A Framework for Defining Declarative Languages

by

Feryal Fulya Horozal

A thesis submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy
in Computer Science

Approved, Thesis Committee

Prof. Dr. Michael Kohlhase, Jacobs University Bremen (Chair)
Prof. Dr. Dieter Hutter, Bremen University
Prof. Dr. Herbert Jäger, Jacobs University Bremen
Prof. Dr. Till Mossakowski, Otto-von-Guericke University of Magdeburg
Prof. Dr. Carsten Schürmann, IT University of Copenhagen

Date of Defense: November 14, 2014

Engineering and Science

Statutory Declaration

I, Feryal Fulya Horozal, hereby declare that I have written this PhD thesis independently, unless where clearly stated otherwise. I have used only the sources, the data and the support that I have clearly mentioned. This PhD thesis has not been submitted for conferral of degree elsewhere.

The main research problems addressed by this work stem from close collaboration with the members of the LATIN project [KMR09, CHK+11], in particular, with Florian Rabe, Michael Kohlhase, Till Mossakowski and Mihai Codescu.

Parts of this thesis are based on or closely related to previously published material or material that is prepared for publication at the time of this writing. These are [HR11], [HR12], [HKR12] and [HRK14].

Bremen, 31.08.2014

Abstract

Declarative languages are an important family of formalisms used for formal knowledge representation in computer science. Examples include logics, type theories, set theories, specification languages and ontology languages. Despite the vast variety of declarative languages, they share a common structure: A possibly infinite collection of theories, which are sets of declarations, and a set of well-formed expressions over each theory. In fact, we can use this common structure as the defining characteristics of declarative languages.

Logical frameworks are abstract formalisms that have been introduced for representing declarative languages and for studying their properties. In particular, they can be used to relate and combine different declarative languages. A key challenge in designing logical frameworks is that they should be capable of generalizing concepts, algorithms and theorems about declarative languages while using as simple and yet expressive primitives as possible.

Abstract logical frameworks, such as institutions, work with abstract categories. This has the advantage that the framework is independent of the structure of the particular declarative language; thus theory-related concepts such as model classes and satisfaction can be formulated generically.

Declarative logical frameworks, such as LF, on the other hand, represent each individual theory of a declarative language as a theory of the framework. This makes it easy to provide generic tool support for concrete theories. However, they do not have an abstract concept to represent the infinite collection of theories, and therefore, they cannot reason about or operate on arbitrary theories.

This PhD thesis contributes to the design of declarative logical frameworks that balances the trade-off between abstract and concrete representations. In particular, we identify a new feature that can be added to a declarative framework as a language primitive: *declaration patterns*. The main idea behind this feature is to capture the common structure of the theories of declarative languages.

More precisely, we exploit the observation that theories consist of a list of declarations each of which must conform to one out of a few patterns fixed by the declarative language. For instance, first-order logic has three such patterns: one each for the declaration of *(i)* function symbols, *(ii)* predicate symbols and *(iii)* axioms. We reify declaration patterns as a language primitive for declarative logical frameworks. While frameworks like LF focus on defining the logical symbols and the expressions, declaration patterns characterize the legal declarations of non-logical symbols by specifying their syntactic shape. Declaration patterns also prove crucial for language translations. We introduce a notion of *pattern-based translation*, which maps legal theories of one declarative language to legal theories of the other language.

We introduce these novel notions in a calculus-based meta-framework for defining declarative languages and translations between them. Our meta-framework is independent of the underlying declarative logical framework and can be instantiated with specific logical frameworks like LF. In particular we develop a new declarative logical framework LFS (LF with sequences) that supports sequences and dependent function spaces that take sequence arguments. LFS proves very useful for defining mathematical operators of flexible arity and ellipses, which we need for writing down the declaration patterns of most declarative languages. We apply and evaluate our pattern-based approach in an atlas of declarative languages, which includes various logics ranging from traditional first-order logic to more complex languages such as polymorphic higher order logic.

Acknowledgements

Hereby I would like to thank my supervisor, Michael Kohlhase, for providing me with a challenging and inspiring research topic and for making it possible for me to pursue it in a very friendly atmosphere in his research group. I am most grateful to him for giving me the freedom to explore different ideas and for his continued encouragement, advice, patience and support in every stage of my thesis.

I am profoundly indebted to Florian Rabe, whose expert insights and guidance have been crucial in shaping my path to achieve this work. I am very grateful to him for the time he spent working with me and sharing with me his knowledge and ideas.

I am particularly thankful to Till Mossakowski and Carsten Schürmann for the very productive discussions with them and for their valuable feedback, as well as to all my colleagues and the students in the KWARC group at Jacobs University Bremen and the LATIN project.

I would like to give my special heartfelt thanks to Andrea Kohlhase, not only for being a very kind and supportive colleague, but also for her sincere friendship and for being there for me at difficult times, supporting me very lovingly, and for motivating me.

Contents

Chapter 1

Introduction

Formal methods is the field of formal specification and verification of software and hardware systems. Based on various mathematical and logic-based techniques, formal methods have been successfully applied in several real-world verification tasks ranging from the verification of processors and traffic systems to security protocols. Such verification tasks are often highly complex and typically require more than one specific formal method to capture the different aspects of the task. For that purpose, a wide rage of highly specialized computer systems have been developed including (semi-)automated theorem provers, model-checkers, constraint-solvers, computer algebra systems and concept classifiers, each of which is based on a specific underlying formalism for formal knowledge representation.

In this chapter, we give an introduction to **declarative languages** — a group of formal knowledge representation languages, which are used pervasively in formal methods and also increasingly in mathematics.

1.1 Declarative Languages

Typically, a declarative language is defined in two steps: *i)* The **theories** of the language are defined, *ii)* the **expressions** of the language over a given theory are defined.

In the first step, we observe that theories almost always have a common structure: They are a list of declarations that introduce new **symbols**. Moreover, each symbol declaration in a theory conforms to a certain pattern that is determined by the declarative language. Function symbols, predicate symbols, type operators, judgments, axioms, theorems, inference rules are some examples of such patterns. To capture this, we introduce the concept of **declaration patterns** in this thesis. Then, theories arise by combining different instances of declaration patterns. In fact, we can use this property as the defining characteristics of declarative languages.

In the second step, the expressions are usually defined via a formal grammar. This generates all possible expressions over a theory including some which might not make sense semantically. An inference system is often used to sort the **well-formed** expressions out from all expressions generated by the grammar.

It is the structure of declaration patterns and their instances that we are interested in this thesis. To fortify our intuition, we will look at different types of declarative languages, as illustrated in figure 1.1, and introduce their declaration patterns. In the following sections, we will elaborate on two different kinds of declarative languages: *logics* and *type theories*.

	Theory	Expressions	Patterns
Logics	Theory	Terms Sentences Proofs	Functions Predicates Axioms
Type Theories	Signature	Terms Types Kinds	Types Constants

Figure 1.1: Declarative Languages, their Theories and Patterns

1.1.1 Logics

Logics are declarative languages that are used to model domains that involve truth and consequence. The vocabulary of a logic consists of a fixed set of logical symbols, an infinite set of variables, and an extensible set of non-logical symbols that are introduced in the theories of the logic. Here, we regard the axioms in a logic as special symbols introduced in the theories of that logic.

We now fortify our intuition by giving individual logics as examples.

First-Order Logic First-order logic (FOL) is a declarative language whose logical symbols are $\top, \bot, \neg, \wedge, \vee, \Rightarrow, \forall, \exists$, and non-logical symbols are function symbols $f_1, f_2, \ldots$ of arity n and predicate symbols $p_1, p_2, \ldots$ of arity n for $n = 0, 1, \ldots$.

Definition 1.1 (First-Order Theories)**.** A first-order theory Σ is a set of declarations that introduce

- n-ary function symbols f
- n-ary predicate symbols p
- axioms F.

Definition 1.1 defines the theories of FOL by precisely describing the form of the symbol declarations allowed in them. The three bullets above are the declaration patterns of FOL.

Definition 1.2 (First-Order Expressions)**.** The grammar of FOL over a FOL-theory Σ is as follows:

Formulas	F	$::=$	$\top$	truth	
			$\bot$	falsity	
			$\neg F$	negation	
			$F \wedge F$	conjunction	
			$F \vee F$	disjunction	
			$F \Rightarrow F$	implication	
			$\forall x.\, F$	universal quantification	
			$\exists x.\, F$	existential quantification	
			$p(t_1, \ldots, t_n)$	atomic formulas	$p \in \Sigma$
			$t_1 = t_2$	equality	
Terms	t	$::=$	x	variables	
			$f(t_1, \ldots, t_n)$	function application	$f \in \Sigma$

where the arity of f is n and the arity of p is n for any natural number n. Expressions produced from the non-terminal F are called *formulas* and those produced from t are called *terms*. Closed formulas are called *sentences*.

Note that the first-order theories are the extensible part of the vocabulary. They extend the grammar by introducing new non-logical symbols.

An example of a FOL-theory is the theory of monoids:

Example 1.3 (FOL-Theory of Monoids). The theory of monoids in FOL declares the following symbols

- a binary function symbol $\circ$, usually written as $f \circ g$ instead of $\circ(f, g)$,
- a nullary function symbol e,
- the axioms

$$\forall x.\forall y.\forall z.\, (x \circ y) \circ z = x \circ (y \circ z)$$
$$\forall x.\, e \circ x = x$$
$$\forall x.\, x \circ e = x$$

  for the associativity of $\circ$ and the neutrality of e.

Sorted First-Order Logic The vocabulary of sorted first-order logic (SFOL) has the same logical connectives as in FOL, but the universal and the existential quantifiers take sorted arguments.

Definition 1.4 (SFOL Theories)**.** A SFOL-theory is a list of declarations that introduce

- sorts s,
- n-ary function symbols f over sorts $s_1, \ldots, s_n$ with return sort s,
- n-ary predicate symbols p over sorts $s_1, \ldots, s_n$.
- axioms F.

Definition 1.5 (SFOL Expressions)**.** The expressions of SFOL are generated by the following grammar:

Formulas	F	$::=$	$\top$	truth	
			$\bot$	falsity	
			$\neg F$	negation	
			$F \wedge F$	conjunction	
			$F \vee F$	disjunction	
			$F \Rightarrow F$	implication	
			$\forall x : s.\, F$	universal quantification	
			$\exists x : s.\, F$	existential quantification	
			$p(t_1, \ldots, t_n)$	atomic formulas	$p \in \Sigma$
			$t_1 = t_2$	equality	
Terms	t	$::=$	x	variables	
			$f(t_1, \ldots, t_n)$	function application	$f \in \Sigma$

Example 1.6 (SFOL-Theory of Vector Spaces). Consider the theory of vector spaces as an SFOL-theory with the following list of symbols is an SFOL-theory:

- a sort *vec* for the set of vector spaces,
- a sort *sca* for the field of scalars,
- a binary function symbol + over sorts *vec*, *vec* with return sort *vec*,
- a nullary function symbol 0 over *sca*,

and the remaining function symbols and the axioms that add *i)* a commutative group of vectors, and *ii)* a field of scalar, which we omit here for the sake of simplicity as we focus on the structure of the declared symbols.

1.1.2 Type Theories

Type theories typically employ a hierarchical classification of their expressions: In two-leveled type theories, such as simple type theory, expressions are classified into **terms** and **types**, where the relation between a term and its type is given by a typing judgment, usually written as $t : A$ for a term t of type A.

In three-leveled type theories, such as polymorphic type theory, expressions are classified into terms, types and **kinds**, where kinds are used for grouping families of types.

Simple Type Theory Simple type theory [Chu40] is a declarative language whose set of primitive symbols consists of the symbols $\to$, λ and @, where $\to$ is a type constructor, λ is a binder for function abstractions and @ is an operator for the application of functions to their arguments.

Definition 1.7 (STT-Theories)**.** A theory of STT is a list of declarations that introduce

- base types a,
- typed constants c of type A, where A is formed by the grammar in definition 1.8.

Definition 1.8 (STT Expressions)**.** The expressions over a STT theory Σ are formed by the following grammar:

Types	A	$::=$	a	base types
			$A \to A$	function types
Terms	t	$::=$	x	variables
			c	constants
			$\lambda x : A.\, t$	function abstractions
			$t\, t$	function applications

where we simply write $s\, t$ for the application $@\, s\, t$.

The **well-formed** expressions over Σ are a subset of the expressions over Σ which is defined by an inference system. Note that the type constructor $\to$ is right-associative, i.e., $A_1 \to A_2 \to A_3$ denotes $A_1 \to (A_2 \to A_3)$, and term application is left-associative, i.e., $t_1\, t_2\, t_3$ denotes $(t_1\, t_2)\, t_3$.

Dependent Type Theory There are various versions of dependent type theory (Martin-Löf type theory [ML74], LF [HHP93], calculus of constructions [CH88]). Here we use LF [HHP93].

DTT extends STT with dependent types, allowing types and kinds to depend on terms. The set of primitive symbols of DTT consists of, in addition to λ and @ for term abstraction and term application, the symbol Π for dependent type construction and similarly one symbol for type abstraction, type application and dependent kind construction. We will use the same symbols for the type level as their analogs for the terms level.

The definition of DTT-theories and DTT-expressions have significant mutual recursion.

Definition 1.9 (DTT-Theories)**.** A theory of DTT is a list of declarations that introduce

- type constants a of kind K, and
- term constants c of type A,

where K and A are formed by the grammar in definition 1.10.

Definition 1.10 (DTT Expressions)**.** The expressions over a DTT theory Σ is formed by the following grammar:

Kinds	K, L	$::=$	$\texttt{type}$	kind of types
			$\Pi x : A.\, K$	kind of type families
Type Families	A, B	$::=$	a	base types
			$\Pi x : A.\, B$	dependent function types
			$\lambda x : A.\, B$	type abstractions
			$A\, M$	type applications
Terms	M, N	$::=$	x	variables
			c	constants
			$\lambda x : A.\, M$	term abstractions
			$M\, N$	term applications

We will visit the inference system for well-formed DTT expressions in section 4.2.

1.2 Language Translations

Analogous to the definition of declarative languages, a language translation is typically specified in two steps: *i)* The translation of the theories of the language is defined, *ii)* the translation of the expressions over a given theory is defined.

It is not surprising that both declarative languages and language translations have a structural parallelism. We illustrate this in figure 1.2.

More specifically, in a translation $\mathcal{T} = (\Theta, \alpha)$ from language $\mathcal{L}_1$ to $\mathcal{L}_2$,

- a function Θ that translates each theory Σ of $\mathcal{L}_1$ to a theory $\Theta(\Sigma)$ of $\mathcal{L}_2$, and
- a family α of functions α_Σ indexed by theories Σ, which translate each expression E over a theory Σ of $\mathcal{L}_1$ to an expression $\alpha_\Sigma(E)$ over the theory $\Theta(\Sigma)$ of $\mathcal{L}_2$.

In particular, each production in the grammar for the formation of expressions corresponds to one case in the expression translation function α_Σ and to one case in the inductive proof of judgment preservation.

Recall the definitions of FOL and SFOL from section 1.1.1. A translation from SFOL to FOL is defined as follows:

Declarative Language	Language Translation
Theories	Theory translation function
Declaration patterns	Declaration pattern translations
Expressions over each theory	Expression translation function for each theory
Grammar	Inductive functions over the grammar
Non-terminals	One translation function for each non-terminal
Inference system	Preservation of judgments

Figure 1.2: Definitional Structure of Declarative Languages and their Translations

Definition 1.11 (SFOL to FOL Theory Translation)**.** Let Σ be an SFOL-theory as in definition 1.4. We give a translation $\mathcal{S}2\mathcal{F} = (\Theta, \alpha)$ from SFOL to FOL:

The translation of Σ into FOL is the FOL-theory $\Theta(\Sigma)$ that consists of

- a unary predicate symbol s for every sort symbol s in Σ,
- an n-ary first-order function symbol f and the axiom

  $$\forall x_1, \ldots x_n.\, s_1(x_1) \wedge \ldots \wedge s_n(x_n) \Rightarrow s(f(x_1, \ldots, x_n))$$

  for every function symbol f in Σ with input arguments of sorts s_1,..., s_n and output sort s,
- an n-ary predicate symbol p[1] for every predicate symbol p in Σ with input arguments of sorts s_1,..., s_n, and

The translation of sorted first-order expressions E is the first-order expression $\alpha_\Sigma(E)$, where α_Σ is defined in definition 1.12.

Note that the theory translation $\Theta(\Sigma)$ is defined by induction of Σ with one case for each declaration pattern.

Definition 1.12 (SFOL to FOL Expression Translations)**.** Let Σ be an SFOL theory. The translation $\alpha_\Sigma(F)$ of Σ-expressions F is defined inductively as follows.

$$\begin{array}{lcl}
\alpha_\Sigma(\top) & = & \top \\
\alpha_\Sigma(\bot) & = & \bot \\
\alpha_\Sigma(\neg F) & = & \neg\, \alpha_\Sigma(F) \\
\alpha_\Sigma(F_1 \wedge F_2) & = & \alpha_\Sigma(F_1) \wedge \alpha_\Sigma(F_2) \\
\alpha_\Sigma(F_1 \vee F_2) & = & \alpha_\Sigma(F_1) \vee \alpha_\Sigma(F_2) \\
\alpha_\Sigma(F_1 \Rightarrow F_2) & = & \alpha_\Sigma(F_1) \Rightarrow \alpha_\Sigma(F_2) \\
\alpha_\Sigma(\forall x : s.\, F) & = & \forall x.\, s(x) \Rightarrow \alpha_\Sigma(F) \\
\alpha_\Sigma(\exists x : s.\, F) & = & \exists x.\, s(x) \wedge \alpha_\Sigma(F) \\
\alpha_\Sigma(p(t_1, \ldots, t_n)) & = & p(\alpha_\Sigma(t_1), \ldots, \alpha_\Sigma(t_n)) \\
\alpha_\Sigma(t_1 = t_2) & = & \alpha_\Sigma(t_1) = \alpha_\Sigma(t_2) \\
\alpha_\Sigma(x) & = & x \\
\alpha_\Sigma(f(t_1, \ldots, t_n)) & = & f(\alpha_\Sigma(t_1), \ldots, \alpha_\Sigma(t_n))
\end{array}$$

Example 1.13 (Theory Translation). The translation of the SFOL-theory of vector spaces from example 1.6 to FOL consists of

[1] A model-theory based translation of SFOL to FOL introduces the axiom $\forall x_1, \ldots x_n.\, \neg s_1(x_1) \vee \ldots \vee \neg s_n(x_n) \Rightarrow \neg p(x_1, \ldots, x_n)$ for p, which we do not require here.

- a unary first-order predicate vec,
- a unary first-order predicate sca,
- a binary first-order function symbol $+$ with the axiom

$$\forall x.\, \forall y.\, vec(x) \wedge vec(y) \Rightarrow vec(+(x, y))$$

- a nullary first-order function symbol 0 with the axiom

$$sca(0)$$

1.3 Extension Principles

Declarative languages are usually coupled with a set of **extension principles**. Here, the word **extension** refers to the act of forming a new theory by adding declarations to an existing theory. For example, the theory of groups extends the theory of monoids by adding a unary function symbol for the inverse operation and the usual axiom about it.

Thus, an **extension principle** is a theory forming operation that takes a theory Σ (and possibly parameters that are typically expressions over Σ) and returns an extension of Σ. For example, explicit definitions are the simplest and most common extension principle: An explicit definition takes a theory Σ and an expression E over Σ, and returns the theory Σ extended with the declaration $c = E$, where c is a fresh name chosen by the user of this extension principle[2].

Extension principles are very common in mathematical practice. We will briefly review some important ones below.

Explicit Definitions Explicit definitions are a standard way of defining functions in first-order logic. An n-ary first-order function symbol can be defined by $f(x_1, \ldots, x_n) = t$, by formulating the value of f applied to its arguments $x_1, \ldots, x_n$.

Implicit Definitions An implicit definition defines a mathematical object by axiomatizing the properties the object has. This is in contrast to explicit definition, which states the definiens directly. For instance, the neutral element e in monoid can be defined implicitly as the unique object that satisfies the neutrality axioms. An implicit definition requires a proof that there is exactly one object that fits this description.

More precisely, an implicit function definition takes

- a theory Σ,
- the name f of the function being defined,
- the inpurt sorts D_1,...,D_n of the function f,
- the output sort C of the function f,
- a predicate P over D_1,..., D_n, C, and
- the proof of totality (existence) and functionality (uniqueness) of f as parameters,

[2] Alternatively, the name c can be given as a parameter to the extension principle.

and returns the theory obtained by extending Σ with the declaration

$$f : D_1 \to \ldots \to D_n \to C$$

and the axiom

$$\forall x_1 : D_1 \ldots \forall x_n : D_n.\, P(x_1, \ldots, x_n, f(x_1, \ldots, x_n)).$$

Example 1.14 (The Neutral Element). Let Σ be the theory of monoids in SFOL, and consist of, in particular, the declarations M and $\circ : M \to M \to M$ for sort M. The neutral element, which is well-known to exist uniquely, can be added by an implicit definition, where M is the sort argument and $P(x)$ is $\forall y : M.\, x \circ y = x \wedge y \circ x = y$.

Example 1.15 (Inverse). Consider the theory of groups obtained from the theory of monoids in example 1.14 by adding the axiom $\forall x : M. \exists y : M. x \circ y = e$. The definition of the inverse function *inv* over the sort M is given by proving the following arguments: The function symbol *inv*, the input sort M, the output sort M, the predicate $P(x) = \exists y : M. x \circ y = e$, the two straightforward proofs for totality and functionality of *inv*, and introduces the declarations $inv : M \to M$ and the axiom $\forall x : M.\, x \circ f(x) = e \wedge f(x) \circ x = e$.

Case-Based Function Definitions Another common extension principle used in mathematics is the case-based function definition. In fact, case-based function definitions are a special case of FOL-style function definitions.

A case-based definition of a unary function f from A to B uses n different cases where each case is guarded by the predicate c_i together with the respective definiens d_i.

$$f(x) = \begin{cases} d_1(x) & \text{if} \quad c_1(x) \\ \vdots & \quad\;\; \vdots \\ d_n(x) & \text{if} \quad c_n(x) \end{cases}$$

The extension principle for the case-based unary function definition takes

- a theory Σ,
- the name f of the function,
- the domain A and the codomain B of f,
- n predicates c_i over A as the cases,
- n unary functions $d_i : A \to B$ as the respective definiens for each case c_i and
- a proof that exactly one of the propositions $c_i(x)$ holds for any element x in the domain A,

and returns the declaration $f : A \to B$ and the axiom

$$\forall x : A.\, (c_i(x) \Rightarrow f(x) = d_i(x)) \wedge \ldots \wedge (c_n(x) \Rightarrow f(x) = d_n(x)).$$

HOL-Style Type Definitions Higher-order logic (HOL) [Chu40] admits λ-abstraction and a description operator, which permits deriving many common extension principles, in particular, implicit definitions.

But there is one primitive extension principle that is commonly accepted in HOL-based formalizations: Gordon/HOL type definition [Gor88] introduces a new type that is axiomatized to be isomorphic to a subtype of an existing type.

This primitive extension principle is used to derive extension principles, e.g., for inductive types, record types and quotient types. HOL-based proof assistants implement the type definition principle as a built-in statement. They also often provide further built-in statements for other definition principles that become derivable in the presence of type definitions, e.g., a definition principle for record types.

For example, in Isabelle/HOL [NPW02], HOL is formalized in the Pure logic underlying the logical framework Isabelle [Pau94]. But because the type definition principle is not expressible in Pure, it is implemented as a primitive Isabelle feature that is only active in Isabelle/HOL.

Uses of Extension Principles

We observe that extension principles are used, in particular, for the following purposes:

- Pragmatic features
- Conservativity
- Macro functionality

All of these three aspects pervade mathematical practice, and therefore should be present and indeed well-supported in computer mathematics. We discuss these three interrelated aspects in more detail below.

Pragmatic Features We introduce the phrase **pragmatic feature** for high-level language features. Typically, a pragmatic feature in a language introduces new syntax that captures the high level meaning of the concept it represents. This permits intuitive formal representations that use a more verbose and a larger vocabulary.

More specifically, a pragmatic feature typically provides object-level syntax, which we will refer to as **pragmatic syntax**, for a meta-level operation. For instance, the extension principles we have reviewed in the previous sections are theory forming operations at the meta-level and they can be added to the syntax of a declarative language as new features.

Pragmatic syntax is given meaning by elaborating it into default syntax of a declarative language. This permits implementing semantic services like validation only for the core language and extending them to the pragmatic features by elaboration. Ultimately, a language might even have multiple pragmatic front-ends geared towards different audiences.

In fact, in languages for formalized mathematics, it is standard practice to define a minimal core language that is extended by macros, functions, or notations. For example, Isabelle [Pau94] provides a rich language of notations, abbreviations, syntax and printing translations, and a number of definitional forms.

This language design has the advantage that only a small, regular sublanguage has to be given a mathematical meaning, but a larger vocabulary that is more intuitive to practitioners of the field can be used for actual representations.

Conservativity Conservativity is the property of an extension that permits a larger syntax while maintaining the same semantic properties.

More concretely, individual declarative languages typically admit a definition of **conservative extension** in the presence of a notion of theorem:

Definition 1.16 (Conservative Extensions)**.** An extension Σ' of Σ is conservative if for every Σ-expression T if T is a theorem over Σ' then it is a theorem over Σ as well.

The definition of the notion of theorem and therefore reasoning for theoremhood differ from language to language. Note that a conservative extension may add theorems about the new symbols it introduces, however the theoremhood over the symbols from Σ is restricted to those theorems over Σ.

Conservative extensions assure that new theories can be built by safe increments: Given a consistent theory Σ, any conservative extension Σ' remains consistent. However, reasoning about conservative extensions is in general undecidable. Therefore, it is common to choose a few extension principles that are known to be conservative and use them to build big theories safely.

The extension principles we have discussed in this section are known to be conservative.

Macro Functionality Extension principles can be used as macros: They introduce abbreviations for frequently occurring (groups of) declarations to increase the efficiency of theory construction. For instance, in narrative formats for mathematics, e.g., the TEX/LATEX format, the TEX layout primitives constitute the core syntax of TEX and macro definitions allow the user to add his own extensions to the language. This extensibility led to the profusion of user-defined LATEX document classes and packages that has made TEX/LATEX so successful.

Chapter 2

State of the Art

In this chapter we give the state of the art of existing abstract formalisms that are developed and used for the purpose of representing declarative languages and studying their properties. Furthermore, we mention notable libraries of language representations given in such formalisms.

2.1 Logical Frameworks

Logical frameworks are abstract formalisms that provide a specific mathematical infrastructure to define declarative languages. In particular, they can be used to relate and combine different declarative languages.

Based on their underlying mathematical infrastructure, logical frameworks can be split into the following two main groups: **Abstract logical frameworks**, which are typically built on category theory, and **declarative logical frameworks**, which are built on type theory. We will briefly introduce these two groups below.

2.1.1 Abstract Logical Frameworks

Abstract logical frameworks typically work with abstract **categories**. Also called **general logics**, these frameworks define what it means to be a logical system. The most important frameworks in this group are the framework of **institutions** [GB92] and general logics by Messeguer [Mes89].

In the following, we will explore abstract logical frameworks in terms of representing *i)* declarative languages, *ii)* language translations, and *iii)* extension principles, the three aspects highlighted in the previous chapter.

Representing Declarative Languages There is no standard definition of a declarative language in abstract logical frameworks. Institutions define logical systems, which are centered around the notion of truth and consequence, and are more specific than declarative languages.

By weakening the definition of a logical system in institutions, a declarative language can typically be defined using a class of theories and a function that assigns for every theory in this class, the set of expressions over this theory. What makes this definition abstract is that it does not assume any concrete structure of the individual theories or the expressions over a theory.

More concretely, we define declarative languages as follows:

Definition 2.1 (Declarative Language)**.** A declarative language $\mathcal{L}$ is a pair $(\mathbf{Th}^{\mathcal{L}}, \mathbf{Exp}^{\mathcal{L}})$, where

- $\mathbf{Th}^{\mathcal{L}}$ is the class of **theories** of $\mathcal{L}$ and
- $\mathbf{Exp}^{\mathcal{L}} : \mathbf{Th}^{\mathcal{L}} \to \mathcal{SET}$ is a function that assigns to each theory Σ in $\mathbf{Th}^{\mathcal{L}}$ the set $\mathbf{Exp}^{\mathcal{L}}(\Sigma)$ of **expressions** over Σ.

Here $\mathcal{SET}$ denotes the class of all sets.

Example 2.2 (First-Order Logic). $\mathbf{Th}^{FOL}$ is the class of first-order theories from definition 1.1. $\mathbf{Exp}^{FOL}(\Sigma)$ consists of the expressions that have the form given by the grammar in definition 1.2 for each theory $\Sigma \in \mathbf{Th}^{FOL}$.

Example 2.3 (Sorted First-Order Logic). $\mathbf{Th}^{SFOL}$ is the class of first-order theories from definition 1.4. $\mathbf{Exp}^{SFOL}(\Sigma)$ consists of the expressions that have the form given by the grammar in definition 1.5 for each theory $\Sigma \in \mathbf{Th}^{SFOL}$.

