disjunction of any conjunct of the antecedent and eliminating the conjunct that holds in both the antecedent and the conclusion by simplifying it into $\top$. Regarding the first conjunct $\phi_1(a) \implies (*)$, assuming $\phi_1(a)$, then all the disjuncts starting with $\neg\phi_1(a)$ will be eliminated, and the $\phi_1(a)$ in the first disjunct will disappear as well because it is simplified into $\top$. This completes the normalization.

  Such a tool is useful when we want to define a function by explicitly specifying where the elements in the domain goes to. For instance, if we want to define the function $2 \times 2 \to 2$, giving the

$$\forall v.\ v = (\bot_2, \bot_2) \implies f(v) = \bot_2 \wedge \mathsf{ELSE} \implies f(v) = \top_2$$

  Gives us the disjunction operation on the two-element set.

5.2.4 Inductive and Coinductive Definitions

We experiment with inductive definitions by mechanizing induction on natural numbers, finite sets, and lists, and with coinductive definitions by constructing co-lists.

Natural Numbers, Finite Sets, and Lists

Our system implements a version of Harrison's [24] inductive relation definition package. After that implementation, to define an inductive subset, we just need to provide the inductive clauses.

5.2.8 Example. As discussed in that last section, we should apply induction to extract the set of natural numbers out of the set $\mathbb{N}_0$ provided by Axiom 4. By providing the clauses

$$(n = \mathsf{z}_0 \implies \mathsf{IN}(n, N)) \ \wedge \ \forall n_0.\, \mathsf{IN}(n_0, N) \wedge n = \mathsf{s}_0(n_0) \implies \mathsf{IN}(n, N)$$

the inductive tools find us a subset of $\mathbb{N}$, as an element of $\mathsf{Pow}(N_0)$, contains only z_0, and every iteration of successor function applied on it.

Using theorem 5.1 together with the specification rule, we extract the subset of $\mathbb{N}_0$, which consists of elements in N, as a constant term $\mathbb{N}$ of the set sort. We call the lifted zero element and successor map 0 and SUC respectively, with SUC obtained by specializing the following lemma with the inclusion from $\mathbb{N}_0$:

$$\begin{aligned}&\forall A\ A_0\ (i : A \to A_0)\ (f_0 : A_0 \to A_0).\\ &\quad \mathsf{Inj}(i) \ \wedge \ (\forall a_1 . \exists a_2.\ f_0(i(a_1)) = i(a_2)) \implies\\ &\qquad \exists! f : A \to A.\ \ \forall a \in A.\ \ i(f(a)) = f_0(i(a))\end{aligned}$$

The constructed $\mathbb{N}$ then can be shown to satisfy the standard induction principle.

$$\begin{aligned}&\mathcal{F}[\mathsf{mem}(\mathbb{N})](0) \wedge (\forall n \in \mathbb{N}.\ \ \mathcal{F}[\mathsf{mem}(\mathbb{N})](n) \implies \mathcal{F}[\mathsf{mem}(\mathbb{N})](\mathsf{SUC}(n))) \implies\\ &\quad \forall n \in \mathbb{N}.\ \ \mathcal{F}[\mathsf{mem}(\mathbb{N})](n)\end{aligned}$$

By instantiating the formula variable $\mathcal{F}$ with concrete properties, we apply the above to perform inductive proofs for ordering and natural number arithmetic. We later use such theorems together with quotient lemmas to construct the set of integers. Recursion on $\mathbb{N}$ is constructed with a further application of inductive relations. As we want to eventually define a function $\mathbb{N} \to X$, that application constructs a subset of $\mathbb{N} \times X$.

Also inductively, we define the predicate $\mathsf{isFinite}$ on members of some set X's power set. The empty subset $\mathsf{Empty}(X)$ is finite, and if $s \in \mathsf{Pow}(X)$ is finite, then the set $\mathsf{Ins}(x, s)$, which inserts x into s, is finite for any $x \in X$. Similar to the counterpart of natural numbers, the principle

of induction on the finiteness of a set is proved as:

$$\begin{aligned}&\mathcal{F}[\mathsf{mem}(\mathsf{Pow}(X))](\mathsf{Empty}(X)) \wedge \\ &(\forall x\ (xs_0 \in \mathsf{Pow}(X)).\ \mathcal{F}[\mathsf{mem}(\mathsf{Pow}(X))](xs_0) \implies \mathcal{F}[\mathsf{mem}(\mathsf{Pow}(X))](\mathsf{Ins}(x, xs_0))) \implies \\ &\quad \forall xs \in \mathsf{Pow}(X).\ \mathsf{isFinite}(xs) \implies \mathcal{F}[\mathsf{mem}(\mathsf{Pow}(X))](xs)\end{aligned}$$

We define a relation $\mathsf{Pow}(X) \nrightarrow \mathbb{N}$ relating a subset of X to its cardinality. By induction on finiteness, we prove each subset is related to a unique natural number, which gives us a function $\mathsf{Pow}(X) \to \mathbb{N}$ that sends a finite subset to its cardinality and sends any infinite subset to 0. The output of the function applied on $s \in \mathsf{Pow}(X)$ is denoted as $\mathsf{Card}(s)$. To build lists over a set X as an "inductive type", we firstly define the subset of $\mathsf{Pow}(\mathbb{N} \times X)$ which encodes a list, such sets are finite sets of the form $\{(0, x_1), \cdots, (n, x_n)\}$. The base case of the induction is the empty subset of $\mathsf{Pow}(\mathbb{N} \times X)$, and the step case inserts the set s started with by the pair $(\mathsf{Card}(s), x)$. Using the same approach we constructed $\mathbb{N}$, we form $\mathsf{List}(X)$. It is then straightforward to prove the list induction principle and define the usual list operations like taking the head, tail, n-th element of the list, and map, etc.

5.2.5 Co-lists

Following the HOL approach, we construct co-lists over sets X, by using maps $\mathbb{N} \to X + 1$ as representatives. The codomain is regarded as X option, whose members either have the form $\mathsf{SOME}(x)$ for $x \in X$, or $\mathsf{NONE}(X)$. First, by dualizing the argument we used to define inductive predicates, we define a coinductive predicate on members ($f \in (X + 1)^{\mathbb{N}}$) expressing that such a member captures a co-list, and we collect the subset where this predicate holds, defining $\mathsf{list_c}(X)$, just as we did for constructing inductive types. Every term of $\mathsf{list_c}(X)$ has a representative: it is either the constant function mapping to $\mathsf{NONE}(X)$, corresponding to the empty co-list $\mathsf{Nil_c}(X)$, or it is the function obtained by attaching an element $x \in X$ to an existing function encoding a co-list. Almost all the HOL4 definitions can be readily translated into SEAR. The only exception is we cannot write expressions such as $\mathsf{THE}(\mathsf{Hd_c}(l))$. Here $\mathsf{Hd_c}$ is the function that returns $\mathsf{SOME}(x)$ when l is a co-list with element x at its front. If l is the empty co-list, then $\mathsf{Hd_c}(l) = \mathsf{NONE}$. In HOL4, THE is the left-inverse of SOME; in SEAR, our (set) parameter X may be empty, and so there is no general value (even if unspecified) for the head of the co-list. So far, this has not been an obstacle in any of our proofs. The HOL proof of the key co-list principle, which states that two co-lists $l_1, l_2 \in \mathsf{list_c}(X)$ are equal if and only

if they are connected by a bisimulation relation R, translates into SEAR, yielding:

$$
\begin{array}{l}
l_1 = l_2 \Leftrightarrow \\
\exists R : \mathsf{list_c}(X) \nrightarrow \mathsf{list_c}(X). \\
\quad \mathsf{Holds}(R, l_1, l_2) \wedge \\
\quad \forall l_3\ l_4 \in \mathsf{list_c}(X).\ \ \mathsf{Holds}(R, l_3, l_4) \implies \\
\qquad\qquad\qquad (l_3 = \mathsf{Nil_c}(X) \wedge l_4 = \mathsf{Nil_c}(X)) \vee \\
\qquad\qquad\qquad \exists (h \in X)\ t_1\ t_2.\ \mathsf{Holds}(R, t_1, t_2) \wedge l_3 = \mathsf{Cons_c}(h, t_1) \wedge l_4 = \mathsf{Cons_c}(h, t_2)
\end{array}
$$

where $\mathsf{Nil_c}(X)$ is the empty co-list over X, and $\mathsf{Cons_c}(h, t)$ is the co-list built by putting element $h \in X$ in front of co-list t. We can perform coinductive proofs on co-lists by the theorem above. For instance, the above helps to prove that $\mathsf{Map_c}$ function, with the usual definition, is functorial.

5.2.6 Quotients

We develop a machinery for constructing quotient sets, which correspond to quotient types in type theory, in our SEAR system. The same approach is also formalized in ETCS. We notice that the quotient package implemented in HOL with its description in [27] cannot be re-implemented into either of these two systems. This treatment of quotient appeals to representatives of equivalence classes, singled out by the choice operator. As every HOL type is non-empty, the choice operator can return an arbitrary element on the input of the empty set. The counterpart of a HOL type here is a SEAR set, and is not guaranteed to own an element. Therefore, we cannot obtain the choice operator even if we assume the Axiom of Choice. Another method, described by Paulson [40], avoids the Axiom of Choice by not using any representative, but using the sets as equivalence classes directly. However, this method requires the application of a form of iterated big union, which lacks a nice presentation in SEAR without the set-builder syntax. We build our approach to avoid all these problems. We only consider full equivalence relations, since partial equivalences become full by restricting their domains. Our approach does not require any form of the Axiom of Choice. We present the standard example of constructing the set of integers and transferring natural number operations onto the counterpart of integers along with our introduction to the general method. The formalization of the quotient group as in the next section also follows this construction.

Given a binary relation R on a set A, the function symbol application $\mathsf{rsi}(R, a) \in \mathsf{Pow}(A)$ for $a \in A$ is called the *relational image* of a, it consists of elements a' such that $\mathsf{Holds}(R, a, a')$. We will eventually form a term $Q : \mathsf{set}$ by extracting the subset of $\mathsf{Pow}(A)$ whose elements are of form $\mathsf{rsi}(R, a)$. Such a Q will be injected into $\mathsf{Pow}(A)$ via an inclusion map. We want the *quotient map* to send an element in A to its equivalence class in Q. For a map $i : Q \to \mathsf{Pow}(A)$ which does capture an inclusion from a quotient, the corresponding quotient map exists. We

define "i as a quotient of the relation R" by:

$$\begin{aligned}&\mathsf{Quot}(R : A \nrightarrow A, i : Q \to \mathsf{Pow}(A)) \Leftrightarrow \\ &\quad \mathsf{Inj}(i) \ \wedge \ \forall s \in \mathsf{Pow}(A).\ (\exists q \in Q.\ s = i(q)) \Leftrightarrow \exists a \in A.\ s = \mathsf{rsi}(R, a)\end{aligned}$$

That is, i injects to the set of relational images of R. Then the quotient map is obtained by two steps: Take an element $a \in A$, we send it to $\mathsf{rsi}(R, a) \in \mathsf{Pow}(A)$ (the function does this is called $\mathsf{Rsi}(R)$). This set is in the image of i, so there is a unique element in q sent to it. The q can be found by applying the left inverse of the injection i. There is a HOL function that takes a map and outputs its left inverse if it is an injection from a non-empty set, and outputs a constant function sending everything to a fixed arbitrary place otherwise. As the same thing happens to the choice operator, the exact same function cannot be constructed in SEAR. But this is easy enough to fix by requiring another input, that is, an element in the domain of the injection. For i from X to Y, and given an element $x \in X$, we define $\mathsf{LINV}(i, x)(y)$ to map $y \in Y$ to x_0 if x_0 is the unique element mapped to y, or to x otherwise, so it is the left inverse of i being an injection. We write the quotient map $\mathsf{Abs}(R, i, q_0)$, and denote the output of this map applied to an element $a \in A$ as $\mathsf{abs}(R, i, q_0, a)$.

5.2.9 Example. We construct the set of integer $\mathbb{Z}$ by quotient the set $\mathbb{N} \times \mathbb{N}$. The relation $R_\mathbb{Z}$ on $\mathbb{N} \times N$ is $(n_1, n_2) \sim (n_3, n_4)$ iff $n_1 + n_4 = n_2 + n_3$. Using 5.1, we take out the subset of $\mathsf{Pow}(\mathbb{N} \times \mathbb{N})$ consisting of equivalence classes of $R_\mathbb{Z}$ as the set $\mathbb{Z}$, and call the inclusion map $i_\mathbb{Z} : \mathbb{Z} \to \mathsf{Pow}(\mathbb{N} \times \mathbb{N})$. The integer zero $0_\mathbb{Z}$ is the element in $\mathbb{Z}$ injected into the equivalence class of the natural number zero $0_\mathbb{N}$. Then the quotient map is $\mathsf{LINV}(i_\mathbb{Z}, 0_\mathbb{N}) \circ \mathsf{Rsi}(R_\mathbb{Z})$

Now we consider lifting functions on the original set into a function on the quotient set. Such a process is always done with the spirit that we can factor through a function through a quotient once the factorization can be proved to be well-defined. We simply make the application of this spirit explicit.

A function $f : A \to B$ *respects* to the relation $R : A \nrightarrow A$ when we have $f(a_1) = f(a_2)$ once $\mathsf{Holds}(R, a_1, a_2)$, denoted $\mathsf{resp1}(f, R)$. Write $\mathsf{ER}(R)$ for R an equivalence relation, we prove:

$$\begin{aligned}&\mathsf{ER}(R) \wedge \mathsf{resp1}(f, R) \wedge \mathsf{Quot}(R, i) \implies \\ &\quad \forall q_0 \in Q.\ \exists! \overline{f} : Q \to B.\ \forall a \in A.\ \overline{f}(\mathsf{abs}(R, i, q_0, a)) = f(a)\end{aligned}$$

This is exactly the "universal property of quotient". The naive theorem already works well for lifting functions with a single argument. By passing from a predicate to a function to the two-element set (Called 2, as in ETCS, it has two elements $\top_I$ and $\bot_I$), which can be constructed in SEAR, it allows us to lift predicates as well.

5.2.10 Example. Consider lifting the inductively-defined argument $\mathsf{Even}(n)$, which means

the natural number $n \in \mathbb{N}$ is even. We define $\mathbb{N} \times \mathbb{N} \mapsto 2$ using 5.2.3 by $(n_1, n_2) \mapsto \top_I$ if they have the same parity, and $\bot_I$ otherwise. Specialize 5.2.6 with it gives a function $\mathbb{Z} \to 2$, expressing evenness of integers.

Functions such as negation has sort $\mathbb{Z} \to \mathbb{Z}$, we may just use 5.2.6 for lifting it as well, from a function $\mathbb{N} \times \mathbb{N} \to \mathbb{Z}$. But for the sack of convenience and generality, we prefer to have a theorem where we are only required to check conditions on the original set. This requires us to consider functions that respect the relation on the domain and codomain. We define $\mathsf{resp}(f : A \to B, r_1, r_2)$ as if $\mathsf{Holds}(r_1, a_1, a_2)$, then $\mathsf{Holds}(r_2, f(a_1), f(a_2))$. Given $f : A \to B$ where A and B are quotiented into Q_1 and Q_2, we want the induced function $Q_1 \to Q_2$ to send an equivalence class that a belongs to the equivalence class $f(a)$ belongs to. We say these two equivalence classes are related by the relation $\mathsf{rext}(f, r_1, r_2)$. The following theorem addressing equivalence relations on both the domain and codomain is very helpful.

$$\begin{aligned}\vdash \forall A\ B\ (f : A \to B)\ (r_1 : A \nrightarrow A)\ (r_2 : B \nrightarrow B)\ Q_1\ Q_2\ (i_1 : Q_1 \to \mathsf{Pow}(A))\ (i_2 : Q_2 \to \mathsf{Pow}(B)).\\ \mathsf{ER}(r1) \wedge \mathsf{ER}(r2) \wedge \mathsf{resp}(f, r_1, r_2) \wedge \mathsf{Quo}(r_1, i_1) \wedge \mathsf{Quo}(r_2, i_2) \implies\\ \exists! q_f : Q_1 \to Q_2.\ \forall q_1 : \mathsf{mem}(Q_1).\ \mathsf{Holds}(\mathsf{rext}(f, r_1, r_2), \mathsf{App}(i_1, q_1), \mathsf{App}(i_2 \circ q_f, q_1))\end{aligned}$$

The original quotient theorem 5.2.6 is realized as the case when r_2 is the trivial equivalence relation defined by equality.

5.2.11 Example. The function $z \mapsto -z$ on $\mathbb{Z}$ is defined from the function $f_0 : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N}$, with $(n_1, n_2) \mapsto (n_2, n_1)$. We prove only on the level of elements of $\mathbb{N}$, to get $\mathsf{resp}(f_0, R_\mathbb{Z}, R_\mathbb{Z})$. Specializing 5.2.6 directly gives us a function $\mathbb{Z} \to \mathbb{Z}$. This function is the negation operator.

There are, however, many functions with multiple arguments, and some of them have outputs in a quotient. That suggests us to give a treatment to function from a product of quotients. We reduce such cases into a case where 5.2.6 is applicable. This is achieved by realizing the product of quotients as a quotient as well in the following way: Given two relations R_1 on A and R_2 on B, we define their product relation as:

$$\mathsf{Holds}(\mathsf{prrel}(R_1, R_2), (a_1, b_1), (a_2, b_2)) \Leftrightarrow \mathsf{Holds}(R_1, a_1, a_2) \wedge \mathsf{Holds}(R_2, b_1, b_2)$$

And given quotients $i_1 : Q_1 \to \mathsf{Pow}(A), i_2 : Q_2 \to \mathsf{Pow}(B)$ of R_1 and R_2, we define a map $\mathsf{ipow2}(i_1, i_2) : Q_1 \times Q_2 \to \mathsf{Pow}(A \times B)$ such that for every pair $(a, b) \in Q_1 \times Q_2$, we have:

$$\mathsf{IN}((a, b), \mathsf{ipow2}(i_1, i_2)(q_1, q_2)) \Leftrightarrow \mathsf{IN}(a, i_1(q_1)) \wedge \mathsf{IN}(b, i_2(q_2))$$

If R_1 and R_2 are both equivalence relations, the relation $\mathsf{prrel}(R_1, R_2)$ and the inclusion map $\mathsf{ipow2}(i_1, i_2)$ form a new quotient. This would allow us to borrow the previous quotient theorems.