The above definition of declarative languages can be considered as a very basic form of abstract frameworks. In fact, most abstract frameworks refine this definition by adding more structure. For instance, entailment systems [FS87] add entailment relation, institutions add model theory, institutions with contexts [Paw95] add contexts and substitution, parchments [GB86] add expressions generated by a signature in a meta-language, which is the closest to declarative frameworks (we will discuss this in section 2.1.2).

Abstract logical frameworks have the advantage that they are independent of the structure of the particular declarative language; thus theory-related concepts can be formulated generically.

Representing Language Translations In the framework of institutions, **institution comorphisms** [GR02] are the standard tool to give translations between two logical systems. By weakening the definition of an institution comorphism, language translations can be typically defined as follows:

Definition 2.4 (Language Translation)**.** Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two declarative languages. A **translation** $\mathcal{T}$ from $\mathcal{L}_1$ to $\mathcal{L}_2$ is a pair $(\mathbf{th}^{\mathcal{T}}, \mathbf{exp}^{\mathcal{T}})$, where

- $\mathbf{th}^{\mathcal{T}} : \mathbf{Th}^{\mathcal{L}_1} \to \mathbf{Th}^{\mathcal{L}_2}$ is a function that maps each theory $\Sigma \in \mathbf{Th}^{\mathcal{L}_1}$ to a theory of $\mathbf{th}^{\mathcal{T}}(\Sigma) \in \mathbf{Th}^{\mathcal{L}_2}$,
- $\mathbf{exp}^{\mathcal{T}}$ is a family of mappings $\mathbf{exp}^{\mathcal{T}}_{\Sigma} : \mathbf{Exp}^{\mathcal{L}_1}(\Sigma) \to \mathbf{Exp}^{\mathcal{L}_2}(\mathbf{th}^{\mathcal{T}}(\Sigma))$ that assigns to each expression $F \in \mathbf{Exp}^{\mathcal{L}_1}(\Sigma)$ an expression $\mathbf{exp}^{\mathcal{T}}_{\Sigma}(F) \in \mathbf{Exp}^{\mathcal{L}_2}(\mathbf{th}^{\mathcal{T}}(\Sigma))$.

We will omit the superscript $\mathcal{T}$ whenever the translation is clear from the context.

Example 2.5 (Translation from SFOL to FOL). $\mathbf{th}(\Sigma)$ consists of the declarations given in definition 1.11 for each $\Sigma \in \mathbf{Th}^{FOL}$. And $\mathbf{exp}_{\Sigma}(F)$ is defined inductively on the syntactic form of F as in definition 1.12.

Representing Extension Principles In abstract logical frameworks, individual extension principles are not the main center of attention. But the property of being a conservative extension is often studied. In particular, abstract frameworks use conservative extensions mainly as a tool for reasoning about theory extensions rather than as a tool for theory construction.

Reasoning for conservativity is in general difficult. For that reason, systems like Hets [MML07] that are based on insitutions often work with a set of selected sufficient criteria for individual conservative extensions, and use conservativity reasoners to discharge proof obligations [CMM13].

2.1.2 Declarative Logical Frameworks

Declarative logical frameworks are special declarative languages in which other languages are represented. We often refer to such languages as **meta-languages**, and to the languages that are represented in them as **object-languages**.

Automath [dB70] was the first system that implemented the ideas of a declarative logical framework. The most significant ones today include the Edinburgh logical framework LF [HHP93], Isabelle [Pau94], Coq [Coq14], Maude [CELM96] and Agda[1] [Nor05].

Formally, a declarative logical framework comprises theories and expressions over its theories:

Example 2.6 (LF). The Edinburgh logical framework LF is based on dependent type theory. **LF-theories** are captured by the definition of DTT-theories in definition 1.9. Similarly, the expressions over an LF-theory, i.e., **LF-expressions**, are captured by definition 1.10 of DTT-expressions.

More specifically, LF-expressions are classified as **LF-kinds** K, kinded **LF-type families** $A : K$ and typed **LF-terms** $M : A$. We refer to LF-type families of the form $A : \texttt{type}$ as **LF-types**. Then an LF-theory consists of symbol declarations for type families $a : K$ and terms $c : A$, where both a and c may have definiens, e.g., the declarations $a : K = L$ and $c : A = M$ are allowed.

LF is implemented in the Twelf system [PS99]. In 2009, a module system for LF was developed and implemented for Twelf: The LF module system [RS09] implements, in particular, **LF-theory morphisms** that allow **theory inclusion** and **theory interpretation**. The concept of theory morphisms for LF was first introduced in [HST94].

In the following we will elaborate on the state of the art of representing declarative languages, their translations and extension principles with respect to LF and its module system as a representative example.

Representing Declarative Languages In declarative frameworks, declarative languages are represented as theories of the framework. In particular, *i)* the theories of the language, as well, are represented as theories of the framework, *ii)* the expressions of the language are represented as expressions of the framework.

For instance, in LF, every declarative language is represented as an LF-theory. In particular, LF-type declarations $a : \texttt{type}$ are used for syntactic classes, e.g., sorts, terms, formulas and judgments, and LF-term declarations $c : A$ are used for individual connectives, quantifiers, sorts, functions, predicates, axioms, etc.

Definition 2.7 (FOL Syntax in LF)**.** The following LF-theory *FOL* is a representation of the FOL syntax:

[1] Agda is an implementation of Martin-Löf dependent type theory that combines features of theorem proving and programming. It was not designed as a logical framework, but can be used as one.

$$
\begin{array}{lcl}
\mathit{term} & : & \texttt{type} \\
\mathit{form} & : & \texttt{type} \\
\mathit{ded} & : & \mathit{form} \to \texttt{type} \\
\mathit{true} & : & \mathit{form} \\
\mathit{false} & : & \mathit{form} \\
\neg & : & \mathit{form} \to \mathit{form} \\
\wedge & : & \mathit{form} \to \mathit{form} \to \mathit{form} \\
\vee & : & \mathit{form} \to \mathit{form} \to \mathit{form} \\
\Rightarrow & : & \mathit{form} \to \mathit{form} \to \mathit{form} \\
\forall & : & (\mathit{term} \to \mathit{form}) \to \mathit{form} \\
\exists & : & (\mathit{term} \to \mathit{form}) \to \mathit{form} \\
\doteq & : & \mathit{term} \to \mathit{term} \to \mathit{form}
\end{array}
$$

More specifically, we introduce two LF-types, *term* and *form*, for FOL terms and formulas, respectively. Moreover, *ded* assigns to each LF-term $F : \mathit{form}$ the LF-type $\mathit{ded}\, F$ of the proofs of F. We use the notation $A \to B$ for $\Pi x : A.B$ when x does not occur in B.

For each logical primitive of FOL, there is an LF-term declared: *true* and *false* correspond to the logical connectives $\top$ for truth and $\bot$ for falsehood, respectively. The unary LF-term $\neg : \mathit{form} \to \mathit{form}$ corresponds to negation taking one argument, say F, of LF-type *form* and returning $\neg F$. The other FOL connectives are represented similarly. The representation of the FOL quantifiers uses higher-order abstract syntax, which we discuss later in this chapter in section 2.1.4: $\forall : (\mathit{term} \to \mathit{form}) \to \mathit{form}$ takes as argument a logical formula with a free variable, and returns a logical formula. $\exists$ is analogous. $\doteq$ corresponds to equality $=$ for FOL-terms. A full representation of first-order logic in LF including its proof theory and model theory is given in [HR11].

Definition 2.8 (SFOL Syntax in LF)**.** The following LF-theory *SFOL* gives a representation of the SFOL syntax:

$$
\begin{array}{lcl}
\mathit{sort} & : & \texttt{type} \\
\mathit{tm} & : & \mathit{sort} \to \texttt{type} \\
\mathit{form} & : & \texttt{type} \\
\mathit{ded} & : & \mathit{form} \to \texttt{type} \\
\mathit{true} & : & \mathit{form} \\
\mathit{false} & : & \mathit{form} \\
\neg & : & \mathit{form} \to \mathit{form} \\
\wedge & : & \mathit{form} \to \mathit{form} \to \mathit{form} \\
\vee & : & \mathit{form} \to \mathit{form} \to \mathit{form} \\
\Rightarrow & : & \mathit{form} \to \mathit{form} \to \mathit{form} \\
\forall & : & \Pi S : \mathit{sort}.\, (\mathit{tm}\, S \to \mathit{form}) \to \mathit{form} \\
\exists & : & \Pi S : \mathit{sort}.\, (\mathit{tm}\, S \to \mathit{form}) \to \mathit{form} \\
\doteq & : & \Pi S : \mathit{sort}.\, \mathit{tm}\, S \to \mathit{tm}\, S \to \mathit{form}
\end{array}
$$

sort is an LF-type representing the universe of all sorts. *tm* assigns to every LF-term $S : \mathit{sort}$ the LF-type $\mathit{tm}\, S$ of the SFOL terms of sort S.

Notation 2.9. In addition to using the calligraphic font, e.g., $\mathcal{L}$, for introducing meta-variables for declarative languages, we will use L to denote the respective object-language that represents $\mathcal{L}$ in LF. Moreover, we will denote the representation of an $\mathcal{L}$-expression E in LF as $\ulcorner E \urcorner$.

Furthermore, each individual $\mathcal{L}$-theory Σ is represented as an LF-theory that extends the respective LF-theory L by adding an LF-declaration for each symbol in Σ.

Definition 2.10 (FOL-Theories in LF)**.** Every FOL-theory Σ is represented as an LF-theory that introduces the declarations

- $f : \underbrace{\mathit{term} \to \ldots \to \mathit{term}}_{n} \to \mathit{term}$ for every n-ary function symbol f in Σ,
- $p : \underbrace{\mathit{term} \to \ldots \to \mathit{term}}_{n} \to \mathit{form}$ for every n-ary predicate symbol p in Σ,
- $a : \mathit{ded}\ulcorner F\urcorner$ for axioms F and some fresh name a.

where $\ulcorner F\urcorner$ is defined in definition 2.11.

Any LF-theory given according to definition 2.10 conforms to the declaration patterns of FOL.

Definition 2.11 (FOL-Expressions in LF)**.** Let Σ be a FOL-theory. Then every FOL-term t or formula F over Σ in context $x_1, \ldots, x_n$ is represented as an LF-expression $\ulcorner t\urcorner : \mathit{term}$ or $\ulcorner F\urcorner : \mathit{form}$, respectively, in context $x_1 : \mathit{term}, \ldots, x_n : \mathit{term}$, where $\ulcorner t\urcorner$ and $\ulcorner F\urcorner$ are defined by an straightforward induction:

$$
\begin{array}{lcl}
\ulcorner \top \urcorner & = & \mathit{true} \\
\ulcorner \bot \urcorner & = & \mathit{false} \\
\ulcorner \neg F \urcorner & = & \neg \ulcorner F \urcorner \\
\ulcorner F \wedge G \urcorner & = & \wedge \ulcorner F \urcorner \ulcorner G \urcorner \\
\ulcorner F \vee G \urcorner & = & \vee \ulcorner F \urcorner \ulcorner G \urcorner \\
\ulcorner F \Rightarrow G \urcorner & = & \Rightarrow \ulcorner F \urcorner \ulcorner G \urcorner \\
\ulcorner \forall x.\, F \urcorner & = & \forall\, \lambda x : \mathit{term}.\, \ulcorner F \urcorner \\
\ulcorner \exists x.\, F \urcorner & = & \exists\, \lambda x : \mathit{term}.\, \ulcorner F \urcorner \\
\ulcorner p(t_1, \ldots, t_n) \urcorner & = & p \ulcorner t_1 \urcorner \ldots \ulcorner t_n \urcorner \\
\ulcorner t_1 = t_2 \urcorner & = & \doteq \ulcorner t_1 \urcorner \ulcorner t_2 \urcorner \\
\ulcorner x \urcorner & = & x \\
\ulcorner f(t_1, \ldots, t_n) \urcorner & = & f \ulcorner t_1 \urcorner \ldots \ulcorner t_n \urcorner
\end{array}
$$

We will use infix notation to write applications of binary LF-symbols to their arguments, e.g., $F \wedge G$ instead of $\wedge\, F\, G$.

Example 2.12 (Theory of Monoid in LF). The following LF-theory represents the first-order theory of neutral element from example 1.3:

$$
\begin{array}{lcl}
e & : & \mathit{term} \\
\circ & : & \mathit{term} \to \mathit{term} \to \mathit{term} \\
\mathit{assoc} & : & \forall\, \lambda x : \mathit{term}.\, \forall\, \lambda y : \mathit{term}.\, \forall\, \lambda z : \mathit{term}.\, x \circ (y \circ z) \doteq (x \circ y) \circ z \\
\mathit{neut}_r & : & \forall\, \lambda x : \mathit{term}.\, x \circ e \doteq x \\
\mathit{neut}_r & : & \forall\, \lambda x : \mathit{term}.\, e \circ x \doteq x
\end{array}
$$

Definition 2.13 (SFOL-Theories in LF)**.** Every FOL-theory Σ is represented as an LF-theory that introduces the declarations

- $s : \mathit{sort}$ for every sort s in Σ,

- $f : tm\, s_1 \to \ldots \to tm\, s_n \to tm\, t$ for every n-ary function symbol f in Σ with input sorts $s_1, \ldots, s_n$ and output sort t,
- $p : tm\, s_1 \to \ldots \to tm\, s_n \to form$ for every n-ary predicate symbol p in Σ with input sorts $s_1, \ldots, s_n$,
- $a : ded\, \ulcorner F \urcorner$ for axioms F and some fresh name a.

where $\ulcorner F \urcorner$ is defined in definition 2.14.

Definition 2.14 (SFOL-Expressions in LF)**.** Let Σ be an SFOL-theory. Then every SFOL-term t of sort s or formula F over Σ in context $x_1, \ldots, x_n$ of sorts $s_1, \ldots, s_n$, respectively, is represented as an LF-expression $\ulcorner t \urcorner : tm\, s$ or $\ulcorner F \urcorner : form$, respectively, in context $x_1 : tm\, s_1 \ldots, x_n : tm\, s_n$, where $\ulcorner t \urcorner$ and $\ulcorner F \urcorner$ are defined by a straightforward induction as in definition 2.11 except for the quantifiers:

$$\begin{array}{lcl} \ulcorner \forall\, x : s.\, F \urcorner & = & \forall\, \lambda x : tm\, s. \ulcorner F \urcorner \\ \ulcorner \exists\, x : s.\, F \urcorner & = & \exists\, \lambda x : tm\, s. \ulcorner F \urcorner \end{array}$$

Example 2.15 (Theory of Vector Spaces in LF). The following LF-theory represents the SFOL theory of vector spaces from example 1.6:

$$\begin{array}{lcl} vec & : & sort \\ sca & : & sort \\ + & : & tm\, vec \to tm\, vec \to tm\, vec \\ 0 & : & tm\, sca \\ \vdots & & \end{array}$$

Here we omit the straightforward encodings of the remaining function symbols and the axioms that introduce a commutative group.

The below table summarizes the representation of declarative languages in LF. Here we refer to an LF-theory that extends an LF-theory L as L**-extension** and to an LF-expression over an LF-theory Σ as Σ**-expression**:

Mathematical Concept	Respective LF Representation
Declarative Language $\mathcal{L}$	LF-theory L
$\mathcal{L}$-theory Σ	L-extension Σ
Σ-expression E	Σ-expression $\ulcorner E \urcorner$

Declarative logical frameworks can provide generic tool support for concrete theories, e.g., type-checking. However, they do not have an abstract concept to represent the infinite collection of theories, and therefore, cannot reason easily about or operate on arbitrary theories.

Representing Language Translations In a declarative framework, where declarative languages are represented as theories of the framework, it is natural to represent language translations as theory morphisms in the framework between the those theories. Moreover, every theory morphism μ induces a function $\overline{\mu}$, called the **homomorphic extension** of μ, for the translation of the theories and the expressions of each theory of the object-language.

In LF, every language translation is given as an LF-theory morphism T. In particular, every $\mathcal{L}$-theory Σ and every Σ-expression E are translated by the homomorphic extension $\overline{T}$ of T (see [Rab13a]).

Definition 2.16 (Translating SFOL to FOL in LF)**.** The translation $\mathcal{S}2\mathcal{F}$ from SFOL to FOL from definition 1.11 can be represented as the LF-theory morphism *S2F* from the LF-theory *SFOL* to *FOL* that maps the symbols in *SFOL* to the LF-expressions given in the table below:

SFOL	*FOL*
sort	$term \to form$
tm	$\lambda s : term \to form.\, term$
form	*form*
ded	$\lambda F : form.\, ded\, F$
true	*true*
false	*false*
$\wedge$	$\lambda F : form.\, \lambda G : form.\, F \wedge G$
$\vee$	$\lambda F : form.\, \lambda G : form.\, F \vee G$
$\Rightarrow$	$\lambda F : form.\, \lambda G : form.\, F \Rightarrow G$
$\forall$	$\lambda s : term \to form.\, \lambda F : term \to form.\, \forall\, \lambda x : term.\, s\, x \Rightarrow F\, x$
$\exists$	$\lambda s : term \to form.\, \lambda F : term \to form.\, \exists\, \lambda x : term.\, s\, x \wedge F\, x$

In particular, the logical symbols in *SFOL* are mapped as in definition 1.11.

Then, the translation of an *SFOL*-theory Σ in LF is given as $\overline{S2F}(\Sigma)$, and of a Σ-expression E as $\overline{S2F}(E)$.

Example 2.17 (Translation of the SFOL-Theory of Vector Spaces to FOL in LF). The following LF-theory is the result of the translation of the *SFOL*-theory of vector spaces (from example 2.15) into *FOL* along the LF-theory morphism *S2F* from example 2.16:

$$\begin{array}{lcl} vec & : & term \to form \\ sca & : & term \to form \\ + & : & term \to term \to term \\ 0 & : & term \\ \vdots & & \end{array}$$

Representing Extension Principles Several implementations of individual logics use a selected number of individual extension principles. Among logical frameworks, Twelf/LF [PS99, HHP93] permits two statements: defined and undefined constants. Isabelle [Pau94] and Coq [Coq14] permit much larger, but still fixed sets that include, for example, recursive case-based function definitions. In particular, in Isabelle [Pau94] the type definition principle from section 1.3 and many others that are derived from it are extensively used. The Mizar system [TB85] uses function definitions and case-based function definitions we discussed in section 1.3.

However, often such systems hard-code the individual extension principles they work with, and do not permit other extension principles than those implemented. Existing logical frameworks typically do not support a mechanism that allows adding new extension principles.

2.1.3 Adequacy

One crucial requirement about logical frameworks is to determine the adequacy of the representation of a language or a translation in that framework. Here adequacy addresses whether the representation given in the logical framework and the actual language or the translation are the same. We will formalize this intuition in the following:

Every representation of a declarative language in a logical framework induces a new declarative language. In particular, we define declarative languages that are induced by language representations in LF:

Definition 2.18 (LF-Induced Languages)**.** Let L be an LF-theory that represents a declarative language. L induces a declarative language $\mathcal{D}^{LF}(L)$, where

- the class $\mathbf{Th}^{\mathcal{D}^{LF}(L)}$ of the theories of $\mathcal{D}^{LF}(L)$ consists of the LF-theories Σ that extend L,
- the set $\mathbf{Exp}^{\mathcal{D}^{LF}(L)}(\Sigma)$ of expressions over a theory $\Sigma \in \mathcal{D}^{LF}(L)$ consists of the LF-expressions over the LF-theory Σ.

Similarly, every representation of a language translation in a logical framework induces a new language translation. In particular, we define language translations that are induced by translation representations in LF using LF's pushouts along inclusions [HST94]:

Definition 2.19 (LF-Induced Language Translations)**.** Let T be an LF-theory morphism from L_1 to L_2. T induces a language translation $\mathcal{D}^{LF}(T)$ from $\mathcal{D}^{LF}(L_1)$ to $\mathcal{D}^{LF}(L_2)$, where

- the function $\mathbf{th}^{\mathcal{D}^{LF}(T)}$ maps each theory Σ of $\mathcal{D}^{LF}(L_1)$ to the LF-theory $\overline{T}(\Sigma)$,
- for each theory Σ of $\mathcal{D}^{LF}(L_1)$, the function $\mathbf{exp}_{\Sigma}^{\mathcal{D}^{LF}(T)}$ is the homomorphic extension $\overline{T^{\Sigma}}$ of the LF-theory morphism T^{Σ}, which completes the pushout diagram below:

$$\begin{array}{ccc} L_1 & \xrightarrow{\quad T \quad} & L_2 \\ \big\downarrow & & \big\downarrow \\ \Sigma & \xrightarrow[\quad T^{\Sigma} \quad]{} & \overline{T}(\Sigma) \end{array}$$

Let us define the bijection between two declarative languages:

Definition 2.20. Two declarative languages $\mathcal{L}_1$ and $\mathcal{L}_2$ are **bijective** if

- $\mathbf{Th}^{\mathcal{L}_1}$ is in bijection with $\mathbf{Th}^{\mathcal{L}_2}$,
- $\mathbf{Exp}^{\mathcal{L}_1}(\Sigma)$ is in bijection with $\mathbf{Exp}^{\mathcal{L}_2}(\Sigma_2)$ for every pair (Σ_1, Σ_2) of bijective $\mathcal{L}_1$ and $\mathcal{L}_2$ theories.

Now using our definitions of LF-induced languages and language translations, we can give a more formal definition of adequacy:

Definition 2.21 (Adequacy for Languages)**.** Let $\mathcal{L}$ be a declarative language and L be an LF theory. We say that L **is adequate for** $\mathcal{L}$ if there is a bijection between $\mathcal{D}^{LF}(L)$ and $\mathcal{L}$.

Definition 2.22 (Adequacy for Translations)**.** Let $\mathcal{T}$ be a language translation from $\mathcal{L}_1$ to $\mathcal{L}_2$ and T be an LF-theory morphism from L_1 to L_2 such that each L_i is adequate for $\mathcal{L}_i$, respectively. We say that T **is adequate for** $\mathcal{T}$ if the diagram below commutes:

$$\begin{array}{ccc} \mathcal{L}_1 & \xrightarrow{\quad\mathcal{T}\quad} & \mathcal{L}_2 \\ \downarrow & & \downarrow \\ \mathcal{D}^{LF}(L_1) & \xrightarrow{\mathcal{D}^{LF}(T)} & \mathcal{D}^{LF}(L_2) \end{array}$$

2.1.4 Design Principles

A key challenge in designing logical frameworks is that they should allow generalizing concepts, algorithms and theorems about declarative languages while using as simple and yet expressive primitives as possible. Here we briefly survey some successful design principles in order to later add one of ours to the literature.

Judgments-as-Types and Proofs-as-Terms The design principle of **judgments-as-types, proofs-as-terms** was developed over several decades starting with the Curry-Howard [CF58, How80] isomorphism and later with Martin-Löf type theory.

The main idea is that judgments of a declarative language $\mathcal{L}$ are represented as types of the declarative logical framework in which $\mathcal{L}$ is represented. Then the derivation of a judgment is represented as a term of that logical framework. More specifically, if J denotes a judgment, then the typing relation $e : J$ introduces a derivation e for J.

This principle is used in frameworks with dependent types like LF [HHP93]. The main advantage of this principle is that it uses a single primitive to unify expressions and judgments.

A variation of the judgments-as-types principle is the **propositions-as-types** principle: Every proposition is represented as a type, and every proof is typed by the proposition it proves. In a logical framework with propositions-as-types, the type system of the framework determines the different syntactic forms propositions may have. For instance, if the type system is equipped with function types, then propositions can be formed using implication. Therefore, the type system of the framework needs to provide one type constructor for every propositional connective. The systems Agda [Nor05], Matita [ACTZ06] and Coq [Coq14] allow this principle.

This is different from the judgments-as-types principle, where each proposition is represented as a term in the framework, and usually a type constructor is used to give one type for every proposition, namely, the judgment for the provability of a given proposition. In particular, for every propositional connective, it suffices to introduce the respective term constructor: E.g., for implication, one declares a new binary symbol $\Rightarrow$ over the type of propositions. Then a type constructor *ded* assigns a type *ded* F for every provable proposition F.

Higher-Order Abstract Syntax The main idea of higher-order abstract syntax is that terms in contexts are in bijection with terms of function types. This permits representing binders, i.e., operators that take a term with free variables as arguments, as higher-order functions.

This design principle unifies operators and binders. For instance, the first-order universal quantifier is represented as a symbol $\forall : (\mathit{term} \to \mathit{form}) \to \mathit{form}$. In particular, the binders of the object-language are represented in terms of the binders of the logical framework.

Theory Categories The theories of a declarative language often naturally form a category. Abstract logical frameworks are typically designed to work with such categories.

In a declarative logical framework with theory categories, object-languages are represented as theories of the framework and translations between the object-languages are represented as theory morphisms of the framework. Moreover, for a specific object-language L, the theories of L and the theory morphisms between L-theories are represented as theories and theory morphisms of the logical framework, respectively. Consequently, both object-languages and their theories are uniformly represented.

One immediate result of representing object-level theories as theories of the logical framework is the following: We have discussed earlier in section 1.1 that every object-language comes with its own repertoire of declaration patterns, e.g., types, functions, predicates, axioms, etc. Depending on the type of declarations the logical framework supports, often different declaration patterns of the object-language are uniformly represented as one specific type of declaration in the logical framework. For example, LF provides two types of symbol declarations: Type family declarations and term declarations. If we take first-order logic as an object-language represented in LF, we observe that all three declaration patterns of FOL, i.e, functions, predicates and axioms, are represented as term declarations in LF.

Module Systems Module systems for declarative languages build large theories out of small reusable components called the **modules**. Typically, a module system is built on top of a theory category, where the modules are the theories of the category and the modular structure is captured by theory morphisms. The key idea of such module systems is that relations between object-level theories like reuse and inheritance are uniformly represented at the module-level as theory morphisms. This permits the module system to be generic for any declarative language.

For abstract logical frameworks, the ASL module system [SW83] provides modularity for an arbitrary institution. For declarative logical frameworks, the MMT system (Module System for Mathematical Theories) [RK13] is a generic module system.

Models-as-Morphisms A model is a mathematical entity that gives a semantic interpretation of a theory. The framework of institutions [GB92] works intensively with models.

In declarative logical frameworks, models have been typically represented using record types. An alternative representation of models in declarative frameworks is given in [Rab13a, HR11] by using theory morphisms. The key idea is that if the semantic realm $\mathcal{S}$ in which models exist is formalized in the framework as a theory, then every model of a theory Σ can be represented as a theory morphism that assigns to each symbol in Σ its interpretation in $\mathcal{S}$. Thus a model of Σ becomes a theory morphism from Σ to $\mathcal{S}$.

This representation has the advantage that semantic translations, i.e., models, are unified with the syntactic translations, i.e., theory morphisms, in the logical framework.

2.2 Foundation-Independent Meta-Frameworks

In the previous sections, we have analyzed the hierarchy of mathematical knowledge representation: mathematical theories (like the theory of monoids) are formalized in a declarative language (like first-order logic), which itself is represented in a declarative logical framework (like LF). A **meta-framework** adds one more level to this hierarchy: logical frameworks themselves are represented in a meta-framework as a **foundation**. Within that context, **foundation-independent** means that the meta-framework does not commit to a particular formal system, type theory or logic; instead every formal system is represented as a theory of the meta-framework. This notion was introduced by the MMT system (Module System for Mathematical Theories) [RK13], and is a key feature of this meta-framework.

A Module System for Mathematical Theories (MMT) MMT provides a minimal number of primitives:

Firstly, MMT uses a single primitive to represent every formal system: Logical frameworks (like LF), logics (like first-order logic) and mathematical theories (like the theory of monoids) are represented uniformly as **MMT theories**, and are related by the **meta-theory** relation. These form theory graphs such as the one in figure 2.1, where simple arrows $\to$ denote theory translations and hooked arrows $\hookrightarrow$ denote the meta-theory relation between two MMT-theories. The theory *FOL* for first-order logic is the meta-theory for *Monoid* and *Ring*. And the theory *LF* for the logical framework LF [HHP93] is the meta-theory of *FOL* and *HOL* for higher-order logic.

MMT uses the meta-theory relation to obtain the semantics of a theory from the semantics of its meta-theory. If we refer to the MMT theories with meta-theory M as M-**theories**, then the semantics of M induces the syntax and semantics of all M-theories. For example, if the syntax and semantics are fixed for *LF*, they determine those of *FOL* and *Monoid*.

Then, an external semantics, e.g., a research article or an implementation, only has to be supplied for the topmost meta-theories, e.g. logical frameworks. MMT refers to the external semantics as **foundation**. For example, the foundation for LF can be given in the form of a type theory. Foundations define in particular the typing relation between expressions, in which MMT is parametric. For example, the foundation for *LF* induces the type-checking relation for all theories with meta-theory *LF*.

MMT-theories contain typed **symbol declarations**, which MMT uses as a single primitive to uniformly represent all statement declarations, e.g., constants, functions, predicates, and via the Curry-Howard correspondence, judgments, inference rules, axioms and theorems. These are differentiated by the type system of the respective meta-theory. An MMT symbol declaration $c[: E_1][= E_2]$ introduces a named atomic expression c with optional type E_1 or definition E_2.