5.2.12 Example. The addition function on $\mathbb{Z}$ is obtained by lifting the function $f : (\mathbb{N}\times\mathbb{N})\times(\mathbb{N}\times\mathbb{N}) \to \mathbb{N}\times\mathbb{N}$ with $((n_1, n_2), (n_3, n_4)) \mapsto (n_1{+}n_3, n_2{+}n_4)$. We have $\mathsf{Quo}(\mathsf{prrel}(R_\mathbb{Z}, R_\mathbb{Z}), \mathsf{ipow2}(i_\mathbb{Z}, i_\mathbb{Z}))$ by the result above, and we already have $\mathsf{Quo}(R_\mathbb{Z}, i_\mathbb{Z})$. It is easy to prove $\mathsf{resp}(f, \mathsf{prrel}(R_\mathbb{Z}, R_\mathbb{Z}), R_\mathbb{Z})$. Note that the domain of $\mathsf{ipow2}(i_\mathbb{Z}, i_\mathbb{Z})$ is $\mathbb{Z}\times\mathbb{Z}$. Applying 5.2.6 with these conditions gives us the addition function $\mathbb{Z}\times\mathbb{Z} \to \mathbb{Z}$.

The group operations in a quotient group, to be constructed in the next section, are also done as above.

5.2.7 Group Theory

Many mathematical results look neater in theorem-provers based on dependent type theory (DTT), since instead of assuming complicated predicates, we can internalize those predicates as types, thereby shortening the statement. By formalizing some group theory, we demonstrate that we can prove similarly neat versions of statements in our simple logic.

We encode a group with underlying set G as an element of $\mathsf{Grp}(\mathsf{G})$. Such a set is constructed from the comprehension schema which injects to the subset of the product $G^{G\times G}\times G^G \times G$ satisfying the usual group axioms. For a group $g \in \mathsf{Grp}(G)$, also by comprehension, we construct the set of its subgroups $\mathsf{sgrp}(g)$ as injected into $\mathsf{Pow}(G)$, and the set of its normal subgroups $\mathsf{nsgrp}(g)$ that injects to $\mathsf{sgrp}(g)$. The problem we mentioned as a pain point for creating a function symbol producing an equalizer object happens here again. As groups are encoded by members of sets, it is possible to compare if two groups are equal, *e.g.*, $g_1 = g_2$, with $g_1, g_2 \in \mathsf{Grp}(G)$. However, if $h_1 \in \mathsf{sgrp}(g_1)$ and $h_2 \in \mathsf{sgrp}(g_2)$, we cannot write $h_1 = h_2$ because such an equality will not type check. But here, we hold this to be appropriate because equality is not the correct way to compare abstract structures such as groups. Even if we wanted to work with equality on groups g_1, g_2, we should compare their representatives or define transferring functions like the ones of sort $\mathsf{sgrp}(g_1) \to \mathsf{sgrp}(g_2)$, which map a subgroup of g_1 to a subgroup of g_2.

Note that the sort $\mathsf{mem}(\mathsf{sgrp}(g))$ is a naturally occurring example for a top-level term, namely g, to occur in a sort. That means we cannot instantiate a formula variable $\mathcal{F}[g_0 \in \mathsf{Grp}(G)](g)$ to be a formula with a free variable of sort $\mathsf{mem}(\mathsf{sgrp}(g))$, because such a formula will become ill-formed with the quantification on g.

For a normal subgroup $N \in \mathsf{nsgrp}(g)$, the underlying set of the quotient group $\mathsf{qgrp}(N)$ has as its underlying set the set of all right cosets of N. The function symbol qgrp only needs to take the group N as argument, since the group being quotiented is contained in the sort information of N. The quotient homomorphism $\mathsf{qhom}(N)$ is a member of $\mathsf{ghom}(g, \mathsf{qgrp}(g))$ of all homomorphisms between the original group and the quotient. Its underlying function $\mathsf{homfun}(\mathsf{qhom}(N))$ sends a group element to its coset.

By construction, each underlying function of a homomorphism respects the relation induced by its kernel. Then the first isomorphism theorem can be obtained by instantiating the quotient mapping theorem as in the last section, giving

$$\begin{array}{l}\forall G_1\ G_2\ \ g_1 \in \mathsf{Grp}(G_1)\ \ g_2 \in \mathsf{Grp}(G_2)\ \ f \in \mathsf{ghom}(g_1, g_2).\\ \quad \exists!\ \overline{f} \in \mathsf{ghom}(\mathsf{qgrp}(\ker(f)), g_2).\\ \qquad \mathsf{Inj}(\mathsf{homfun}(\overline{f}))\ \wedge\ \mathsf{homfun}(\overline{f}) \circ \mathsf{qmap}(\ker(f)) = \mathsf{homfun}(f)\end{array} \tag{5.5}$$

This is a nice illustration of the strengths of the "DTT style".

Discussion

Our approach to group theory is very different from its counterpart in HOL. Firstly, the HOL type α group is inhabited by values that must record their underlying carrier set. Secondly, the HOL quotient group function takes two α groups and outputs a term of $(\alpha \to \mathsf{bool})$ group, which is proved to satisfy the group axioms if the first term satisfies the group axioms and the second term is a normal subgroup. Further, as HOL types cannot depend on terms, we certainly cannot construct the type of all homomorphisms between two groups. There is actually a trade-off between choosing the HOL style and the DTT style of stating theorems. Whereas the first isomorphism theorem is clearly better in DTT style (5.5), the second and third isomorphism theorems in DTT style can look complicated, with a great deal of coercions happening under the covers.[2] Since the HOL quotient group only takes two groups of the same type, we can use exactly the same term for the ambient group and its subgroup and do not need to construct different terms to regard the same group as subgroups of an ambient group. In this case, the convenience of the HOL style (using assumptions) is evident. We can choose each style in our system, so users can try both approaches and compare them. To find the best form of a statement, we may try combining the two approaches: we do not always have to create a subset once we come up with a new property, but we may use them as assumptions as well.

5.2.8 Modal Model Theory

In recent work, we developed a mechanization of some basic modal logic theory [51]. While defining the notion of being preserved under simulation, we observed that if a property of a modal formula is defined in terms of the behavior of the formula on all models, then such a property cannot be faithfully captured by HOL. Such an issue can be resolved by choosing a dependent sorted foundation and doing the proof in our logic. We demonstrate this here by mechanizing the proof that characterizes formulas preserved under simulation as those are equivalent to a positive existential formula in SEAR.

[2] Of course, DTT *systems* offer the ability to write statements in HOL's predicate-heavy style as well.

Using roughly the general method introduced at the end of Harrison [24], we first construct the "type" (actually a set in SEAR) of modal formulas over variables drawn from the set V. We then denote the set of modal formulas over V as $\mathsf{form}(V)$. A Kripke model on a set W of such formulas is an element of $\mathsf{Pow}(W \times W) \times \mathsf{Pow}(V)^W$ (written as $\mathsf{model}(W, V)$ in the following paragraphs). The first component encodes the model's reachability relation, while the second encodes the variable valuation. Satisfaction of modal formulas can then be defined in the standard way, and if ϕ is satisfied at w in the model M, we write $M, w \Vdash \phi$.

The two key definitions of this proof are that of simulation, and of being preserved under simulation (written as PUS below). A simulation between two models M_1 and M_2 with underlying world sets W_1 and W_2 is a relation $R : W_1 \nrightarrow W_2$. It is a concept describing that once we can make a transition from w_1 in M_1, a corresponding transition can be taken in M_2 from any element that w_1 is related to. The is identical to its counterpart in HOL, and the predicate is written $\mathsf{Sim}(R, M_1, M_2)$. The definition of "preserved under simulation" is more interesting. This refers to the property for a formula's truth value to be preserved in any two models with a simulation between them. In the textbook description, the default foundation is ZF, so there is no notion of the "type" of a model. However, in simple type theory, the collection of all models ranges over all types. Therefore, the fact that we cannot quantify over types in HOL does raise an issue. In our previous formalization, we did not have any option but to encode a strictly weaker version by adding an extra type parameter, and defining "preserved under simulation between models of a certain type". Thankfully, due to the fact that we are now allowed to quantify over sets in SEAR, we can state this definition faithfully. As it is supposed to be, in SEAR, such a definition is a predicate symbol taking a formula only as its argument.

$$
\begin{aligned}
&\forall V\ (\phi \in \mathsf{form}(V)).\\
&\quad \mathsf{PUS}(\phi) \Leftrightarrow\\
&\quad \forall W_1\ W_2\ (R : W_1 \nrightarrow W_2)\ (w_1 \in W_1)\ (w_2 \in W_2)\\
&\qquad (M_1 \in \mathsf{model}(W_1, A))\ \ (M_2 \in \mathsf{model}(W_2, A)).\\
&\qquad\quad \mathsf{Sim}(R, M_1, M_2) \wedge \mathsf{Holds}(R, w_1, w_2)\ \wedge\ M_1, w_1 \Vdash \phi\ \Longrightarrow\ M_2, w_2 \Vdash \phi
\end{aligned}
$$

Quantification over sets also makes the formalized notion of formula equivalence simpler: two formulas are equivalent if their truth value agrees on all models. Again, now we are not restricted to only being able to define "agree on a particular type of models", but can simply define the predicate on two formulas, denoted $\phi_1 \sim \phi_2$. Under these definitions, the proofs of both directions of theorem 2.78 in [16] can be faithfully translated, yielding the two formal

statements:

$$\forall V\ (\phi \in \mathsf{form}(V))\ (\phi_0 \in \mathsf{form}(V)).\ \mathsf{PE}(\phi_0) \wedge \phi \sim \phi_0 \implies \mathsf{PUS}(\phi)$$

$$\forall V\ (\phi \in \mathsf{form}(V)).\ \mathsf{PUS}(\phi) \implies \exists \phi_0 \in \mathsf{form}(V).\ \mathsf{PE}(\phi_0) \wedge \phi \sim \phi_0$$

Clearly, the two directions can be put together into an if-and-only-if, hence giving the full form of the characterization theorem, which cannot even be stated in HOL.

$$\forall V\ (\phi \in \mathsf{form}(V)).\ \mathsf{PUS}(\phi) \Leftrightarrow \exists f_0 \in \mathsf{form}(V).\ \mathsf{PE}(\phi_0) \wedge \phi \sim \phi_0)$$

Such a theorem can be stated in ETCS as well, but it does not mean that it is provable in ETCS. As an intermediate step of the proof in [16], we are required to define the positive consequence of a formula ψ, i.e., the set $\{\phi \mid \psi \vDash \phi \ \wedge$ an extra condition$\}$. The $\vDash$ here means "semantical consequence", which means ϕ is true if ψ is true, in any model. Again, this requires quantification on models. But also not that this set requires the theorem 5.1 to construct, so the quantification appears in the formula to be instantiated into the formula variable there. Therefore, a translation from this proof does require *unbounded* comprehension, where ETCS is not sufficient.

5.2.9 Existence of Large Sets

Whereas iterating the procedure of taking the power set by infinite times is impossible in HOL due to foundational issues, the collection axiom schema in SEAR makes it possible. The statement of the SEAR collection axiom is formalized as:

5.2.13 Axiom 5.

$$\begin{aligned}&\exists B\ Y\ (p : B \to A)\ (M : B \nrightarrow Y).\\ &\quad (\forall S\ (i : S \to Y)\ (b \in B).\\ &\qquad \mathsf{isset}(i, \{y \mid \mathsf{Holds}(M, b, y)\}) \implies \mathcal{F}[a_0 : \mathsf{mem}(A), X_0 : \mathsf{set}](p(b), S)) \wedge\\ &\quad (\forall (a \in A)\ X.\ \mathcal{F}[\mathsf{mem}(A), \mathsf{set}](a, X) \implies \exists b.\ p(b) = a)\end{aligned}$$

with $\mathcal{F}[a_0 : \mathsf{mem}(A), X_0\mathsf{set}]$ a formula variable, to be instantiated to be a predicate on an element of A and a set.

Using this axiom, we will prove:

$$\forall A.\ \exists P.\ \forall n \in \mathbb{N}.\ \exists i : \mathsf{Pow}^n(A) \to P.\ \mathsf{Inj}(i)$$

Here the $\mathsf{Pow}^n(A)$ is "the" n-th power set of A. Note that the induction principle on natural numbers does not allow us to take a set as an argument, and does not allow the output to be

a set as well. To create this function symbol, we start by defining a predicate $\mathsf{nPow}(n, A, B)$, which means B is *an* n-th power set of A. We then prove such B is unique up to bijection, hence the specification rule applies. In the following, we write $\mathsf{P}(s) \in \mathsf{Pow}(\mathsf{Pow}(A))$ for the set of subsets of $s \in \mathsf{Pow}(A)$. For $s_1 \in \mathsf{Pow}(A)$ and $s_2 \in \mathsf{Pow}(B)$, we write $|s_1| = |s_2|$ for s_1 and s_2 have the same cardinality. We write $\mathsf{Whole}(A) \in \mathsf{Pow}(A)$ as the subset of A consisting of all members of A.

We define $\mathsf{nPow}(n, A, B)$ if there exists a set X and a function $f : X \to \mathbb{N}$ such that $|f^{-1}(0)| = |\mathsf{Whole}(A)|$, $|f^{-1}(n)| = |\mathsf{Whole}(B)|$, and for each $n_0 < n$, $|f^{-1}(n_0^+)| = |\mathsf{P}(f^{-1}(n_0))|$. Such a function f records a sequence of power set relations, in this case, we write $\mathsf{nPowf}(n, A, B, f)$. By induction on n_0, $\mathsf{nPow}(n, A, B, f)$, implies $\mathsf{nPow}(n_0, A, \mathsf{m2s}(f^{-1}(n_0)), f)$ for each $n_0 \leq n$.

If $\mathsf{nPow}(n, A, B_1)$ and $\mathsf{nPow}(n, A, B_2)$, we can infer B_1 and B_2 have the same cardinality by induction on n. The base case is trivial. Assume $f_1 : X_1 \to \mathbb{N}$ witnesses $\mathsf{nPow}(n^+, A, C_1)$ and $f_2 : X_2 \to \mathbb{N}$ witnesses $\mathsf{nPow}(n^+, A, C_2)$, as $n < n^+$, we have f_1, f_2 witness that their preimage at n is an n-th powerset of A, and hence by inductive hypothesis has the same cardinality. Therefore, the cardinality of C_1 and C_2 are equal as power sets of sets with the same cardinality.

Now we prove the existence of these iterated power sets. Suppose we have $\mathsf{nPowf}(n, A, B, f_0 : X \to \mathbb{N})$, we construct $f' : \mathsf{Pow}(X + 1) \to \mathbb{N}$ such that $\mathsf{nPowf}(n^+, A, \mathsf{Pow}(B), f')$. Define $f : X \to \mathbb{N}$ such that as if $f_0(x) \leq n$ then $f(x) = f_0(x)$, else $f(x) = n^{++}$, then we have $\mathsf{nPowf}(n, A, B, f : X \to \mathbb{N})$, and n^+ is not in the range of f. According to the definition of nPow, there exists an injection $B \to X$, and thus an injection $i : \mathsf{Pow}(B) \to \mathsf{Pow}(X)$. We define the function $f' : \mathsf{Pow}(X + 1) \to \mathbb{N}$ as:

$$f'(s) = \begin{cases} f(x) & \text{if } s = \{\mathsf{SOME}(x)\} \\ n^+ & \text{if } \exists xs \in \mathsf{Pow}(X).\ i(xs) = s_0 \wedge s = \{\mathsf{NONE(X)}\} \cup s_0 \\ n^{++} & \text{else} \end{cases}$$

It follows that $|f'^{-1}(n_0)| = |f^{-1}(n_0)|$ for $n_0 \leq n$, and the preimage of n^+ is a copy of $\mathsf{Pow}(B)$, so f' witnesses $\mathsf{Pow}(B)$ is the n^+-th power set of A.

To prove the existence of the large set. By specializing the axiom of collection, we obtain a set B, a function $p : B \to \mathbb{N}$, a set Y, and a relation $M : B \nrightarrow Y$ satisfying:

$$(\forall S\ (i : S \to Y)\ (b \in B).\ \mathsf{isset}(i, \{y \mid \mathsf{Holds}(M, b, y)\}) \implies \mathsf{nPow}(p(b), A, S)) \wedge$$
$$(\forall n \in \mathbb{N}\ \ X.\ \mathsf{nPow}(n, A, X)) \implies \exists b \in B.\ p(b) = n)$$

The set Y is the large set we want to construct. For any $n \in \mathbb{N}$, we have $\mathsf{nPow}(n, A, \mathsf{Pow}^n(A))$, and thus there exists a $b \in B$ with $p(b) = n$. For this b, Let $H(b)$ denote the set of

elements y such that $\mathsf{Holds}(M, b, y)$, then $\mathsf{minc}(H(b))$ gives an injection $\mathsf{m2s}(H(b)) \to Y$. As $\mathsf{nPow}(n, A, \mathsf{m2s}(H(b)))$ and also $\mathsf{nPow}(n, A, \mathsf{Pow}^n(A))$, by uniqueness proved above, there exists a bijection $j : \mathsf{Pow}^n(A) \to \mathsf{m2s}(H(b))$. The composition $\mathsf{minc}(H(b)) \circ j$ is the desired injection.

Future work on Beth cardinals The above proves the existence of a super large set, which already breaks the limitation in HOL. But SEAR is even more powerful than that. Actually, it is equipped with the power to prove the minimal set that we can inject each iterated power set of a set A exists and is unique up to *unique* structure-preserving isomorphism. The uniqueness of such an isomorphism is critical. The minimal such subset cannot be obtained directly from applying Axiom 5. We require a derived "replacement schema" for this task. Such a schema is more meta-theoretic than those that can be captured in one statement within SEAR. We cannot state a form of it within a theorem, even with the help of formula variables. This is because it has a form for each well-formed dependency list $\vec{v}$ that can be used to quantify on formulas, and hence have an instance, expressed with a formula variable $\mathcal{F}[\vec{v}]$, for each such $\vec{v}$. Despite the fact that the general proof in [47] borrows a lot of category theory from the meta-level, each instance of provable within the SEAR theory[3].

The notion of context as in [47] is the same as us, except that it is a set in our logic and is a list consisting of the elements in our set. The replacement schema says if for each element $a \in A$, there is a context θ satisfies $\phi(a, \theta)$, and moreover, such a context is unique up to unique isomorphism, then there exists a minimal, universal context $\tilde{\theta}$ such that each context θ such that $\phi(a, \theta)$ can be recovered from $\tilde{\theta}$.

The recovery will rely on a map from the universal context to the set A. To avoid categorical terminology, we say for each piece of information in the context, we associate it with an element in A. For each $a \in A$, we can take out all the information from $\tilde{\theta}$ that is associated with it to form a new context, considered as "the fragment θ_a of θ over a". Then the context θ_a satisfies $\phi(a, \theta_a)$. The universal context $\tilde{\theta}$ is minimal because it does not contain any redundant information. This is evident from the indexing map to the set A, which shows every piece of information in $\tilde{\theta}$ serves for consisting a part of *the* unique context for some element.

In what follows, we keep things simple by taking the set A to be $\mathbb{N}$. The countably infinite iterated power set of $\mathbb{N}$ is called the *Beth number* $\beth_\omega$, already very interesting.

We formalize the proof of the unique existence of the structure-preserving isomorphism, which is already quite involved. The first task is to present a definition to express that "the n-th iterated power set of $\mathbb{N}$ is unique for each $n \in \mathbb{N}$". We define a predicate, that captures "the set X is the disjoint union of the n-th iterated power set of N" by the predicate with four inputs:

[3] This is confirmed via personal communication

$$
\begin{aligned}
&\forall n \in \mathbb{N}\ X\ (p : X \to \mathbb{N})\ (R : X \nrightarrow X)\ (z : \mathbb{N} \to X).\\
&\quad \mathsf{Upows}(n, p, R, z) \Leftrightarrow\\
&\quad \mathsf{IM}(p) = \mathsf{Les}(n)\ \wedge\ \mathsf{Inj}(z)\ \wedge\ \mathsf{IM}(z) = \mathsf{Fib}(p, 0_{\mathbb{N}})\ \wedge\\
&\quad (\forall n_0 \in \mathbb{N}.\\
&\qquad n_0 < n\ \Longrightarrow\\
&\qquad\ \exists A\ (i : A \to X)\ (pi : \mathsf{Pow}(A) \to X).\\
&\qquad\ \mathsf{Inj}(i)\ \wedge\ \mathsf{Inj}(pi)\ \wedge\ \mathsf{IM}(i) = \mathsf{Fib}(p, n_0)\ \wedge\ \mathsf{IM}(pi) = \mathsf{Fib}(p, {n_0}^+)\ \wedge\\
&\qquad\ (\forall (a \in A)\ (s \in \mathsf{Pow}(A)).\ \mathsf{Holds}(R, i(a), pi(s)) \Leftrightarrow \mathsf{IN}(a, s))) \wedge\\
&\quad (\forall x \in X\ s \in X.\ \mathsf{Holds}(R, x, s)\ \Longrightarrow\ p(x)^+ = p(s))
\end{aligned}
$$

The predicate indicates "the set X resembles a disjoint union of power sets N, $\mathsf{Pow}(\mathbb{N})$, $\mathsf{Pow}(\mathsf{Pow}(\mathbb{N}))$, $\cdots$, $\mathsf{Pow}^n(\mathbb{N})$", The projection $p : X \to \mathbb{N}$, considered as an indexing map, sends an element $x \in X$ to $n \in \mathbb{N}$ if x belongs to the $\mathsf{Pow}^n(\mathbb{N})$ component of the disjoint union. We use the notation $\mathsf{Fib}(p, n)$ to denote the preimage of p at a point n. The iteration starts with $\mathbb{N}$, whose copy in X is found as $\mathsf{Fib}(p, 0_{\mathbb{N}})$. The relation R captures the disjoint union of all the membership relations between these iterated power sets. The second-to-last conjunct says for each number n_0 below n, there is a pair of sets A and $\mathsf{Pow}(A)$ injected into the fiber of n_0 and ${n_0}^+$. Two elements in the image of i and pi are related if and only if they come from a membership relation from A and $\mathsf{Pow}(A)$. The final clause says the relation R only holds for elements that come from adjacent power sets.

We construct by induction on n, using the induction principle 5.2.4, that for each n, the existence of X, $p : X \to \mathbb{N}$, $R : X \nrightarrow X$ and $z : \mathbb{N} \to X$ such that $\mathsf{Upows}(n, p, R, z)$. The base case is trivial by taking $\mathbb{N}$ itself. For the base case, assume $\mathsf{Upows}(n, p, R, z)$, for any injection $i : A \to X$ whose image is $\mathsf{Fib}(p, n)$, we use $X + \mathsf{Pow}(A)$ to capture the $n + 1$-th iteration, with all of $\mathsf{Pow}(A)$ mapped to $n + 1$. The relation on X is the one from the previous step, and the new relation on $X + \mathsf{Pow}(A)$ will just add the relation that relates the elements in the image of i to the sets in $\mathsf{Pow}(A)$ it belongs to.

Given $\mathsf{Upows}(n, p_1 : X_1 \to \mathbb{N}, R_1 : X_1 \nrightarrow X_1, z_1 : \mathbb{N} \to X_1)$ and $\mathsf{Upows}(n, p_2 : X_2 \to \mathbb{N}, R_2 : X_2 \nrightarrow X_2, z_2 : \mathbb{N} \to X_2)$, a structural preserving isomorphism consists of two functions $f_1 : X_1 \to X_2$ and $f_2 : X_2 \to X_1$, satisfying the equations $p_2 \circ f_1 = p_1$, $f_1 \circ z_1 = z_2$, $p_2 \circ f_2 = p_1$, $f_2 \circ z_1 = z_2$. Moreover, the relation R_1 has to be the image of the relation R_1 under f_1, and vice versa. Given two such isomorphisms, we prove they coincide on every fiber of p to prove they are equal on the whole domain. This is again an induction on n. The base case is ensured by the equations on z_1 and z_2. For the step case, we assume they agree on the fiber on n and prove they agree on the fiber on n^+. We have by definition of Upows that there exist injections $i : A \to X_1$ and $pi : \mathsf{Pow}(A) \to X_1$ on the fiber of n and n^+. The inductive hypothesis says $f_1 \circ i = f_2 \circ i$, and it is sufficient to show $f_1 \circ pi = f_2 \circ pi$. We introduce

another auxiliary lemma, saying for two pairs of inclusions $i_1 : A_1 \to X_1$, $i_2 : A_2 \to X_1$ inject to $\mathsf{Fib}(p, n)$, and maps $pi_1 : \mathsf{Pow}(A_1) \to X_1$, $pi_2 : \mathsf{Pow}(A_2) \to X_1$ injects to $\mathsf{Fib}(p, n^+)$, if they both correspond to the membership relation, then there is an isomorphism $k : A_1 \to A_2$ such that $i_2 \circ k = i_1$ and $pi_2 \circ \mathsf{Image}(k) = pi_1$. We take the i_1 and i_2 in this lemma to be $f_1 \circ pi$ and $f_2 \circ pi$. Then the isomorphism μ satisfies $(f_2 \circ i) \circ k = f_1 \circ i$ and also $(f_2 \circ i) \circ \mathsf{Image}(k) = f_1 \circ k$ Recall $f_1 \circ i = f_2 \circ i$, so $(f_2 \circ i) \circ k = f_2 \circ i$. But $f_2 \circ i$ is a mono, so $k = \mathsf{Id}(A)$, immediately gives the result.

The existence of the isomorphism is proved by induction on n, using the embedding from the meta-tuple satisfies $\mathsf{Upows}(n, p_1 : X_1 \to \mathbb{N}, R_1 : X_1 \nrightarrow X_1, z_1 : \mathbb{N} \to X_1)$ into the one satisfies $\mathsf{Upows}(n^+, p_2 : X_2 \to \mathbb{N}, R_2 : X_2 \nrightarrow X_2, z_2 : \mathbb{N} \to X_2)$, and extend the bijection to cover the extra $n + 1$-th power set of $\mathbb{N}$.

5.2.10 The collapse of the λ-cube

A measurement of how powerful a type theory is can be measured by fitting it into the λ-cube. From the origin representing the dependency "terms on terms", there are three axes, representing dependencies of terms on types, types on terms, and types on types. Here "dependency" is universally conceived as "being able to construct a kind of thing out of another kind of thing". For instance, HOL includes terms depending on terms, through the mechanism of function application; terms depending on types, such as the polymorphism in the term "empty list"; and types depending on types, as seen in the polymorphic type of trees, where different types detail the example. But HOL does not have types depending on terms, so this dimension is missing. In contrast, Coq and Lean have all of the above, and in addition, allow types to depend on terms. Indeed, this is what is understood by the term "dependent types".

Such dependencies naturally occur in mathematical constructions. It is naturally thought of as the application of meta-functions, they are "functions" that may not exist in the theory, and do not have to be representable by a function term As we have already seen, we can construct the set of all groups with a fixed underlying set, this can be regarded as a function in the meta-level, from the collection of sets to the collection of groups. Such a function does not exist in the SEAR theory but is a function symbol.

It is clear that all these constructions can be done through the use of complicated sets in ZF set theory. But then, ZF does not provide any formal support on type checking. Being able to type check is important because it makes sure we are writing down statements that express interesting mathematical facts. Whereas we can represent in dependent type theory, the type checking is complicated and takes a longer time to perform. By providing a "type check" just in four kinds of things, capturing four interesting kinds of object that are sufficient to do mathematics, SEAR provide a sweet spot on this balance.

We now claim that whereas it is clear that the origin, which represents the dependency "terms on terms", is simply function applications and can certainly be captured by function symbols in first-order logic since our machinery allows constructing function symbols [4] out of functions, it is moreover, possible to simulate all the remaining three dimensions in SEAR with function symbols as well. In SEAR, the λ-cube collapses to the point of "function application" only. It is witnessed by the following examples.

Terms depend on types (polymorphism) Polymorphism means that we can construct different terms with the same role for different types. This is doable in SEAR with the obvious witnesses of the empty (co-)list. We write $\mathsf{Nil}(A)$ to denote the empty list with elements taken from the set A, for any A. This is done by imposing the constraint on the sort of output, due to the fact that our function symbols are sort-checked by pattern-matching.

Types depend on other types In SEAR, the role of a type as in type theory is played by sets, we can apply function symbols on set terms and produce a set. The analog of the standard example of creating trees for each type is already presented in modal model theory section, where we create the set of formulas with propositional symbols from each set A.

Types depend on terms We enable our types to depend on terms as a primitive feature in our logic. Differing from DTT logics, the dependency in our logic must happens in a certain way restricted by the signature. We are not allowed to build an arbitrary sort out of a given term. But what the signature requires is just making sure which role will a term play. Standard constructions can be easily carried out. Let us take a favorite example on the dependent type of "vectors of length n" on a type A:

5.2.14 Example. We can use our dependent sort to capture the list-appending map. In DTT systems, let A be the type where the items of lists come from, the append map has type $\mathsf{list}(A, n_1) \to \mathsf{list}(A, n_2) \to \mathsf{list}(A, n_1 + n_2)$. In SEAR, for a set A, we can create the set $\mathsf{list}(A, n)$ of lists of length n for each n, with a canonical inclusion map $\mathsf{iLn}(A, n) : \mathsf{list}(A, n) \to \mathsf{List}(A)$. For each $l \in \mathsf{list}(A, n)$, we then have $\mathsf{Length}(\mathsf{iLn}(A, n)(l)) = n$. Trivially, by abbreviating $\mathsf{RepLn}(l) := \mathsf{iLn}(A, n)(l)$, we can prove theorems like:

$$\forall A\ (n_1 \in \mathbb{N})\ (n_2 \in \mathbb{N})\ (l_1 \in \mathsf{list}(A, n_1))\ (l_2 \in \mathsf{list}(A, n_2)).$$
$$\mathsf{RepLn}(l_1) = \mathsf{RepLn}(l_2) \implies n_1 = n_2$$

We have that appending a list with length n_1 with a list of length n_2 yields a list of length $n_1 + n_2$. This theorem can be lifted into the dependent sort. We prove for each pair of list $l_1 \in \mathsf{list}(A, n_1)$

[4] Note that there is no notion of "function symbols" in type theory

and $l_2 \in \mathsf{list}(A, n_2)$, we have a unique list in $\mathsf{list}(A, n_1 + n_2)$. This gives us a function symbol for appending, allows us to produce the function term. $\mathsf{Appendn}(l_1, l_2) : \mathsf{list}(A, n_1 + n_2)$.

As above, the canonical way for "types depend on terms" in SEAR is arguably "term depends on terms".

In conclusion, although we do not obtain the layering through type theory, we do not lose any aspect of its power. This is done by choosing powerful axioms, indicating various of type theories are not the only ways to layer in computer formalization of mathematics.

Chapter 6

Mechanizing CCAF

The category of categories as a mathematical foundation, given by Colin McLarty [33], is an attempt at axiomatizing category theory. It is the only existing publication of such an axiomatization besides Lawvere's ETCC, that has already been pointed out to be flawed, at present. However, by mechanizing it, we catch an error in CCAF as well. The two sorts of this system are that of the categories and that of the functors, which depend on the source and the target category. Categories are usually denoted as letters $A, B, C, \cdots$. A functor f (also g, h, ...) from A to B is written $f : A \to B$. Higher dimensional information, a natural transformation, say, is given by a functor to a functor category. Equalities are only allowed between functor terms. Table 6.1 presents primitive symbols from McLarty [33].

In what follows, as the translations are mostly straightforward, we are not going to present all the formalized statements but focus on explaining the parts that are less straightforward to understand. In particular, the critical choices we made, and the differences between our formalization and the original paper. The section titles in this chapter follow those of McLarty [33]. This allows this chapter to be read in parallel with the original paper.

6.1 The categories 1 and 2

Axiom 1 imposes *selected* products, coproducts, and exponentials to the system. This is done by defining the predicates isPr etc. by their universal property and declaring the primitive function symbols accordingly. They have the definition that precisely looks like what we did before for ETCS and SEAR. Then we state as axioms that the terms constructed by the application of these function symbols satisfy the desired properties. However, whereas these notions are on sets in the two set theories, now the notions are on categories, so $\pi_1(A, B) : A \times B \to A$ means the projection on the first component is a functor.

6.1.1 Axiom $\mathbf{CC_1}$. *The meta-category* **CAT** *(not a term in our logic) is a cartesian closed category with all finite limits and colimits, where the initial object $\varnothing$ is not isomorphic to the*

Symbol	Input	Output	Source	Predicate/Function
$\circ$	$[f : A \to B,\ g : B \to C]$	$g \circ f : A \to C$	signature	Function
Id	$[A : \mathrm{Cat}]$	$\mathsf{Id}(A) : A \to A$	signature	Function
$\times$	$[A : \mathrm{Cat},\ B : \mathrm{Cat}]$	$A \times B : \mathrm{Cat}$	CC_1	Function
π_1	$[A : \mathrm{Cat},\ B : \mathrm{Cat}]$	$\pi_1(A, B) : A \times B \to A$	CC_1	Function
π_2	$[A : \mathrm{Cat},\ B : \mathrm{Cat}]$	$\pi_2(A, B) : A \times B \to B$	CC_1	Function
$+$	$[A : \mathrm{Cat},\ B : \mathrm{Cat}]$	$A + B : \mathrm{Cat}$	CC_1	Function
i_1	$[A : \mathrm{Cat},\ B : \mathrm{Cat}]$	$i_1(A, B) : A \to A + B$	CC_1	Function
i_2	$[A : \mathrm{Cat},\ B : \mathrm{Cat}]$	$i_2(A, B) : B \to A + B$	CC_1	Function
Exp	$[A : \mathrm{Cat},\ B : \mathrm{Cat}]$	$B^A : \mathrm{Cat}$	CC_1	Function
ev	$[A : \mathrm{Cat},\ B : \mathrm{Cat}]$	$\mathsf{ev}(A, B) : A \times B^A \to B$	CC_1	Function
$\varnothing$	[]	$\varnothing : \mathrm{Cat}$	CC_1	Function
$\mathbf{1}$	[]	$\mathbf{1} : \mathrm{Cat}$	CC_1	Function
$\mathbf{2}$	[]	$\mathbf{2} : \mathrm{Cat}$	CC_2	Function
0	[]	$0 : \mathbf{1} \to \mathbf{2}$	CC_2	Function
1	[]	$1 : \mathbf{1} \to \mathbf{2}$	CC_2	Function
$\mathbf{E}$	[]	$\mathbf{E} : \mathrm{Cat}$	Comment after CC_3	Function
ε_1	[]	$\varepsilon_1 : \mathbf{2} \to \mathbf{E}$	Comment after CC_3	Function
ε_2	[]	$\varepsilon_2 : \mathbf{2} \to \mathbf{E}$	Comment after CC_3	Function
$\mathbf{3}$	[]	$\mathbf{3} : \mathrm{Cat}$	CC_4	Function
α	[]	$\alpha : \mathbf{2} \to \mathbf{3}$	CC_4	Function
β	[]	$\beta : \mathbf{2} \to \mathbf{3}$	CC_4	Function
γ	[]	$\gamma : \mathbf{2} \to \mathbf{3}$	CC_4	Function
isop	$[A : \mathrm{Cat},\ B : \mathrm{Cat}]$	-	CC_7	Predicate
isopf	$[f : A \to B,\ f' : A' \to B']$	-	CC_7	Predicate

Table 6.1: Primitive symbols required by CCAF

terminal object **1**.

$$\vdash \forall A\ B.\ \mathsf{isPr}(\pi_1(A,B), \pi_2(A,B))$$

$$\vdash \forall A\ B.\ \mathsf{iscoPr}(i_1(A,B), i_2(A,B))$$

$$\vdash \forall X\ H\ (f : X \to H)\ Y\ (g : Y \to H).\ \exists P\ (p : P \to X)\ (q : P \to Y).\ \mathsf{isPb}(f,g,p,q)$$

$$\vdash \forall H\ X\ (f : H \to X)\ Y\ (g : H \to Y).\ \exists P\ (p : X \to P)\ (q : Y \to P).\ \mathsf{isPo}(f,g,p,q)$$

$$\vdash \mathsf{intl}(\varnothing)\ \wedge\ \mathsf{tml}(\mathbf{1})\ \wedge\ \varnothing \ncong \mathbf{1}$$

$$\vdash \forall A\ B.\ \mathsf{isExp}(\mathsf{ev}(A,B))$$

Also as before, we denote the exponential object B^A, and $\mathsf{ev}(A,B)$ is of sort $A \times B^A \to B$. The unique transposition $X \to B^A$ for a functor $f : A \times X \to B$ is created by the function specification rule as $\mathsf{Tp}(f)$. Conversely, given an $h : X \to B^A$, we denote by $\overline{h}$ the corresponding functor $A \times X \to B$. Also, note that the notion of isomorphism in above refers to isomorphism between categories, that is, refers to composition of functors instead of arrows within a category.