MMT-theories are related via **MMT theory morphisms**, which represent translations of mathematical theories, languages, functors and models uniformly. In figure 2.1, the theory translations from *Monoid* to *Ring* and from *FOL* to *HOL* are MMT theory morphisms. A special form of MMT theory morphisms are **inclusion morphisms**, which permit a theory to inherit from another one.

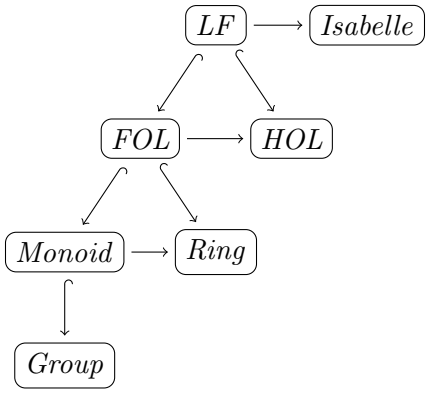


Figure 2.1: An MMT Theory Graph

MMT expressions are a fragment of OpenMath [BCC+04] objects and represent mathematical objects such as terms, types, formulas, and proofs uniformly. MMT expressions are formed from variables x, constants c, applications $E\,E_1\,\ldots\,E_n$ of functions E to a sequence of arguments E_i, and bindings $E_1\,\Gamma.\,E_2$ that use a binder E_1, a context Γ of bound variables, and a scope E_2. Contexts Γ consist of variables $x[:\,E]$ that can optionally attribute a type E.

MMT itself is not aware of the typing relation between the expressions and delegates the resolution of the typing judgments to the foundation under consideration. That way, all MMT results are obtained for arbitrary foundations. For example, MMT guarantees that theory morphisms translate objects in a typing- and truth-preserving way. This is crucial for the reuse of results in large networks of theories.

Finally, MMT provides a module system for building large theories and theory morphisms via reuse and inheritance. The module level of MMT introduces named **theory declarations** $\%\texttt{theory}\ T = \{\Sigma\}$, for theories Σ and named **theory morphism declarations** $\%\texttt{view}\ \mu = \{\sigma\}$ that are called **views** for theory morphisms σ.

Moreover, theories can include other theories T via $\%\texttt{include}\ T$, and each theory may declare its meta-theory T via $\%\texttt{meta}\ T$. Similarly, MMT views can include other ones.

Example 2.23 (MMT-Theories). Below we give an MMT theory for the logical framework LF. `type`, $\to$, and *lam* are untyped constants representing the primitives of LF. Here the symbol @ represents the binary application operator, i.e., $@\,A\,B$ denotes the application $A\,B$ of A to B.

$\%\texttt{theory}\ LF\ =\ \{$
  `type`
  Π
  $\to$
  λ
  @
$\}$

$\%\texttt{theory}\ Forms\ =\ \{$
  `meta` LF
  $form$: `type`
  ded : $form \to$ `type`
$\}$

Then the MMT-theory *Forms* introduces all symbols needed to declare logical connectives and inference rules of a logic. *form* is the type of logical formulas and *ded* is a constant that assigns to each logical formula F : *form* the type *ded* F of its proof. The syntax and semantics of this theory are defined in terms of its meta-theory LF.

The LATIN Meta-Framework Along the lines of the ideas of MMT, the Logic Atlas and Integrator (LATIN) project [KMR09] introduced the theoretical concept of **logical meta-framework** [CHK+12] that combines the approaches of the abstract and the declarative logical frameworks without being biased in any of the two directions. The LATIN meta-framework (LMF) is foundation-independent and allows representing logical frameworks as declarative languages given by categories of theories. In particular, it can be instantiated with logical frameworks such as LF [HHP93], Isabelle [Pau94] and rewriting logic [MOM94].

LMF specializes to logics using theory graphs. More precisely, the concrete logical framework chosen in LMF provides a distinguished theory *Base* that declares the primitive logical notions. Then a logic is represented as a span consisting of *i)* the syntax, *ii)* the proof-theory, and *iii)* the model-theory of that logic.

In particular, LMF guarantees that each such representation of a logic defines an institution for that logic, and more generally, each theory graph in a logical framework leads to a graph of institutions and comorphisms.

LMF, along with instantiations for specific logical frameworks, has been integrated within the Heterogeneous Tool Set (Hets, [MML07]), a set of tools for multi-logic specifications based on the framework of institutions that permits performing heterogeneous proof about heterogeneous theories formulated in logics. This integration allows new logics to be added in Hets (and in its logic graph) from their formalization in a logical framework in a declarative fashion, allowing logic formalizations to be combined with the theorem proving technology [CHJ+13].

Within the instantiation of LMF with LF, an atlas of logics and logic translations has been created in a modular way, ranging from propositional logic, first-order, modal, description and higher-order logics to foundational set and type theories, which we will elaborate further on below.

2.3 Libraries of Language Representations

Libraries of language representations are given in several projects including Logosphere [PSK+03], LATIN [KMR09] and Hets [MML07].

In the LATIN project [KMR09], an atlas of logics and their translations were represented within the LATIN meta-framework LFM instantiated with the logical framework LF. The LATIN atlas contains highly modularized formalizations of various logics, type theories, foundations of mathematics, algebra, and category theory. Among the logics formalized in the LATIN atlas are propositional (PL), first (FOL) and higher-order logic (HOL), sorted ($SFOL$) and dependent first-order logic ($DFOL$), description logics (DL), modal (ML) and common logic (CL) as illustrated in figure 2.2. Single arrows ($\longrightarrow$) in this diagram denote translations between formalizations and hooked arrows ($\hookrightarrow$) denote imports. Other formal systems formalized in the LATIN atlas are Zermelo-Fraenkel set theory, Isabelle's higher-order logic, and Mizar's set theory (the formalizations are given in [IR11]). Type theory formalization includes the λ-cube [Bar92]. The modular representation of λ-cube allows combining the different corners of the cube in any way. Notable special cases were published as [HR11] and [IR11].

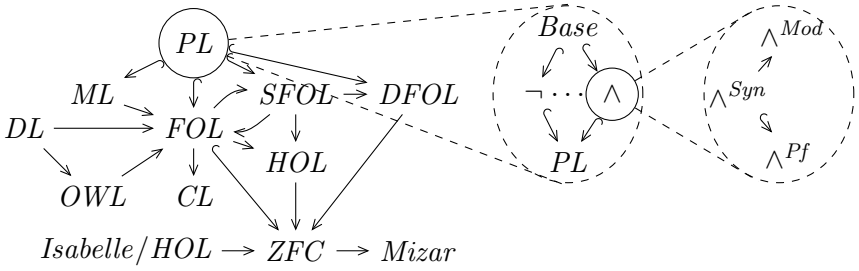


Figure 2.2: Logics and their Interrelations in the LATIN Logic Atlas

A high-level overview of a fragment of the logic atlas is given in the left part of figure 2.2. The whole graph is significantly more complex as we use the LF module system to obtain a maximally modular design of logics. For example, propositional, higher-order, modal, and description logics are formed from orthogonal modules for the individual connectives, quantifiers, and axioms. For example, the $\wedge$-connective is only declared once in the whole logic atlas and imported into the various logics and foundations and related to the type theoretic product via the Curry-Howard correspondence. Moreover, we use individual modules for syntax, proof theory and model theory so that the same syntax can be combined with different interpretations. For example, the formalization of $\wedge$ consists of the signatures $\wedge^{Syn}$ for syntax containing the connective itself, $\wedge^{Pf}$ for proof theory containing natural deduction style inference rules for $\wedge$, and $\wedge^{Mod}$ for model theory containing a meta-language (in this case *ZFC*) to axiomatize the properties of models of $\wedge$, an interpretation of $\wedge$ in *ZFC* and axioms specifying its truth values. The whole modularized logic atlas comprises over 1000 LF theories and morphisms. Despite the achievements of the LATIN project, it revealed major insufficiencies in logic representations in existing logical frameworks, which serve as essential motivation for the developments in this thesis. We elaborate on these further in chapter 3.

Chapter 3

Research Problems and Methodology

In this chapter we discuss the research problems that motivate the work in this thesis, formulate our research objectives and give an overview of our methodologies towards achieving those goals.

In the previous chapter we took a closer look at the state-of-art of representing declarative languages, language translations and extension principles in declarative logical frameworks as well as of foundation-independent meta-frameworks and libraries of language representations. In the following we will elaborate on the major problems we have encountered in them, in particular within the framework of LF [HHP93] and the LATIN logic atlas [KMR09].

3.1 Representing Declarative Languages

In current declarative logical frameworks, we see three main problems regarding the representation of declarative languages. We will discuss each problem below:

Over-generation A major problem is the **over-generation** problem. Here we use the term over-generation whenever a concept induced by the framework contains more members than the actual concept that is represented in the framework.

In particular, in a representation L of a declarative language $\mathcal{L}$ in LF, we observe that the class $\mathbf{Th}^{\mathcal{D}(L)}$ of theories of an LF-induced declarative language is over-generated (recall definition 2.18): It consists of all LF-theories Σ that extend the LF-theory L — including those LF-theories which do not conform to the respective definition of $\mathcal{L}$-theories in LF, i.e., to the declaration patterns of $\mathcal{L}$. We will refer to such theories as **ill-shaped** theories. Moreover, we will refer to an LF-theory Σ that extends L **pattern-valid with respect to** L if it consists of only declarations that conform to the declaration patterns of $\mathcal{L}$.

For example, recall definition 2.10 of FOL-theories in LF. It precisely specifies the LF-declarations that are allowed in the LF representation of a FOL-theory, and example 2.12 gives an example of an LF-theory that is pattern-valid with respect to *FOL*. However, it is very easy to write an LF-theory that is a member of $\mathbf{Th}^{\mathcal{D}(L)}$, but does not conform to definition 2.10:

Example 3.1 (Over-Generation of Theories). Consider the following LF-theory which extends the LF-theory for FOL from definition 2.7 with the following declarations:

$$
\begin{array}{lcl}
f_1 & : & \mathit{form} \to \mathit{term} \\
f_2 & : & \mathit{term} \to \mathit{form} \to \mathit{form} \\
f_3 & : & \mathit{form} \to \mathit{form} \to \mathit{form}
\end{array}
$$

Here, the LF-declarations do not conform to definition 2.10 and therefore do not introduce first-order function or predicate symbols, even though they are well-formed declarations in LF.

The underlying cause of the over-generation problem in the above example is that definition 2.10 is not part of the formal representation of FOL in LF. In fact, the formal syntax of LF does not provide a means to formally give such definitions as a part of an LF representation of a declarative language.

There are two main consequences of the over-generation problem: Firstly, a framework that over-generates the theories of an object-language L is not able to determine which LF-theories are pattern-valid with respect to L. We argue that a logical framework should be capable of validating the well-defined theories of its object-languages.

Secondly, the class of theories of the induced language is bigger than the class of theories of the actual language $\mathcal{L}$ that is represented and consequently, adequacy fails for the representation of $\mathcal{L}$-theories.

In particular, the LATIN logic atlas [KMR09] suffers from the latter problem. In figure 3.1, we give an overview of some of the logics in the LATIN logic atlas with respect to their adequacy:

Languages	Expression-Adequate	Theory-Adequate
ML	✓	×
FOL	✓	×
SFOL	✓	×
HOL	✓	×

Figure 3.1: Overview of Language Representations

Here ML stands for modal logic and HOL for higher-order logic. Moreover, **expression-adequate** means that the representation of the expressions of the language is adequate. And similarly, **theory-adequate** means that the representation of the theories of the language is adequate. The logics represented in the LATIN logic atlas are expression-adequate, but not theory-adequate.

Lack of Support for Arbitrary Theories Definitions like definition 2.10 specify the syntactic shape of an *arbitrary* theory of an object-language in LF. Lacking the ability to express such definitions within a framework also means that the framework cannot talk about arbitrary theories, and consequently, cannot express algorithms or meta-theorems that take an arbitrary theory as input. One example of such an application is language translations, which we will discuss in section 3.2.

Lack of Precision An even more striking problem arises when a logical framework cannot define the theories of the object-languages in it: Consider the logics SFOL and dependent first-order logic (DFOL) [Rab06]. Even though these are two different languages, their representations in LF would look the same. In fact, these languages only differ with respect to their theories — i.e., their declaration patterns: DFOL-theories are allowed to declare dependent sorts, whereas SFOL-theories are not. Similarly, the representations of

SFOL and a polymorphic version of it, e.g., the one [BP12] of TPTP [SS98] look the same in LF (we will come back to this example later in chapter 9). In fact, all three logics differ only in what kind of operators are allowed to declare sorts in their theories. This is an important part of the definition of these languages that we would like to capture when representing them in a logical framework.

3.2 Representing Language Translations

In declarative logical frameworks, in particular in LF, we see three main problems regarding the representation of language translations. We will elaborate on those problems using the following terminology:

Let L_i be LF-representations of the languages $\mathcal{L}_i$. Then we say that a LF-representation $T : L_1 \to L_2$ of a translation $\mathcal{T}$ from $\mathcal{L}_1$ to $\mathcal{L}_2$ is

- **adequate for expressions** if it induces the intended translation of expressions, i.e., the induced expression translation function $\mathbf{exp}_{\Sigma}^{\mathcal{D}^{LF}(T)}$ from definition 2.19 is adequate in the sense of definition 2.22,
- **valid for theories** if it induces a mapping that maps an L_1-pattern-valid LF-theory to an L_2-pattern-valid LF-theory,
- **adequate for theories** if it induces the intended translation of theories, i.e., the induced theory translation function $\mathbf{th}^{\mathcal{D}^{LF}(T)}$ from definition 2.19 is adequate in the sense of definition 2.22.

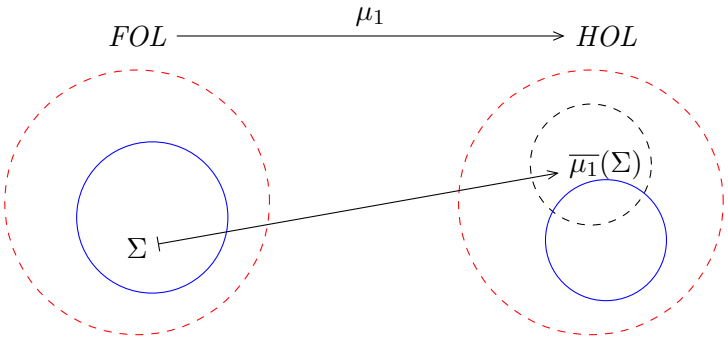
Note that these three properties can be generalized and applied to an arbitrary declarative logical framework. Here we will focus on LF.

For each of the above-mentioned properties, we give examples of language translations from the LATIN logic atlas in figure 3.2. Here, $FOL \to HOL$ is the straightforward embedding of FOL in HOL. $SFOL \to FOL$ is the translation from definition 1.11. And $ML \to FOL$ is the standard Kripke-model-based translation of modal logic to FOL, which makes the worlds of a Kripke model explicit in FOL.

Translations	Expression-Adequate	Theory-Valid	Theory-Adequate
$FOL \to HOL$	✓	×	×
$SFOL \to FOL$	✓	✓	×
$ML \to FOL$	✓	✓	✓

Figure 3.2: Overview of Representing Language Translations

The translation $\mu_1 : FOL \to HOL$ is adequate for expressions, but neither valid nor adequate for theories: It maps the symbol declarations $f : term \to \ldots \to term$ in a FOL-theory Σ to LF-declarations $f : tm\, i \to \ldots \to tm\, i$, where $i : tp$ is an LF-symbol denoting the HOL-type of individuals. The resulting declaration $f : tm\, i \to \ldots \to tm\, i$ in $\overline{\mu_1}(\Sigma)$ does not conform to a HOL-pattern. An adequate mapping would yield the declaration $f : tm\,(i \Longrightarrow \ldots \Longrightarrow i)$ that represents HOL-functions in LF. Therefore, the translation $FOL \to HOL$ is not well-defined for theories and consequently not theory-adequate.

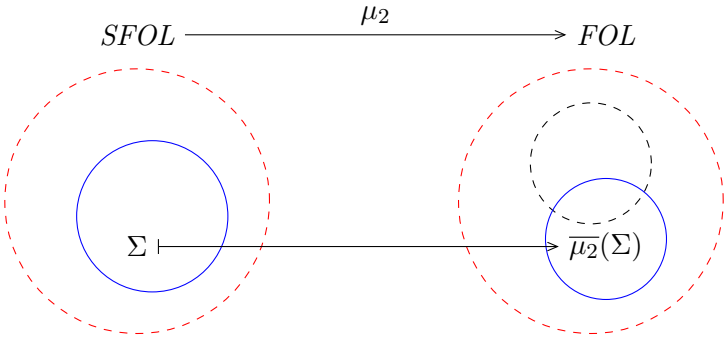


Red = LF-theories
Blue = Pattern-valid LF theories

The translation $\mu_2 : SFOL \to FOL$ in LF is valid for theories. For instance, the declaration $f : tm\, A_1 \to \ldots \to tm\, A_n \to tm\, B$ in an *SFOL*-theory Σ for $A_i : sort$ and $B : sort$ is mapped to the declaration

$$f : \underbrace{term \to \ldots term}_{n} \to term$$

in $\overline{\mu_1}(\Sigma)$, which conforms to a *FOL*-pattern.



Red = LF-theories
Blue = Pattern-valid LF theories

However, the translation is not adequate for theories: The resulting LF-theory $\overline{\mu_2}(\Sigma)$ does not contain the necessary axioms should be generated due to elimination of *SFOL*-sorts.

The translation $ML \to FOL$ in LF satisfies all three properties above. This is because the translation from ML to FOL can be adequately represented by an LF theory morphism.

3.3 Representing Extension Principles

We discussed in section 2.1 that several implementations of logical systems use individual extension principles and provide a hard-coded implementation for them. We also mentioned that existing logical frameworks do not provide a representation mechanism for extension principles that would allow coupling the representation of declarative languages with their respective extension principles. This has been generally overlooked by in the existing logical frameworks.

For example, in axiomatic set theory with only a predicate symbol $\in$ and the usual axioms, the construction of first-order terms is only possible via the extension principle of implicit function definitions. Without this, it would become practically infeasible to develop any sophisticated mathematical content. If we use higher-order logic as a foundation of mathematics, then similarly, without the type definition principle, it would become practically impossible to develop large formalizations.

Languages for mathematics commonly permit a variety of extension principles. But existing implementations of logical systems do not include a generic mechanism that permits introducing new extension principles. The problem is that instead a fixed set is built into the object-language in the framework, and thus any extension principle that is desired needs to be hard-coded in the implementation.

Moreover, we know that reasoning about conservativity is undecidable. Current systems, such as Isabelle [Pau94] or Hets [MML07] can define conservativity theoretically, but must employ a set of heuristic criteria to check for practical cases. At the very least, it is desirable to allow representing the extension principles that are known to be conservative. Ideally, the framework would help the user to prove conservativity for extension principles.

3.4 Research Objectives

O1: Pattern-Based Representations We want to design a foundation-independent meta-framework that permits theory-adequate representations of declarative languages and their translations. The framework must be capable of representing the definition of the theories, in particular the declaration patterns, and use this representation to validate theories that conform to that definition. In particular, the over-generation problem from section 3.1 must be solved.

Furthermore, the framework must permit representing the definition of theory translations between declarative languages. In particular, the represented translations must guarantee that pattern-valid theories in the framework are translated to pattern-valid theories.

O2: Representing Extension Principles The designed meta-framework must provide a generic mechanism for representing arbitrary extension principles. In particular, individual representations of extension principles must be available for the languages represented in the framework via language translation.

O3: An Atlas of Adequate Representations Based on our meta-framework, we want to create an atlas of adequately-represented languages and language translations. The atlas must include commonly-used extension principles and their translations between various languages in the atlas.

3.5 Methodology

This thesis contributes to the design of declarative logical frameworks by identifying a new design principle that balances the trade-off between representing individual theories and representing arbitrary theories: Declaration patterns. The main idea behind declaration pattens is to capture the common syntactic structure of the theories of declarative languages.

In the following, we will outline the main ideas of our solution to the research problems in the previous sections.

3.5.1 Declaration Patterns

We exploit the observation that theories typically consist of a list of declarations each of which conforms to one out of a few patterns fixed by the declarative language. For instance, definition 1.1 of FOL-theories suggests three declaration patterns: one each for the declaration of *(i)* function symbols, *(ii)* predicate symbols and *(iii)* axioms. We represent each of these three different types of declarations as one declaration pattern.

While declarative logical frameworks like LF focus on defining the logical symbols and the expressions, declaration patterns characterize the legal declarations of non-logical symbols by specifying their syntactic shape. Moreover, in a legal theory of a language $\mathcal{L}$, each declaration must match one of the declaration patterns of $\mathcal{L}$. Then, theories of $\mathcal{L}$ may only contain declarations that conform to a declaration pattern of $\mathcal{L}$.

The syntactic elements we utilize for declaration patterns are intuitive concepts that are already at our disposal: We take ordinary theories and parametrize them by expression arguments. Our approach finds a good balance between language expressivity versus complexity while we are able to express a wide range of concepts using only a simple syntactic primitive. In particular, declaration patterns allow the uniform representation of

- the shape of the legal declarations in theories,
- extension principles and
- repetitive groups of declarations / abbreviations (pragmatic statements).

The expressivity of our approach comes from one relaxation we allow in the translation of declaration patterns: They are not required to completely preserve the structure in their body. This gives us great flexibility in expressing how certain forms of declarations should be translated.

Nevertheless, our approach has certain limitations: Not every language translation is expressible as a theory morphism since we do not have induction on theories. Using declaration patterns, we can express only those translations that map every instance of one pattern to a certain set of declarations in the other language.

3.5.2 Sequences

Interestingly, we have to introduce one additional primitive concept that is technically independent but practically necessary for declaration patterns: We usually need sequences of expressions to define the declaration patterns of a declarative language $\mathcal{L}$ even if the expressions of $\mathcal{L}$ do not use sequences. For example, in theories of first-order logic, the declaration pattern for function symbols use a sequence of terms — the arguments of the function symbol. Therefore, we introduce expression sequences and natural numbers to our design.

At this point, there are two very crucial and orthogonal questions to answer: *i)* at which language level should sequence expressions be introduced, *ii)* what primitives should be used for sequence expressions.

The answer to the first question has three options: *i)* the level of the theories of a declarative language, e.g., developing a theory of sequences in a specific logic, *ii)* the level

of declarative languages, e.g., adding sequences to a specific logic as a fixed interpreted sort, *iii)* the level of foundations, e.g., introducing sequences in a specific foundation.

The level theories does not work for our purposes, because declaration patterns are defined at the level of declarative languages for any theory of the language. Then the second option of introducing sequences in specific logics where they are needed might sound more plausible. This would suggest that sequences should be part of those languages that need them, which would, in contrary, make the specific logics too complicated.

Therefore, our goal is to add sequences at the level of foundations. This corresponds most closely to informal mathematics where sequences and ellipses are assumed to be given at the informal meta-level and not explicitly defined at the logical or set theoretical level.

3.5.3 Modular Foundations

While sequences are needed in the logical framework to express the declaration patterns of several languages in practice, they should be part of the logical foundations only when they are actually needed. Therefore, it is desirable to support modularity for foundations, i.e., logical frameworks, to combine different foundational features and to enhance their re-usability for different applications.

Reusable foundational features are typically syntactic primitives like constructors for expressions, notations, typing rules, algorithms and rule-based implementations like type-checking, type reconstruction or parsing. A crucial requirement for reusing foundational features is to represent them in (possibly finite) set-like structures so that standard mathematical operations like union can be applied to combine them.

Therefore, we represent the syntax of a foundation as a set of primitive symbols, which permits combining different syntaxes by taking the union of those sets. Moreover, the reuse of syntactic concepts and definitions require some form of syntax translation.

We take one step forward and look into the structure of the semantics of foundations in addition to their syntax. In particular, we pursue a **rule-based** approach, where the semantics of a foundation is given as a set of inference rules. Then, meta-properties and algorithms that proceed by induction on the set of inference rules of the foundation can be reused by combining respective rules.

The key novelty we use for modular foundations is an abstract representation of the inference rules for the expressions in a foundation. In particular, we represent the inference system of a foundation as a set of rules, so-called **foundational-rules**, that conform to a rule scheme. This rule scheme covers a wide range of inference rules including, for example, typing and equality rules for dependent function types.

3.6 Thesis Outline

This thesis is structured as follows:

In chapters 4 and 5 we introduce our calculus-based, foundation-independent meta-framework TFI for representing declarative languages and their translations. TFI is designed in two main parts: The first part is introduced in chapter 4, in which we develop *modular foundations*, a novel feature that allows combining exisiting foundations to define new ones within TFI. We give LF as an example of a foundation in TFI.

In chapter 5, we introduce the primitive concepts of TFI. The key novelty in this chapter is the notion of *theory families* and their *instantiations*. Theory families will serve as the syntactic means to write down the declaration patterns of a language in TFI/$\mathcal{F}$.

In chapter 6, we introduce a new foundation within TFI for sequences, and combine it with the foundation LF, resulting in a novel foundation LFS that supports ellipses, dependent functions of flexible arity and sequence arguments.

Then, in chapter 7, we employ our developments from chapter 5, in particular theory families and their instantiations, to define declarative languages and their theories in TFI/$\mathcal{F}$, and define language translations that respect declaration patterns.

In chapter 8, we show how to represent extension principles in TFI and employ our definition of language translation from chapter 7 to translate extension principles between declarative languages.

Chapter 9 presents a collection of case studies we carried out to evaluate our developments. In particular, we give an atlas of declarative languages and translations between them represented in TFI with LFS being the underlying foundation of the representations. Then, we enrich the language atlas with specific extension principles we defined in our framework and use the translations in our atlas to translate them.

Chapter 10 concludes this thesis with a summary and discusses future work.

Chapter 4

Modular Foundations

Foundations, in our sense, are formal meta-languages that are assumed to be a priori given as opposed to object-languages that are defined within a meta-language. Typical examples include declarative logical frameworks like LF [HHP93] or Isabelle [Pau94].

In this chapter we give an abstract definition of foundation and introduce mathematical operations that permit combining existing foundations modularly. More specifically, we define a foundation as a set of primitive symbols and a set of inference rules that define the semantics of these primitives.

In section 4.1 and section 4.2 we describe the representation of the syntax and the inference system of a foundation, respectively. In section 4.4, we define module operations for inheritance, union, and translation of foundations to define new foundations from existing ones.

We will use our definition of foundation in chapter 5 as a generic base layer on top of which we define declarative languages and translations.

4.1 Syntax

When we talk about the *syntax* of a foundation, we consider *i)* the primitive symbols of a foundation, *ii)* the theories of the foundations, *iii)* the expressions over a given theory of the foundation, and *iv)* the notations used for those expressions.

We introduce the primitive symbols of a foundation in *foundational theories*:

Definition 4.1 (Foundational Theory)**.** A **foundational theory** is a set. We call the elements of this set **foundational symbols**.

Foundational symbols are typically identifiers like strings or URIs, e.g., MMT URIs [RK13]. The notion of foundational theory as a set of symbols is first introduced in the work of [RK13][1]. We use the same concept here because it is simple and general.

Next, we will define the theories of a foundation and the expressions over a given theory by using a context-free grammar. We will use a Backus-Naur-Form style notation to introduce our context-free grammars. Non-terminal symbols are introduced on the left of ::=, and the vertical bar | is used to separate the expressions and denotes a choice. We use $[E]$ to denote optional components E. The sign $+$, as in E^+, denotes the occurrence of at least one meta-variable E.

Definition 4.2 (Theories of a Foundation)**.** The **theories** of a foundation are formed by the following grammar in figure 4.1:

[1] Foundational theories are meta-theories in MMT.

Theories	Σ	$::=$	$\cdot \mid \Sigma, x : E[= E]$
Expressions	E	$::=$	$s \mid x \mid \beta(E\,;\,\Sigma\,;\,E) \mid @(E; E^+)$

Figure 4.1: Grammar

More specifically, theories Σ are formed by declarations $x : E[= E']$ of non-primitive symbols x of type E and optional definiens E'.

The **expressions** E are OpenMath [BCC+04] objects and are formed by

- foundational symbols s,
- symbols x that are declared in theories Σ,
- bindings $\beta(E\,;\,\Sigma\,;\,E')$, where E binds the symbols in Σ in the scope E' and Σ consists of symbol declarations of the form $x : E$ only (no defieniens) and
- applications $@(E; E_1, \ldots, E_n)$, where E is applied to $E_1, \ldots, E_n$.

Notation 4.3 ($\mathcal{F}$-Theories). We will refer to *i)* the theories of a foundation $\mathcal{F}$ as **$\mathcal{F}$-theories**, and *ii)* the expressions over an $\mathcal{F}$-theory Σ as **Σ-expressions**. Moreover, we will denote *i)* the set of Σ-expressions as $Exp^{\mathcal{F}}(\Sigma)$ and omit the superscript $\mathcal{F}$ whenever the foundation is clear from the context, and *ii)* the set of expressions over the foundational symbols of $\mathcal{F}$ only, i.e., $Exp^{\mathcal{F}}(\cdot)$, as $Exp^{\mathcal{F}}$.