Each functor $f : A \to B$ induces a functor $X^f : X^B \to X^A$ defined by the term $\mathsf{Tp}(\mathsf{ev}(B,X) \circ \langle f \circ \pi_1(A, X^B), \pi_2(A, X^B)\rangle)$. This functor is regarded as sending a functor $B \to X$ to a functor $A \to X$ via composition by f. Note that this function symbol takes two arguments: a category X, where there is no notion of equality, and a functor, where equality can be written. Here are two ways to prove $X^f = X^g$ for $f = g$. The first way is to use congruence between formula variables, which immediately gives the result: the formula $\forall f : A \to B.\ X^f = X^g$ is well-formed, so from $f = g$, we have $X^f = X^g \Leftrightarrow X^g = X^g$, where the right-hand side is true. But we actually recommend the second way: we expand the definition to give the result. This uses the fact that despite we do not have equality between categories, function symbols applied to the same category can be equated if they look exactly the same. This is by reflexivity between equations. Therefore, we have $\pi_1(A, X^B) = \pi_1(A, X^B)$, with the equality $f = g$ we can derive the intended equation. Note this proof is entirely syntactical, we do not even need to check the output of the functors is always the same.

The categories **1** and $\varnothing$ as in Axiom 1 are to be thought of as the categories with a single object and only the identity arrow on it and an empty category, respectively. Again, we use $!_A$ to denote the only functor $A \to 1$. An important idea to keep in mind when reading theorems in CCAF (and actually, in category theory in general) is that a functor embeds the shape from the source category as a possibly distorted copy into the target category. A first example is that an object in a category A is defined in CCAF as a functor $\mathbf{1} \to A$, by analogy with ETCS, where an arrow from the singleton set to a set A selects an element. With the same idea, an arrow in A should be an embedding of the shape $\cdot \to \cdot$ to A. Axiom CC_2 gives the existence of such a category.

6.1.2 $\mathbf{CC_2}$. *There are constants* $0, 1 : \mathbf{1} \to \mathbf{2}$ *such that the identity* two *on* $\mathbf{2}$, $\mathsf{zero} := 0 \circ !_{\mathbf{2}}$ *and* $\mathsf{one} := 1 \circ !_{\mathbf{2}}$ *are distinct and the only endofunctors of* $\mathbf{2}$. *Moreover, the object* $\mathbf{2}$ *is a generator: any pair of parallel functors are either equal or differ at some functor from* $\mathbf{2}$.

$$\vdash \mathsf{zero} \neq \mathsf{one} \ \wedge \ \mathsf{zero} \neq \mathsf{two} \ \wedge \ \mathsf{one} \neq \mathsf{two} \tag{0}$$

$$\vdash \forall f : \mathbf{2} \to \mathbf{2}.\ f = \mathsf{zero} \ \vee \ f = \mathsf{one} \ \vee \ f = \mathsf{two} \tag{1}$$

$$\vdash \forall A\ B\ f\ g : A \to B.\ f \neq g \implies \exists a : \mathbf{2} \to A.\ f \circ a \neq g \circ a \tag{2}$$

Although not directly obtainable from the only two axioms here, with the later axiom 6.3.1, we will eventually prove (0) and (1) are distinct in the sense that the pullback of these arrows is the initial category (for the terminology, a *pullback* of two functors can be thought as how much they overlap). That is, $\mathbf{2}$ has exactly two objects and a non-identity arrow, hence three arrows in total. We will abbreviate 0 and 1 as $\mathsf{exact3(zero, one, two)}$. An arrow in a category A is a functor $f : \mathbf{2} \to A$. Its domain and codomain are given by $f \circ 0, f \circ 1 : \mathbf{1} \to A$, and denoted as $\mathsf{dom}(f)$ and $\mathsf{cod}(f)$ respectively. We define the identity $\mathsf{id}(a)$ on an object $a : 1 \to A$ as $a \circ !_{\mathbf{2}}$. As a fact that admits an ETCS (or SEAR) counterpart, we point out that we can later prove every non-initial category has an arrow. But the counterpart of well-pointedness is now different in CCAF. Since a set consists of elements, which in structural set theory have no internal structure, a category has arrows, which link between objects. Therefore, the "extensionality" of functors requires that two functors are equal if they agree on all the arrows, i.e., agree on all the functors from $\mathbf{2}$. This implies two equal functors agree over all objects, which can be expressed as: for $F, G : A \to B$, we have $F = G \implies \forall a : \mathbf{1} \to A.\ F \circ a = G \circ a$. This makes $\mathbf{2}$ a *generator*. To prove this, the axiom

6.1.3 $\mathbf{CC_3}$. *The coproduct* $1 + 1 : \mathbf{1} + \mathbf{1} \to \mathbf{2}$ *is not an epimorphism.*

$$\vdash \neg \mathsf{Epi}(1 + 1)$$

is added. Moreover, we can prove $\mathbf{2}$ is up to isomorphism the only generator.

6.2 Composition

Whereas we automatically get the notion of composition of functors in the signature, with effective type-checking, the notion of composition within a category is not primitive in CCAF. We now investigate how would we derive such a notion. From now on, we write for arrows $f, g : \mathbf{2} \to A$ the "composable" predicate $\mathsf{cpsb}(g, f)$ to mean $\mathsf{dom}(g) = \mathsf{cod}(f)$.

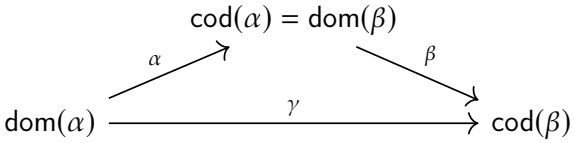
A composition in a category A will give a commutative triangle in A, with the idea "functors as shapes", we then require a triangle-shaped category, provided by CC_4.

6.2.1 $\mathbf{CC_4}$. *The category* **3** *is the pushout below, with an arrow* $\gamma : \mathbf{2} \to \mathbf{3}$ *such that* $\gamma \circ 0 = \alpha \circ 0$ *and* $\gamma \circ 1 = \beta \circ 1$

$$\begin{array}{ccc} 1 & \xrightarrow{1} & 2 \\ \downarrow{\scriptstyle 0} & & \downarrow{\scriptstyle \alpha} \\ 2 & \xrightarrow{\beta} & 3 \end{array}$$

$$\vdash \mathsf{isPo}(1, 0, \alpha, \beta) \ \wedge \ \mathsf{dom}(\gamma) = \mathsf{dom}(\alpha) \ \wedge \ \mathsf{cod}(\gamma) = \mathsf{cod}(\beta)$$

A pushout is regarded as a gluing. In our case, we glue together two copies of the category **2** by putting them in a line and gluing a head and a tail from these two copies together. The resulting category **3** hence has a "longer" arrow, obtained by connecting two copies of the arrow two of **2**. Before having functor comprehension schema, we cannot prove **3** is exactly a commutative triangle, but actually it is, pictured as:



This is sufficient for us to define composition. Once we have $\mathsf{cpsb}(g, f)$, the universal property of a pushout will give us a functor $\mathbf{3} \to A$ representing the composed triangle. By composing with α, β, and γ, we take parts from the triangle as arrows in A. The original arrows f and g are taken out by composing with α and β respectively. The composition is obtained by composing with γ.

We now want a function symbol that takes two composable arrows and gives the composition. Formally, we will prove:

$$\forall f \ g : \mathbf{2} \to A. \exists! t : \mathbf{3} \to A. (\mathsf{cpsb}(f, g) \wedge t \circ \alpha = f \wedge t \circ \beta = g) \ \vee (\neg \mathsf{cpsb}(f, g) \wedge t = \mathsf{dom}(f) \circ !_3)$$

By specification, we create a function symbol such that $\Delta(f, g) : \mathbf{3} \to A$ gives the composing triangle of the two arrows once they are composable. When the two inputs are not composable, we still need an output, but the result would be an uninteresting trivial arrow. We define the composition $g \circ_A f$ as $\Delta(f, g) \circ \gamma$. The domain and codomain information $\mathsf{dom}(g \circ_A f) = \mathsf{dom}(f)$ and $\mathsf{cod}(g \circ_A f) = \mathsf{cod}(g)$ are immediate by the pushout, and so is the fact that composing with identity preserves the arrow.

The sorts of the inputs and the composed arrow are hence all the same and the composition is a total function symbol. Since the domain and codomain of an arrow are not intrinsic,

composibility has to be declared via predicates. The sort information only supports type-checking between functors. We will later put natural transformation in the same layer as functors, so the type-checking is not helpful there either.

Composing any functor $A \to B$ with a functor $\mathbf{3} \to A$ will give a functor $\mathbf{3} \to B$. In other words, commutative triangles are preserved by functor application. This gives "functor preserves composition" for free. We prove it as a theorem:

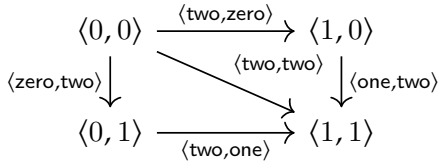
$$\vdash \forall A\ f\ g : \mathbf{2} \to A.\ \mathsf{cpsb}(g, f) \implies \forall B\ k : A \to B.\ k \circ (g \circ_A f) = (k \circ g) \circ_B (k \circ f) \quad (6.1)$$

The compositions in $\mathbf{2}$ will play an important role afterward.

6.2.2 Theorem 4.

$$\vdash \mathsf{one} \circ_{\mathbf{2}} \mathsf{two} = \mathsf{two} \ \wedge\ \mathsf{two} \circ_{\mathbf{2}} \mathsf{zero} = \mathsf{two}$$

We want to prove the associativity of the composition of arrows. This requires techniques to manipulate commutative squares and explore the connection between commutative squares in the intuitive sense and arrows in the arrow category. We observe that $\mathbf{2} \times \mathbf{2}$ consists of four objects and nine arrows, as pictured below.



A commutative square in A is then a functor $c : \mathbf{2} \times \mathbf{2} \to A$. The edges of c are obtained by composing with the corresponding arrows of $\mathbf{2} \times \mathbf{2}$. For instance, the left edge is $c \circ \langle \mathsf{zero}, \mathsf{two} \rangle : \mathbf{2} \to A$. We write $\mathsf{csT}(c), \mathsf{csB}(c), \mathsf{csL}(c), \mathsf{csR}(c)$ to denote the top, bottom, left and right edges respectively. Equivalently, such a functor is an arrow $s : \mathbf{2} \to A^{\mathbf{2}}$. The domain and codomain of such an arrow will be the top and bottom edges of the corresponding commutative square. The edges of its transpose are given by:

$$\begin{aligned}
\mathsf{csT}(\overline{s}) &= \mathbf{2} \xrightarrow{\langle \mathsf{two}, !_{\mathbf{2}} \rangle} \mathbf{2} \times \mathbf{1} \xrightarrow{\overline{\mathsf{dom}(s)}} A \\
\mathsf{csB}(\overline{s}) &= \mathbf{2} \xrightarrow{\langle \mathsf{two}, !_{\mathbf{2}} \rangle} \mathbf{2} \times \mathbf{1} \xrightarrow{\overline{\mathsf{cod}(s)}} A \\
\mathsf{csL}(\overline{s}) &= \mathbf{2} \xrightarrow{s} A^{\mathbf{2}} \xrightarrow{A^0} A^{\mathbf{1}} \xrightarrow{\cong} A \\
\mathsf{csR}(\overline{s}) &= \mathbf{2} \xrightarrow{s} A^{\mathbf{2}} \xrightarrow{A^1} A^{\mathbf{1}} \xrightarrow{\cong} A
\end{aligned}$$

Under the above definition, the commutativity of a square is proved as:

$$\vdash \forall A\ s : \mathbf{2} \times \mathbf{2} \to A.\ \mathsf{csR}(s) \circ_A \mathsf{csT}(s) = \mathsf{csB}(s) \circ_A \mathsf{csL}(s)$$

Conversely, every two compositions that agree give a commutative square.

6.2.3 Theorem 7. *For composable pairs f, g and f', g' such that $g \circ_A f = g' \circ_A f'$, there exists a corresponding commutative square.*

$$\begin{aligned}\vdash \forall A\ f\ g\ f'\ g' : \mathbf{2} \to A.\ \mathsf{cpsb}(g, f)\ \wedge\ \mathsf{cpsb}(g', f')\ \wedge\ g \circ_A f = g' \circ_A f'\ \Longrightarrow \\ \exists q : \mathbf{2} \times \mathbf{2} \to A.\ \mathsf{csT}(q) = f\ \wedge\ \mathsf{csR}(q) = g\ \wedge\ \mathsf{csL}(q) = f'\ \wedge\ \mathsf{csB}(q) = g'\end{aligned}$$

The final ingredient we need is that commutative squares can be pasted together side by side to form a new commutative square. The pasted square will have as top and bottom arrows the composition of the arrows in the component squares.

Theorem 6.2.1.

$$\begin{aligned}\vdash \forall A\ c_1\ c_2 : \mathbf{2} \times \mathbf{2} \to A.\ \mathsf{csR}(c_1) = \mathsf{csL}(c_2)\ \Longrightarrow \\ \exists c : \mathbf{2} \times \mathbf{2} \to A.\ \mathsf{csL}(c) = \mathsf{csL}(c_1)\ \wedge\ \mathsf{csR}(c) = \mathsf{csR}(c_2)\ \wedge \\ \mathsf{csT}(c) = \mathsf{csT}(c_2) \circ_A \mathsf{csT}(c_1)\ \wedge\ \mathsf{csB}(c) = \mathsf{csB}(c_2) \circ_A \mathsf{csB}(c_1)\end{aligned}$$

Now we are equipped to prove associativity.

6.2.4 Theorem 8.

$$\vdash \forall A\ f\ g\ h : \mathbf{2} \to A.\mathsf{cpsb}(g, h)\ \wedge\ \mathsf{cpsb}(f, g)\ \Longrightarrow\ (f \circ_A g) \circ_A h = f \circ_A g \circ_A h$$

Proof. By Theorem 7, there exist commutative squares $\begin{array}{ccc} \cdot & \xrightarrow{h} & \cdot \\ \downarrow{\scriptstyle h} & & \downarrow{\scriptstyle g} \\ \cdot & \xrightarrow{g} & \cdot \end{array}$ and $\begin{array}{ccc} \cdot & \xrightarrow{g} & \cdot \\ \downarrow{\scriptstyle g} & & \downarrow{\scriptstyle f} \\ \cdot & \xrightarrow{f} & \cdot \end{array}$. By 6.2.1, we then obtain a commutative square $\begin{array}{ccc} \cdot & \xrightarrow{h \circ_A g} & \cdot \\ \downarrow{\scriptstyle h} & & \downarrow{\scriptstyle f} \\ \cdot & \xrightarrow{g \circ_A f} & \cdot \end{array}$. The result follows by commutativity. □

6.3 Functorial comprehension

Like the axiom of specification in terms of set theory, there is also a comprehension schema in CCAF, declared as an axiom. There is no existing comprehension schema for forming new categories. Nevertheless, given two categories, we have a helpful comprehension schema constructing functors between them. Here is the axiom CC_5.

6.3.1 $\mathbf{CC_5}$. *If R is a functorial relation between arrows of A and arrows of B, then R defines a functor $A \to B$.*

$$
\begin{array}{l}
\{A,\ B\} \\
\vdash (\forall f : \mathbf{2} \to A.\ \exists! g : \mathbf{2} \to B.\ \mathcal{R}[f_0 : \mathbf{2} \to A,\ g_0 : \mathbf{2} \to B](f, g)) \wedge \\
\quad (\forall f : \mathbf{2} \to A\ g : \mathbf{2} \to B.\ \mathcal{R}[f_0 : \mathbf{2} \to A,\ g_0 : \mathbf{2} \to B](f, g) \implies \\
\qquad \mathcal{R}[f_0 : \mathbf{2} \to A,\ g_0 : \mathbf{2} \to B](\mathsf{id}(\mathsf{dom}(f)), \mathsf{id}(\mathsf{dom}(g))) \wedge \\
\qquad \mathcal{R}[f_0 : \mathbf{2} \to A,\ g_0 : \mathbf{2} \to B](\mathsf{id}(\mathsf{cod}(f)), \mathsf{id}(\mathsf{cod}(g)))) \wedge \\
\quad (\forall f\ g : \mathbf{2} \to A\ h : \mathbf{2} \to B. \\
\qquad \mathsf{cpsb}(g, f) \wedge \mathcal{R}[f_0 : \mathbf{2} \to A,\ g_0 : \mathbf{2} \to B](g \circ_A f, h) \implies \\
\qquad \forall f'\ g' : \mathbf{2} \to B. \\
\qquad\quad \mathcal{R}[f_0 : \mathbf{2} \to A,\ g_0 : \mathbf{2} \to B](f, f') \wedge \mathcal{R}[f_0 : \mathbf{2} \to A,\ g_0 : \mathbf{2} \to B](g, g') \implies h = g' \circ_A f') \\
\implies \\
\quad \exists F : A \to B.\ \forall a : 2 \to A\ b : 2 \to B.\ \mathcal{R}[f_0 : \mathbf{2} \to A,\ g_0 : \mathbf{2} \to B](a, b) \Leftrightarrow F \circ a = b
\end{array}
$$

The first clause captures functionality; the second clause captures the preservation of identity and the third clause forces the arrow related to the composite of g and f to be the composite of the one related to g and the one related to f. If F_1 and F_2 are both determined by $\mathcal{R}$, then the conclusion of CC_5 says they agree on arrows from $\mathbf{2}$, so they are the same because $\mathbf{2}$ is a generator. Now we are allowed to define functors directly by establishing the association via predicates, by instantiating the formula variable $\mathcal{R}[f_0 : \mathbf{2} \to A,\ g_0 : \mathbf{2} \to B]$. An easy application of CC_5 will be proving isomorphisms are precisely mono-epis $F : A \to B$ by explicitly defining an inverse $B \to A$ using the relation associating an arrow $f : \mathbf{2} \to B$ to the unique arrow $g : \mathbf{2} \to A$ such that $F \circ f = g$.

We can now treat adjoint functors, an important topic in category theory. We start introducing the notions required. Firstly, we have to define *natural transformations*. These "live" in another dimension above functors, but do not use a separate sort, instead, they are captured by functors $\eta : A \to B^{\mathbf{2}}$. The functors $B^{\mathsf{zero}}, B^{\mathsf{one}} : B^{\mathbf{2}} \to B$ (by abuse of notation, using $B^{\mathbf{1}} \cong B$) take the domain and codomain of an arrow. We define $\mathsf{Dom}(\eta) := B^{\mathsf{zero}} \circ \eta$ and $\mathsf{Cod}(\eta) := B^{\mathsf{one}} \circ \eta$, then η is a natural transformation from F and G iff F and G are its domain and codomain, in which case we write $\mathsf{Nt}(\eta, F, G)$. The identity natural transformation is $\mathbf{I}(F) := \mathsf{Tp}(\pi_2(\mathbf{2}, B)) \circ F$. On each object $a : \mathbf{1} \to A$, the component η_a is $\overline{\eta \circ a} \circ \langle \mathsf{Id}(\mathbf{2}), !_{\mathbf{2}} \rangle$. Under such a realization, each $f : \mathbf{2} \to A$ will give a commutative square in A.

Theorem 6.3.1. The naturality square is arranged as:

$$\begin{array}{ccc} F_1(A_1) & \xrightarrow{\eta_{A_1}} & F_2(A_1) \\ \downarrow{\scriptstyle F_1(f)} & & \downarrow{\scriptstyle F_2(f)} \\ F_1(A_2) & \xrightarrow{\eta_{A_2}} & F_2(A_2) \end{array}$$

$$\begin{aligned} \vdash\ & \forall F_1\ F_2\ (\eta : A \to B^{\mathbf{2}})\ (f : \mathbf{2} \to A). \\ & \mathsf{Nt}(\eta, F_1, F_2) \implies \\ & \mathsf{csT}(\overline{\eta \circ f}) = \eta_{\mathsf{dom}(f)} \wedge \mathsf{csB}(\overline{\eta \circ f}) = \eta_{\mathsf{cod}(f)} \wedge \\ & \mathsf{csL}(\overline{\eta \circ f}) = F_1 \circ f \wedge \mathsf{csR}(\overline{\eta \circ f}) = F_2 \circ f \end{aligned}$$

Natural transformations also satisfy extensionality: if two of them agree on all the components then they are equal. This roots from functor extensionality as in CC_2-2. To compose natural transformations, we dualize the pushout square as in CC_4 into a pullback square.

$$\begin{array}{ccc} B^{\mathbf{3}} & \xrightarrow{B^{\beta}} & B^{\mathbf{2}} \\ \downarrow{\scriptstyle B^{\alpha}} & & \downarrow{\scriptstyle B^{0}} \\ B^{\mathbf{2}} & \xrightarrow{B^{1}} & B \end{array}$$

The pullback implies for $F_1, F_2 : A \to B^{\mathbf{2}}$ satisfy $\mathsf{Dom}(F_2) = \mathsf{Cod}(F_1)$, then there is a functor $v : A \to B^{\mathbf{3}}$. The composition of $F_2 \cdot F_2$ is then $B^{\gamma} \circ v$. Associativity, naturality, components, domain and codomain of compositions are straightforward to check. By specializing CC_5, we can construct natural transformations by comprehension.

Theorem 6.3.2.

$$\begin{aligned} & \{C,\ D\} \\ \vdash\ & \forall (F_1 : C \to D)\ (F_2 : C \to D). \\ & (\forall (c : \mathbf{1} \to C)\ (cpc : \mathbf{2} \to D). \\ & \quad \mathcal{R}[c_0 : \mathbf{1} \to C,\ cpc_0 : \mathbf{2} \to D](c, cpc) \implies \mathsf{dom}(cpc) = F_1 \circ c \wedge \mathsf{cod}(cpc) = F_2 \circ c) \wedge \\ & (\forall c : \mathbf{1} \to C.\ \exists! cpc : \mathbf{2} \to D.\ \mathcal{R}[c_0 : \mathbf{1} \to C,\ cpc_0 : \mathbf{2} \to D](c, cpc)) \wedge \\ & (\forall (f : \mathbf{2} \to C)\ (c_1 : \mathbf{1} \to C)\ (c_2 : \mathbf{1} \to C)\ (cpc_1 : \mathbf{2} \to D)\ (cpc_2 : \mathbf{2} \to D). \\ & \quad \mathsf{dom}(f) = c_1 \wedge \mathsf{cod}(f) = c_2 \wedge \\ & \quad \mathcal{R}[c_0 : \mathbf{1} \to C,\ cpc_0 : \mathbf{2} \to D](c_1, cpc_1) \wedge \mathcal{R}[c_0 : \mathbf{1} \to C,\ cpc_0 : \mathbf{2} \to D](c_2, cpc_2) \implies \\ & \quad (F_2 \circ f) \circ cpc_1 = cpc_2 \circ (F_1 \circ f)) \implies \\ & \quad \exists \eta : C \to D^{\mathbf{2}}.\ \mathsf{Nt}(\eta, F_1, F_2) \wedge \forall c : \mathbf{1} \to C.\ \mathcal{R}[c_0 : \mathbf{1} \to C,\ cpc_0 : \mathbf{2} \to D](c, \eta_c) \end{aligned}$$

Proof. The relation between $f : \mathbf{2} \to C$ and $g : \mathbf{2} \to D^{\mathbf{2}}$ is given by

$$\mathcal{R}(\mathsf{dom}(f), \mathsf{csT}(\overline{g})) \wedge \mathcal{R}(\mathsf{cod}(f), \mathsf{csB}(\overline{g})) \wedge \mathsf{csL}(\overline{g}) = F_1 \circ f \wedge \mathsf{csR}(\overline{g}) = F_2 \circ f$$

That is, f is sent to g iff g is the commutative square for the arrow f. □

The next required ingredient is *whiskering*, which amounts to composing a functor with a natural transformation. Precomposing a functor $H : X \to A$ with a natural transformation $\eta : A \to B^{\mathbf{2}}$ gives a natural transformation $X \to B^{\mathbf{2}}$ with components $(\eta *_L H)_x = \eta_{H \circ x}$. This is called *left whiskering*. As for post-composition, consider the covariant exponentiation map $H_{\mathbf{2}} : B^{\mathbf{2}} \to C^{\mathbf{2}}$ induced by $H : B \to C$, then we have $H_{\mathbf{2}} \circ \eta$ is called *right whiskering*. The components are $(H *_R \eta)_x = H \circ \eta_x$.

McLarty's paper takes the following triangular identity definition of adjunction:

6.3.2 Adjunction.

$$
\begin{aligned}
&\vdash \forall A\ X\ (L : X \to A)\ (R : A \to X)\ (\eta : X \to X^{\mathbf{2}})\ (\epsilon : A \to A^{\mathbf{2}}).\\
&\quad \mathsf{Adj}(L, R, \eta, \epsilon) \Leftrightarrow\\
&\qquad \mathsf{Nt}(\eta, \mathsf{Id}(X), R \circ L) \ \wedge\ \mathsf{Nt}(\epsilon, L \circ R, \mathsf{Id}(A)) \wedge\\
&\qquad \epsilon *_L L \cdot L *_R \eta = \mathbf{I}(L) \ \wedge\ R *_R \epsilon \cdot \eta *_L R = \mathbf{I}(R)
\end{aligned}
$$

Theorem 13 characterizes adjunctions alternatively, as families of universal arrows. A family of universal arrows from a functor $F : X \to A$ to an object $a : \mathbf{1} \to A$ consists of pairs $(x : \mathbf{1} \to X, f : \mathbf{2} \to A)$ such that for any $x' : \mathbf{1} \to X$ and $f' : \mathbf{2} \to A$ from $F \circ x'$ and to a, it factorizes uniquely as $f' = f \circ F(\hat{f})$ for some $\hat{f}$. We write $\mathsf{UFrom}(F, a, x, f)$ in that case.

6.3.3 Theorem 13. *If the pairs such that $U(x, f)$ form a family of universal arrows from F to the (common) codomain of all these f, and each object a of A uniquely determines a pair (x, f) such that f has codomain a and $U(x, f)$, then F admits a right adjoint G with unit $\eta : X \to X^{\mathbf{2}}$ and count $\epsilon : A \to A^{\mathbf{2}}$. On each object a, the component ϵ_a is the unique arrow with codomain a such that $U(G \circ a, \epsilon_a)$. Moreover, the adjunction is strictly (as the contrast to uniqueness up to isomorphism) uniquely determined.*

$$
\begin{aligned}
&\vdash \forall X\ A\ (F : X \to A).\\
&\quad (\forall (x : \mathbf{1} \to X)\ (f : \mathbf{2} \to A).\ U[x_0 : \mathbf{1} \to X, f_0 : \mathbf{2} \to A](x, f) \implies \mathsf{UFrom}(F, \mathsf{cod}(f), x, f)) \wedge\\
&\quad (\forall (a : \mathbf{1} \to A).\ \exists (x : \mathbf{1} \to X)\ (f : \mathbf{2} \to A).\ \mathsf{cod}(f) = a \ \wedge\ U[x_0 : \mathbf{1} \to X, f_0 : \mathbf{2} \to A](x, f)) \wedge\\
&\quad (\forall (a : \mathbf{1} \to A)\ (x_1 : 1 \to X)\ (x_2 : 1 \to X)\ (f_2 : 2 \to A).\\
&\qquad \mathsf{cod}(f_1) = a \ \wedge\ U[x_0 : \mathbf{1} \to X, f_0 : \mathbf{2} \to A](x_1, f_1) \wedge\\
&\qquad \mathsf{cod}(f_2) = a \wedge U[x_0 : \mathbf{1} \to X, f_0 : \mathbf{2} \to A](x_2, f_2) \implies\\
&\quad \exists!(G : A \to X)\ (\eta : X \to X^{\mathbf{2}})\ (\epsilon : A \to A^{\mathbf{2}}).\\
&\qquad \mathsf{Adj}(F, G, \eta, \epsilon) \ \wedge\ \forall a : 1 \to A.\mathsf{cod}(\epsilon_a) = a \ \wedge\ U[x_0 : \mathbf{1} \to X, f_0 : \mathbf{2} \to A](G \circ a, \epsilon_a)
\end{aligned}
$$

6.4 Duals

The difference between McLarty [33] and our formalization is most evident when it comes to the topic of dual categories.

In McLarty's approach, there is declared as primitive a function symbol $-^{\mathsf{op}}$ such that A^{op} is *a* chosen opposite for the category A. It is taken as an axiom that $(A^{\mathsf{op}})^{\mathsf{op}} = A$. From here, we already note that this is not possible in our setting, because we do not have equality on categories. The next axiom is even more problematic. It imposes that for $F : A \to B$, F^{op} is of sort $A^{\mathsf{op}} \to B^{\mathsf{op}}$ and $(F^{\mathsf{op}})^{\mathsf{op}} = F$. We are not allowed to state such an axiom in our system, even if we allow equalities between categories, because it will not "sort check": the functor F has sort $A \to B$ whereas $(F^{\mathsf{op}})^{\mathsf{op}}$ has sort $A^{\mathsf{op}\mathsf{op}} \to B^{\mathsf{op}\mathsf{op}}$.

We choose to take an approach using a predicate. Such a predicate has to be primitive because there is in general no functor from a category to its dual. In particular, there does not exist a functor sending an arrow to its opposite. We take as primitive the predicate $\mathsf{isop}(A, A')$ expressing that the category A' is the opposite of A, and $\mathsf{isopf}(f : A \to B, f' : A' \to B')$ as expressing the functor f' is the one corresponding to f under the duality. We impose an axiom on $\mathsf{isopf}(f : A \to B, f' : A' \to B')$ only when given $\mathsf{isop}(A, A')$ and $\mathsf{isop}(B, B')$. It would be more uniform to add the condition that the duality between the categories is implied by the duality of functors, but it is not required for any of these proofs, hence we leave it out.

Given the lack of the function symbol, our proofs in this part largely rely on *uniqueness*: the proof of $a = b$, where $\mathsf{isopf}(b_0, b)$ is given, usually done by proving $\mathsf{isopf}(b_0, a)$.

Our axiomatization of duality is given as:

6.4.1 $\mathbf{CC_7}$.

$$\vdash \forall A.\ \exists A'.\mathsf{isop}(A, A')$$
$$\vdash \forall A\ A'.\ \mathsf{isop}(A, A') \implies \mathsf{isop}(A', A)$$
$$\vdash \forall A\ A'\ B\ B'.\ \mathsf{isop}(A, A') \land \mathsf{isop}(B, B') \implies \forall (F : A \to B).\ \exists!(F' : A' \to B').\ \mathsf{isopf}(F, F')$$
$$\vdash \forall (F : A \to B)\ (F' : A' \to B').\ \mathsf{isopf}(F, F') \implies \mathsf{isopf}(F', F)$$
$$\vdash \forall A\ B\ (f : A \to B)\ A'\ B'\ (f' : A' \to B')\ C\ (g : B \to C)\ C'\ (g' : B' \to C').$$
$$\mathsf{isopf}(f, f') \land \mathsf{isopf}(g, g') \implies \mathsf{isopf}(g \circ g, g' \circ f')$$
$$\vdash \forall A\ A'.\ \mathsf{isop}(A, A') \implies \mathsf{isopf}(\mathsf{Id}(A), \mathsf{Id}(A'))$$

By the one-to-one of the opposite functor between a pair of opposite categories, we can easily check the opposite of a discrete category is discrete and the opposite of a monic functor is monic.

Theorem 6.4.1.

$$\vdash \forall D\ D'.\ \mathsf{Disc}(D)\ \wedge\ \mathsf{isop}(D, D')\ \Longrightarrow\ \mathsf{Disc}(D')$$
$$\vdash \forall A\ B\ (f : A \to B)\ A'\ B'\ (f' : A' \to B').$$
$$\mathsf{Mono}(f)\ \wedge\ \mathsf{isop}(A, A')\ \wedge\ \mathsf{isop}(B, B')\ \wedge\ \mathsf{isopf}(f, f')\ \Longrightarrow\ \mathsf{Mono}(f')$$

Since our axiomatization differs from McLarty's paper, we will spell out more details for the proofs in this chapter so a reader can be convinced that we are capturing the same idea as McLarty's original work.

Preservation of opposite arrows under composition will not be given as primitive but be proved as a consequence, as we assume, in addition:

6.4.2 Duality of 1, 2 and 3.

$$\vdash \mathsf{isop}(\mathbf{1}, \mathbf{1})\ \wedge\ \mathsf{isop}(\mathbf{2}, \mathbf{2})\ \wedge\ \mathsf{isop}(\mathbf{3}, \mathbf{3})\ \wedge\ \mathsf{isopf}(0, 1)$$

Note that then the opposite of an object again gives an object and the opposite of an arrow again gives an arrow, both live in the opposite category. This allows us to write when given $\mathsf{isop}(A, A')$, statements such as $\mathsf{isopf}(a : \mathbf{1} \to A, a' : \mathbf{1} \to A')$ and $\mathsf{isopf}(f : \mathbf{2} \to A, f' : \mathbf{2} \to A')$ when talking about the corresponding objects and arrows in the following proofs.

Before we arrive at the conclusion on the composition of arrows, we prove the following lemmas consecutively.