Definition 4.4 (Domain of a Theory)**.** We define the **domain** of a theory Σ inductively on the declarations in Σ:

$$\begin{array}{lcl} dom(\cdot) & = & \emptyset \\ dom(\Sigma, x : E[= E']) & = & dom(\Sigma) \cup \{x\} \end{array}$$

Definition 4.5 (Free Occurring Symbols)**.** We define the set $FV(E)$ of symbols that occur **free** in an expression E inductively on the formation of E:

$$\begin{array}{lcl} FV(x) & = & \{x\} \\ FV(s) & = & \emptyset \\ FV(\beta(E\,;\,\Sigma\,;\,E')) & = & FV(E) \cup FV(\Sigma) \cup (FV(E') \setminus dom(\Sigma)) \\ FV(@(E; E_1, \ldots, E_n)) & = & FV(E) \cup FV(E_1) \cup \ldots \cup FV(E_n) \end{array}$$

Similarly, we define the free symbols of Σ:

$$\begin{array}{lcl} FV(\cdot) & = & \emptyset \\ FV(\Sigma, x : E[= E']) & = & FV(\Sigma) \cup (FV(E)[\cup FV(E')] \setminus dom(\Sigma)) \end{array}$$

Moreover, we say that the occurrences of the symbols $x_1, \ldots, x_n$ in Σ in bindings $\beta(E\,;\,\Sigma\,;\,E')$ are **bound**.

Note that foundational symbol s do not occur free or bound in expressions as they are primitive symbols.

At this point, we could already define the substitution of free occurring symbols, but we find it more elegant to introduce that as a special case of morphism application in section 5.1.3.

Typically, a foundational theory is presented with a set of **notations** for the expressions. We will not formally restrict the notations a foundation may use, and will introduce them in an ad hoc manner whenever they are intuitive and improve the readability of the syntax of the foundation.

The following example introduces the syntax of LF using our definition of syntax.

Example 4.6 (LF Syntax). The foundational theory of LF is the set $\{\texttt{type}, \texttt{kind}, \lambda, \Pi, \texttt{app}\}$. These symbols permit the formation of expressions of the form:

- $\beta(\lambda\,;\,\Sigma\,;\,E)$ for λ-abstractions that bind the symbols in Σ over E,
- $\beta(\Pi\,;\,\Sigma\,;\,E)$ for dependent function types and kinds that bind the symbols in Σ over E,
- $@(\texttt{app}; E, E_1, \ldots, E_n)$ for applications of expressions E to $E_1, \ldots, E_n$.

The type system for LF in example 4.10 will restrict the use of $\beta(E\,;\,\Sigma\,;\,E')$ to the foundational symbols λ and Π only.

We introduce the following notations:

- $\lambda x_1 : A_1. \ldots \lambda x_n : A_n. E$ for bindings $\beta(\lambda\,;\,\Sigma\,;\,E)$ where $\Sigma = x_1 : A_n, \ldots, x_n : A_n$,
- $\Pi x_1 : A_1. \ldots \Pi x_n : A_n. E$ for bindings $\beta(\Pi\,;\,\Sigma\,;\,E)$ where $\Sigma = x_1 : A_1, \ldots, x_n : A_n$, and $A_1 \to \ldots \to A_n \to E$ whenever no x_i occurs in $A_1, \ldots, A_n$ and E.
- $E E_1 \ldots E_n$ for applications $@(\texttt{app}; E, E_1, \ldots, E_n)$ (application is left-associative).

4.2 Type System

The type system of a foundation typically employs *i)* a typing judgment to specify the well-formed expressions, *ii)* an equality judgment on the well-formed expressions, and *iii)* inference rules for these judgments.

We will use the judgments in figure 4.2 to specify the type system of any given foundation $\mathcal{F}$: The judgments $\Sigma \vdash E : E'$ and $\Sigma \vdash E \doteq E'$ are standard typing and equality judgments for well-formed $\mathcal{F}$-expressions E and E' over a well-formed theory Σ. In particular, the equality judgment states E and E' are equal if they are well-formed.

Note that we do not distinguish between terms, types or kinds syntactically at this level.

Additionally, we have two new judgments:

- The judgment $\Sigma \vdash E\ \mathit{Inhabitable}$ states that E can be inhabited by the expressions over a $\mathcal{F}$-theories (e.g., as types or kind in LF).
- The judgment $\Sigma \vdash E\ \mathit{Quantifiable}$ states that inhabitants of E can be bound by $\mathcal{F}$-binders (e.g., such as in $\lambda x : E. E'$ in LF).

We will use these judgments as auxiliary judgments in the formation rules for binders.

$\Sigma \vdash E : E'$	E is a well-formed expression inhabiting (well-formed) E' over Σ.
$\Sigma \vdash E \doteq E'$	E is equal to E' over Σ (if E and E' are well-formed).
$\Sigma \vdash E\ \mathit{Inhabitable}$	E is inhabitable over Σ.
$\Sigma \vdash E\ \mathit{Quantifiable}$	E is quantifiable over Σ.

Figure 4.2: Foundational Judgments

Note that judgments and rules over them are foundation-independent, whereas the derivability of a specific judgment is foundation dependent.

In the following we introduce a rule schema that we will use to constrain the inference rules of a foundation:

Definition 4.7 (Foundational Rule)**.** A **foundational rule** is an inference rule with a set $\mathcal{M}$ of meta-variables and instantiates the following rule schema:

$$\frac{\Sigma, \boldsymbol{\Psi^1} \vdash \mathcal{S}_1 \quad \ldots \quad \Sigma, \boldsymbol{\Psi^m} \vdash \mathcal{S}_m}{\Sigma \vdash \mathcal{S}}\,\mathcal{R}$$

where Σ is meta-variable we use for $\mathcal{F}$-theories, $\boldsymbol{\Psi^j}$ consists of ${x_1}^j : A_1^j, \ldots, {x_l}^j : A_l^j$ and the succedents $\mathcal{S}$, $\mathcal{S}_1$, …, $\mathcal{S}_m$ range over the forms of the succedents of the foundational judgments in figure 4.2. The succedents and A_k^j may contain symbols from Ψ_j, and may refer to the foundational symbols of $\mathcal{F}$ and to the meta-variables in $\mathcal{M}$. Moreover, the succedents may contain substitution.

Note that our rule schema for foundational rules allow one meta-level operation in the judgments used in them: substitution. This is a safe decision, because we show in lemma 5.35 of chapter 5 that substitution is preserved under morphism application.

Now we can define a foundation as a pair of syntax and a type system for that syntax:

Definition 4.8 (Foundation)**.** A **foundation** $\mathcal{F}$ is a pair $(Sym^{\mathcal{F}}, Rules^{\mathcal{F}})$ of a foundational theory $Sym^{\mathcal{F}}$ and a set $Rules^{\mathcal{F}}$ of inference rules that consists of

- a collection of foundational rules,
- the rules in figure 4.3 and figure 4.4, which we call **structural rules**, and
- the rules of the meta-framework[2] in which $\mathcal{F}$ is used, which we call **framework rules**.

$$\frac{\Sigma \vdash E \doteq E' \quad \Sigma \vdash F_1 \doteq F_1' \quad \ldots \quad \Sigma \vdash F_n \doteq F_n'}{\Sigma \vdash \mathbb{@}(E; F_1, \ldots, F_n) \doteq \mathbb{@}(E'; F_1', \ldots, F_n')}\,\mathcal{R}AppCong$$

$$\frac{\Sigma \vdash E \doteq E' \quad \Sigma \vdash \Psi \doteq \Psi' \quad \Sigma, \Psi \vdash F \doteq F'}{\Sigma \vdash \beta(E\,;\,\Psi\,;\,F) \doteq \beta(E'\,;\,\Psi'\,;\,F')}\,\mathcal{R}BindCong$$

$$\frac{\Sigma \vdash E_1 : E_2 \quad \Sigma \vdash E_1 \doteq E_1' \quad \Sigma \vdash E_2 \doteq E_2'}{\Sigma \vdash E_1' : E_2'}\,\mathcal{R}TypeCong$$

Figure 4.3: Congruence Rules

The intuition behind our definition is that every foundation comes with a set of foundational rules that instantiate the rule schema $\mathcal{R}$. Note that according to our definition, every foundation contains congruence and equivalence rules.

In the rule $\mathcal{R}BindCong$ of figure 4.3, $\Sigma \vdash \Psi \doteq \Psi'$ is an auxiliary judgment for the equality of theories, which is defined as follows:

$$\frac{}{\Sigma \vdash \cdot \doteq \cdot}\,empty$$

[2]We introduce our meta-framework in chapter 5.

$$\frac{}{\Sigma \vdash E \doteq E}\,\mathcal{R}Refl \qquad \frac{\Sigma \vdash E_1 \doteq E_2}{\Sigma \vdash E_2 \doteq E_1}\,\mathcal{R}Sym$$

$$\frac{\Sigma \vdash E_1 \doteq E_2 \quad \Sigma \vdash E_2 \doteq E_3}{\Sigma \vdash E_1 \doteq E_3}\,\mathcal{R}Trans$$

Figure 4.4: Equivalence Rules

$$\frac{\Sigma \vdash \Psi \doteq \Psi' \quad \Sigma, \Psi \vdash T \doteq T' \quad [\Sigma, \Psi \vdash D \doteq D']}{\Sigma \vdash \Psi, x : T[= D] \ \doteq \ \Psi', x : T'[= D']}\,concat$$

As a very simple example we give a foundation for a type theory with only types and kinds.

Example 4.9. $Types = (Sym^{Types}, Rules^{Types})$, where $Sym^{Types} = \{\texttt{type}, \texttt{kind}\}$ and $Rules^{Types}$ consists of the structural rules in figure 4.3 and figure 4.4, the foundational rules in figure 4.5 and the framework rules in figure 5.10.

$$\frac{}{\Sigma \vdash \texttt{type} : \texttt{kind}}\,type$$

$$\frac{\Sigma \vdash A : \texttt{type}}{\Sigma \vdash A\ Inhabitable}\,typeInh$$

$$\frac{\Sigma \vdash B : \texttt{kind}}{\Sigma \vdash B\ Inhabitable}\,kindInh$$

Figure 4.5: Typing Rules for Types

4.3 Examples

Now we can recover the syntax and the type system of LF as a foundation in the sense of definition 4.8:

Example 4.10. We give the foundation $LF = (Sym^{LF}, Rules^{LF})$, where Sym^{LF} is the set in example 4.6 and $Rules^{LF}$ instantiates the rule schema $\mathcal{R}$ in definition 4.7 by the typing rules in figure 4.5 and figure 4.6, and the conversion rules in figure 4.7.

Note that figure 4.6 does not contain a rule for the case of well-formed non-logical symbols declared in the LF-theories Σ. We introduce look-up rules for symbols declared in $\mathcal{F}$-theories in figure 5.10 as a part of our framework in chapter 5 and every foundation $\mathcal{F}$ that we introduce in our framework will inherit these rules directly from the foundation. Similarly, Sym^{LF} does not contain a symbol for the equality of LF expressions as our definition of foundation encompasses equality rules.

$$\frac{\Sigma \vdash A : \texttt{type}}{\Sigma \vdash A\ Quantifiable}\ typeQuan$$

$$\frac{\Sigma \vdash A\ Quantifiable \quad \Sigma, x : A \vdash E_1\ Inhabitable \quad \Sigma, x : A \vdash E_1 : E_2}{\Sigma \vdash \Pi x : A.\ E_1 : E_2}\ depType$$

$$\frac{\Sigma \vdash A\ Quantifiable \quad \Sigma, x : A \vdash E_2\ Inhabitable \quad \Sigma, x : A \vdash E_1 : E_2}{\Sigma \vdash \lambda x : A.\ E_1 : \Pi x : A.\ E_2}\ abstr$$

$$\frac{\Sigma \vdash E_1 : \Pi x : A.\ E_2 \quad \Sigma \vdash M : A}{\Sigma \vdash E_1\ M : [M/x]E_2}\ app$$

Figure 4.6: Typing Rules of *LF*

$$\frac{\Sigma, x : A \vdash E : E' \quad \Sigma \vdash M : A}{\Sigma \vdash (\lambda x : A.\ E)\ M \doteq [M/x]E}\ \beta\text{-}Conv$$

$$\frac{\Sigma, x : A \vdash x : A \quad \Sigma \vdash E : \Pi x : A.\ E'}{\Sigma \vdash \lambda x : A.\ (E\ x) \doteq E}\ \eta\text{-}Conv$$

Figure 4.7: β- and η-Conversions

Note that our formulation of the LF typing rules as foundational rules in figure 4.6 differs from the formulation in [HHP93]. In our formulation of LF rules, we first categorize LF expressions into *i)* those which are allowed to introduce bound variables, and *ii)* those which can be inhabited. In the former case, one can only quantify over LF types $A : \texttt{type}$ and thus only bind typed symbols $x : A$ so that λ and Π-binders cannot bind type variables. In the latter case, inhabitable LF expressions are LF types $E : \texttt{type}$ or LF kinds $E : \texttt{kind}$.

In both the λ-formation and Π-formation rules, we make sure by adding the premise $\Sigma \vdash A\ Quantifiable$ that only term symbols $x : A$ may be bound. The premise that E is inhabitable over $\Sigma, x : A$ permits us to unify the λ and Π-formation rules for type and kind level abstraction.

Figure 4.7 gives the β- and η-equality rules.

Now let us consider the following foundation that introduces natural numbers:

Example 4.11 (Naturals). We define $Nat = (Sym^{Nat}, Rules^{Nat})$ as follows:

$$Sym^{Nat} = Sym^{Types} \cup \{\texttt{nat}, \texttt{zero}, \texttt{one}, \texttt{add}, \texttt{sub}, \texttt{leq}, \texttt{refl}, \texttt{trans}, \texttt{least}, \texttt{monotone1}, \texttt{monotone2}\}$$

The following expressions can be formed over Sym^{Nat}:

- $@(\texttt{add}; E_1, E_2)$ for the sum of two expressions E_1 and E_2.

- $@(\texttt{sub}; E_1, E_2)$ for the subtraction of expression E_2 from E_1.
- $@(\texttt{leq}; E_1, E_2)$ for the less-than or equal relation between two expressions E_1 and E_2.
- $@(\texttt{trans}; E_1, E_2)$, $@(\texttt{monotone1}; E)$ and $@(\texttt{monotone2}; E)$ for transitivity and monotonicity axioms in figure 4.10.

Expression	Notation
`zero`	0
`one`	1
$@(\texttt{add}; E_1, E_2)$	$E_1 + E_2$
$@(\texttt{sub}; E_1, E_2)$	$E_1 - E_2$
$@(\texttt{leq}; E_1, E_2)$	$E_1 \leq E_2$

Figure 4.8: Notations for *Nat* Expressions

$Rules^{Nat}$ consists of the rules in $Rules^{Types}$, figure 4.9 and figure 4.10.

$$\frac{}{\Sigma \vdash \texttt{nat} : \texttt{type}}\ typeNat$$

$$\frac{\Sigma \vdash m : \texttt{nat} \quad \Sigma \vdash n : \texttt{nat}}{\Sigma \vdash m \leq n : \texttt{type}}\ typeLeq$$

$$\frac{}{\Sigma \vdash 0 : \texttt{nat}}\ natZero \qquad \frac{}{\Sigma \vdash 1 : \texttt{nat}}\ natOne$$

$$\frac{\Sigma \vdash m : \texttt{nat} \quad \Sigma \vdash n : \texttt{nat}}{\Sigma \vdash m + n : \texttt{nat}}\ natAdd$$

$$\frac{\Sigma \vdash m : \texttt{nat} \quad \Sigma \vdash n : \texttt{nat} \quad \Sigma \vdash \_ : m \leq n}{\Sigma \vdash n - m : \texttt{nat}}\ natSub$$

Figure 4.9: Typing Rules for Natural Numbers

The rules in figure 4.9 are straightforward. It is nevertheless worth mentioning the rule *typeLeq*. We use the propositions-as-types [CF58, How80] paradigm to represent propositions on natural numbers: $\leq$ is a type judgment on natural numbers, i.e., a type constructor for `nat`, where $m \leq n$ is the type of the expressions that proves that m is less-than or equal to n. Moreover, subtraction $n - m$ is only allowed as long as the result is a positive natural number (i.e., whenever the type $m \leq n$ is inhabited).

By choosing $\leq$ to be a primitive symbol, we are able to introduce axioms about natural numbers as foundational rules in figure 4.10 that introduce $+$ as a commutative monoid

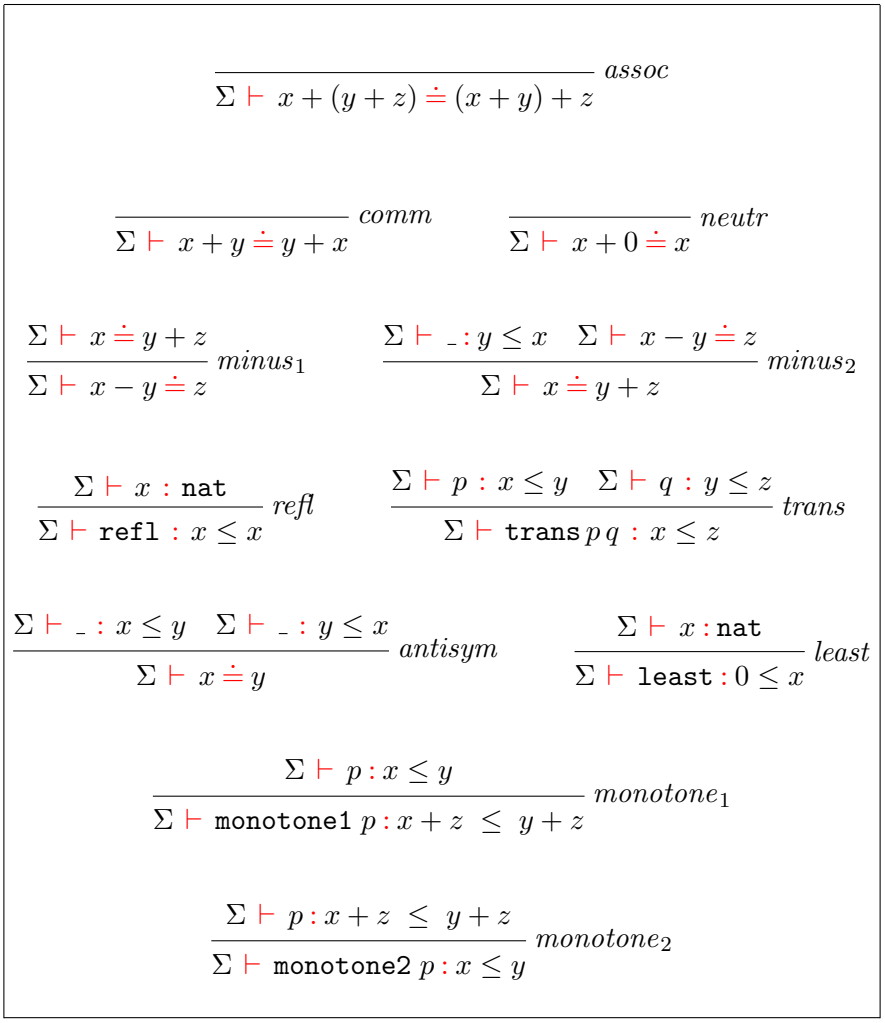


Figure 4.10: Rules for Natural Numbers

with the neutral element 0 and $\leq$ as an ordering with the least element 0. Additionally, we introduce two rules for the axiom $x = y + z$ if and only if $x - y = z$ and two rules for $+$ being monotonous with respect to $\leq$. The set of axioms in figure 4.10 exclude induction, but permit deriving the usual computations on the naturals using addition and subtraction.

Notation 4.12. We will use any meta-level natural number $2, 3, \ldots$ as an abbreviation of the corresponding natural number expression in the obvious way starting with 2 denoting $1 + 1$, and continue adding 1, e.g., 3 denoting $(1 + 1) + 1$, and so on.

Definition 4.13 (Normal *Nat*-Expressions)**.** We say that a *Nat*-expression is **normal** if it has the form $\underbrace{1 + \ldots + 1}_{n}$ for some meta-level natural number n.

Note that a *Nat*-expression without free occurring symbols is provably equal to a normal one.

4.4 Modularity

Given a foundation like *Types* in example 4.9, we can easily add more symbols to its set of primitives and couple those with appropriate inference rules in order to obtain more interesting foundations. For that purpose, we define **inclusion of foundations**:

Definition 4.14 (Inclusion of Foundations)**.** Let $\mathcal{F} = (Sym^{\mathcal{F}}, Rules^{\mathcal{F}})$ and $\mathcal{G} = (Sym^{\mathcal{G}}, Rules^{\mathcal{G}})$ be two foundations. We say that $\mathcal{G}$ **includes** $\mathcal{F}$, denoted as $\mathcal{F} \hookrightarrow \mathcal{G}$, if and only if we have $Sym^{\mathcal{F}} \subseteq Sym^{\mathcal{G}}$ and $Rules^{\mathcal{F}} \subseteq Rules^{\mathcal{G}}$.

Example 4.15. We have $Types \hookrightarrow Nat$ and $Types \hookrightarrow LF$.

Next, we use the set-theoretic notion of union to introduce the **union of foundations**:

Definition 4.16 (Union of Foundations)**.** Let $\mathcal{F} = (Sym^{\mathcal{F}}, Rules^{\mathcal{F}})$ and $\mathcal{G} = (Sym^{\mathcal{G}}, Rules^{\mathcal{G}})$ be two foundations. The **union** $\mathcal{F} \cup \mathcal{G}$ is the pair $(Sym^{\mathcal{F}} \cup Sym^{\mathcal{G}}, Rules^{\mathcal{F}} \cup Rules^{\mathcal{G}})$.

We now define what it means to apply a symbol mapping to expressions, theories, foundational judgments, and foundational rules:

Definition 4.17 (Symbol Mappings)**.** A **symbol mapping** $f : Sym^{\mathcal{F}} \to Exp^{\mathcal{G}}$ maps the foundational symbols in $Sym^{\mathcal{F}}$ to expressions over the foundational symbols of $\mathcal{G}$.

Moreover, we define the **extension** $\overline{f}$ of f to *i)* the expressions in $Exp^{\mathcal{F}}$, *ii)* $\mathcal{F}$-theories, *iii)* the succedents of the foundational judgments in figure 4.2, and *iv)* the foundational rules in $Rules^{\mathcal{F}}$ as follows:

- Expressions $E \in Exp^{\mathcal{F}}$: $\overline{f}$ replaces all occurrences of $s_1, \ldots, s_n \in Sym^{\mathcal{F}}$ in E with $f(s_1), \ldots, f(s_n)$, respectively.
- $\mathcal{F}$-theories $\Sigma = x_1 : E_1, \ldots, x_n : E_n$ are mapped to $\overline{f}(\Sigma) = x_1 : \overline{f}(E_1), \ldots, x_n : \overline{f}(E_n)$.
- The succedents of the foundational judgments in figure 4.2 are mapped as follows:

$$\begin{array}{lcl} \Sigma \vdash \overline{f}(E : E') & = & \Sigma \vdash \overline{f}(E) : \overline{f}(E') \\ \Sigma \vdash \overline{f}(E \doteq E') & = & \Sigma \vdash \overline{f}(E) \doteq \overline{f}(E') \\ \Sigma \vdash \overline{f}(E\ \mathit{Inhabitable}) & = & \Sigma \vdash \overline{f}(E)\ \mathit{Inhabitable} \\ \Sigma \vdash \overline{f}(E\ \mathit{Quantifiable}) & = & \Sigma \vdash \overline{f}(E)\ \mathit{Quantifiable} \end{array}$$

  where Σ is a meta-variable for a theory.

- Every foundational rule of the form

$$\frac{\Sigma, \boldsymbol{\Psi^1} \vdash \mathcal{S}_1 \quad \ldots \quad \Sigma, \boldsymbol{\Psi^m} \vdash \mathcal{S}_m}{\Sigma \vdash \mathcal{S}}$$

  is mapped to the rule

$$\frac{\Sigma, \overline{f}(\boldsymbol{\Psi^1}) \vdash \overline{f}(\mathcal{S}_1) \quad \ldots \quad \Sigma, \overline{f}(\boldsymbol{\Psi^m}) \vdash \overline{f}(\mathcal{S}_m)}{\Sigma \vdash \overline{f}(\mathcal{S})}$$

Now we use definition 4.17 to define a more complex operation that permits manipulating the foundational symbols included from a foundation into another one:

Definition 4.18 (Foundation Morphisms)**.** Let $\mathcal{F}$ and $\mathcal{G}$ be two foundations. We say that a (partial) symbol mapping $f : Sym^{\mathcal{F}} \to Exp^{\mathcal{G}}$ is a **(partial) foundation morphism** from $\mathcal{F}$ to $\mathcal{G}$ if $\overline{f}$ maps (some) all foundational rules in $Rules^{\mathcal{F}}$ to a derivable rule over $Rules^{\mathcal{G}}$.

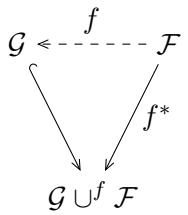
Definition 4.19 (Generative Unions)**.** Let $f : Sym^{\mathcal{F}} \to Exp^{\mathcal{G}}$ be a partial foundation morphism from $\mathcal{F}$ to $\mathcal{G}$ and $\mathcal{D} \subseteq Rules^{\mathcal{F}}$ be the set of rules $\mathcal{R}$ such that $\overline{f}(\mathcal{R})$ is derivable. The **generative union $\mathcal{G} \cup^f \mathcal{F}$ of $\mathcal{G}$ and $\mathcal{F}$ along** f is the foundation defined by

$$\begin{array}{lcl} Sym^{\mathcal{G}\cup^f\mathcal{F}} & = & Sym^{\mathcal{G}} \cup (Sym^{\mathcal{F}} \setminus dom(f)) \\ Rules^{\mathcal{G}\cup^f\mathcal{F}} & = & Rules^{\mathcal{G}} \cup \{\overline{f'}(\mathcal{R}) \,|\, \mathcal{R} \in Rules^{\mathcal{F}} \setminus \mathcal{D}\} \end{array}$$

and $f^* : Sym^{\mathcal{F}} \to Exp^{\mathcal{G}\cup^g\mathcal{F}}$ is the foundation morphism from $\mathcal{F}$ to $\mathcal{G} \cup^f \mathcal{F}$ defined as follows:

$$f^*(s) = \begin{cases} f(s) & \text{if } s \in dom(f) \\ s & \text{otherwise} \end{cases}$$

Intuitively, generative unions allow including a foundation via an instantiation for some of its foundational symbols. The generative union $\mathcal{G} \cup^f \mathcal{F}$ includes the foundation $\mathcal{G}$ and includes the foundation $\mathcal{F}$ along the partial symbol mapping f as the diagram below illustrates.



In SML [MTHM97], a similar construction can be achieved by

```
include G, include F with f.
```

In MMT [RK13], structure declarations in an MMT-theory allow such inclusions with an assignment f. In development graphs [MAH06], these correspond to definitional links.

4.5 Discussion

The more a system knows about the semantics of a foundation, the more it can facilitate reuse. For example, MMT [RK13] and Hets [MML07] regard the semantics of a foundation abstractly as an oracle and therefore have very little structure to work with. In particular, MMT does not look into the structure of the inference system of a foundation and only axiomatizes certain semantic properties that must be satisfied by individual foundations. In Hets, each foundation is implemented individually. In that case, foundations treated as oracles have to come with their own definitions, their own proofs of meta-properties and implementations of algorithms. Therefore, implementations and proofs have to be done for every individual foundation.

Our rule-based definition of foundation allows us to exploit the structure of the foundational rules, therefore, to prove certain meta-properties by induction on the derivation of each kind of foundational rule once and for all foundations. For example, the preservation of typing along theory morphisms is one major meta-property that we prove for all foundations that conform to our definition (see theorem 5.40).

Our rule schema for foundational rules is relatively simple and has broad coverage: It covers most declarative languages, in particular, pure type systems [Ber90] and systems based on Martin Löf type theory [ML74].

Modular foundations is a novel development that permits defining new foundations reusing existing ones and interrelating different foundations. In chapter 6, we will exploit foundation morphisms and generative unions to define two new foundations modularly.

Chapter 5

Theory Families and Instantiations

In this chapter we develop our foundation-independent meta-framework TFI for mathematical theories and their translations. TFI is built around our generic foundation layer from chapter 4 and refines it by adding *theory morphisms*. This development is very similar to that of MMT [RK13]. Then, TFI introduces the notion of *theory families* and their *instantiations*. These are novel concepts that characterize TFI.

We introduce the syntax of TFI in section 5.1, and the type system in section 5.2. We end the chapter with a discussion in section 5.3.

	Grammar	Inference System
Language	section 5.1.1	section 5.2.1
Homomorphic Extensions	section 5.1.3	section 5.2.2
Elaboration	section 5.1.3	section 5.2.2

Figure 5.1: Chapter Overview

5.1 Syntax

We will assume an arbitrary fixed foundation $\mathcal{F}$ throughout this chapter and define our concepts relative to that foundation.

5.1.1 Grammar

In TFI we do not distinguish between theories and contexts, and merge these two concepts together. This has the consequence that constants and variables become one unified notion.

Terminology 5.1. We call this unified notion of constants and variables **symbols**.