Theorem 6.4.2.

$$\vdash \forall A\ A'\ (a : \mathbf{2} \to A)\ (a' : \mathbf{2} \to A').\ \mathsf{isop}(A, A')\ \wedge\ \mathsf{isopf}(a, a')\ \Longrightarrow$$
$$\mathsf{isopf}(\mathsf{dom}(a), \mathsf{cod}(a'))\ \wedge\ \mathsf{isopf}(\mathsf{cod}(a), \mathsf{dom}(a'))$$
$$\vdash \mathsf{isopf}(\alpha, \beta)\ \wedge\ \mathsf{isopf}(\beta, \alpha)\ \wedge\ \mathsf{isopf}(\mathsf{dom}(\alpha), \mathsf{cod}(\beta))\ \wedge\ \mathsf{isopf}(\gamma, \gamma)$$
$$\vdash \forall A\ (f : \mathbf{2} \to A)\ (g : \mathbf{2} \to A).\ \mathsf{cpsb}(g, f)\ \Longrightarrow$$
$$\forall A'\ (f' : \mathbf{2} \to A)\ (g' : \mathbf{2} \to A).\ \mathsf{isop}(A, A')\ \wedge\ \mathsf{isopf}(f, f')\ \wedge\ \mathsf{isopf}(g, g')\ \Longrightarrow$$
$$\mathsf{cpsb}(f', g')\ \wedge\ \mathsf{isopf}(\Delta(f, g), \Delta(g', f'))$$

Proof. By definition of domain and codomain, proving $\mathsf{isopf}(\mathsf{dom}(a), \mathsf{cod}(a'))$ amounts to proving $\mathsf{isopf}(a \circ 0, a' \circ 1)$. Then (1) follows by CC_7-5 and the fact that $\mathsf{isopf}(0, 1)$.

To obtain $\mathsf{isopf}(\alpha, \beta)$ and $\mathsf{isopf}(\beta, \alpha)$, we use the fact that whenever we have two non-identity arrows t_1, t_2 in $\mathbf{3}$ such that $t_2 \circ_{\mathbf{3}} t_1 = \gamma$, we must have $t_2 = \beta$ and $t_1 = \alpha$. As the identity arrows in $\mathbf{3}$ are the only ones that have the same domain and codomain, the proof is completed by checking the domain and codomain using (1). We then get $\mathsf{isopf}(\mathsf{dom}(\alpha), \mathsf{cod}(\beta))$ from (1).

To prove γ is its own opposite, we use the fact that it is the only arrow from $\mathsf{dom}(\alpha)$ to $\mathsf{cod}(\beta)$. If $\mathsf{isopf}(\gamma, f')$, we deduce $\mathsf{isopf}(\mathsf{dom}(\alpha), \mathsf{cod}(f'))$ and $\mathsf{isopf}(\mathsf{cod}(\beta), \mathsf{dom}(f'))$ from (1), then we are done because $\mathsf{isopf}(\mathsf{dom}(\alpha), \mathsf{cod}(\beta))$.

Given $\mathsf{cpsb}(g, f)$, then we get $\mathsf{cpsb}(f', g')$ because $\mathsf{cod}(g')$ is the opposite of $\mathsf{dom}(g)$ and $\mathsf{dom}(f')$ is the opposite of $\mathsf{dom}(f)$, where the opposite of $\mathsf{dom}(g) = \mathsf{cod}(f)$ is unique. Assuming $\mathsf{isopf}(\Delta(f, g), t)$, to check $t = \Delta(g', f')$ we check $t \circ \alpha = g'$ and $t \circ \beta = f'$. As α and β are the inverse of each other, we prove both of them obtained by applying CC_7-5 on $\Delta(f, g) \circ \beta = g$ and $\Delta(f, g) \circ \alpha = f$.

□

Theorem 6.4.3.

$$\begin{aligned}
&\vdash \forall A\ (f : \mathbf{2} \to A)\ (g : \mathbf{2} \to A).\ \mathsf{cpsb}(g, f) \implies \\
&\quad \forall A'\ (f' : \mathbf{2} \to A)\ (g' : \mathbf{2} \to A).\ \mathsf{isop}(A, A') \wedge \mathsf{isopf}(f, f') \wedge \mathsf{isopf}(g, g') \implies \\
&\quad\quad \mathsf{isopf}(g \circ_A f, f' \circ_{A'} g')
\end{aligned}$$

Proof. By definition of arrow composition, we aim to show $\mathsf{isopf}(\Delta(f, g) \circ \gamma, \Delta(g', f') \circ \gamma)$. The result is followed by the lemmas above and CC_7-5. □

Instead of stating it as an extra axiom, we can prove the dualized version of CC_5 as the comprehension schema of defining functors from an opposite category.

6.4.3 Theorem 18.

$$\begin{aligned}
&\{A,\ B\} \\
&\vdash (\forall (f : \mathbf{2} \to A).\ \exists!(g : \mathbf{2} \to B).\ \mathcal{R}'[f_0 : \mathbf{2} \to A', g_0 : \mathbf{2} \to B](f, g)) \wedge \\
&\quad (\forall (f : \mathbf{2} \to A)\ (g : \mathbf{2} \to B).\ \mathcal{R}'[f_0 : \mathbf{2} \to A', g_0 : \mathbf{2} \to B](f, g) \implies \\
&\qquad \mathcal{R}'[f_0 : \mathbf{2} \to A', g_0 : \mathbf{2} \to B](\mathsf{id}(\mathsf{dom}(f)), \mathsf{id}(\mathsf{cod}(g))) \wedge \\
&\qquad \mathcal{R}'[f_0 : \mathbf{2} \to A', g_0 : \mathbf{2} \to B](\mathsf{id}(\mathsf{cod}(f)), \mathsf{id}(\mathsf{dom}(g)))) \wedge \\
&\quad (\forall (f : \mathbf{2} \to A)\ (g : \mathbf{2} \to A)\ (h : \mathbf{2} \to B). \\
&\qquad \mathsf{cpsb}(g, f) \wedge \mathcal{R}'[f_0 : \mathbf{2} \to A', g_0 : \mathbf{2} \to B](g \circ_A f, h) \implies \\
&\qquad \forall (f' : \mathbf{2} \to B)\ (g' : \mathbf{2} \to B). \\
&\qquad\quad \mathcal{R}'[f_0 : \mathbf{2} \to A', g_0 : \mathbf{2} \to B](f, f') \wedge \mathcal{R}'[f_0 : \mathbf{2} \to A', g_0 : \mathbf{2} \to B](g, g') \implies h = g' \circ_A f') \\
&\quad \implies \\
&\qquad \forall A'.\ \mathsf{isop}(A, A') \implies \\
&\qquad\quad \exists F : A' \to B. \\
&\qquad\quad \forall (a : \mathbf{2} \to A)\ (a' : \mathbf{2} \to A'). \\
&\qquad\quad \mathsf{isopf}(a, a') \implies \forall\ b : \mathbf{2} \to B.\ \mathcal{R}'[f_0 : \mathbf{2} \to A', g_0 : \mathbf{2} \to B](a, b) \Leftrightarrow F \circ a = b
\end{aligned}$$

Proof. Instantiate CC_5 with the relation $\mathcal{R}[f_0 : \mathbf{2} \to A', g_0 : \mathbf{2} \to B]$ with $\exists f_1 : \mathbf{2} \to A.\ \mathsf{isopf}(f_1, f_0) \wedge \mathcal{R}'(f_1, g_0)$ and use 6.4.3 to check the functoriality clause on composition.

□

The next theorem states that each category has a unique opposite category up to isomorphism (not merely categorical equivalence). As for the original approach in the paper, it is captured by declaring two operations that give A^{op} and $A^{\mathsf{op}'}$ as opposite categories of A, and regarding the assignments $\mathsf{op}_1 : A \mapsto A^{\mathsf{op}}\ f \mapsto f^{\mathsf{op}}$ and $\mathsf{op}_2 : A \mapsto A^{\mathsf{op}'},\ f \mapsto f^{\mathsf{opf}'}$ as functors $\mathbf{CAT} \to \mathbf{CAT}$. For each A, we consider the unique functor $A^{\mathsf{op}} \to A^{\mathsf{op}'}$ specified by $f^{\mathsf{opf}} \mapsto f^{\mathsf{opf}'}$. Each of such a functor can be regarded as the component of a natural transformation from op_1 to op_2. However, the domain and codomain of the op_1 and op_2 here are $\mathbf{CAT}$, so they are meta-functors. Therefore, instead of using the notion of natural transformation as we used in the definition of adjunction, this time we have to state this condition separately. Another notable difference is that we choose not to use any operation but merely predicates, to axiomatize duality between categories. As a result, instead of formulating the theorem between a pair of chosen operations, we are only able to state it for any two opposite categories of the same category.

Given a category A, we write $\mathsf{isDiso}(A, i : A_1 \to A_2)$ when A_1 and A_2 are both duals of A and i is an isomorphism building the correspondence between copies of the same arrow from A.

$$\begin{aligned}
\vdash\ & \forall A\ A_1\ A_2\ (i : A_1 \to A_2).\ \mathsf{isDiso}(A, i) \Leftrightarrow \\
& (\mathsf{isop}(A, A_1) \wedge \mathsf{isop}(A, A_2) \wedge \mathsf{Iso}(i) \wedge \\
& (\forall (f : \mathbf{2} \to A)\ (f_1 : \mathbf{2} \to A_1)\ (f_2 : \mathbf{2} \to A_2).\ \mathsf{isopf}(f, f_1) \wedge \mathsf{isopf}(f, f_2) \Leftrightarrow i \circ f_1 = f_2))
\end{aligned}$$

Our formulation of Theorem 19, as displayed below, is actually a stronger version than the counterpart in the paper.

6.4.4 Theorem 19.

$$\begin{aligned}
\vdash\ & \forall A\ A_1\ A_2\ (i_A : A_1 \to A_2)\ B\ B_1\ B_2\ (i_B : B_1 \to B_2). \\
& \mathsf{isDiso}(A, i_A) \wedge \mathsf{isDiso}(B, i_B) \Longrightarrow \\
& \forall (f : A \to B)\ (f_1 : A_1 \to B_1)\ (f_2 : A_2 \to B_2). \\
& \mathsf{isopf}(f, f_1) \wedge \mathsf{isopf}(f, f_2) \Longrightarrow f_2 \circ i_A = i_B \circ f_1 \\
\vdash\ & \forall A\ A_1\ A_2.\ \mathsf{isop}(A, A_1) \wedge \mathsf{isop}(A, A_2) \Longrightarrow \exists! i : A_1 \to A_2.\ \mathsf{isDiso}(A, i)
\end{aligned}$$

Proof. The naturality equation is proved by extensionality. By definition of i_B, checking $i_B \circ f_1 \circ a = f_2 \circ i_A \circ a$ for $a : \mathbf{2} \to A_1$ amounts to checking the arrow opposite to $f_2 \circ i_A \circ a$ is the arrow opposite to $f_1 \circ a$. As the opposite of arrows always exists, it suffices to give an arrow that is the opposite of both. Take the unique arrow $b : \mathbf{2} \to B$ such that $\mathsf{isopf}(f_1 \circ a, b)$ is the witness, we prove we also have $\mathsf{isopf}(b, f_2 \circ i_A \circ a)$. This is by two

applications of the fact that opposites are preserved by composition: We have $b = f \circ a_0$ where a_0 is the opposite of a_0 in A, and the result follows because $\mathsf{isopf}(f, f_2)$ and $\mathsf{isopf}(a_0, i_A \circ a)$.

The uniqueness part of the unique existence is trivial. To establish existence, we apply Theorem 18 by taking the relation $\mathcal{R}(f : \mathbf{2} \to A, g : \mathbf{2} \to A_2)$ to be $\mathsf{isopf}(f, g)$. This gives our desired functor from, in particular, the opposite category A_1 of A to A_2. □

6.5 Internal categories

Given two categories C_1 and C_0, with functors $i : C_0 \to C_1$, $d_0, d_1 : C_1 \to C_0$, we can regard C_0 as resembling a collection of objects and C_1 as resembling a collection of arrows, with the given functors specifying the identity, domain, and codomain. If in addition, we have a functor giving the "composition" of arrows, plus some conditions are satisfied, then all the information intuitively specifies a category, which we call an *internal category*. In summary, an internal category consists of the information as displayed in the following diagram:

$$C_1 \times_{C_0} C_1 \xrightarrow{r} C_1 \underset{d_0}{\overset{\overset{i}{\leftarrow}}{\underset{d_1}{\rightrightarrows}}} C_0$$

Here the object $C_1 \times_{C_0} C_1$ is *the* pullback of the "codomain" functor d_1 and the "domain" functor d_0. The composition functor r is from this pullback, which means that it only composes pairs of internal arrows with matching domain and codomain.

To formalize the definition of an internal category, we expect to define a predicate given the four arrows above. However, to specify the r, we need to fix the pullback first. There are two options: carrying the projection arrows in the pullback as inputs as well or fixing a canonical choice of the pullback, so the definition only takes the above four inputs. As the latter makes the definition look simpler, we take it as our choice.

This in turn requires us to create function symbols (relying on the specification rule, as before) that produce a pullback. The pullback of functors $F_1 : X \to Z$ and $F_2 : Y \to Z$ consists of three pieces of information, given by function symbols Pbo, Pba_1, and Pba_2. The left square below is the pullback square using our notations whereas the right square uses the usual textbook notations.

$$\begin{array}{ccc} \mathsf{Pbo}(F_1, F_2) & \xrightarrow{\mathsf{Pba}_2(F_1,F_2)} & Y \\ {\scriptstyle \mathsf{Pba}_1(F_1,F_2)}\downarrow & & \downarrow{\scriptstyle F_2} \\ X & \xrightarrow{F_1} & Z \end{array} \qquad \begin{array}{ccc} X \times_Z Y & \xrightarrow{p_2} & Y \\ {\scriptstyle p_1}\downarrow & & \downarrow{\scriptstyle F_2} \\ X & \xrightarrow{F_1} & Z \end{array}$$

By construction, we have $\mathsf{isPb}(f, g, \mathsf{Pba}_1(f, g), \mathsf{Pba}_2(f, g))$ for any functor f and g once they are functors to the same category. Some defining clauses for being an internal category, written

as $\mathsf{Icat}(d_0, d_1, i, r)$, are trivial to express. These are:

$$\text{Domain of identity :} \quad d_0 \circ i = \mathsf{Id}(C_0) \tag{6.2}$$

$$\text{Codomain of identity :} \quad d_1 \circ i = \mathsf{Id}(C_0) \tag{6.3}$$

$$\text{Domain of composition :} \quad d_0 \circ r = d_0 \circ \mathsf{Pba}_1(d_1, d_0) \tag{6.4}$$

$$\text{Codomain of composition :} \quad d_1 \circ r = d_1 \circ \mathsf{Pba}_2(d_1, d_0) \tag{6.5}$$

The remaining ones are the identity laws and the associativity of internal composition. In diagrams with mathematical language, the identity law for pre-composition is expressed as:

$$\begin{array}{ccc} C_0 \times_{C_1} C_1 & \xrightarrow{i \times \mathsf{Id}(C_1)} & C_1 \times_{C_1} C_1 \\ & \searrow^{p_2} & \downarrow r \\ & & C_1 \end{array}$$

The object $C_0 \times_{C_1} C_1$, regarded as object-arrow pairs such that the first entry is the domain of the second entry, is formalized as the pullback $\mathsf{Pbo}(\mathsf{Id}(C_0), d_0)$. The functor $i \times \mathsf{Id}(C_1)$ is the unique arrow induced by the "internal identity" functor and the identity functor, the factorization through the pullback does exist because of 1.5. However, we are not able to use this mathematical notation to formalize this arrow by applying a function symbol to the two arguments.

Creating function symbols to capture such induced arrows might seem to be possible, as we have done this for the composition of arrows. But in contrast with what we did for defining the Δ function symbol in Section 1.2, here we do not have a universally applicable machinery to associate each pair of arrows from any category to a pullback object. The heart of the difference is that even the precondition as in the definition of Δ does not hold, we are guaranteed the existence of the arrow of the sort $\mathbf{3} \longrightarrow A$ when given a $\mathbf{2} \longrightarrow A$. However, it is no longer true for pullback objects: before checking the commutativity condition, there is no way to make sure that there exists an arrow to the pullback. As a consequence, the function specification rule does not apply here.

The only remaining solution is to use equations to characterize the top arrow, which results in the definition below:

6.5.1 identity law for pre-composition.