The syntax of TFI is given by the grammar in figure 5.1.1, which extends the grammar in figure 4.1 from chapter 4. We explain the non-terminal symbols and their productions below.

Theories are formed from the empty theory, denoted as $\cdot$, by adding symbol declarations. More specifically, a theory Σ consists of a list of **declarations** of the form:

- declarations $x : E[\ = E']$ of **expression symbols** x of type E and with optional definition E',

Theories	Σ	::=	$\cdot \mid \Sigma, x : E[= E] \mid \underline{\Sigma, \varphi : \tau = \Phi} \mid \underline{\Sigma, \iota : \Phi}$
Theory morphisms	σ	::=	$\cdot \mid \sigma, x := E \mid \underline{\sigma, \varphi := \Phi} \mid \underline{\sigma, \iota := I}$
Kinds	τ	::=	$\underline{\texttt{theory}} \mid \underline{(x : E)\,\tau}$
Theory Families	Φ	::=	$\underline{\varphi} \mid \underline{\{\Sigma\}} \mid \underline{(x : E)\,\Phi} \mid \underline{\Phi(E)}$
Instances	I	::=	$\underline{\iota} \mid \underline{\{\sigma\}}$
Expressions	E	::=	$s \mid x \mid \underline{I.x} \mid \beta(E\,;\,\Sigma\,;\,E) \mid @(E;E)$

Figure 5.2: TFI Grammar

- declarations $\varphi : \tau = \Phi$ of **theory family symbols** φ of kind τ and with definition Φ and
- declarations $\iota : \Phi$ of **instance symbols** ι of type Φ.

Theory morphisms map symbols of one theory to objects in another theory. More specifically, a theory morphism σ consists of a list of **assignments** of the form:

- $x := E$ that map symbols x to expressions E,
- $\varphi := \Phi$ that map symbols φ to theory families Φ, and
- $\iota := I$ that map symbols ι to instances I.

Note that theory morphisms do not map foundational symbols s. Foundational symbols are automatically mapped to themselves.

Terminology 5.2 (Fragments). Let Σ, Ψ be a theory. We say that Σ, Ψ **includes** Σ or that Σ, Ψ **extends** Σ **with** Ψ. We call Ψ a **theory fragment** of Σ, Ψ.

Analogous to theory fragments, we can talk about fragments of theory morphisms: Let σ, ψ be a theory morphism that maps symbols in Σ, Ψ to objects in Σ'. We say that ψ is a **morphism fragment** of σ, ψ.

We reify theory fragments as objects that may occur in theories themselves. Then *theory families* are a result of performing object-level operations on the reified theory fragments.

More specifically, **theory families** Φ are theory fragments that are parameterized by expression level declarations. Consequently, they are formed by four constructors:

- Reified theory fragments $\{\Psi\}$,
- abstractions $(x : E)\,\Phi$ that parameterize a theory family Φ by a declaration $x : E$,
- applications $\Phi(E)$ that provide an argument E to a theory family Φ, and
- names φ to refer to previously declared theory families.

Definition 5.3. We say that the theory fragment Ψ in Σ, Ψ is **simple** if it does not contain any theory family declarations $\varphi : \tau = \Phi$.

Moreover we say that Ψ is **strict** if it consists of only declarations of the form

$$x : E = E' \qquad \text{and} \qquad i : \Phi$$

such that whenever Φ contains $\{\Psi'\}$ then Ψ' is strict. In that case, we say that Φ is **strict**.

Note that our operations on theory fragments are first-order: Abstraction over theory families is restricted to object level expressions $x : E$ only.

One crucial decision in our language design is the use of **kinds**. We use kinds to classify theory families and to distinguish them from typed expressions. In particular, **kinds** τ classify theory families into *i*) atomic ones $\{\Psi\}$, which are kinded by the base kind `theory`, and *ii*) those that are parameterized by an expression symbol $x : E$, which are kinded by the kind $(x : E)\,\tau$. We will elaborate on the kinds in our type system in section 5.2.

We call theory families that are kinded by `theory` *object-level theories*. Note that these are different than the meta-level theories Σ.

Our approach of using kinds to classify theory families at the object level is in parallel to the typing system object level expressions have and ensures that we do not use expression bindings for theory families.

Analogous to theory fragments, we reify morphism fragments as objects that occur in theories. We call these objects *instances*. The notion of instances goes together with the notion of theory families. Intuitively, instances are objects that arise by instantiating theory families.

More specifically, **instances** I are formed from reified morphism fragments $\{\psi\}$ and names ι. The type system of TFI has a hierarchy that is similar to those in well-known type systems like LF [HHP93]. Instances I are typed by theory families Φ that are kinded by `theory`.

In addition to the OpenMath expressions from grammar 4.1, TFI permits projections $I.x$ of instances to symbols x.

5.1.2 Examples

Now we will give some examples of the concepts we introduced in section 5.1.1. For our examples, we will instantiate $\mathcal{F}$ with the foundation LF from example 4.10.

Theories and Theory Morphisms

Notation 5.4. We will use the following notations to write named theories and theory morphisms:

- $T = \{\Sigma\}$ for a theory named T with body Σ and
- $\mu = \{\sigma\}$ for a theory morphism named μ with body σ.

Furthermore, we will use the following features for convenience and readability:

- infix notation $E_1\, O\, E_2$ for the application $O\, E_1\, E_2$ of binary operators O to arguments E_1 and E_2,
- implicit arguments in symbol declarations,
- omitted types of bound variables.

Such notations can be declared formally in frameworks like MMT [RK13], but we omit the notation language for simplicity.

The following example introduces a well-known formalization of simple type theory in LF:

Example 5.5 (Simple Type Theory). We introduce simple type theory as a TFI/LF-theory named STT.

$$
\begin{array}{l}
STT \;=\; \{ \\
\quad tp \;:\; \texttt{type} \\
\quad tm \;:\; tp \to \texttt{type} \\
\quad \iota \;:\; tp \\
\quad o \;:\; tp \\
\quad \Longrightarrow \;:\; tp \to tp \to tp \\
\quad lam \;:\; tm\, A \to tm\, B \to tm\, (A \Longrightarrow B) \\
\quad app \;:\; tm\, (A \Longrightarrow B) \to tm\, A \to tm\, B \\
\}
\end{array}
$$

Here, we use implicit arguments $A : tp$ and $B : tp$ in the declaration of lam and app.

The following example is a small fragment of the formalization of axiomatic set theory in LF given in [HR11].

Example 5.6 (Set Theory). We introduce axiomatic set theory as an TFI/LF-theory named $Sets$:

$$
\begin{array}{l}
Sets \;=\; \{ \\
\quad set \;:\; \texttt{type} \\
\quad form \;:\; \texttt{type} \\
\quad ded \;:\; form \to \texttt{type} \\
\quad \in \;:\; set \to set \to form \\
\quad Fun \;:\; set \to set \to set \\
\quad lam \;:\; set \to (set \to set) \to set \\
\quad app \;:\; set \to set \to set \\
\quad Bool \;:\; set \\
\}
\end{array}
$$

The following example introduces the interpretation of simple type theory in terms of axiomatic set theory as a TFI/LF-theory morphism:

Example 5.7 (Theory Morphisms). The theory morphism $STT2Sets$ maps the symbols in STT from example 5.5 to objects over the theory $Sets$ from example 5.6.

$$
\begin{array}{l}
STT2Sets \;=\; \{ \\
\quad tp \;:=\; set \\
\quad tm \;:=\; \lambda s : set.\ set \\
\quad o \;:=\; Bool \\
\quad \Longrightarrow \;:=\; \lambda s : set.\ \lambda t : set.\ Fun\, s\, t \\
\quad lam \;:=\; \lambda s : set.\ \lambda t : set.\ lam\, s\, t \\
\quad app \;:=\; \lambda f : set.\ \lambda x : set.\ app\, f\, x \\
\}
\end{array}
$$

Notation 5.8. Given named theories $S = \{\Sigma\}$ and $T = \{\Sigma, \Psi\}$, we will occasionally define T as

$$
\begin{array}{l}
T \;=\; \{ \\
\quad \texttt{include}\ S \\
\quad \Psi \\
\}
\end{array}
$$

and say that T *extends* S *with* Ψ.

Now, we introduce the theory of magma as a theory in TFI/LF:

Example 5.9 (Magma). Consider the TFI/LF-theory STT from example 5.5. We introduce the theory of magma by extending STT with the following declarations and name the resulting theory *PointedMagma*:

$$
\begin{array}{l}
\mathit{PointedMagma} \;=\; \{ \\
\quad \texttt{include}\ \mathit{STT} \\
\quad M \;:\; \mathit{tp} \\
\quad e \;:\; \mathit{tm}\, M \\
\quad \circ \;:\; \mathit{tm}\,(M \Longrightarrow M \Longrightarrow M) \\
\}
\end{array}
$$

Theory Families and Instances

Notation 5.10. We will use the following notations for theory families throughout this thesis: If Γ is a possibly empty list of declarations $x_1 : E_1, \ldots, x_n : E_n$, then we will write

- $(\Gamma)\,\texttt{theory}$ for $(x_1 : E_1) \ldots (x_n : E_n)\,\texttt{theory}$,
- $(\Gamma)\,\{\Psi\}$ for $(x_1 : E_1) \ldots (x_n : E_n)\,\{\Psi\}$,
- $\Phi(x_1, \ldots, x_n)$ or $\Phi\, x_1 \ldots x_n$ for $\Phi(x_1) \ldots (x_n)$.

Example 5.11 (Theory Families). We extend STT from example 5.5 with the following declarations of theory families and refer to this extension as STT^+:

$$
\begin{array}{l}
\mathit{STT}^+ \;=\; \{ \\
\quad \texttt{include}\ \mathit{STT} \\
\quad \mathit{typeDecl} \;:\; \texttt{theory} \;=\; \{x : \mathit{tp}\} \\
\quad \mathit{TypedSym} \;:\; (T : \mathit{tp})\,\texttt{theory} \;=\; (T : \mathit{tp})\,\{x : \mathit{tm}\, T\} \\
\}
\end{array}
$$

typeDecl represents declarations of type constants a in simple type theory. Then, *TypedSym* represents declarations of typed terms $t : T$ in simple type theory for a simple type T.

Now, we extend *Sets* from example 5.6 with the following declarations of theory families and will refer to this extension as Sets^+:

$$
\begin{array}{l}
\mathit{Sets}^+ \;=\; \{ \\
\quad \texttt{include}\ \mathit{Sets} \\
\quad \mathit{setDecl} \;:\; \texttt{theory} \;=\; \{s : \mathit{set}\} \\
\quad \mathit{ax} \;:\; (F : \mathit{form})\,\texttt{theory} \;=\; (F : \mathit{form})\,\{m : \mathit{ded}\, F\} \\
\}
\end{array}
$$

setDecl represents the introduction of individual sets s in set theory. Then, ax represents the introduction of axioms F, using the judgments-as-types approach, as symbols declarations $m : \mathit{ded}\, F$.

Now, we give assignments that map the theory family symbols in STT^+ to theory families over Sets^+.

$$
\begin{array}{lcl}
\mathit{typeDecl} & := & \mathit{setDecl} \\
\mathit{TypedSym} & := & (T : \mathit{set})\,\{x : \mathit{set}, m : \mathit{ded}\, x \in T\}
\end{array}
$$

The first assignment represents that for every type declaration $a : tp$ in simple type theory, a set declaration is introduced.The second assignment represents that for every declaration of a typed constant $t : T$ in simple type theory, a set $x : set$ and an axiom for x are introduced.

Notation 5.12. In the examples we give in the remainder of this document, we will omit writing the kind τ in the declarations of theory family symbols $\varphi : \tau = \Phi$ and will simply write $\varphi = \Phi$ instead as τ can be easily inferred from Φ.

Recall that in example 5.9, we introduced the TFI/LF-theory *PointedMagma* as an example of an STT-theory. *PointedMagma* extended STT^+ with declarations of expression symbols. Alternatively, we can give the theory of pointed magma as the following TFI/LF-theory that extends STT with instance declarations:

Example 5.13 (Instances). The following declarations introduce instances of the theory families in example 5.11.

$$
\begin{array}{l}
PointedMagma' \ = \ \{ \\
\quad \texttt{include}\ STT^+ \\
\quad M' \ : \ typeDecl \\
\quad e' \ : \ TypedSym(M'.x\,) \\
\quad \circ' \ : \ TypedSym(M'.x \implies M'.x \implies M'.x\,) \\
\}
\end{array}
$$

$M'.x$ is the projection of the declaration $x : tp$ in *typeDecl* out of M'.

Notation 5.14. In the remainder of this document, a theory family may not necessarily give a concrete name to a declaration in the theory family, but rather use _ like in

$$\varphi : \tau = \{\_ : A = B\}$$

as the name. In that case, whenever there is an instance instantiating φ, e.g., $\iota : \varphi$, then we will write the projection $\iota.\_$ of ι simple as ι for the sake of readability.

5.1.3 Meta-Level Definitions

In this section, we develop three main definitions by induction on the grammar 5.1.1: *i) homomorphic extensions of theory morphisms*, *ii) transformation along theory morphisms*, and *iii) elaboration of theories*.

Notation 5.15. Let σ be a theory morphism. We denote the theory morphism $\sigma, x_1 := x_1, \ldots, x_n := x_n$ by $\sigma^{x_1,\ldots,x_n}$.

Remark 5.16. The function definitions in this section do not check whether the input is a well-formed object. We define well-formed TFI objects in section 5.2 and use these definitions for only well-formed input arguments.

Homomorphic Extensions Our definition of homomorphic extensions maps every object of the TFI grammar from one theory to another. Substitution is subsumed by this definition.

Definition 5.17 (Homomorphic Extension of Theory Morphisms)**.** Let σ be a theory morphism. The **homomorphic extension $\overline{\sigma}$ of σ** is the following family of mappings defined for each non-terminal in our grammar 5.1.1:

- $\overline{\sigma}(\Phi)$ is defined by:

$$\begin{array}{lcll} \overline{\sigma}(\varphi) & = & \Phi & \text{for } \varphi := \Phi \text{ in } \sigma \\ \overline{\sigma}((x : E)\,\Phi) & = & (x : \overline{\sigma}(E))\,\overline{\sigma^x}(\Phi) & \\ \overline{\sigma}(\Phi(E)) & = & \overline{\sigma}(\Phi)(\overline{\sigma}(E)) & \\ \overline{\sigma}(\{\Psi\}) & = & \{\overline{\sigma}(\Psi)\} & \end{array}$$

- $\overline{\sigma}(\tau)$ is defined by:

$$\begin{array}{lcl} \overline{\sigma}(\texttt{theory}) & = & \texttt{theory} \\ \overline{\sigma}((x : E)\,\tau) & = & (x : \overline{\sigma}(E))\,\overline{\sigma^x}(\tau) \end{array}$$

- $\overline{\sigma}(I)$ is defined by:

$$\begin{array}{lcll} \overline{\sigma}(\iota) & = & I & \text{for } \iota := I \text{ in } \sigma \\ \overline{\sigma}(\{\psi\}) & = & \{\overline{\sigma}(\psi)\} & \end{array}$$

- $\overline{\sigma}(E)$ is defined by.

$$\begin{array}{lcll} \overline{\sigma}(s) & = & s & \\ \overline{\sigma}(x) & = & E & \text{for } x := E \text{ in } \sigma \\ \overline{\sigma}(I.x\,) & = & \overline{\sigma}(I).x & \\ \overline{\sigma}(@(E; E')) & = & @(\overline{\sigma}(E); \overline{\sigma}(E')) & \\ \overline{\sigma}(\beta(E\,;\,\Psi\,;\,E')) & = & \beta(\overline{\sigma}(E)\,;\,\overline{\sigma}(\Psi)\,;\,\overline{\sigma^{\Psi}}(E')) & \end{array}$$

  where $\overline{\sigma^{\Psi}}$ denotes the morphism $\sigma^{x_1,\ldots,x_n}$ for $x_1, \ldots, x_n \in dom(\Psi)$.

- $\overline{\sigma}(\Psi)$ is defined by:

$$\begin{array}{lcl} \overline{\sigma}(\cdot) & = & \cdot \\ \overline{\sigma}(\Psi, x : E[= E']) & = & \overline{\sigma}(\Psi),\, x : \overline{\sigma * \Psi}(E)[= \overline{\sigma * \Psi}(E')] \\ \overline{\sigma}(\Psi, \iota : \Phi) & = & \overline{\sigma}(\Psi),\, \iota : \overline{\sigma * \Psi}(\Phi) \\ \overline{\sigma}(\Psi, \varphi : \tau = \Phi) & = & \overline{\sigma}(\Psi),\, \varphi : \overline{\sigma * \Psi}(\tau) = \overline{\sigma * \Psi}(\Phi) \end{array}$$

  where $\sigma * \Psi$ is defined in definition 5.18.

- $\overline{\sigma}(\psi)$ is defined by:

$$\begin{array}{lcl} \overline{\sigma}(\cdot) & = & \cdot \\ \overline{\sigma}(\psi, x := E) & = & \overline{\sigma}(\psi),\, x := \overline{\sigma}(E) \\ \overline{\sigma}(\psi, \iota := I) & = & \overline{\sigma}(\psi),\, \iota := \overline{\sigma}(I) \\ \overline{\sigma}(\psi, \varphi := \Phi) & = & \overline{\sigma}(\psi),\, \varphi := \overline{\sigma}(\Phi) \end{array}$$

Intuitively, this definition captures the following diagram:

$$\begin{array}{ccc} \Sigma & \xrightarrow{\quad\sigma\quad} & \Sigma' \\ \big\downarrow & & \big\downarrow \\ \Sigma, \Psi & & \Sigma', \overline{\sigma}(\Psi) \end{array}$$

Transformation along Morphisms Now, we define a theory morphism to complete the diagram above:

Definition 5.18 (Transformation along Morphisms)**.** Let Σ, Ψ and Σ' be theories, and σ be a theory morphism that maps symbols in Σ to objects over Σ'. The **transformation for Ψ along σ** is the theory morphism, denoted as $\sigma * \Psi$, that maps symbols in Σ, Ψ to objects in $\Sigma', \overline{\sigma}(\Psi)$ and is defined recursively as follows:

$$\begin{array}{lcl} \sigma * \cdot & = & \sigma \\ \sigma * (\Psi, x : E[= E']) & = & (\sigma * \Psi), x := x \\ \sigma * (\Psi, i : \Phi) & = & (\sigma * \Psi), i := i \\ \sigma * (\Psi, \varphi : \tau = \Phi) & = & (\sigma * \Psi), \varphi := \varphi \end{array}$$

definition 5.17 and definition 5.18 together capture the following commutative diagram:

$$\begin{array}{ccc} \Sigma & \xrightarrow{\sigma} & \Sigma' \\ \downarrow & & \downarrow \\ \Sigma, \Psi & \xrightarrow{\sigma * \Psi} & \Sigma', \overline{\sigma}(\Psi) \end{array}$$

Example 5.19. (Continuing example 5.9). We apply the theory morphism *STT2Sets* from example 5.7 to the theory *PointedMagma* in example 5.9 and get the following theory:

$$\overline{STT2Sets}(PointedMagma) = \{M : set, e : set, \circ : set\}$$

Furthermore, we have the following theory morphism as illustrated in the diagram below:

$$\begin{array}{ccc} STT & \xrightarrow{STT2Sets} & Sets \\ \downarrow & & \downarrow \\ STT, PointedMagma & \xrightarrow{STT2Sets * PointedMagma} & Sets, \overline{STT2Sets}(PointedMagma) \end{array}$$

$$STT2Sets * PointedMagma = \{M := M, e := e, \circ := \circ\}$$

Elaboration Now we will talk about the elaboration of TFI/$\mathcal{F}$ theories. The idea is that every instance declaration $\iota : \Phi$ in a TFI/$\mathcal{F}$-theory is elaborated to a list of expression symbol declarations $x_i : E_i[D_i]$ that Φ consists of in its body.

Definition 5.20 (Elaboration)**.** We define the **elaboration** of a theory Σ recursively as follows:

$$\begin{array}{lcl} \mathcal{E}(\cdot) & = & \cdot \\ \mathcal{E}(\ \Sigma, x : E = E'\ ) & = & \mathcal{E}(\Sigma), x : E = E' \\ \mathcal{E}(\ \Sigma, \iota : \Phi\ ) & = & \mathcal{E}(\Sigma), \iota.x_1 : \overline{\gamma}(E_1)[= \overline{\gamma}(D_1)], \ldots, \iota.x_n : \overline{\gamma}(E_n)[= \overline{\gamma}(D_n)] \\ & & \text{for } \Phi{\downarrow_{\delta\beta}} = \{\Psi\} \text{ and } \mathcal{E}(\Psi) = \{x_1 : E_1[= D_1], \ldots, x_n : E_n[= D_n]\} \\ & & \text{and } \gamma = id_\Sigma, x_1 := i.x_1, \ldots, x_n := i.x_n \\ \mathcal{E}(\ \Sigma, \varphi : \tau = \Phi\ ) & = & \mathcal{E}(\Sigma), \varphi : \tau = \Phi \end{array}$$

where $\Phi{\downarrow_{\delta\beta}}$ is introduced in definition 5.27.

Note that the names x_i that result from the elaboration of theories are qualified names of the form $\iota_1.\ \ldots\ .\iota_m.y_i$ for some number m for each i.

Example 5.21. The elaboration $\mathcal{E}(\mathit{PointedMagma}')$ of the TFI/LF-theory $\mathit{PointedMagma}'$ in example 5.13 is the following theory:

$$\begin{array}{lcl} M'.x & : & tp \\ e'.x & : & tm\, M'.x \\ \circ'.x & : & tm\,(M'.x \implies M'.x \implies M'.x\,) \end{array}$$

Note that the resulting theory is isomorphic to *PointedMagma* in example 5.9 up to renaming of symbols.

Now we define some auxiliary functions that we will use in the remainder of this chapter:

Definition 5.22 (Identity Morphism)**.** Let Σ be a theory. We define the **identity theory morphism** id_Σ on Σ by induction on the declarations in Σ as follows:

$$\begin{array}{lcl} id_{\cdot} & = & \cdot \\ id_{\Sigma,x:E[=E']} & = & id_\Sigma, x := x \\ id_{\Sigma,\iota:\Phi} & = & id_\Sigma, \iota := \iota \\ id_{\Sigma,\varphi:\tau=\Phi} & = & id_\Sigma, \varphi := \varphi \end{array}$$

Definition 5.23 (Morphism Composition)**.** Let σ_1 be a theory morphism from Σ_1 to Σ and σ_2 be a theory morphism from Σ to Σ_2. The **composition** of σ_1 and σ_2, denoted as $\sigma_1\,;\sigma_2$, is a theory morphism from Σ_1 to Σ_2 defined as

$$\sigma_1\,;\sigma_2 \;=\; \overline{\sigma_2}(\sigma_1).$$

As we are allowed to designate expression symbols in theory families as parameters, it is natural to substitute values for them. Substitution can be defined as a special theory morphism that maps the designated symbols to the parameters we want to substitute them with, and every other symbol to itself. Definition 5.24 introduces substitution in expressions, theory families, instances and kinds.

Definition 5.24 (Substitution)**.** Let Σ be a theory and $M ::= E \mid I \mid \Phi \mid \tau$ be over Σ. We define the **substitution of $x_1, \ldots, x_n$ in M with the expressions $E_1, \ldots, E_n$**, denoted as $[E_1/x_1, \ldots, E_n/x_n]\,M$, as follows:

$$[E_1/x_1, \ldots, E_n/x_n]\,M = \overline{(id_\Sigma,\, x_1 := E_1, \ldots,\, x_n := E_n)}(M)$$

Definition 5.25 (β-Normal Form)**.** Given a fixed theory Σ, we define the **β-normal form** $\Phi{\downarrow_\beta}$ of a theory family Φ as follows:

$$\begin{array}{lcll} \{\Psi\}{\downarrow_\beta} & = & \{\Psi\} & \\ \Phi(E){\downarrow_\beta} & = & \begin{cases} ([E/x]\,\Phi'){\downarrow_\beta} & \text{if } \Phi{\downarrow_\beta} = (x : E')\,\Phi' \\ \Phi{\downarrow_\beta}\,(E) & \text{otherwise} \end{cases} & \\ ((x : E)\,\Phi){\downarrow_\beta} & = & (x : E)\,\Phi{\downarrow_\beta} & \\ \varphi{\downarrow_\beta} & = & \varphi & \end{array}$$

Definition 5.26 (δ-Normal Form)**.** Let theory Σ be a theory. We define the **δ-normal form** $\Phi{\downarrow_\delta}$ of a theory family Φ as follows:

$$\begin{array}{lcll} \{\Psi\}{\downarrow_\delta} & = & \{\Psi\} & \\ \Phi(E){\downarrow_\delta} & = & \Phi{\downarrow_\delta}\,(E) & \\ ((x : E)\,\Phi){\downarrow_\delta} & = & (x : E)\,\Phi{\downarrow_\delta} & \\ \varphi{\downarrow_\delta} & = & \Phi'{\downarrow_\delta} & \text{for } \varphi : \tau = \Phi' \text{ in } \Sigma \end{array}$$

Definition 5.27 ($\delta\beta$-Normal Form)**.** Given a fixed theory Σ, we define the **$\delta\beta$-normal form** $\Phi{\downarrow_{\delta\beta}}$ of a theory family Φ as

$$\Phi{\downarrow_{\delta\beta}} \;=\; (\Phi{\downarrow_\delta}){\downarrow_\beta}$$

Both the β- and the δ-normalizations trivially terminate. Hence, the $\delta\beta$-normalization terminates. Moreover, the definitions of each three normal forms are deterministic. Hence, they are confluent.

More specifically, the $\delta\beta$-normal forms $\Phi{\downarrow_{\delta\beta}}$ have following shape:

$$(x_1 : E_1) \ldots (x_n : E_n)\{\Psi\} \tag{5.1}$$

for some number n.

The β-normal forms $\Phi{\downarrow_\beta}$ have the following shape in addition to 5.1:

$$(x_1 : E_1) \ldots (x_n : E_n)\varphi \tag{5.2}$$

for some number n.

Definition 5.28 (Domain of a Theory)**.** We extend the definition of the **domain** of a theory Σ from definition 4.4:

$$\begin{array}{lcl} dom(\cdot) & = & \emptyset \\ dom(\Sigma, x : E[\,= E']) & = & dom(\Sigma) \cup \{x\} \\ dom(\Sigma, \iota : \Phi) & = & dom(\Sigma) \cup \{\iota\} \\ dom(\Sigma, \varphi : \tau = \Phi) & = & dom(\Sigma) \cup \{\varphi\} \end{array}$$

5.2 Type System

In this section, we present the type system of TFI.

5.2.1 Judgments and Rules

In figure 5.3 we give the judgments TFI uses in addition to the judgments for foundations in figure 4.2.

We use the judgment $\vdash \Sigma\ \mathit{Theory}$ to denote well-formed theories Σ and the judgment $\vdash \sigma\ :\ \Sigma \to \Sigma'$ to denote well-formed theory morphisms σ from Σ to Σ'.

We use the judgment $\Sigma \vdash \tau\ \mathit{Kind}$ to denote well-formed kinds τ. Well-formed theory families are kinded, and we use the judgment $\Sigma \vdash \Phi : \tau$ to denote well-formed theory families Φ of kind τ. Both Φ and τ may contain symbols from Σ.

We use the judgment $\Sigma \vdash I : \Phi$ to denote well-formed instances I that are typed theory families Φ. Both I and Φ may contain expressions that depend on symbols in Σ.

$\vdash \Sigma\ \mathit{Theory}$	Σ is a well-formed theory.
$\vdash \sigma\ :\ \Sigma \to \Sigma'$	σ is a well-formed theory morphism from Σ to Σ'.
$\Sigma \vdash \tau\ \mathit{Kind}$	τ is a well-formed kind over Σ.
$\Sigma \vdash \Phi : \tau$	Φ is a well-formed theory family of kind τ over Σ.
$\Sigma \vdash I : \Phi$	I is a well-formed instance of family Φ over Σ.
$\Sigma \vdash \Phi \leq_\tau \Phi'$	Φ' refines Φ of kind τ over Σ.

Figure 5.3: Main Judgments

Thus, for every non-terminal in our grammar in figure 5.1.1, we have one judgment for the well-formedness of the objects it produces. Similarly, for each judgment in figure 5.3, we have one inference rule that for each of its production.

Finally, we use the judgment $\Sigma \vdash \Phi \leq_\tau \Phi'$ to denote a **refinement** relation between two theory families, which intuitively capture the meta-level inclusion relation between two theories at the object-level.

Now we will define each judgment in figure 5.3:

Well-Formed Theories figure 5.4 shows the formation rules that define the judgment $\vdash \Sigma\ \mathit{Theory}$ for **well-formed theories**. Theories are formed recursively by adding symbol declarations starting from the empty theory. All symbol declarations in a theory Σ must use different names.

$$\frac{}{\vdash \cdot\ \mathit{Theory}}\ \mathit{emptyTheory}$$

$$\frac{\vdash \Sigma\ \mathit{Theory} \quad \Sigma \vdash E_1\ \mathit{Inhabitable} \quad [\Sigma \vdash E_2 : E_1] \quad x \notin \mathit{dom}(\Sigma)}{\vdash \Sigma, x : E_1[\,= E_2]\ \ \mathit{Theory}}\ \mathit{addConstant}$$

$$\frac{\vdash \Sigma\ \mathit{Theory} \quad \Sigma \vdash \Phi : \texttt{theory} \quad \iota \notin \mathit{dom}(\Sigma)}{\vdash \Sigma, \iota : \Phi\ \ \mathit{Theory}}\ \mathit{addInstance}$$

$$\frac{\vdash \Sigma\ \mathit{Theory} \quad \Sigma \vdash \Phi : \tau \quad \varphi \notin \mathit{dom}(\Sigma)}{\vdash \Sigma, \varphi : \tau = \Phi\ \ \mathit{Theory}}\ \mathit{addTheoryFamily}$$

Figure 5.4: Theory Formation Rules

In figure 5.4 we have one rule for adding each kind of symbol declaration in a theory: term symbols x are typed by well-formed term expressions E that are inhabitable, instance symbols i are typed by well-formed theory families Φ, and theory family symbols φ are kinded. If symbol declarations have a definiens, then the definiens must have the same type as the symbol. Recall that theory family declarations have mandatory definiens.

Well-Formed Theory Morphisms figure 5.5 shows the formation rules that define the judgment $\vdash \sigma : \Sigma \to \Sigma'$ for **well-formed theory morphisms**. Theory morphisms $\sigma : \Sigma \to \Sigma'$ into a fixed well-formed theory Σ' are formed recursively by adding symbol assignments starting from the empty theory morphism $\cdot$. In particular, the formation of $\sigma : \Sigma \to \Sigma'$ is structurally analogous to the formation of its domain Σ: Whenever a new symbol declaration is added to Σ, a corresponding mapping of that symbol is added to σ.

$$\frac{\vdash \Sigma\ \mathit{Theory}}{\vdash \cdot : \cdot \to \Sigma}\ \mathit{emptyMorph}$$

$$\frac{\vdash \sigma : \Sigma \to \Sigma' \quad \Sigma' \vdash E : \overline{\sigma}(E') \quad [\Sigma \vdash E \doteq \overline{\sigma}(E')]}{\vdash \sigma, x := E : \Sigma, x : E'[= E''] \to \Sigma'}\ \mathit{mapConstant}$$

$$\frac{\vdash \sigma : \Sigma \to \Sigma' \quad \Sigma' \vdash I : \overline{\sigma}(\Phi)}{\vdash \sigma, \iota := I : \Sigma, \iota : \Phi \to \Sigma'}\ \mathit{mapInstance}$$

$$\frac{\vdash \sigma : \Sigma \to \Sigma' \quad \Sigma' \vdash \Phi : \overline{\sigma}(\tau) \quad \Sigma' \vdash \overline{\sigma}(\Phi'{\downarrow_\delta}) \leq_{\overline{\sigma}(\tau)} \Phi}{\vdash \sigma, \varphi := \Phi : \Sigma, \varphi : \tau = \Phi' \to \Sigma'}\ \mathit{mapTheoryFamily}$$

Figure 5.5: Theory Morphism Formation Rules

Moreover, every symbol in the domain of σ is mapped according to its type. For instance, the symbol $x : E'$ is mapped to E such that its type E' is preserved, i.e., $E : \overline{\sigma}(E')$. This is analogous for the case of instance symbols and theory family symbols. If a symbol declaration provides a definiens as well, then the symbol is mapped to the translation of the definiens with the exception of theory family symbols. In the case of definiens E'', the assignment of x to E is predetermined by $\overline{\sigma}(E'')$.

Symbols are mapped to type preserving expressions. Defined symbols are mapped to expressions that are provably equal to the translation of the definiens.

In the case of the mapping of $\varphi : \tau = \Phi'$, the theory family Φ assigned to φ is allowed to include the translation of the definiens Φ'. This unusual relaxation is very crucial to achieve theory translations in section 7.2. This implies that the definitional equality of theory families along theory morphisms is not necessarily preserved. This is also the reason why the main theorem of this chapter (theorem 5.40) is nontrivial.

Well-Formed Kinds figure 5.6 shows the formation rules that define the judgment $\Sigma \vdash \tau\ \mathit{Kind}$ for **well-formed kinds**. The base kind `theory` is a well-formed kind. The formation of dependent kinds $(x : E)\,\tau$ is as usual.

Well-Formed Theory Families figure 5.7 shows the kinding rules that define the judgment $\Sigma \vdash \Phi : \tau$ for **well-formed theory families**. We use kinds, instead of types, to establish the well-formed theory families so that we can distinguish theory families from typed expressions. Atomic theory families $\{\Psi\}$ are kinded by the base kind `theory` and

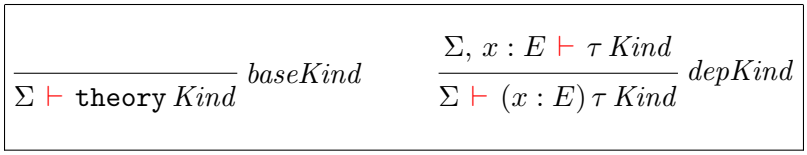


Figure 5.6: Kind Formation Rules

formed from theory fragments Ψ that extend theories Σ. Therefore, $\{\Psi\}$ is well-formed if the extension Σ, Ψ is well-formed. Note that we have one restriction on the formation of Ψ: Ψ must be simple and therefore does not contain theory family declarations. This excludes the formation of nested theory families. The remaining rules are standard typing rules for abstraction, application and look-up.

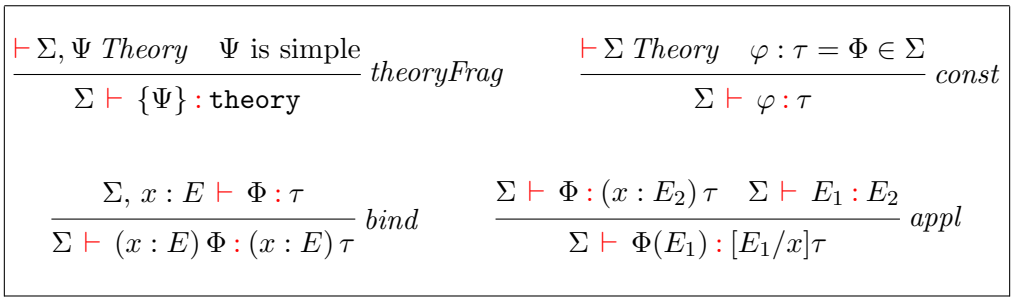


Figure 5.7: Kinding Rules for Theory Families

Well-Formed Instances Figure 5.8 show the typing rules that define the judgment $\Sigma \vdash I : \Phi$ for **well-formed instances**.

Instances are typed by theories $\Phi : \texttt{theory}$. The rule *morphFrag* is the introduction rule for $\{\Psi\}$: $\{\psi\}$ is typed by $\{\Psi\}$ if the theory morphism id_Σ, ψ is a well-formed theory morphism from Σ, Ψ to Σ.

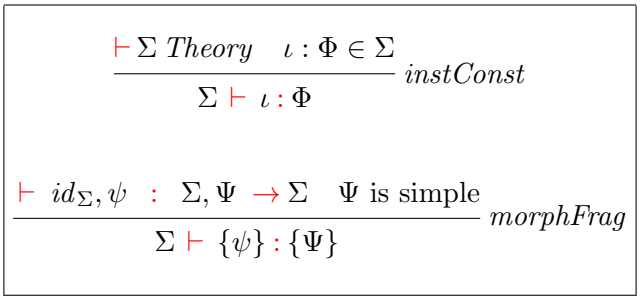


Figure 5.8: Typing Rules for Instances

Refinement of Theory Families Just like theory families internalize meta-level theories in the object-level, the judgment $\Sigma \vdash \Phi \leq_\tau \Phi'$ internalizes the meta-level theory inclusion relation in the object-level. Intuitively, $\Sigma \vdash \Phi \leq_\tau \Phi'$ means that the declarations in Φ are included in those of Φ'. We call this judgment the **refinement** judgment and say that Φ' refines Φ at kind τ.

Before we formally define the refinement judgment, we need to recall the well-established notion of *theory inclusion*:

Definition 5.29 (Inclusion Morphism)**.** Let $\vdash \Sigma$ *Theory* and $\vdash \Sigma'$ *Theory*. We say that Σ' **includes** Σ, denoted as $\vdash \Sigma \hookrightarrow \Sigma'$, if and only if $\vdash id_\Sigma : \Sigma \to \Sigma'$. In this case, we call $id_\Sigma : \Sigma \to \Sigma'$ the **inclusion morphism** from Σ to Σ'.

We define the refinement judgment by the rules given in figure 5.9. In particular, we have one rule for each kind of theory families. Note that in rule *parTheoryInc*, Φ' is a refinement of Φ if $\Phi'(x)$ refines $\Phi(x)$ for all values of $x : E$. This is similar to functional extensionality.

$$\frac{\vdash \Sigma, \Psi \hookrightarrow \Sigma, \Psi' \quad \Phi{\downarrow_{\delta\beta}} = \{\Psi\} \quad \Phi'{\downarrow_{\delta\beta}} = \{\Psi'\}}{\Sigma \vdash \Phi \leq_{\texttt{theory}} \Phi'}\ \mathit{theoryFragInc}$$

$$\frac{\Sigma, x : E \vdash \Phi(x) \leq_\tau \Phi'(x)}{\Sigma \vdash \Phi \leq_{(x:E)\,\tau} \Phi'}\ \mathit{parTheoryInc}$$

Figure 5.9: Theory Family Refinement

Well-Formed Expressions The type system of TFI is generic. Expressions come from a specific foundation together with a type system that govern their well-formedness: The judgment $\Sigma \vdash E : E'$ is with respect to a foundation $\mathcal{F}$. TFI introduces typing rules only for expression symbols x and $I.x$, and inherits from the foundation $\mathcal{F}$ the typing rules for all other expressions.

Fig. 5.10 shows the typing rules TFI has for the judgment $\Sigma \vdash E : E'$ for any foundation.

$$\frac{\vdash \Sigma\ \mathit{Theory} \quad x : E[= E'] \in \Sigma}{\Sigma \vdash x : E}\ \mathit{constType}$$

$$\frac{\Sigma \vdash I : \Phi \quad \Phi{\downarrow_{\delta\beta}} = \{\Psi\} \quad \mathcal{E}(\Psi) = \Psi_0, x : E[= E'], \Psi_1 \quad \mathit{dom}(\Psi_0) = \{y_1, \ldots, y_n\}}{\Sigma \vdash I.x : [I.y_1/y_1, \ldots, I.y_n/y_n]\,E}\ \mathit{elabType}$$

Figure 5.10: Typing Rules for Expressions

Equality of Expressions In addition to the equality rules in chapter 4.2, we have two more rules for the equality of expressions in figure 5.11.

5.2.2 Preservation of Judgments

In this section, we establish the main result of this chapter: We show that the homomorphic extension of theory morphisms preserves all judgments. This is non-trivial due to the rule *mapTheoryFamily* in figure 5.5. Moreover, we show that theory elaboration preserves the well-formedness of theories.

$$\frac{\vdash \Sigma\ \mathit{Theory} \quad x : E = E' \in \Sigma}{\Sigma \vdash x \doteq E'}\ \mathit{constDef}$$

$$\frac{\Sigma \vdash I : \Phi \quad \Phi{\downarrow_{\delta\beta}} = \{\Psi\} \quad \mathcal{E}(\Psi) = \Psi_0, x : E = E', \Psi_1 \quad \mathit{dom}(\Psi_0) = \{y_1, \ldots, y_n\}}{\Sigma \vdash I.x \doteq [I.y_1\,/y_1, \ldots, I.y_n\,/y_n]\,E'}\ \mathit{elabDef}$$

Figure 5.11: Equality Judgments for Expression Symbols

First we will introduce a couple of lemmas that will be used in our main theorem.

Normalization Lemmas We have two lemmas on the δ- and β-normal forms:
The following lemma introduces the subject reduction property for theory families.

Lemma 5.30 (Subject Reduction)**.** *If we have* $\Sigma \vdash \Phi : \tau$*, then we have* $\Sigma \vdash \Phi{\downarrow_{\delta\beta}} : \tau$*.*

Proof. By definition 5.27 of $\delta\beta$-form, it suffices to show that

1. if we have $\Sigma \vdash \Phi : \tau$, then we have $\Sigma \vdash \Phi{\downarrow_{\delta}} : \tau$, and
2. if we have $\Sigma \vdash \Phi : \tau$, then we have $\Sigma \vdash \Phi{\downarrow_{\beta}} : \tau$.

In both cases, the proof proceeds by a straightforward induction on the derivation of $\Sigma \vdash \Phi : \tau$. □

The following lemma guarantees that the $\delta\beta$-form of any well-formed atomic theory family has the form of $\{\Psi\}$ for some Ψ.

Lemma 5.31 (Normal Forms)**.** *If we have* $\Sigma \vdash \Phi : \texttt{theory}$*, then* $\Phi{\downarrow_{\delta\beta}} = \{\Psi\}$ *for some* Ψ*.*

Proof. Assume that $\Sigma \vdash \Phi : \texttt{theory}$. We know that by definition 5.26, the δ-normal form $\Phi{\downarrow_{\delta}}$ replaces all occurrences of symbols φ in Φ with their respective definiens, and by definition 5.25 $\Phi{\downarrow_{\delta\beta}}$ has the form of $(x_1 : E_1) \ldots (x_n : E_n)\{\Psi\}$ for some Ψ. Since Φ has kind `theory`, by lemma 5.30, $(x_1 : E_1) \ldots (x_n : E_n)\{\Psi\}$ has kind `theory`, which is only possible if $n = 0$ by inspection of our inference rules. □

Morphism Lemmas Now we show properties about theory morphisms from section 5.1.3. The identity morphism on Σ from definition 5.22 is indeed a well-formed theory morphism from Σ to itself:

Lemma 5.32 (Identity Morphism)**.** *Assume* $\vdash \Sigma\ \mathit{Theory}$*. Then we have* $\vdash id_\Sigma : \Sigma \to \Sigma$*.*

Proof. The proof proceeds by a straightforward induction on the formation of Σ. □

To prove that the composition of two well-formed morphisms is well-formed, we require the preservation of the typing judgments along theory morphisms, which we show in theorem 5.40.

The following lemma introduces the homomorphic extension of morphism composition in terms of the individual morphisms that are composed.

Lemma 5.33 (Homomorphic Extension of Morphism Composition)**.** *Assume* $\vdash \sigma_1 : \Sigma_1 \to \Sigma$ *and* $\vdash \sigma_2 : \Sigma \to \Sigma_2$*. Then we have*

$$\overline{\sigma_1 ; \sigma_2}(M) = \overline{\sigma_2}(\overline{\sigma_1}(M))$$

for $M ::= E \mid I \mid \tau \mid \Phi \mid \Psi \mid \psi$*.*

Proof. The proof proceeds by a straightforward definition expansion of definition 5.17 of $\overline{\sigma}$ and definition 5.23 of morphism composition. □

In particular, the morphisms introduced in notation 5.15 are indeed well-formed:

Lemma 5.34. *If* $\vdash \sigma : \Sigma \to \Sigma'$*, then* $\vdash \sigma^x : \Sigma, x : E[= E'] \to \Sigma', x : \overline{\sigma}(E)[= \overline{\sigma}(E')]$*.*

Proof. The proof proceeds by the rule *mapConstant*. □

We have the following lemma for applying the homomorphic extension $\overline{\sigma}$ of a theory morphism $\sigma : \Sigma \to \Sigma'$ to a substitution of an expression E, an instance I or a theory family Φ.

Lemma 5.35 (Substitution Lemma)**.** *If we have* $\vdash \Sigma, \Psi$ *Theory and* $\vdash \sigma : \Sigma \to \Sigma'$*, then we have:*

$$\overline{\sigma}([E_1/x_1, \ldots, E_n/x_n]\, M) = [\overline{\sigma}(E_1)/x_1, \ldots, \overline{\sigma}(E_n)/x_n]\, \overline{\sigma^{x_1,\ldots,x_n}}(M)$$

for E_i *over* Σ *and* $M ::= E \mid \Phi \mid I \mid \tau$ *over* Σ, Ψ *where* $dom(\Psi) = \{x_1, \ldots, x_n\}$*.*

Proof. Starting from the left-hand side of the equality, we have:

$$\overline{\sigma}([E_1/x_1, \ldots, E_n/x_n]\, M) \quad = \quad \overline{id_\Sigma,\, x_1 := E_1, \ldots\, x_n := E_n\, ; \sigma}(M) \tag{5.3}$$

Starting from the right-hand side of the equality, we have:

$$[\overline{\sigma}(E_1)/x_1, \ldots, \overline{\sigma}(E_n)/x_n]\, \overline{\sigma^{x_1,\ldots,x_n}}(M) \quad =$$

$$\overline{\sigma,\, x_1 := x_1, \ldots, x_n := x_n\, ; id_{\Sigma'},\, x_i := \overline{\sigma}(E_i)}(M) \tag{5.4}$$

for $1 \leq i \leq n$.

Then, it suffices to show that the morphisms in 5.3 and in 5.4 are equal for $1 \leq i \leq n$, which is straightforward to show by checking it for each declaration in Σ, Ψ. □

Lemma 5.36 (Commutativity)**.** *Assume* $\vdash id_\Sigma, \psi : \Sigma, \Psi \to \Sigma$*, where* Ψ *is simple and* $\vdash \sigma : \Sigma \to \Sigma'$*. Then the following diagram commutes:*

$$\begin{array}{ccc} \Sigma, \Psi & \xrightarrow{\quad id_\Sigma, \psi \quad} & \Sigma \\ {\scriptstyle \sigma * \Psi}\downarrow & & \downarrow{\scriptstyle \sigma} \\ \Sigma', \overline{\sigma}(\Psi) & \xrightarrow{\quad id_{\Sigma'}, \overline{\sigma}(\psi) \quad} & \Sigma' \end{array}$$

Consequently, we have

$$\overline{\sigma * \Psi}(\overline{id_\Sigma, \psi}(M)) = \overline{id_{\Sigma'}, \overline{\sigma}(\psi)}(\overline{\sigma * \Psi}(M))$$

for $M ::= E \mid \Phi \mid I \mid \tau$ *over* Σ, Ψ*.*

Proof. We show that $\sigma * \Psi\, ; (id_{\Sigma'}, \overline{\sigma}(\psi)) = (id_\Sigma, \psi)\, ; \sigma$ holds by a straightforward induction on the list of declarations in Σ, Ψ and by the definition of $\sigma * \Psi$ and $\overline{\sigma}$. □

Refinement Lemmas Now, we state some properties about our judgment $\Sigma \vdash \Phi \leq \Phi'$ as a lemma and show that it is pre-order relation. This justifies our $\leq$ notation.

Lemma 5.37 (Pre-order)**.** *The following rules are admissible.*

$$\frac{\Sigma \vdash \Phi : \tau}{\Sigma \vdash \Phi \leq \Phi}\ \textit{reflexive}$$

$$\frac{\Sigma \vdash \Phi_1 \leq \Phi_2 \quad \Sigma \vdash \Phi_2 \leq \Phi_3}{\Sigma \vdash \Phi_1 \leq \Phi_3}\ \textit{transitive}$$

$$\frac{\Sigma, x : A \vdash \Phi_1 \leq \Phi_2}{\Sigma \vdash [E/x]\Phi_1 \leq [E/x]\Phi_2}\ \textit{substClosure}$$

Proof. The proofs proceed by a straightforward induction on the derivation of the premises of each rule. □

Lemma 5.38 ($\delta\beta$-Form)**.** *We have* $\Sigma \vdash \Phi \leq \Phi'$ *if and only if* $\Sigma \vdash \Phi{\downarrow_{\delta\beta}} \leq \Phi'{\downarrow_{\delta\beta}}$.

Proof. The proof proceeds by lemma 5.37 and the definition of $\delta\beta$-form. □

Elaboration Lemmas

Lemma 5.39. *If* $\vdash \Sigma \hookrightarrow \Sigma'$, *then* $\vdash \mathcal{E}(\Sigma) \hookrightarrow \mathcal{E}(\Sigma')$.

Proof. The proof follows immediately from the definition of elaboration. □

Preservation along Morphism Now we can present our main theorem of this chapter:

Theorem 5.40 (Preservation of Judgments)**.** *For an arbitrary foundation* $\mathcal{F}$, *if* $dom(\Sigma)$ *and* $dom(\Sigma')$ *are disjoint, then the following rules are admissible:*

$$\frac{\vdash \sigma \ :\ \Sigma \to \Sigma' \quad \Sigma \vdash E : E'}{\Sigma' \vdash \overline{\sigma}(E) : \overline{\sigma}(E')}\ (1a) \qquad \frac{\vdash \sigma \ :\ \Sigma \to \Sigma' \quad \Sigma \vdash E \doteq E'}{\Sigma' \vdash \overline{\sigma}(E) \doteq \overline{\sigma}(E')}\ (1b)$$

$$\frac{\vdash \sigma \ :\ \Sigma \to \Sigma' \quad \Sigma \vdash E\ \textit{Inhabitable}}{\Sigma' \vdash \overline{\sigma}(E)\ \textit{Inhabitable}}\ (1c) \qquad \frac{\vdash \sigma \ :\ \Sigma \to \Sigma' \quad \Sigma \vdash E\ \textit{Quantifiable}}{\Sigma' \vdash \overline{\sigma}(E)\ \textit{Quantifiable}}\ (1d)$$

$$\frac{\vdash \sigma \ :\ \Sigma \to \Sigma' \quad \Sigma \vdash I : \Phi}{\Sigma' \vdash \overline{\sigma}(I) : \overline{\sigma}(\Phi)}\ (2) \qquad \frac{\vdash \sigma \ :\ \Sigma \to \Sigma' \quad \Sigma \vdash \tau\ \textit{Kind}}{\Sigma' \vdash \overline{\sigma}(\tau)\ \textit{Kind}}\ (3)$$

$$\frac{\vdash \sigma \ :\ \Sigma \to \Sigma' \quad \Sigma \vdash \Phi : \tau}{\Sigma' \vdash \overline{\sigma}(\Phi) : \overline{\sigma}(\tau)}\ (4) \qquad \frac{\vdash \sigma \ :\ \Sigma \to \Sigma' \quad \Sigma \vdash \Phi : \tau}{\Sigma' \vdash \overline{\sigma}(\Phi{\downarrow_\delta}) \leq \overline{\sigma}(\Phi)}\ (5)$$

If $dom(\Psi)$ *and* $dom(\Sigma')$ *are disjoint, then the following rules are admissible:*

$$\frac{\vdash \sigma \ :\ \Sigma \to \Sigma' \quad \vdash \Sigma, \Psi\ \textit{Theory}}{\vdash \Sigma', \overline{\sigma}(\Psi)\ \textit{Theory}}\ (6a) \qquad \frac{\vdash \sigma \ :\ \Sigma \to \Sigma' \quad \vdash \Sigma, \Psi\ \textit{Theory}}{\vdash \sigma * \Psi \ :\ \Sigma, \Psi \to \Sigma', \overline{\sigma}(\Psi)}\ (6b)$$

In addition to disjoint domains $dom(\Psi)$ and $dom(\Sigma')$, if Ψ is simple, then the following rules are admissible:

$$\frac{\vdash \sigma \ : \ \Sigma \to \Sigma' \quad \vdash \Sigma, \Psi \ \mathit{Theory}}{\vdash \Sigma', \overline{\sigma}(\mathcal{E}(\Psi)) \longhookrightarrow \Sigma', \mathcal{E}(\overline{\sigma}(\Psi))} \ (7) \qquad \frac{\vdash \sigma \ : \ \Sigma \to \Sigma' \quad \vdash id_\Sigma, \psi \ : \ \Sigma, \Psi \to \Sigma}{\vdash id_{\Sigma'}, \overline{\sigma}(\psi) \ : \ \Sigma', \overline{\sigma}(\Psi) \to \Sigma'} \ (8)$$

Proof. The proof proceeds by mutual induction on the derivation of the second hypothesis in each rule above:

1. The rules from which the premises *i)* $\Sigma \vdash E : E'$, *ii)* $\Sigma \vdash E \doteq E'$, *iii)* $\Sigma \vdash E \ \mathit{Inhabitable}$ and *iv)* $\Sigma \vdash E \ \mathit{Quantifiable}$ can be derived are specific cases of the foundational rule in definition 4.7, except for the rule *constType* and the rule *elabType* in figure 5.10. Therefore, it suffices to proceed the proof for (1*a*), (1*b*), (1*c*), (1*d*) in three cases:

   - Case Foundational Rule:

     $$\frac{\Sigma, \mathbf{\Psi^1} \vdash \mathcal{S}_1 \quad \dots \quad \Sigma, \mathbf{\Psi^m} \vdash \mathcal{S}_m}{\Sigma \vdash \mathcal{S}} \ \mathcal{R}$$

     We want to prove that $\Sigma' \vdash \overline{\sigma}(\mathcal{S})$, where $\overline{\sigma}(\mathcal{S})$ denotes the application of $\overline{\sigma}$ to the expressions that occur in the respective succedent S, more precisely $\overline{\sigma}(E : E')$ denotes $\overline{\sigma}(E) : \overline{\sigma}(E')$, $\overline{\sigma}(E \doteq E')$ denotes $\overline{\sigma}(E) \doteq \overline{\sigma}(E')$, $\overline{\sigma}(E \ \mathit{Inhabitable})$ denotes $\overline{\sigma}(E) \ \mathit{Inhabitable}$ and $\overline{\sigma}(E \ \mathit{Quantifiable})$ denotes $\overline{\sigma}(E) \ \mathit{Quantifiable}$.
     By induction hypothesis, we have

     $$\Sigma', \overline{\sigma}(\mathbf{\Psi^1}) \vdash \overline{\sigma}(\mathcal{S}_1) \ \dots \ \Sigma', \overline{\sigma}(\mathbf{\Psi^m}) \vdash \overline{\sigma}(\mathcal{S}_m)$$

     where $\overline{\sigma}(\mathbf{\Psi_i})$ denotes the application of $\overline{\sigma}$ to all expressions that occur in Ψ'. By applying the foundational rule to these premises, we get $\Sigma' \vdash \overline{\sigma}(\mathcal{S})$ as desired.
     This shows that the judgments $\Sigma \vdash E \ \mathit{Inhabitable}$ and $\Sigma \vdash E \ \mathit{Quantifiable}$ are preserved along morphism application $\overline{\sigma}$.

   To complete the proof for the remaining two judgments $\Sigma \vdash E : E'$ and $\Sigma \vdash E \doteq E'$, we consider the following cases:

   - **Case** *constType***:**

     $$\frac{\vdash \Sigma \ \mathit{Theory} \quad x : E[= E'] \in \Sigma}{\Sigma \vdash x : E} \ \mathit{constType}$$

     We want to show that $\Sigma' \vdash \overline{\sigma}(x) : \overline{\sigma}(E)$.

     Since we have $\vdash \sigma \ : \ \Sigma \to \Sigma'$ and $x \ : \ E \ = \ E'$ is declared in Σ, we know from the rule *mapConstant* that σ maps x to a well-formed expression E'' of type $\overline{\sigma}(E)$ over Σ'. By definition 5.17, we have $\overline{\sigma}(x) = E''$. Hence we have $\Sigma' \vdash \overline{\sigma}(x) : \overline{\sigma}(E)$.

- **Case** *elabType*:

$$\frac{\Sigma \vdash I : \Phi \quad \Phi{\downarrow_{\delta\beta}} = \{\Psi\} \quad \mathcal{E}(\Psi) = \Psi_0, x : E[= E'], \Psi_1 \quad dom(\Psi_0) = \{y_1, \ldots, y_n\}}{\Sigma \vdash I.x : [I.y_1 / y_1, \ldots, I.y_n / y_n]\, E}\ \textit{elabType}$$

We want to show that

$$\Sigma' \vdash \overline{\sigma}(I.x) : \overline{\sigma}([I.y_1 / y_1, \ldots, I.y_n / y_n]\, E).$$

Firstly, by applying the induction hypothesis for (2) to $\Sigma \vdash I : \Phi$, we get

$$\Sigma' \vdash \overline{\sigma}(I) : \overline{\sigma}(\Phi) \tag{5.5}$$

Secondly, by applying $\overline{\sigma}$ to both sides in the premise $\Phi{\downarrow_{\delta\beta}} = \{\Psi\}$ and by definition of $\overline{\sigma}(\{\Psi\})$, we get

$$\overline{\sigma}(\Phi{\downarrow_{\delta\beta}}) = \{\overline{\sigma}(\Psi)\} \tag{5.6}$$

Then by taking the $\delta\beta$-form of both sides in 5.6, we get

$$\overline{\sigma}(\Phi{\downarrow_{\delta\beta}}){\downarrow_{\delta\beta}} = \{\overline{\sigma}(\Psi)\}{\downarrow_{\delta\beta}} \tag{5.7}$$

Then by applying the definition of $\delta\beta$-form in the right hand-side of 5.7, we get

$$\overline{\sigma}(\Phi{\downarrow_{\delta\beta}}){\downarrow_{\delta\beta}} = \{\overline{\sigma}(\Psi)\} \tag{5.8}$$

It is straightforward to show that

$$\overline{\sigma}(\Phi{\downarrow_{\delta}}){\downarrow_{\delta\beta}} = \overline{\sigma}(\Phi{\downarrow_{\delta\beta}}){\downarrow_{\delta\beta}} \tag{5.9}$$

Then from 5.8 and 5.9, we get

$$\overline{\sigma}(\Phi{\downarrow_{\delta}}){\downarrow_{\delta\beta}} = \{\overline{\sigma}(\Psi)\} \tag{5.10}$$

Thirdly, it follows straightforward from $\Sigma \vdash I : \Phi$ that

$$\Sigma \vdash \Phi : \texttt{theory} \tag{5.11}$$

Then, by applying the induction hypothesis for (5) to 5.11, we get

$$\Sigma' \vdash \overline{\sigma}(\Phi{\downarrow_{\delta}}) \leq_{\texttt{theory}} \overline{\sigma}(\Phi) \tag{5.12}$$

Since the rule *theoryFragInc* is the only primitive rule to derive 5.12 and we have 5.10, we know that

$$\vdash \Sigma', \overline{\sigma}(\Psi) \hookrightarrow \Sigma', \Psi' \tag{5.13}$$

$$\text{where } \overline{\sigma}(\Phi){\downarrow_{\delta\beta}} = \{\Psi'\} \tag{5.14}$$

for a fresh meta-variable Ψ'.