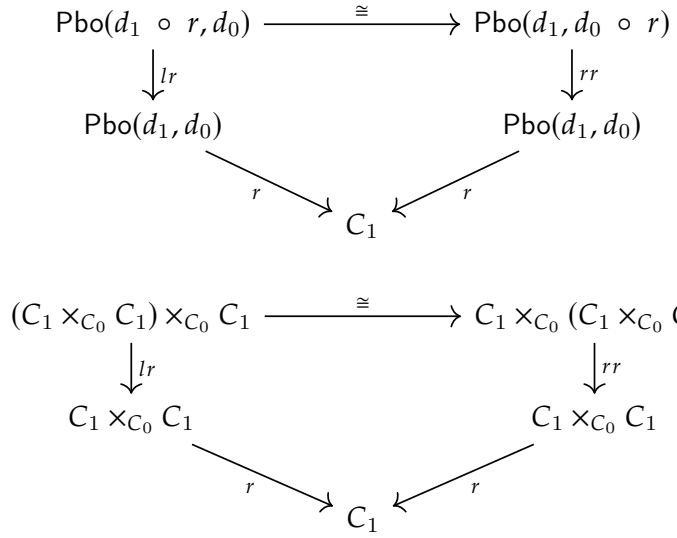
$$\begin{aligned}
&\vdash \forall C_0\ C_1\ (d_0 : C_1 \to C_0)\ (d_1 : C_1 \to C_0)\ (i : C_0 \to C_1)\ (r : \mathsf{Pbo}(d_1, d_0) \to C_1).\\
&\quad \mathsf{lidL}(d_0, d_1, i, r) \Leftrightarrow\\
&\quad \forall ci_1 : \mathsf{Pbo}(\mathsf{Id}(C_0), d_0) \to \mathsf{Pbo}(d_1, d_0).\\
&\qquad \mathsf{Pba}_1(d_1, d_0) \circ ci_1 = i \circ \mathsf{Pba}_1(\mathsf{Id}(C_0), d_0) \wedge \mathsf{Pba}_2(d_1, d_0) \circ ci_1 = \mathsf{Pba}_2(\mathsf{Id}(C_0), d_0)\\
&\qquad \Longrightarrow r \circ ci_1 = \mathsf{Pba}_2(\mathsf{Id}(C_0), d_0)
\end{aligned}$$

By the universal property of the pullback, despite the quantification, the functor ci_1 satisfying the equations is unique. If we replace the right-hand side with

$$\begin{aligned}
&\exists ci_1 : \mathsf{Pbo}(\mathsf{Id}(C_0), d_0) \to \mathsf{Pbo}(d_1, d_0).\\
&\qquad \mathsf{Pba}_1(d_1, d_0) \circ ci_1 = i \circ \mathsf{Pba}_1(\mathsf{Id}(C_0), d_0) \wedge \mathsf{Pba}_2(d_1, d_0) \circ ci_1 = \mathsf{Pba}_2(\mathsf{Id}(C_0), d_0) \wedge\\
&\quad r \circ ci_1 = \mathsf{Pba}_2(\mathsf{Id}(C_0), d_0)
\end{aligned}$$

The definition does not change.

The advantage of the "chosen pullback" function symbol is evident when it comes to the formalization of the associativity clause. Below, the top diagram is the diagrammatic definition using formalized function symbols and the corresponding diagram in the mathematical symbols is on the bottom.



Using the same strategy of capturing the unique induced arrow with characterizing equations, internal associativity is formalized as:

6.5.2 Internal Associativity.

$$
\begin{aligned}
&\vdash \mathsf{lassoc}(d_0, d_1, i, r) \Leftrightarrow \\
&\quad \forall lr : \mathsf{Pbo}(d_1 \circ r, d_0) \to \mathsf{Pbo}(d_1, d_0) \\
&\quad aiso : \mathsf{Pbo}(d_1 \circ r, d_0) \to \mathsf{Pbo}(d_1, d_0 \circ r) \\
&\quad rr : \mathsf{Pbo}(d_1, d_0 \circ r) \to \mathsf{Pbo}(d_1, d_0). \\
&\qquad \mathsf{Pba}_1(d_1, d_0) \circ l = r \circ \mathsf{Pba}_1(d_1 \circ r, d_0) \wedge \mathsf{Pba}_2(d_1, d_0) \circ l = \mathsf{Pba}_2(d_1 \circ r, d_0) \wedge \\
&\qquad \mathsf{Pba}_1(d_1, d_0) \circ rr = \mathsf{Pba}_1(d_1, d_0 \circ r) \wedge \mathsf{Pba}_2(d_1, d_0) \circ rr = r \circ \mathsf{Pba}_2(d_1, d_0 \circ r) \wedge \\
&\qquad \mathsf{Pba}_1(d_1, d_0 \circ r) \circ aiso = \mathsf{Pba}_1(d_1, d_0) \circ \mathsf{Pba}_1(d_1 \circ r, d_0) \wedge \\
&\qquad \mathsf{Pba}_1(d_1, d_0) \circ \mathsf{Pba}_2(d_1, d_0 \circ r) \circ aiso = \mathsf{Pba}_2(d_1, d_0) \circ \mathsf{Pba}_1(d_1 \circ r, d_0) \wedge \\
&\qquad \mathsf{Pba}_2(d_1, d_0) \circ \mathsf{Pba}_2(d_1, d_0 \circ r) \circ aiso = \mathsf{Pba}_2(d_1 \circ r, d_0) \\
&\qquad\quad \Longrightarrow r \circ l = r \circ rr \circ aiso
\end{aligned}
$$

As before, the definition above admits an equivalent characterization using existential quantifiers instead.

Conjuncting the former 4 clauses with the above completes the definition of internal category.

Between two internal categories, we have the notion of *internal functors*. Let C_0, C_1 and D_0, D_1 and functors below form internal categories, a pair of functors $f_0 : C_0 \to D_0$ and $f_1 : C_1 \to D_1$ form an internal functor if the two diagrams:

$$
\begin{array}{ccc}
C_1 \times_{C_0} C_1 & \xrightarrow{f_1 \times f_1} & D_1 \times_{D_0} D_1 \\
\downarrow{\scriptstyle r} & & \downarrow{\scriptstyle r'} \\
C_1 & \xrightarrow{\quad f_1 \quad} & D_1
\end{array}
\qquad
\begin{array}{ccccc}
C_1 & \overset{d_0}{\underset{d_1}{\rightrightarrows}} & C_0 & \xrightarrow{\quad i \quad} & C_1 \\
\downarrow{\scriptstyle f_1} & & \downarrow{\scriptstyle f_0} & & \downarrow{\scriptstyle f_1} \\
D_1 & \overset{d'_0}{\underset{d'_1}{\rightrightarrows}} & D_0 & \xrightarrow{\quad i' \quad} & D_1
\end{array}
$$

commutes. The right diagram expresses the preservation of identity and domain and codomain of internal arrows, expressing the clauses:

$$
\begin{aligned}
\text{Preservation of domain :} \quad & d'_0 \circ f_1 = f_0 \circ d_0 && (6.6) \\
\text{Preservation of codomain :} \quad & d'_1 \circ f_1 = f_0 \circ d_1 && (6.7) \\
\text{Preservation of identity :} \quad & i' \circ f_0 = f_1 \circ i && (6.8)
\end{aligned}
$$

The left diagram expresses the preservation of internal composition, with the top arrow captured by quantification. It holds if and only if:

$$
\begin{aligned}
&\forall ff : \mathsf{Pbo}(d_1, d_0) \to \mathsf{Pbo}(d'_1, d'_0). \\
&\quad \mathsf{Pba}_1(d'_1, d'_0) \circ ff = f_1 \circ \mathsf{Pba}_1(d_1, d_0) \wedge \mathsf{Pba}_2(d'_1, d'_0) \circ ff = f_1 \circ \mathsf{Pba}_2(d_1, d_0) \\
&\quad \Longrightarrow r' \circ ff = f_1 \circ r
\end{aligned}
$$

is true.

The proof of theorems in this section mostly boils down to unwinding and checking the definition of internal categories and internal functors. As their definition uses chosen pullback instead of arbitrary pullback, when we are given an assumption of having an internal category/functor, the clauses that immediately appear only work to a particular pullback, and when a proof of being an internal category/functor is required, we are required to prove it for a particular pullback instead of an arbitrary pullback. We address such an issue and enable full flexibility by proving an alternative version of each clause that refers to a composition, using a predicate stating "is the internal composition".

6.5.3 "Is the internal composition".

$$\begin{aligned}\vdash \forall & C_0\ C_1\ (d_0 : C_1 \to C_0)\ (d_1 : C_1 \to C_0)\ C_1C_1\ (p_1 : C_1C_1 \to C_1)\ (p_2 : C_1C_1 \to C_1)\ (r : C_1C_1 \to C_1) \\ & A\ (f : A \to C_1)\ (g : A \to C_1)\ (gf : A \to C_1). \\ & \mathsf{isio}(d_0, d_1, p_1, p_2, r, g, f, gf) \Leftrightarrow \\ & \mathsf{isPb}(d_1, d_0, p_1, p_2) \ \wedge\ d_0 \circ r = d_0 \circ p_1 \ \wedge\ d_1 \circ r = d_1 \circ p_2 \ \wedge\ d_0 \circ g = d_1 \circ f \ \wedge \\ & \exists fg_0 : A \to C_1C_1.\ p1 \circ fg_0 = f \ \wedge\ p_2 \circ fg_0 = g \ \wedge\ r \circ fg_0 = gf\end{aligned}$$

It is natural to define such a predicate. As we discussed before, we are not able to create a function symbol that produces the factorization through the pullback. Note that instead of only regarding "elements", which are generally regarded as functors from $\mathbf{1}$, of C_1, we use A as the source of f and g. This is required by the commutative clauses, and replacing the A with $\mathbf{1}$ will weaken the definition. But using the fact that $\mathbf{2}$ is the generator, one can prove an equivalent version with A replaced by $\mathbf{2}$. Taking terminology from topos theory, we call the functors $A \to C_1$ "generalized internal arrows".

This notion of internal composition takes the pullback information as a part of the input. Once we have assumptions about one pullback and want to move to another choice of pullback, we use the following lemma to transfer.

6.5.4 Compatibility of internal composition respects to a different choice of pullbacks. *For two pullbacks and the canonical isomorphism between them, the composition of two generalized internal arrows is the same with respect to the same internal composition.*

$$\begin{aligned}\vdash \forall & C_0\ C_1\ (d_0 : C_1 \to C_0)\ (d_1 : C_1 \to C_0) \\ & C_1C_1\ (p_1 : C_1C_1 \to C_1)\ (p_2 : C_1C_1 \to C_1)\ Pb\ (p'_1 : Pb \to C_1)\ (p'_2 : Pb \to C_1) \\ & i : Pb \to C_1C_1. \\ & \mathsf{isPb}(d_1, d_0, p_1, p_2) \ \wedge\ \mathsf{isPb}(d_1, d_0, p'_1, p'_2) \ \wedge\ p_1 \circ i = p'_1 \ \wedge\ p_2 \circ i = p'_2 \implies \\ & \quad \forall r\ A\ (f : A \to C_1)\ (g : A \to C_1)\ (gf : A \to C_1). \\ & \qquad \mathsf{isio}(d_0, d_1, p_1, p_2, r, g, f, gf) \Leftrightarrow \mathsf{isio}(d_0, d_1, p'_1, p'_2, r \circ i, g, f, gf)\end{aligned}$$

Therefore, our decision to pick one pullback to state the definitions does not result in any real inflexibility.

Using the predicate for "being the composition", we can now reformulate the commutative clauses of all of the previous diagrams into a more workable version in terms of generalized internal arrows.

- Given the composition has the correct domain and codomain and the identity has the correct codomain, the identity law for post composition holds if and only if for every generalized internal object c and generalized internal arrow a, the internal composition of a and the generalized identity on c is a itself.

$$\begin{aligned}
&\forall C_1\ C_0\ (d_0 : C_1 \to C_0)\ (d_1 : C_1 \to C_0)\ (i : C_0 \to C_1)\ r.\\
&\quad d_1 \circ i = \mathsf{Id}(C_0) \wedge d_0 \circ r = d_0 \circ \mathsf{Pba}_1(d_1, d_0) \wedge d_1 \circ r = d_1 \circ \mathsf{Pba}_2(d_1, d_0) \implies\\
&\quad (\mathsf{lidL}(d_0, d_1, i, r) \Leftrightarrow\\
&\qquad \forall T\ (c : T \to C_0)\ (a : T \to C_1).\\
&\qquad\quad d_0 \circ a = c \implies \mathsf{isio}(d_0, d_1, \mathsf{Pba}_1(d_1, d_0), \mathsf{Pba}_2(d_1, d_0), r, a, i \circ c, a))
\end{aligned}$$

- If the internal composition gives the correct domain and codomain, then internal associativity is satisfied if, for all generalized internal arrows, there exists a common arrow that is the composition of the generalized internal arrows obtained by internal composition in either order.

$$\begin{aligned}
&\vdash \forall C_1\ C_0\ (d_0 : C_1 \to C_0)\ (d_1 : C_1 \to C_0)\ (i : C_0 \to C_1)\ (r : \mathsf{Pbo}(d_1, d_0) \to C_1).\\
&d_0 \circ r = d_0 \circ \mathsf{Pba}_1(d_1, d_0) \wedge d_1 \circ r = d_1 \circ \mathsf{Pba}_2(d_1, d_0) \implies\\
&(\mathsf{lassoc}(d_0, d_1, i, r) \Leftrightarrow\\
&\forall T\ (t_3 : T \to C_1)\ (t_2 : T \to C_1)\ (t_1 : T \to C_1).\\
&d_0 \circ t_2 = d_1 \circ t_1 \wedge d_0 \circ t_3 = d_1 \circ t_2 \implies \exists t_{321}\ t_{32}\ t_{21} : \mathbf{2} \to C_1.\\
&\mathsf{isio}(d_0, d_1, \mathsf{Pba}_1(d_1, d_0), \mathsf{Pba}_2(d_1, d_0), r, t_2, t_1, t_{21}) \wedge\\
&\mathsf{isio}(d_0, d_1, \mathsf{Pba}_1(d_1, d_0), \mathsf{Pba}_2(d_1, d_0), r, t_3, t_2, t_{32}) \wedge\\
&\mathsf{isio}(d_0, d_1, \mathsf{Pba}_1(d_1, d_0), \mathsf{Pba}_2(d_1, d_0), r, t_{32}, t_1, t_{321}) \wedge\\
&\mathsf{isio}(d_0, d_1, \mathsf{Pba}_1(d_1, d_0), \mathsf{Pba}_2(d_1, d_0), r, t_3, t_{21}, t_{321}))
\end{aligned}$$

- Given two internal categories and a pair of functors such that the internal domain and codomain are preserved. Internal composition is preserved if and only if for every composable pair of generalized internal arrows, the composition of their image is the

image of their composition.

$$
\begin{array}{l}
\forall C_0\ C_1\ (d_0 : C_1 \to C_0)\ (d_1 : C_1 \to C_0)\ (i : C_0 \to C_1)\ r \\
\quad D_0\ D_1\ (d_0' : D_1 \to D_0)\ (d_1' : D_1 \to D_0)\ (i' : D_0 \to D_1)\ r'\ (f_0 : C_0 \to D_0)\ (f_1 : C_1 \to D_1). \\
\quad\quad \mathsf{Icat}(d_0, d_1, i, r) \wedge \mathsf{Icat}(d_0', d_1', i', r') \wedge d_0' \circ f_1 = f_0 \circ d_0 \wedge d_1' \circ f_1 = f_0 \circ d_1 \implies \\
\quad\quad\quad (\mathsf{Ipreso}(d_0, d_1, i, r, d_0', d_1', i', r', f_0, f_1) \Leftrightarrow \\
\quad\quad\quad\quad \forall T\ (f : T \to C_1)\ (g : T \to C_1).\ d_0 \circ g = d_1 \circ f \implies \\
\quad\quad\quad\quad\quad \exists gf : T \to C_1. \\
\quad\quad\quad\quad\quad\quad \mathsf{isio}(d_0, d_1, \mathsf{Pba}_1(d_1, d_0), \mathsf{Pba}_2(d_1, d_0), r, g, f, gf) \wedge \\
\quad\quad\quad\quad\quad\quad \mathsf{isio}(d_0', d_1', \mathsf{Pba}_1(d_1', d_0'), \mathsf{Pba}_2(d_1', d_0'), r', f_1 \circ g, f_1 \circ f, f_1 \circ gf))
\end{array}
$$

The first standard example where internal category naturally occurs is that each actual category A gives an internal category. The category of internal arrows is given by $A^{\mathbf{2}}$. We write this fact as $\mathsf{Icat}(\mathsf{Id}_0(A), \mathsf{Id}_1(A), \mathsf{Ii}(A), \mathsf{Ir}(A))$, where $\mathsf{Id}_0(A)$ and $\mathsf{Id}_1(A)$ are functors $A^0, A^1 : A^{\mathbf{2}} \to A$ as in Section 1.3, the internal identity is $\mathsf{Tp}(\pi_2(2, A))$ and the internal composition is given by composing A^γ with the canonical isomorphism $\mathsf{Pbo}(\mathsf{Id}_0(A), \mathsf{Id}_1(A)) \cong A^{\mathbf{3}}$.

A functor $F : A \to B$ then induces a functor between their corresponding internal categories. The effect on arrows of A is simply composition, which suggests that the corresponding $\mathsf{Sq}(F) : A^{\mathbf{2}} \to B^{\mathbf{2}}$ is given by $\mathsf{Tp}(F \circ \mathsf{ev}(\mathbf{2}, A))$. This correspondence is actually one-to-one: Existence amounts to checking the conditions for being an internal functor. As for uniqueness, it is elegantly proved as the following neat result:

6.5.5 Theorem 24. *If two functors $F : A \to B$ and $G : A^{\mathbf{2}} \to B^{\mathbf{2}}$ form an internal functor between internal categories of actual categories, then G is the functor $\mathsf{Sq}(F)$.*

$$
\begin{array}{l}
\vdash \forall A\ B\ (F : A \to B)\ (G : A^{\mathbf{2}} \to B^{\mathbf{2}}). \\
\quad \mathsf{IFun}(\mathsf{Id}_0(A), \mathsf{Id}_1(A), \mathsf{Ii}(A), \mathsf{Ir}(A), \\
\quad\quad \mathsf{Id}_0(B), \mathsf{Id}_1(B), \mathsf{Ii}(B), \mathsf{Ir}(B), F, G) \implies G = \mathsf{Sq}(F)
\end{array}
$$

Remark on Theorem 27 We formalize 26 out of 27 theorems that appear in McLarty [33]. Theorem 27 is about comprehension of categories, i.e., forming a new category out of existing categories. In private communication, McLarty and I determined that the theorem statement is flawed. The theorem is attempting to describe situations in which an internal category corresponds to an existing external category but turns out to be more subtle than expected, as many versions of the comprehension schema in many foundations. Now we are not even sure what would even be the correct statement of this statement. Therefore, our formalization catches an important error to be fixed as a piece of future work.

Chapter 7

Conclusion and Future Work

We end the thesis by summarizing our work and pointing to future directions that could lead to more interesting research.