Forthly, applying $\overline{\sigma}$ to $\mathcal{E}(\Psi)$ yields

$$\overline{\sigma}(\mathcal{E}(\Psi)) = \overline{\sigma}(\Psi_0), x : \overline{\sigma * \Psi}(E)[= \overline{\sigma * \Psi}(E')], \overline{\sigma}(\Psi_1) \tag{5.15}$$

Then, by applying lemma 5.39 to 5.13 and by the definition of elaboration, we get

$$\vdash \Sigma', \mathcal{E}(\overline{\sigma}(\Psi)) \hookrightarrow \Sigma', \mathcal{E}(\Psi') \tag{5.16}$$

Then, by applying the induction hypothesis for (7), we get

$$\vdash \Sigma', \overline{\sigma}(\mathcal{E}(\Psi)) \hookrightarrow \Sigma', \mathcal{E}(\overline{\sigma}(\Psi)) \tag{5.17}$$

Therefore, we have

$$\vdash \Sigma', \overline{\sigma}(\mathcal{E}(\Psi)) \hookrightarrow \Sigma', \mathcal{E}(\Psi') \tag{5.18}$$

Thus, we have

$$\mathcal{E}(\Psi') = \Psi'_0, x : \overline{\sigma * \Psi}(E)[\, = \overline{\sigma * \Psi}(E')], \Psi'_1 \tag{5.19}$$

for some Ψ'_0 and Ψ'_1.
Since $dom(\Sigma)$ and $dom(\Sigma')$ are disjoint, by definition of $\overline{\sigma}$ we have

$$dom(\overline{\sigma}(\Psi_0)) \supseteq \{y_1, \ldots, y_n\} \tag{5.20}$$

Because of 5.20, we know that $\overline{\sigma * \Psi_0}(E)$ only contains $y_1, \ldots, y_n$. Therefore, by applying the rule *elabType* to 5.5, 5.14, 5.19 and 5.20, we get

$$\Sigma' \vdash \overline{\sigma}(I).x \,:\, [\overline{\sigma}(I).y_1 \,/y_1, \ldots, \overline{\sigma}(I).y_n \,/y_n]\, \overline{\sigma * \Psi_0}(E) \tag{5.21}$$

Thus, by applying lemma 5.35 to 5.21, we get

$$\Sigma' \vdash \overline{\sigma}(I).x \,:\, \overline{\sigma}([I.y_1 \,/y_1, \ldots, I.y_n \,/y_n]\, E) \tag{5.22}$$

Hence, applying the definition of $\overline{\sigma}$ for $I.x$ in 5.22 yields

$$\Sigma' \vdash \overline{\sigma}(I.x\,) : \overline{\sigma}([I.y_1 \,/y_1, \ldots, I.y_n \,/y_n]\, E).$$

as desired.
This shows that the typing judgment $\Sigma \vdash E : E'$ for expressions is preserved along morphism application $\overline{\sigma}$.

The proof of the cases for the rules *constDef* and *elabDef* are analogous to proofs of the cases for the rules *constType* and *elabType*, respectively. This shows that the equality judgment $\Sigma \vdash E \doteq E'$ is preserved along morphism application $\overline{\sigma}$.

2. Derivation of the judgment $\Sigma \vdash I : \Phi$.

- **Case** *instConst*:

$$\frac{\vdash \Sigma\ \mathit{Theory} \quad i : \Phi \text{ in } \Sigma}{\Sigma \vdash i : \Phi}\ \mathit{instConst}$$

This follows immediately from rule *mapInstance*.

- **Case** *morphFrag*:

$$\frac{\vdash id_{\Sigma}, \psi \ : \ \Sigma, \Psi \to \Sigma}{\Sigma \vdash \{\psi\} : \{\Psi\}} \ \mathit{morphFrag}$$

By applying the induction hypothesis for (8) to the premise of *morphFrag*, we get

$$\vdash id_{\Sigma'}, \overline{\sigma}(\psi) \ : \ \Sigma', \overline{\sigma}(\Psi) \to \Sigma' \tag{5.23}$$

By applying the rule *morphFrag* to 5.23, we get

$$\Sigma' \vdash \{\overline{\sigma}(\psi)\} : \{\overline{\sigma}(\Psi)\} \tag{5.24}$$

Thus, applying the definition of $\overline{\sigma}$ for $\{\psi\}$ and $\{\Psi\}$ to 5.24 yields

$$\Sigma' \vdash \overline{\sigma}(\{\psi\}) : \overline{\sigma}(\{\Psi\}).$$

This shows that the typing judgment $\Sigma \vdash I : \Phi$ for instances is preserved along morphism application $\overline{\sigma}$.

3. Derivation of the judgment $\Sigma \vdash \tau \ \mathit{Kind}$.

- **Case** *baseKind*:

$$\frac{}{\Sigma \vdash \texttt{theory} \ \mathit{Kind}} \ \mathit{baseKind}$$

By definition of $\overline{\sigma}$ for `theory`, we have $\Sigma' \vdash \overline{\sigma}(\texttt{theory}) \ \mathit{Kind}$.

- **Case** *depKind*:

$$\frac{\Sigma, x : E \vdash \tau \ \mathit{Kind}}{\Sigma \vdash (x : E)\, \tau \ \mathit{Kind}} \ \mathit{depKind}$$

By lemma 5.34 and the induction hypothesis for (3), we have

$$\Sigma', x : \overline{\sigma}(E) \vdash \overline{\sigma^x}(\tau) \ \mathit{Kind} \tag{5.25}$$

By applying the rule *depKind* to 5.25, we get

$$\Sigma', x : \overline{\sigma}(E) \vdash (x : \overline{\sigma}(E))\, \overline{\sigma^x}(\tau) \ \mathit{Kind} \tag{5.26}$$

Thus, applying the definition of $\overline{\sigma}$ for $(x : \overline{\sigma}(E))\, \overline{\sigma^x}(\tau)$ in 5.26 yields

$$\Sigma', x : \overline{\sigma}(E) \vdash \overline{\sigma}((x : E)\, \tau) \ \mathit{Kind}.$$

This shows that the well-formedness judgment $\Sigma \vdash \tau \ \mathit{Kind}$ for kinds is preserved along morphism application $\overline{\sigma}$.

4. Derivation of the judgment $\Sigma \vdash \Phi : \tau$.

- **Case** *theoryFrag*:

$$\frac{\vdash \Sigma, \Psi \ \mathit{Theory}}{\Sigma \vdash \{\Psi\} : \texttt{theory}} \ \mathit{theoryFrag}$$

By applying the induction hypothesis for (6*a*) to $\vdash \Sigma, \Psi \ \mathit{Theory}$, we get

$$\vdash \Sigma', \overline{\sigma}(\Psi) \ \mathit{Theory} \tag{5.27}$$

By applying the rule *theoryFrag* to 5.27, we get

$$\Sigma' \vdash \{\overline{\sigma}(\Psi)\} : \texttt{theory} \tag{5.28}$$

Thus applying the definition of $\overline{\sigma}$ for $\{\Psi\}$ and `theory` in 5.28 yields

$$\Sigma' \vdash \overline{\sigma}(\{\Psi\}) : \overline{\sigma}(\texttt{theory}).$$

- **Case** *bind*:

$$\frac{\Sigma, x : E \vdash \Phi : \tau}{\Sigma \vdash (x : E)\,\Phi : (x : E)\,\tau}\ \mathit{bind}$$

By lemma 5.34 and the induction hypothesis for (4), we have

$$\Sigma', x : \overline{\sigma}(E) \vdash \overline{\sigma^x}(\Phi) : \overline{\sigma^x}(\tau) \tag{5.29}$$

By applying the rule *bind* to 5.29, we get

$$\Sigma' \vdash (x : \overline{\sigma}(E))\,\overline{\sigma^x}(\Phi) : (x : \overline{\sigma}(E))\,\overline{\sigma^x}(\tau) \tag{5.30}$$

Thus applying the definition of $\overline{\sigma}$ for $(x : E)\,\Phi$ and $(x : E)\,\tau$ in 5.30 yields

$$\Sigma' \vdash \overline{\sigma}((x : E)\,\Phi) : \overline{\sigma}((x : E)\,\tau).$$

- **Case** *appl*:

$$\frac{\Sigma \vdash \Phi : (x : E_2)\,\tau \quad \Sigma \vdash E_1 : E_2}{\Sigma \vdash \Phi\,E_1 : [E_1/x]\tau}\ \mathit{appl}$$

By applying the induction hypotheses (4) and (1*a*) to the premises of *appl*, we get

$$\Sigma' \vdash \overline{\sigma}(\Phi) : \overline{\sigma}((x : E_2)\,\tau) \tag{5.31}$$
$$\Sigma' \vdash \overline{\sigma}(E_1) : \overline{\sigma}(E_2) \tag{5.32}$$

By applying the definition of $\overline{\sigma}$ for $(x : E_2)\,\tau$ to 5.31 and applying the rule *appl* to 5.31 and 5.32, we get

$$\Sigma' \vdash \overline{\sigma}(\Phi)\,\overline{\sigma}(E_1) : [\overline{\sigma}(E_1)/x]\overline{\sigma^x}(\tau) \tag{5.33}$$

By lemma 5.35, we get

$$\Sigma' \vdash \overline{\sigma}(\Phi)\,\overline{\sigma}(E_1) : \overline{\sigma}([E_1/x]\tau) \tag{5.34}$$

Thus applying the definition of $\overline{\sigma}$ for $\Phi\,E_1$ to 5.34 yields

$$\Sigma' \vdash \overline{\sigma}(\Phi\,E_1) : \overline{\sigma}([E_1/x]\tau).$$

- **Case** *const*:

$$\frac{\vdash \Sigma\ \mathit{Theory} \quad \varphi : \tau = \Phi \text{ in } \Sigma}{\Sigma \vdash \varphi : \tau}\ \mathit{const}$$

This follows immediately from rule *mapTheoryFamily*.

This shows that the kinding judgment $\Sigma \vdash \Phi : \tau$ for theory families is preserved under morphism application $\overline{\sigma}$.

5. Derivation of the judgment $\Sigma \vdash \Phi : \tau$.

- **Case** *theoryFrag*

$$\frac{\vdash \Sigma,\ \Psi\ \mathit{Theory}}{\Sigma \vdash \{\Psi\} : \mathtt{theory}}\ \mathit{theoryFrag}$$

We want to show that $\Sigma' \vdash \overline{\sigma}(\{\Psi\}\downarrow_\delta) \leq \overline{\sigma}(\{\Psi\})$.

By definition 5.26 and the definition of $\overline{\sigma}$ for $\{\Psi\}$, we have

$$\begin{aligned} \overline{\sigma}(\{\Psi\}\downarrow_\delta) &= \overline{\sigma}(\{\Psi\}) \\ &= \{\overline{\sigma}(\Psi)\} \end{aligned} \tag{5.35}$$

By lemma 5.37, we have $\Sigma' \vdash \{\overline{\sigma}(\Psi)\} \leq \{\overline{\sigma}(\Psi)\}$.
Hence, by 5.35, we have

$$\Sigma' \vdash \overline{\sigma}(\{\Psi\}\downarrow_\delta) \leq \overline{\sigma}(\{\Psi\}).$$

- **Case** *const*

$$\frac{\vdash \Sigma\ \mathit{Theory} \quad \varphi : \tau = \Phi \text{ in } \Sigma}{\Sigma \vdash \varphi : \tau}\ \mathit{const}$$

Since we have $\varphi : \tau = \Phi$ in Σ and Σ is well-formed, we know via rule *addTheoryFamily* that

$$\Sigma \vdash \Phi : \tau \tag{5.36}$$

By applying the induction hypothesis for (5) to 5.36, we get

$$\Sigma' \vdash \overline{\sigma}(\Phi\downarrow_\delta) \leq \overline{\sigma}(\Phi) \tag{5.37}$$

Since $\vdash \sigma : \Sigma \to \Sigma'$, we know via rule *mapTheoryFamily* that there exists a well-formed theory family Φ' over Σ' such that $\varphi := \Phi'$ in σ and a well-formed theory morphism σ_0 from Σ_0 to Σ' such that

$$\Sigma' \vdash \overline{\sigma_0}(\Phi) \leq \Phi'$$

where Σ_0 is the fragment of Σ that does not contain $\varphi : \tau = \Phi$. Since $\overline{\sigma_0}(\Phi) = \overline{\sigma}(\Phi)$, we have

$$\Sigma' \vdash \overline{\sigma}(\Phi) \leq \Phi' \tag{5.38}$$

By transitivity (lemma 5.37) of 5.37 and 5.38, we have

$$\Sigma' \vdash \overline{\sigma}(\Phi\downarrow_\delta) \leq \Phi'$$

We have $\varphi\downarrow_\delta = \Phi\downarrow_\delta$ and $\overline{\sigma}(\varphi) = \Phi'$. Thus, we have

$$\Sigma' \vdash \overline{\sigma}(\varphi\downarrow_\delta) \leq \overline{\sigma}(\varphi)$$

- **Case** *bind*

$$\frac{\Sigma,\, x : E \vdash\ \Phi : \tau}{\Sigma \vdash\ (x : E)\, \Phi : (x : E)\, \tau}\ \mathit{bind}$$

We want to show

$$\Sigma' \ \vdash\ \overline{\sigma}(((x : E)\, \Phi)\!\downarrow_\delta) \leq \overline{\sigma}((x : E)\, \Phi) \tag{5.39}$$

Since we have $\vdash \sigma\ :\ \Sigma \to \Sigma'$, we also have $\vdash \sigma^x\ :\ \Sigma, x : E \to \Sigma', x : \overline{\sigma}(E)$. By applying induction hypothesis for (5) with $\sigma^x : \Sigma, x : E \to \Sigma', x : \overline{\sigma}(E)$ to the premise $\Sigma,\, x : E \vdash\ \Phi : \tau$, we get

$$\Sigma', x : \overline{\sigma}(E) \ \vdash\ \overline{\sigma^x}(\Phi\!\downarrow_\delta) \leq \overline{\sigma^x}(\Phi) \tag{5.40}$$

Since $\overline{\sigma^x}(\Phi\downarrow_\delta)$ and $\overline{\sigma^x}(\Phi)$ are β-equal to $((x\ :\ \overline{\sigma}(E))\, \overline{\sigma^x}(\Phi\downarrow_\delta))(x)$ and $((x\ :\ \overline{\sigma}(E))\, \overline{\sigma^x}(\Phi))(x)$, respectively, we have

$$\Sigma', x : \overline{\sigma}(E) \ \vdash\ ((x : \overline{\sigma}(E))\, \overline{\sigma^x}(\Phi\!\downarrow_\delta))(x) \leq ((x : \overline{\sigma}(E))\, \overline{\sigma^x}(\Phi))(x) \tag{5.41}$$

By applying the rule *parTheoryInc* to 5.41, we get

$$\Sigma' \ \vdash\ (x : \overline{\sigma}(E))\, \overline{\sigma^x}(\Phi\!\downarrow_\delta) \leq (x : \overline{\sigma}(E))\, \overline{\sigma^x}(\Phi) \tag{5.42}$$

By applying definition 5.17 of $\overline{\sigma}$ and definition 5.26 of δ-normal form to 5.42, we get 5.39 as desired.

- **Case** *appl*

$$\frac{\Sigma \vdash\ \Phi : (x : A)\, \tau \quad \Sigma \vdash\ E : A}{\Sigma \vdash\ \Phi(E) : [E/x]\tau}\ \mathit{appl}$$

We want to show that

$$\Sigma' \ \vdash\ \overline{\sigma}(\Phi(E)\!\downarrow_\delta) \leq \overline{\sigma}(\Phi(E)) \tag{5.43}$$

By applying induction hypothesis for (5) and (4) to $\Sigma \vdash\ \Phi : (x : A)\, \tau$, and by definition 5.17, we get

$$\Sigma' \ \vdash\ \overline{\sigma}(\Phi\!\downarrow_\delta) \leq \overline{\sigma}(\Phi) \tag{5.44}$$

$$\Sigma' \ \vdash\ \overline{\sigma}(\Phi) : (x : \overline{\sigma}(A))\, \overline{\sigma^x}(\tau) \tag{5.45}$$

Since we have 5.45, we know that 5.44 is derived from rule *parTheoryInc*. Therefore, we have

$$\Sigma', x : \overline{\sigma}(A) \ \vdash\ \overline{\sigma}(\Phi\!\downarrow_\delta)(x) \leq \overline{\sigma}(\Phi)(x) \tag{5.46}$$

By applying rule *substClosure* in lemma 5.37 to 5.46, we get

$$\Sigma' \ \vdash\ \overline{\sigma}(\Phi\!\downarrow_\delta)(\overline{\sigma}(E)) \leq \overline{\sigma}(\Phi)(\overline{\sigma}(E)) \tag{5.47}$$

By applying definition 5.17 to 5.47, we get

$$\Sigma' \ \vdash\ \overline{\sigma}(\Phi\!\downarrow_\delta\, (E)) \leq \overline{\sigma}(\Phi(E)) \tag{5.48}$$

Hence, by definition 5.26 of δ-normal form for $\Phi(E)$, we get 5.43 as desired.

This shows that the refinement judgment $\Sigma \vdash \Phi \leq_\tau \Phi'$ for theory families is preserved along morphism application $\overline{\sigma}$.

6. Derivation of the judgment $\vdash \Sigma, \Psi\ \mathit{Theory}$.

   - **Case** *emptyTheory*:

   $$\frac{}{\vdash \cdot\ \mathit{Theory}}\ \mathit{emptyTheory}$$

   a) We want to show that $\vdash \Sigma', \overline{\sigma}(\cdot)\ \mathit{Theory}$: This follows immediately by definition 5.17 of $\overline{\sigma}$ and the premise $\vdash \sigma\ :\ \cdot \to \Sigma'$.

   b) We want to show that $\vdash \sigma * \cdot\ :\ \cdot \to \Sigma', \overline{\sigma}(\cdot)$. This follows immediately by definition 5.18 of $\sigma * \cdot$ and definition 5.17 of $\overline{\sigma}(\cdot)$, and the premise $\vdash \sigma\ :\ \cdot \to \Sigma'$.

   - **Case** *addConstant*: Assume $\Psi = \underline{\Psi_0, x : E_1[\,= E_2]}$. Then we have the following derivation via rule *addConstant*:

   $$\frac{\vdash \Sigma,\ \Psi_0\ \mathit{Theory} \quad \Sigma, \Psi_0 \vdash E_1\ \mathit{Inhabitable} \quad [\Sigma, \Psi_0 \vdash E_2 : E_1] \quad x \notin \mathit{dom}(\Sigma, \Psi_0)}{\vdash \Sigma, \Psi_0,\ x : E_1[\,= E_2]\ \mathit{Theory}}\ \mathit{addConstant}$$

   We want to show that

   a) $\vdash \Sigma', \overline{\sigma}(\Psi_0, x : E_1[\,= E_2])\ \mathit{Theory}$.

   b) $\vdash \sigma * (\Psi_0, x : E_1 = E_2)\ :\ \Sigma, (\Psi_0, x : E_1 = E_2) \to \Sigma', \overline{\sigma}(\Psi_0, x : E_1 = E_2)$.

   By applying the induction hypotheses (6*a*), (1*c*) and (1*a*) respectively to the first three premises of *addConstant* with morphism $\sigma * \Psi_0 : \Sigma,\ \Psi_0 \to \Sigma'$, we get

   $$\vdash \Sigma',\ \overline{\sigma}(\Psi_0)\ \mathit{Theory} \tag{5.49}$$

   $$\Sigma',\ \overline{\sigma}(\Psi_0) \vdash \overline{\sigma * \Psi_0}(E_1)\ \mathit{Inhabitable} \tag{5.50}$$

   $$\Sigma',\ \overline{\sigma}(\Psi_0) \vdash \overline{\sigma * \Psi_0}(E_2) : \overline{\sigma * \Psi_0}(E_1) \tag{5.51}$$

   Since $\mathit{dom}(\Psi_0,\ x : E_1 = E_2)$ and $\mathit{dom}(\Sigma')$ are disjoint, we know that x is fresh for Σ'. Then, by applying the rule *addConstant* to 5.49, 5.50 and 5.51, we get

   $$\vdash \Sigma',\ \overline{\sigma}(\Psi_0),\ x : \overline{\sigma * \Psi_0}(E_1)[\,= \overline{\sigma * \Psi_0}(E_2)]\ \mathit{Theory} \tag{5.52}$$

   a) Thus, by applying the definition of $\overline{\sigma}$ for $\Psi_0,\ x : E_1[\,= E_2]$ in 5.52 we get

   $$\vdash \Sigma',\ \overline{\sigma}(\Psi_0,\ x : E_1 = E_2)\ \mathit{Theory}$$

   as desired.

   b) By induction hypothesis for (6*b*), we have

   $$\vdash \sigma * \Psi_0\ :\ \Sigma,\ \Psi_0 \to \Sigma',\ \overline{\sigma}(\Psi_0) \tag{5.53}$$

   Therefore, we have

   $$\vdash \sigma * \Psi_0\ :\ \Sigma,\ \Psi_0 \to \Sigma',\ \overline{\sigma}(\Psi_0),\ x : \overline{\sigma * \Psi_0}(E_1)[\,= \overline{\sigma * \Psi_0}(E_2)] \tag{5.54}$$

   Since we have 5.52 and

   $$x : \overline{\sigma * \Psi_0}(E_1)[\,= \overline{\sigma * \Psi_0}(E_2)] \ \in\ \Sigma, \overline{\sigma}(\Psi_0), x : \overline{\sigma * \Psi_0}(E_1)[\,= \overline{\sigma * \Psi_0}(E_2)]$$

we apply rule *constType* to get

$$\Sigma', \overline{\sigma}(\Psi_0), x : \overline{\sigma * \Psi_0}(E_1)[= \overline{\sigma * \Psi_0}(E_2)] \vdash x : \overline{\sigma * \Psi_0}(E_1) \tag{5.55}$$

By applying rule *mapConstant* to 5.54, 5.55 and 5.51, we get

$$\vdash \underline{\sigma * \Psi_0}, x := x \ : \ \underline{\Sigma, \Psi_0, x : E_1[= E_2]} \to \Sigma', \overline{\sigma}(\Psi_0), x : \overline{\sigma * \Psi_0}(E_1)[= \overline{\sigma * \Psi_0}(E_2)].$$

By definition 5.18, we get

$$\vdash \sigma * (\Psi_0, x : E_1[= E_2]) \ : \ \Sigma, \Psi_0, x : E_1[= E_2] \to \Sigma', \overline{\sigma}(\Psi_0, x : E_1[= E_2])$$

as desired.

- **Case** *addInstance*: Assume $\Psi = \Psi_0, \iota : \Phi$. Then we have the following derivation via rule *addInstance*:

$$\frac{\vdash \Sigma, \Psi_0 \ \mathit{Theory} \quad \Sigma, \Psi_0 \vdash \Phi : \texttt{theory} \quad \iota \notin \mathit{dom}(\Sigma, \Psi_0)}{\vdash \Sigma, \Psi_0, \iota : \Phi \ \mathit{Theory}} \ \mathit{addInstance}$$

We want to show that

a) $\vdash \Sigma', \overline{\sigma}(\Psi_0, \iota : \Phi)$ *Theory*.

b) $\vdash \sigma * (\Psi_0, \iota : \Phi) \ : \ \Sigma, \Psi_0, \iota : \Phi \to \Sigma', \overline{\sigma}(\Psi_0, \iota : \Phi)$.

By applying the induction hypothesis for (6*a*) and (6*b*) respectively to the first premise of *addInstance* with theory morphism $\sigma : \Sigma \to \Sigma'$, we get

$$\vdash \Sigma', \overline{\sigma}(\Psi_0) \ \mathit{Theory} \tag{5.56}$$

$$\vdash \sigma * \Psi_0 \ : \ \Sigma, \Psi_0 \to \Sigma', \overline{\sigma}(\Psi_0) \tag{5.57}$$

By applying the induction hypothesis for (4) and (2) respectively to the second premise of *addInstance* with theory morphism $\sigma * \Psi_0 : \Sigma, \Psi_0 \to \Sigma', \overline{\sigma}(\Psi_0)$ and by the definition of $\overline{\sigma}$ for `theory`, we get

$$\Sigma', \overline{\sigma}(\Psi_0) \vdash \overline{\sigma * \Psi_0}(\Phi) : \texttt{theory} \tag{5.58}$$

Since $\mathit{dom}(\Psi_0, \iota : \Phi)$ and $\mathit{dom}(\Sigma')$ are disjoint, by definition of $\overline{\sigma}$ for Ψ, we have

$$i \notin \mathit{dom}(\Sigma', \overline{\sigma}(\Psi_0)) \tag{5.59}$$

Then, by applying the rule *addInstance* to 5.56, 5.58 and 5.59, we get

$$\vdash \Sigma', \overline{\sigma}(\Psi_0), \iota : \overline{\sigma * \Psi_0}(\Phi) \ \mathit{Theory} \tag{5.60}$$

a) Thus, applying the definition of $\overline{\sigma}$ for $\Psi_0, \iota : \Phi$ in 5.60 yields

$$\vdash \Sigma', \overline{\sigma}(\Psi_0, \iota : \Phi) \ \mathit{Theory}$$

as desired.

b) Since we have 5.60, we know that $\iota : \overline{\sigma * \Psi_0}(\Phi)$ is in

$$\Sigma', \overline{\sigma}(\Psi_0), \iota : \overline{\sigma * \Psi_0}(\Phi)$$

Then by applying the rule *instConst* to 5.60, we get

$$\Sigma', \overline{\sigma}(\Psi_0), \iota : \overline{\sigma * \Psi_0}(\Phi) \vdash \iota : \overline{\sigma * \Psi_0}(\Phi) \tag{5.61}$$

Since we have 5.57, we also have

$$\vdash \sigma * \Psi_0 \ : \ \Sigma, \Psi_0 \to \Sigma', \overline{\sigma}(\Psi_0), \iota : \overline{\sigma * \Psi_0}(\Phi) \tag{5.62}$$

Since $dom(\Psi_0, \iota : \Phi)$ and $dom(\Sigma')$ are disjoint, by applying the definition of $\overline{\sigma}$ for Ψ in 5.61 and 5.62, we have

$$\vdash \sigma * \Psi_0 \ : \ \Sigma, \Psi_0 \to \Sigma', \overline{\sigma}(\Psi_0, \iota : \Phi) \tag{5.63}$$

$$\Sigma', \overline{\sigma}(\Psi_0, \iota : \Phi) \vdash \ \iota : \overline{\sigma * \Psi_0}(\Phi) \tag{5.64}$$

Then by applying the rule *mapInstance* to 5.63 and 5.64, we get

$$\vdash (\sigma * \Psi_0), \iota := \iota \ : \ \Sigma, \Psi_0, \iota : \Phi \to \Sigma', \overline{\sigma}(\Psi_0, \iota : \Phi) \tag{5.65}$$

Since $dom(\Psi_0, \iota : \Phi)$ and $dom(\Sigma')$ are disjoint, by applying the definition of $\sigma * \Psi$ in 5.65, we get

$$\vdash \sigma * (\Psi_0, \iota : \Phi) \ : \ \Sigma, \Psi_0, \iota : \Phi \to \Sigma', \overline{\sigma}(\Psi_0, \iota : \Phi)$$

as desired.

- **Case** *addTheoryFamily*: Assume $\Psi = \underline{\Psi_0, \varphi : \tau = \Phi}$. Then we have following derivation via rule *addTheoryFamily*:

$$\frac{\vdash \Sigma, \Psi_0 \ \mathit{Theory} \quad \Sigma, \Psi_0 \vdash \ \Phi : \tau \quad \varphi \notin dom(\Sigma, \Psi_0)}{\vdash \Sigma, (\Psi_0, \varphi : \tau = \Phi) \ \mathit{Theory}} \ \mathit{addTheoryFamily}$$

We want to show that

a) $\vdash \Sigma', \overline{\sigma}(\Psi_0, \varphi : \tau = \Phi) \ \mathit{Theory}$.

b) $\vdash \sigma * (\Psi_0, \varphi : \tau = \Phi) \ : \ \Sigma, (\Psi_0, \varphi : \tau = \Phi) \to \Sigma', \overline{\sigma}(\Psi_0, \varphi : \tau = \Phi)$.