7.1 Conclusion

We review the path we travel along: We begin by setting our goal as designing a theorem-proving system that is suitable for experimenting with traditional-style mathematical foundations, followed by an argument convincing the reader that this is a desirable goal.

Before we work on designing a new system, we make sure that we indeed need a new one by examining existing systems. We highlighted their advantages, but draw the conclusion that none of them are suitable for our particular purpose.

We develop the kernel of our system in Chapter 2. The syntax is constructed by three layers: sorts, terms, and formulas. We highlight our usage of formula variables, which allows us to present axiom schemata in a readable manner, and the context, which prevents us from performing proofs with non-existing terms. Moreover, we provide our function specification rule, which equips users to create function symbols according to any equivalence relation they like, and which could serve as a Skolemization rule. Along with the presentation of our logic, we give some examples showing how various mathematical foundations are captured in our logic, showing our system is versatile.

The implementation of the theorem prover based on our system is introduced in Chapter 3. We explain how we implement the primitive rules on terms and formula instantiation. Our parser is also specially designed, where the type-inference appeals to a unification algorithm.

In Chapter 4, we address possible concerns about the apparent higher-orderness of formula variables. In this chapter, we describe the formalization of our system in the theorem prover HOL4. We formally verify that the well-formedness of our syntax is preserved by operations we have implemented, and hence our theorem prover only produces well-formed theorems.

Eventually, we reach the goal that formula variables could be effectively eliminated. They do not add any extra power to our system and are no more than a convenience.

The rest of the thesis consists of examples. Two structural set theories are formalized in Chapter 5. The first one is the very classic example, namely Lawvere's ETCS. By automating the establishment of internal logic, we make proving theorems in ETCS a smooth procedure. We could hence use ETCS as simply as proving theorems in HOL since its internal logic equips it with such power. Meanwhile, we maintain the syntactical simplicity of first-order logic since the higher-order is a derived notion. A foundation in a completely different flavor, which is the so-called "categorical foundation", is formalized in Chapter 6. Despite the fact that the section on dual categories in McLarty's original paper does not admit a direct translation to our system due to the "type"(sort) issue, our re-axiomatization provides a reasonable alternative that captures all the theorems in that section.

As an overall summary, our system is simple yet powerful enough to formalize a significant amount of mathematics. Our system encodes mathematical presentations naturally. In particular, it is possible to work with axiom schemata just as we do when writing a paper proof. Formula variables enable such expressions and are safe to use. By exploring mathematical foundations for theorem proving, we discover that their usage could fix some problems arising due to the lack of expressiveness of simpler systems. Moreover, their complexity is certainly far below dependently typed systems, suggesting that using DTT and its extensions is not the only way to get the desired power. In conclusion, by choosing suitable mathematical foundations, our system can be specialized into theorem-proving systems that sit at a sweet spot between simplicity and expressiveness.

7.2 Future Work

From here, more interesting work could be done pursuing many angles. We list some options.

Library Construction Certainly, for a new theorem prover, we need to catch up with the pioneers, whose libraries are already rich, so we can compare proofs in our systems and in others. In other words, we would like more mechanization to be done. The Formalizing 100 Theorems list [4] might be taken as a to-do list.

Formalization could take place in the existing foundation DiaToM/SEAR. It would also be interesting to develop new systems in DiaToM, following the approach laid out in Chapter 2.

More proof tools, such as derived rules and tactics that are possibly specific to a particular system, could be developed alongside formalizations. For instance, we could develop decision procedures for natural numbers/integer arithmetics, fully automated quotients, induction, and

co-induction packages, as well as those for record types. Another broad topic is the transfer of proofs from other systems. Some mathematical systems have much similarity in the sense that one notion in one system has a counterpart in the other. As we saw in our formalization of ETCS and SEAR, some proofs in one of them could be copied and pasted into another one to directly work. It would be ideal if we instead had an algorithm to transfer all the theorems, so we guarantee the uniformness of the transfer and avoid repeating the same proof twice.

In addition to transferring proofs within the DiaToM prover, it should be possible to develop programs that migrate proofs from other systems into ours, and ours into others, as well. For instance, all the proofs we have done so far in all these three theories should be able to be transferred into a dependently typed system such as Lean. In the other direction, if the constructors and types that are specific to dependent type theory are not used, we could transfer a DTT theorem into our setting.

It would be also interesting to investigate different approaches for constructing a proof. For instance, it would be possible to complete the construction of Beth cardinals as mentioned in Section 5.2.9 either by proving a restricted version of structural replacement theorem within SEAR, or, according to the following paragraphs, prove the full structural replacement theorem using CCAF, and then develop a mechanism of applying this CCAF axiom to SEAR and get the set of Beth numbers.

Variants and Enrichment of the Logic Whereas all the current formalization in DiaToM is done in classical logic, variants for FOL are of mathematical interest when they are put together with foundations as well. Mild modification to the kernel to support, say intuitionistic logic, could be adopted for formalizing foundations with the same axioms but different available logical operations.

According to the discussion at the end of Chapter 2, we might change the implementation of formula variable instantiation to support syntax checking before instantiation.

Putting Theories Together We would like to investigate applying a theory as a meta-theory to reasoning about another theory. The picture is motivated by [2], where CCAF serves as the meta-theory and ETCS as the object-theory. CCAF is a theory about inter-category reasoning, whereas the whole workspace of ETCS is an instance of a category. Such a category could be captured by a CCAF term. As CCAF can prove general statements starting with the quantification "for every category...", after proving say, every monomorphism is injective on elements, we could apply it to a category satisfying ETCS: The application will ask us to prove that the definition of a monomorphism in ETCS coincide with the notions specified in CCAF, and we could capture the concept "elements" properly in CCAF. Once it is proved, we should be able to instantiate the "for every" to be the category we work in for ETCS and take

the conclusion. The interesting problem is determining how the procedure would be carried out formally.

More Verification We outline many possible extensions above. Each of them should be carried out carefully. We then suggest that before each adjustment is added, we should do some verification to make sure it does not break the current system. Apart from that, as the current formalization in HOL is only up to the elimination of formula variables, it would be helpful to also verify our type-inference algorithm and the further step of translation to sorted FOL. And also the backward transfer, answering which part of sorted FOL does our logic capture. Even when it is not in the kernel, formalizing will also be helpful for us to implement more tactics and simplification tools, to make sure they are working effectively. As far as we have experimented, HOL is a sufficiently good candidate for further formalization, and we expect more verification to take place from there.

Appendix A

Installing and Using DiaToM

A.1 Installation

A potential user might be interested in install the program and run the proofs interactively. The source code for this implementation is available from https://github.com/u5943321/DiaToM Firstly, install the most recent version of Poly/ML from polyml.org. In this directory, build the system with make. It will build an executable called vscore. Executing vscore starts a Poly/ML REPL. Various logical formalisations can then be loaded. For example, in the SEAR directory, run

```
../vscore
```

and then

```
> use "SEARmaster.ML";
```

Similarly, use the ETCSmaster.ML file, having started vscore in the ETCS directory.

A.2 User Interface

Even before we formally introducing the logic, it makes sense to demonstrate how the theorem prover works. In the following example we consider the structural set theory SEAR, and we want to prove for every two elements x, y of the set A and B respectively, there exists an element in the product set $A \times B$ that projects to x and y. The goal displayed in the implemented theorem prover looks as follows:

```
> val it =

  ------------------------------------------------------------------------
  !(A : set)  (B : set)  (x : mem(A#))  (y : mem(B#)).
              ?!(r : mem(A# * B#)).
                App(p1(A#, B#), r#) = x# & App(p2(A#, B#), r#) = y#:
  proofmanager.proof
```

The "#" symbol after a variable indicates that it is bound. After stripping the quantifier, we store the free variables that appears in the list of *context* above the assumption list, as:

```
 > # 1 subgoal:
 val it =
    BA(y : mem(B))(x : mem(A))

    ------------------------------------------------------------------------
    ?!(r : mem(A * B)). App(p1(A, B), r#) = x & App(p2(A, B), r#) = y
```

We use tactics to put relevant theorems into the assumption list and expand the definition of existential quantifier, yields:

```
> # # 1 subgoal:
val it =
   BA(y : mem(B))(x : mem(A))
   1.!(x : mem(A))  (y : mem(B)).
               ?(r : mem(A * B)).
                 App(p1(A, B), r#) = x# & App(p2(A, B), r#) = y#
   2.!(r : mem(A * B))  (s : mem(A * B)).
               App(p1(A, B), r#) = App(p1(A, B), s#) &
               App(p2(A, B), r#) = App(p2(A, B), s#) ==> r# = s#
   ------------------------------------------------------------------------
   ?(r : mem(A * B)).
               (App(p1(A, B), r#) = x & App(p2(A, B), r#) = y) &
               !(r' : mem(A * B)).
                 App(p1(A, B), r'#) = x & App(p2(A, B), r'#) = y ==> r'# = r#
   : proofmanager.proof
```

There is a tactic allowing us to specialize assumptions. Here we specialize the bound x and y into the free variables x and y in the context. This gives us an r satisfies two equations.

```
> # # 1 subgoal:
val it =
   BA(y : mem(B))(x : mem(A))(r : mem(A * B))
   1.!(r : mem(A * B))  (s : mem(A * B)).
               App(p1(A, B), r#) = App(p1(A, B), s#) &
               App(p2(A, B), r#) = App(p2(A, B), s#) ==> r# = s#
   2.App(p1(A, B), r) = x
   3.App(p2(A, B), r) = y
   ------------------------------------------------------------------------
   ?(r : mem(A * B)).
               (App(p1(A, B), r#) = x & App(p2(A, B), r#) = y) &
               !(r' : mem(A * B)).
                 App(p1(A, B), r'#) = x & App(p2(A, B), r'#) = y ==> r'# = r#
   : proofmanager.proof
```

We feed the free variable r on the end of the context list to be the witness of the the existential quantifier. The goal then becomes:

```
> # 1 subgoal:
val it =
   BA(y : mem(B))(x : mem(A))(r : mem(A * B))
   1.!(r : mem(A * B))  (s : mem(A * B)).
               App(p1(A, B), r#) = App(p1(A, B), s#) &
               App(p2(A, B), r#) = App(p2(A, B), s#) ==> r# = s#
   2.App(p1(A, B), r) = x
   3.App(p2(A, B), r) = y
   ------------------------------------------------------------------------
   (App(p1(A, B), r) = x & App(p2(A, B), r) = y) &
             !(r' : mem(A * B)).
               App(p1(A, B), r'#) = x & App(p2(A, B), r'#) = y ==> r'# = r
   : proofmanager.proof
```

from where we can apply the assumptions to finish the proof.

Bibliography

[1] Axiom schemes of separation. https://ncatlab.org/nlab/show/axiom+of+separation. Accessed: 2023-11-01.

[2] A discussion between Colin McLarty and Andrei Rodin about structuralism and categorical foundations of mathematics. http://philomatica.org/wp-content/uploads/2013/02/colin.pdf. Accessed: 2023-11-01.

[3] ETCS with elements. https://ncatlab.org/nlab/show/ETCS+with+elements. Accessed: 2023-11-01.

[4] Formalizing 100 theorems. https://www.cs.ru.nl/~freek/100/. Accessed: 2023-11-01.

[5] HOL theorem prover. https://hol-theorem-prover.org/. Accessed: 2023-11-10.

[6] The LCF approach to theorem proving. https://www.cl.cam.ac.uk/~jrh13/slides/manchester-12sep01/slides.pdf. Accessed: 2023-11-01.

[7] SEAR. https://ncatlab.org/nlab/show/SEAR. Accessed: 2023-11-01.

[8] Size issues. https://ncatlab.org/nlab/show/size+issues. Accessed: 2023-11-01.

[9] Structural ZFC. https://ncatlab.org/nlab/show/structural+ZFC. Accessed: 2023-11-01.

[10] Structurally presented set theory. https://ncatlab.org/nlab/show/structurally+presented+set+theory. Accessed: 2023-11-01.

[11] Theorem proving in lean. https://lean-lang.org/theorem_proving_in_lean/axioms_and_computation.html. Accessed: 2023-11-07.

[12] Oskar Abrahamsson, Magnus O. Myreen, Ramana Kumar, and Thomas Sewell. Candle: A verified implementation of HOL Light. In *Interactive Theorem Proving (ITP)*. LIPIcs, 2022.

[13] Peter Aczel. *Local Constructive Set Theory and Inductive Definitions*, pages 189–207. Springer Netherlands, Dordrecht, 2011.

[14] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Dedukti: a logical framework based on the $\lambda\pi$-calculus modulo theory, 2023.

[15] Jon Barwise. *Admissible Sets and Structures*. Perspectives in Logic. Cambridge University Press, 2017.

[16] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001.

[17] George Boolos. Iteration again. *Philosophical Topics*, 17(2):5–21, 1989.

[18] Anthony Bordg, Lawrence Paulson, and Wenda Li. Simple type theory is not too simple: Grothendieck's schemes without dependent types. *Experimental Mathematics*, 31(2):364–382, apr 2022.

[19] Kevin Buzzard, Johan Commelin, and Patrick Massot. Formalising perfectoid spaces. *CoRR*, abs/1910.12320, 2019.

[20] John Cartmell. Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243, 1986.

[21] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing.

[22] Emmanuel Gunther, Miguel Pagano, and Pedro Sánchez Terraf. Formalization of forcing in isabelle/zf. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 221–235, Cham, 2020. Springer International Publishing.

[23] Thomas Hales, Mark Adams, Gertrud Bauer, Dat Dang, John Harrison, Truong Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Nguyen, Truong Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, An Ta, Trân Trung, Diep Trieu, and Roland Zumkeller. A formal proof of the kepler conjecture. *Forum of Mathematics, Pi*, 5, 01 2015.

[24] John Harrison. Inductive definitions: automation and application. In Phillip J. Windley, Thomas Schubert, and Jim Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications: Proceedings of the 8th International Workshop*, volume 971 of *Lecture Notes in Computer Science*, pages 200–213, Aspen Grove, Utah, 1995. Springer-Verlag.

[25] Douglas R. Hofstadter. *Gödel, Escher, Bach: an Eternal Golden Braid*. Basic Books Inc., 1979.

[26] M. Randall Holmes. Alternative axiomatic set theories. In *Stanford Encyclopedia of Philosophy*. 2008.

[27] Peter V. Homeier. A design structure for higher order quotients. In Joe Hurd and Tom Melham, editors, *Theorem Proving in Higher Order Logics*, pages 130–146, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[28] Kenneth Kunen. *Set Theory: An Introduction to Independence Proofs*. Elsevier, 2006.

[29] F. William Lawvere. An elementary theory of the category of sets. *Proceedings of the National Academy of Sciences*, 52(6):1506–1511, 1964.

[30] F. William Lawvere. The category of categories as a foundation for mathematics. In S. Eilenberg, D. K. Harrison, S. MacLane, and H. Röhrl, editors, *Proceedings of the Conference on Categorical Algebra*, pages 1–20, Berlin, Heidelberg, 1966. Springer Berlin Heidelberg.

[31] Michael Makkai. First order logic with dependent sorts, with applications to category theory. Available from https://www.math.mcgill.ca/makkai/folds/foldsinpdf/FOLDS.pdf, 1995.

[32] Jean-Pierre Marquis. *Unfolding FOLDS*, pages 136–162. Oxford University Press, Oxford,OX2, 1989.

[33] Colin McLarty. Axiomatizing a category of categories. *The Journal of Symbolic Logic*, 56(4):1243–1260, 1991.

[34] Norman Megill and David A. Wheeler. *Metamath: A Computer Language for Mathematical Proofs*. 2019.

[35] Adam Naumowicz, Artur Kornilowicz, and Adam Grabowski. Mizar hands-on tutorial, 2023. http://mizar.org/cicm_tutorial/mizar.pdf.

[36] Steven Obua. Partizan games in Isabelle/HOLZF. In Kamel Barkaoui, Ana Cavalcanti, and Antonio Cerone, editors, *Theoretical Aspects of Computing - ICTAC 2006*, pages 272–286, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[37] Lawrence Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3(2):119–149, 1983.

[38] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer Berlin, Heidelberg, 1994.

[39] Lawrence C. Paulson. The relative consistency of the axiom of choice mechanized using isabelle/zf. *LMS Journal of Computation and Mathematics*, 6:198–248, 2003.

[40] Lawrence C. Paulson. Defining functions on equivalence classes. *ACM Trans. Comput. Logic*, 7(4):658–675, oct 2006.

[41] Frank Pfenning. *Logical Frameworks—A Brief Introduction*, pages 137–166. Springer Netherlands, Dordrecht, 2002.

[42] Michael D. Potter. *Set Theory and its Philosophy: A Critical Introduction*. Oxford University Press, Oxford, England, 2004.

[43] W. V. Quine. New foundations for mathematical logic. *The American Mathematical Monthly*, 44(2):70–80, 1937.

[44] Florian Rabe. First-order logic with dependent types. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning*, pages 377–391, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[45] Emily Riehl. An elementary theory of the category of sets. https://golem.ph.utexas.edu/category/2014/01/an_elementary_theory_of_the_ca.html. Accessed: 2023-11-01.

[46] Ieke Moerdijk Saunders Mac Lane. *Sheaves in Geometry and Logic*. Springer New York, NY, 1994.

[47] Michael Shulman. Comparing material and structural set theories. *Annals of Pure and Applied Logic*, 170(4):465–504, 2019.

[48] Andrzej Trybulec. Tarski-Grothendieck set theory, 1990.

[49] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[50] Petr Vopenka. In *Mathematics in the alternative set theory*. 1979.

[51] Yiming Xu and Michael Norrish. Mechanised modal model theory. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *International Joint Conference on Automated Reasoning (IJCAR), Paris, France*, volume 12166 of *Lecture Notes in Computer Science*, pages 518–533. Springer, 2020.