By applying the induction hypothesis for (6*a*) and (6*b*) respectively to the first premise of *addTheoryFamily* with theory morphism $\sigma : \Sigma \to \Sigma'$, we get

$$\vdash \Sigma', \overline{\sigma}(\Psi_0) \ \mathit{Theory} \tag{5.66}$$

$$\vdash \sigma * \Psi_0 \ : \ \Sigma, \Psi_0 \to \Sigma', \overline{\sigma}(\Psi_0) \tag{5.67}$$

By applying the induction hypothesis (4) respectively to the second premise of *addTheoryFamily* and to 5.67, we get

$$\Sigma', \overline{\sigma}(\Psi_0) \vdash \ \overline{\sigma * \Psi_0}(\Phi) : \overline{\sigma * \Psi_0}(\tau) \tag{5.68}$$

Since $dom(\Psi_0, \varphi : \tau = \Phi)$ and $dom(\Sigma')$ are disjoint, by definition of $\overline{\sigma}$ for Ψ, we have

$$\varphi \notin dom(\Sigma', \overline{\sigma}(\Psi_0)) \tag{5.69}$$

Then, by applying the rule *addTheoryFamily* to 5.66, 5.68, 5.69, we get

$$\vdash \Sigma', \overline{\sigma}(\Psi_0), \varphi : \overline{\sigma * \Psi_0}(\tau) = \overline{\sigma * \Psi_0}(\Phi) \ \mathit{Theory} \tag{5.70}$$

a) Thus, applying the definition of $\overline{\sigma}$ for $\Psi_0, \varphi : \tau = \Phi$ in 5.70 yields

$$\vdash \Sigma', \overline{\sigma}(\Psi_0, \varphi : \tau = \Phi) \ \mathit{Theory}$$

as desired.

b) Since we have 5.67, we also have

$$\vdash \sigma * \Psi_0 \ : \ \Sigma, \Psi_0 \to \Sigma', \overline{\sigma}(\Psi_0), \varphi : \overline{\sigma * \Psi_0}(\tau) = \overline{\sigma * \Psi_0}(\Phi) \tag{5.71}$$

Since we have 5.70, we know that

$$\varphi : \overline{\sigma * \Psi_0}(\tau) = \overline{\sigma * \Psi_0}(\Phi) \in \underline{\Sigma', \overline{\sigma}(\Psi_0), \varphi : \overline{\sigma * \Psi_0}(\tau) = \overline{\sigma * \Psi_0}(\Phi)} \tag{5.72}$$

$$\varphi{\downarrow_{\delta\beta}} = \overline{\sigma * \Psi_0}(\tau) \tag{5.73}$$

Then by applying the rule *const* to 5.70 and 5.72, we get

$$\Sigma', \overline{\sigma}(\Psi_0), \varphi : \overline{\sigma * \Psi_0}(\tau) = \overline{\sigma * \Psi_0}(\Phi) \vdash \varphi : \overline{\sigma * \Psi_0}(\tau) \tag{5.74}$$

Since $dom(\Psi_0, \varphi : \tau = \Phi)$ and $dom(\Sigma')$ are disjoint, by applying the definition of $\overline{\sigma}$ for Ψ in 5.74 and 5.71, we have

$$\vdash \sigma * \Psi_0 \ : \ \Sigma, \Psi_0 \to \Sigma', \overline{\sigma}(\Psi_0, \varphi : \tau = \Phi) \tag{5.75}$$

$$\Sigma', \overline{\sigma}(\Psi_0, \varphi : \tau = \Phi) \vdash \varphi : \overline{\sigma * \Psi_0}(\tau) \tag{5.76}$$

By the reflexivity rule in lemma 5.37, we have

$$\Sigma', \overline{\sigma}(\Psi_0, \varphi : \tau = \Phi) \vdash \overline{\sigma * \Psi_0}(\Phi){\downarrow_{\delta\beta}} \leq \overline{\sigma * \Psi_0}(\Phi){\downarrow_{\delta\beta}} \tag{5.77}$$

Since we have 5.73, by appyling the definition of $\delta\beta$-form (definition 5.27), we get

$$\Sigma', \overline{\sigma}(\Psi_0, \varphi : \tau = \Phi) \vdash \overline{\sigma * \Psi_0}(\Phi){\downarrow_{\delta\beta}} \leq \varphi{\downarrow_{\delta\beta}} \tag{5.78}$$

By applying lemma 5.38 to 5.78, we get

$$\Sigma', \overline{\sigma}(\Psi_0, \varphi : \tau = \Phi) \vdash \overline{\sigma * \Psi_0}(\Phi) \leq \varphi \tag{5.79}$$

Then by applying the rule to 5.71, 5.76 and 5.79, we get

$$\vdash \sigma * (\Psi_0, \varphi : \tau = \Phi) \ : \ \Sigma, (\Psi_0, \varphi : \tau = \Phi) \to \Sigma', \overline{\sigma}(\Psi_0, \varphi : \tau = \Phi)$$

as desired.

7. Derivation of the judgment $\vdash \Sigma, \Psi$ *Theory*. We proceed by induction on the declarations in Ψ. Since Ψ is simple, we have the following three cases:

- **Case** $\Psi = \cdot$**:** By definition 5.29 of inclusion morphism and lemma 5.32, we get $\vdash \Sigma' \hookrightarrow \Sigma'$ immediately.
- **Case** $\Psi = \underline{\Psi_0, x : E[= E']}$**:** We want to show that

$$\vdash \Sigma', \overline{\sigma}(\mathcal{E}(\Psi_0, x : E[= E'])) \hookrightarrow \Sigma', \mathcal{E}(\overline{\sigma}(\Psi_0, x : E[= E']))$$

  This follows immediately from the definition of elaboration and $\overline{\sigma}$.
- **Case** $\Psi = \underline{\Psi_0, \iota : \Phi}$**:** We want to show that

$$\vdash \Sigma', \overline{\sigma}(\mathcal{E}(\Psi_0, \iota : \Phi)) \hookrightarrow \Sigma', \mathcal{E}(\overline{\sigma}(\Psi_0, \iota : \Phi))$$

Firstly, by definition of elaboration and $\overline{\sigma}$, we have

$$\overline{\sigma}(\mathcal{E}(\Psi_0, \iota : \Phi)) = \overline{\sigma}(\mathcal{E}(\Psi_0)), \iota.\Psi_1 \tag{5.80}$$

where $\iota.\Psi_1$ denotes the list of symbol declarations that replace $\iota : \Phi$ after the elaboration of Ψ_1 for

$$\overline{\sigma * \Psi_0}(\Phi{\downarrow_{\delta\beta}}) = \{\Psi_1\} \tag{5.81}$$

Similarly, by definition of elaboration and $\overline{\sigma}$, we have

$$\mathcal{E}(\overline{\sigma}(\Psi_0, \iota : \Phi)) = \mathcal{E}(\overline{\sigma}(\Psi_0)), \iota.\Psi_2 \tag{5.82}$$

where $\iota.\Psi_2$ denotes the list of symbol declarations that replace $\iota : \overline{\sigma * \Psi_0}(\Phi)$ after the elaboration Ψ_2 for

$$\overline{\sigma * \Psi_0}(\Phi){\downarrow_{\delta\beta}} = \{\Psi_2\} \tag{5.83}$$

Secondly, from the derivation of $\vdash \Sigma, \Psi_0, \iota : \Phi$ *Theory*, we know that

$$\Sigma \vdash \Phi : \tau \tag{5.84}$$

Then, by applying the induction hypothesis for (5) with morphism $\overline{\sigma * \Psi_0} : \Sigma, \Psi_0 \to \Sigma'$ to 5.84, we get

$$\Sigma' \vdash \overline{\sigma * \Psi_0}(\Phi{\downarrow_{\delta}}) \leq_{\texttt{theory}} \overline{\sigma * \Psi_0}(\Phi) \tag{5.85}$$

We know that $\overline{\sigma * \Psi_0}(\Phi{\downarrow_{\delta\beta}}) = \overline{\sigma * \Psi_0}(\Phi{\downarrow_{\delta}})$. Then we have

$$\overline{\sigma * \Psi_0}(\Phi{\downarrow_{\delta}}){\downarrow_{\delta\beta}} = \{\Psi_1\} \tag{5.86}$$

Since the rule *theoryFragInc* is the only primitive rule from which 5.85 can be derived, we know that

$$\vdash \Sigma', \Psi_1 \hookrightarrow \Sigma', \Psi_2 \tag{5.87}$$

Thus, from 5.83, 5.86 and 5.87, we know that all declarations $x_i : E_i[= D_i]$ in $\overline{\sigma * \Psi_0}(\Phi{\downarrow_{\delta\beta}})$ are also in $\overline{\sigma * \Psi_0}(\Phi)$.

Hence, we have

$$\vdash \Sigma', \overline{\sigma}(\mathcal{E}(\Psi_0, \iota : \Phi)) \hookrightarrow \Sigma', \mathcal{E}(\overline{\sigma}(\Psi_0, \iota : \Phi))$$

as desired.

8. Derivation of the judgment $\vdash id_\Sigma, \psi : \Sigma, \Psi \to \Sigma$. We look at all the derivations of this judgment where *i)* $\psi = \cdot$, and *ii)* $\psi \neq \cdot$.

   - **Case** $\psi = \cdot$**:** By lemma 5.32, we immediately get $\vdash id_{\Sigma'} : \Sigma' \to \Sigma'$. Then by definition of $\overline{\sigma}$ for ψ_0, we get $\vdash id_{\Sigma'}, \cdot : \Sigma', \cdot \to \Sigma'$ as desired.

- **Case** $\psi = \underline{\psi_0, x := E}$**:** Then we have the following derivation of the second premise of (8) via the rule *mapConstant*.

$$\frac{\vdash id_\Sigma, \psi_0 \ : \ \Sigma, \Psi_0 \to \Sigma \quad \Sigma \vdash E : \overline{id_\Sigma, \psi_0}(E_1) \quad [E = \overline{id_\Sigma, \psi_0}(E_2)]}{\vdash id_\Sigma, (\psi_0, x := E) \ : \ \Sigma, (\Psi_0, x : E_1[= E_2]) \to \Sigma'} \ mapConstant$$

We want to show that $\vdash id_{\Sigma'}, \overline{\sigma}(\psi_0, x := E) \ : \ \Sigma', \overline{\sigma}(\Psi_0, x : E_1[= E_2]) \to \Sigma'$. First, by applying the induction hypothesis for (8) to $\vdash id_\Sigma, \psi_0 \ : \ \Sigma, \Psi_0 \to \Sigma$ with morphism $\sigma : \Sigma \to \Sigma'$, we get

$$\vdash id_{\Sigma'}, \overline{\sigma}(\psi_0) \ : \ \Sigma', \overline{\sigma}(\Psi_0) \to \Sigma' \tag{5.88}$$

Secondly, by applying the induction hypothesis for (1*a*) to $\Sigma \vdash E : \overline{id_\Sigma, \psi_0}(E_1)$ with morphism $\sigma : \Sigma \to \Sigma'$, we get

$$\Sigma' \vdash \overline{\sigma}(E) : \overline{\sigma}(\overline{id_\Sigma, \psi_0}(E_1)) \tag{5.89}$$

By applying lemma 5.33 to 5.89, we get

$$\Sigma' \vdash \overline{\sigma}(E) : \overline{(id_\Sigma, \psi_0) \,;\, \sigma}(E_1) \tag{5.90}$$

By applying lemma 5.36 to 5.90, we get

$$\Sigma' \vdash \overline{\sigma}(E) : \overline{\sigma * \Psi_0 \,;\, (id_{\Sigma'}, \overline{\sigma}(\psi_0))}(E_1) \tag{5.91}$$

By applying lemma 5.33 to 5.91, we get

$$\Sigma' \vdash \overline{\sigma}(E) : \overline{(id_{\Sigma'}, \overline{\sigma}(\psi_0))}(\overline{\sigma * \Psi_0}(E_1)) \tag{5.92}$$

Thirdly, we apply $\overline{\sigma}$ to the both sides of the equation $E = \overline{id_\Sigma, \psi_0}(E_2)$ to get

$$\overline{\sigma}(E) = \overline{\sigma}(\overline{id_\Sigma, \psi_0}(E_2)) \tag{5.93}$$

By applying lemma 5.33 to the right-hand side of 5.93 we get

$$\overline{\sigma}(E) = \overline{(id_\Sigma, \psi_0) \,;\, \sigma}(E_2) \tag{5.94}$$

By applying lemma 5.36 to the right-hand side of 5.94 we get

$$\overline{\sigma}(E) = \overline{\sigma * \Psi_0 \,;\, (id_{\Sigma'}, \overline{\sigma}(\psi_0))}(E_2) \tag{5.95}$$

By applying lemma 5.33 to the right-hand side of 5.95 we get

$$\overline{\sigma}(E) = \overline{(id_{\Sigma'}, \overline{\sigma}(\psi_0))}(\overline{\sigma * \Psi_0}(E_2)) \tag{5.96}$$

Then, by applying the rule *mapConstant* to 5.88, 5.92 and 5.96, we get

$$\vdash id_{\Sigma'}, \overline{\sigma}(\psi_0), x := \overline{\sigma}(E) \ : \ \Sigma', \overline{\sigma}(\Psi_0), x : \overline{\sigma * \Psi_0}(E_1)[= \overline{\sigma * \Psi_0}(E_2)] \to \Sigma' \tag{5.97}$$

Thus, applying the definition of $\overline{\sigma}$ in 5.97 yields

$$\vdash id_{\Sigma'}, \overline{\sigma}(\psi_0, x := E) \ : \ \Sigma', \ \overline{\sigma}(\Psi_0, x : E_1[= E_2]) \to \Sigma'$$

as desired.

- **Case** $\psi = \underline{\psi_0, \iota := I}$**:** Then we have the following derivation of the second premise of (8) via the rule *mapInstance*.

$$\frac{\vdash id_\Sigma, \psi_0 \ : \ \Sigma, \Psi_0 \to \Sigma \quad \Sigma \vdash I : \overline{id_\Sigma, \psi_0}(\Phi)}{\vdash id_\Sigma, (\psi_0, \iota := I) \ : \ \Sigma, (\Psi_0, \iota : \Phi) \to \Sigma} \ \mathit{mapInstance}$$

We want to prove that $\vdash id_{\Sigma'}, \overline{\sigma}(\psi_0, \iota := I) \ : \ \Sigma', \overline{\sigma}(\Psi_0, \iota : \Phi) \to \Sigma'$.

By applying the induction hypothesis for (8) and (2) to the first two premises above and by lemma 5.33 and lemma 5.36, we get

$$\vdash id_{\Sigma'}, \overline{\sigma}(\psi_0) \ : \ \Sigma', \overline{\sigma}(\Psi_0) \to \Sigma' \tag{5.98}$$

$$\Sigma' \vdash \overline{\sigma}(I) : \overline{id_{\Sigma'}, \overline{\sigma}(\psi_0)}(\overline{\sigma * \Psi_0}(\Phi)) \tag{5.99}$$

Then by applying the rule *mapInstance* to 5.98, 5.99, and the definition of $\overline{\sigma}$, we get

$$\vdash id_{\Sigma'}, \overline{\sigma}(\psi_0, \iota := I) \ : \ \Sigma', \overline{\sigma}(\Psi_0, \iota : \Phi) \to \Sigma' \tag{5.100}$$

as desired.

- Case $\psi = \underline{\psi_0, \varphi := \Phi}$ does not exist since Ψ is simple.

□

Preservation under Elaboration We have introduced theory elaboration in definition 5.20. Here we state that the elaboration of a well-formed theory Σ yields a well-formed theory.

Theorem 5.41. *Assume* $\vdash \Sigma$ *Theory. Then we have* $\vdash \mathcal{E}(\Sigma)$ *Theory.*

Proof. Assume a TFI/$\mathcal{F}$ theory Σ. By definition of elaboration, we know that the theory family declarations $\varphi : \tau = \Phi$ and expression declarations $x : E[= E']$ remain unmodified. It suffices to show that the elaboration of instance declarations $i : \Phi$ yield well-formed declarations. Since Σ is a well-formed theory, we know that all the instance declarations $\iota : \Phi$ in Σ are well-formed as well. Then, by the rule *elabType*, we know that the projections $\iota.x$ of ι are well-formed. □

5.3 Discussion

In this chapter, we have introduced our foundation-independent framework TFI for representing declarative languages and their translations. The main novelty of TFI is the notion of theory families, which we have introduced as a new primitive for declarative frameworks. This is orthogonal to the well-established primitives that formal languages have.

Theory families are in principle very similar to record types in the sense that both of them are essentially a "container" of a list of declarations. The type system of TFI clearly separates theory families from the type system of the foundations that can be added in TFI. Thus, abstractions of a foundation cannot bind over theory families. TFI uses a kinding relation to separate theory families from the types of a foundation. In practice, it

could be useful to allow instances as parameters of a theory family (see concrete examples in Chapter 6).

The main difference between record types and theory families is that theory families may have type declarations in their body, which record types can only do in the presence of universes such as in the calculus of constructions [CH88] and Martin-Löf type theory [ML74].

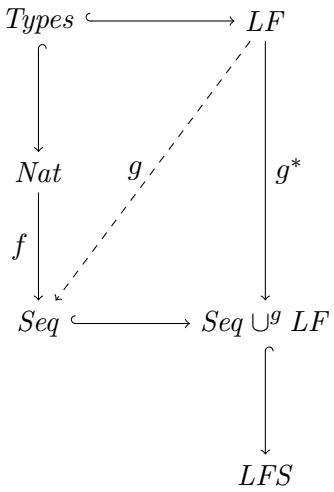
A necessary characteristic of TFI is that definitional equality on theory families is not preserved along theory morphisms (recall rule (5) of theorem 5.40). This relaxation is the key feature that allows adequate theory translations in chapter 7.

Chapter 6

A Logical Framework with Sequences

In section 3.5 we mentioned that sequences of expressions are often needed to define the declaration patterns of a declarative language, e.g., for n-ary function symbol declarations in first-order logic, and discussed that sequences should be introduced at the level of foundations.

In this chapter we develop two new foundations within TFI specifically for that purpose. We develop the foundation *Seq* (section 6.1), which represents sequences as first class citizens in a type theory, and the foundation *LFS* (section 6.2), which is based on a type theory with dependent function spaces of flexible arity and sequence arguments.



Both *Seq* and *LFS* are foundations in the sense of definition 4.8 from chapter 4. *Seq* is an extension of the foundation *Nat* from example 4.11, and introduces new primitives to construct sequence expressions and ellipses. *LFS* is intuitively an extension of the foundation *LF* from example 4.10 with the sequence expressions from *Seq*.

The diagram on the right illustrates the modular structure of the formation of these two new foundations where normal $\longrightarrow$ and dashed arrow $\dashrightarrow$ denote total and partial foundation morphisms from definition 4.18, respectively. The hooked arrow $\hookrightarrow$ denotes inclusion of foundations as in definition 4.14. We will explain the diagram in detail in the following sections.

6.1 Sequences

6.1.1 Syntax

Sym^{Seq} introduces the following symbols as primitives for constructing sequence expressions and ellipses: `empty`, `concat`, `index`, `ellipsis`, `length` and `seqtype`. It also contains the same symbols in Sym^{Nat} from example 4.11 except for the symbol `type` $\in Sym^{Nat}$. The relation between the foundations *Nat* and *Seq* is given by the foundation morphism $f : Sym^{Nat} \to Exp^{Seq}$ that maps the symbol `type` to the expression @(`seqtype`; 1) and all other symbols in Sym^{Nat} themselves.

$$f(x) = \begin{cases} @(\texttt{seqtype}; 1) & \text{if } x = \texttt{type} \\ x & \text{otherwise} \end{cases} \tag{6.1}$$

Before we discuss the *Seq* expressions formed over Sym^{Seq} in detail, we will introduce notations for them in figure 6.1, and we use these notations in the remainder of this thesis in addition to the notations introduced in figure 4.8. We use the meta-variables E, E' for arbitrary expressions formed over Sym^{Seq}.

Expression	Notation
$\texttt{empty}$	$\cdot$
$@(\texttt{concat}; E, E')$	E, E'
$@(\texttt{index}; E, E')$	$E_{E'}$
$\beta(@(\texttt{ellipsis}; 1, E)\,;\, x\,;\, E')$	$[E']^{E}_{x=1}$
$@(\texttt{length}; E)$	$\lvert E \rvert$
$@(\texttt{seqtype}; E)$	$\texttt{type}^{E}$

Figure 6.1: Notations for *Seq*-Expressions

We classify *Seq*-expressions into three syntactic categories: *term sequences*, *type sequences* and *kinds*. As we will discuss in full detail in section 6.1.2, term sequences are typed by type sequences; and type sequences are kinded. The grammar in figure 6.2 shows the formation of the expressions in each syntactic category. We will use the meta-variables S, T for term sequences, U, V for type sequences and K for kinds.

Kinds	K	$::=$	$\texttt{type}^{S}$
Type Sequences	$U, V, \ldots$	$::=$	$\cdot \mid U, V \mid U_S \mid [U]^{S}_{x=1} \mid a \mid \texttt{nat} \mid S \leq T$
Term Sequences	$S, T, \ldots$	$::=$	$\cdot \mid S, T \mid S_T \mid [S]^{T}_{x=1} \mid x \mid 0 \mid 1 \mid S + T \mid S - T \mid \lvert S \rvert \mid \lvert U \rvert \mid \lvert K \rvert$

Figure 6.2: Grammar of *Seq*

Term sequences S, T are formed from

- the **empty term sequence** $\cdot$,
- the **concatenation** S, T of two term sequences S and T,
- the **n-th element** S_n of a term sequence S,
- the **term sequence ellipsis** $[S(x)]^{n}_{x=1}$ that takes an argument $n : \texttt{nat}$ and binds the symbol $x : \texttt{nat}$ in S,
- **term sequence symbols** x,
- the natural numbers 0 and 1,
- the sum $n + m$ of two natural numbers n and m,
- the difference $n - m$ of the natural number n from m,
- the **lengths** $|S|$, $|U|$ and $|K|$ of sequences S, U and K, respectively.

$[S(x)]^{n}_{x=1}$ is a **sequence ellipsis** constructor for term sequences. The intended meaning of $[S(x)]^{n}_{x=1}$ is that it reduces to the term sequence $S(1), \ldots, S(n')$ by substituting for x in S the natural numbers $1, \ldots, n'$ whenever n reduces to a natural number n'. We

can also think of $[S(x)]_{x=1}^{n}$ as a special map operator that applies to each element of the sequence $1, \ldots, n'$ the mapping $x \mapsto S(x)$.

Type sequences U, V are formed analogously. In particular, we have **type sequence symbols** a, the type `nat` of natural numbers, the type $m \leq n$ for the less-than or equal relation between two natural numbers m and n, and the analogous constructors for **type sequence concatenation** U, V as well as the **type sequence ellipsis** constructor $[U(x)]_{x=1}^{n}$.

Notation 6.1. We will write E^n for the sequence $[E]_{x=1}^{n}$ where x does not occur free in E. Since we use both the superscript E^n and the subscript E_n notation for *Seq*-expressions, we will use the notation $E^{(n)}$ whenever we want to use enumerated meta-variables as in $E^{(1)}, \ldots, E^{(n)}$.

Note that the base kind `type` of *Types* is not part of *Seq*. Instead, we have **kinds** $\texttt{type}^n$ for $n : \texttt{nat}$. The intuition for sequence kinds is that type sequences of length n are kinded by $\texttt{type}^n$.

Notation 6.2. We will write `type` for $\texttt{type}^1$.

Examples Now we will fortify our intuitions by a number of useful examples, which we will use as abbreviations for a number of common operations on sequences.

First, we define the **reversal** of a sequence expression:

Example 6.3 (Revert). The following operation takes a sequence expression E as argument and reverts the order of the elements in E:

$$\texttt{reverse}\, E = [E_{|E|+1-x}]_{x=1}^{|E|}.$$

Next, we define the prefix operator that returns the first n elements of a sequence:

Example 6.4 (Prefix). The operator `prefix` takes a sequence expression E, a natural number n and a proof q that $n \leq |E|$ and returns the first n elements in E:

$$\texttt{prefix}\, E\, n\, q = [E_x]_{x=1}^{n}.$$

If the argument n is 0, then $[E_x]_{x=1}^{0}$ reduces to the empty sequence (as we will mention in the conversion rules for *Seq* in section 6.1.3), and $\texttt{prefix}\, E\, 0\, q$ returns the empty sequence.

Similarly, we can now define the suffix operator that returns the elements of a sequence starting from its $n+1$-th element:

Example 6.5 (Suffix). The operator `suffix` takes a sequence expression E, a natural number n and a proof q that $n \leq |E|$ and returns the sequence of elements of E starting from index $n+1$:

$$\texttt{suffix}\, E\, n\, q = [E_{x+n}]_{x=1}^{|E|-n}$$

Our sequence ellipsis constructor $[E]_{x=1}^{n}$ has a fixed starting index, namely 1. Using the reversal of a sequence, it is straightforward to define a new ellipsis operator that constructs $E(n), \ldots, E(1)$. We give that operator a new notation $[E]_{1}^{x=n}$:

Example 6.6 (Reversal of Ellipsis).

$$[E(x)]_{1}^{x=n} \;=\; \texttt{reverse}\, [E(x)]_{x=1}^{n}$$

for $n \leq |E|$.

More generally, we can define **generalized sequence ellipsis** $E(m), \ldots, E(n)$ for $m \leq n$ as follows:

Example 6.7 (Generalized Sequence Ellipses).

$$[E(x)]_{x=m}^{n} \;=\; [E(m+x-1)]_{x=1}^{n-m+1}$$

for $m \leq n$ and $n \leq |E|$.

Similarly, we can define the reversal of the generalized ellipsis $E(n), \ldots, E(m)$ for $m \leq n$:

$$[E(x)]_{m}^{x=n} \;=\; \texttt{reverse}\,[E(x)]_{x=m}^{n}$$

for $m \leq n$ and $n \leq |E|$.

6.1.2 Type System

In figure 6.3 we give the typing and equality judgments for well-formed *Seq* expressions. Note that these judgments are special cases of the foundational judgments in figure 4.2. The judgments for well-formed *Seq* theories and theory morphisms are inherited from our framework TFI (see figure 5.3).

Judgment	Meaning
$\Sigma \vdash E\ \mathit{Inhabitable}$	E is an inhabitable sequence expression over Σ.
$\Sigma \vdash S : U$	S is a well-formed term of type U over Σ.
$\Sigma \vdash U : K$	U is a well-formed type of kind K over Σ.
$\Sigma \vdash K : \texttt{kind}$	K is a well-formed kind over Σ.
$\Sigma \vdash S \doteq T$	Term sequence S is equal to T over Σ.
$\Sigma \vdash U \doteq V$	Type sequence U is equal to V over Σ.
$\Sigma \vdash K \doteq L$	Kind K is equal to L over Σ.

Figure 6.3: Judgments of *Seq*

We use the judgment $\Sigma \vdash S : U$ to denote the typing relation between well-formed term sequences S and well-formed type sequences U over Σ. We will often write $\Sigma \vdash S : U$ (where $U : \texttt{type}^n$ is implied) as $\Sigma \vdash S : U : \texttt{type}^n$ by giving the kind $\texttt{type}^n$ in gray color to keep track of the length of S and U for the purpose of documentation. This is redundant because the length is statically known, but it helps with the readability of the inference rules. Most importantly, term sequences are typed by type sequences of the same length, and type sequences are kinded by $\texttt{type}^n$, where n is their length.

The judgment $\Sigma \vdash U : K$ denotes the kinding relation between well-formed type sequences U and well-formed kinds K over Σ. And the judgment $\Sigma \vdash K : \texttt{kind}$ denotes well-formed kinds K over Σ.

Moreoever, we have a separate equality judgment for well-formed term sequences, type sequences and kinds, respectively.

Now we define the set $\mathit{Rules}^{\mathit{Seq}}$ of the inference rules of *Seq*. The foundational rules in $\mathit{Rules}^{\mathit{Seq}}$ are

- the translations $\overline{f}(\mathcal{R})$ of the rules $\mathcal{R}$ in $\mathit{Rules}^{\mathit{Nat}}$ except for the rules *type* and *typeInh* from figure 4.5, along the foundation morphism f from 6.1,
- the typing rules in figure 6.4, figure 6.5, figure 6.6, figure 6.7, figure 6.8 and

- the equality rules in figure 6.9 and figure 6.10.

It is easy to check that $\overline{f}$ maps each rule in $Rules^{Nat}$ to a derivable rule in Seq.

Inhabitable Expressions Inhabitable Seq-expressions are type sequences $U : type^n$. We give this as a rule in figure 6.4.

$$\frac{\Sigma \vdash U : \mathtt{type}^n}{\Sigma \vdash U\ Inhabitable}\ typeInhab$$

Figure 6.4: Inhabitable Seq-Expressions

$$\frac{\Sigma \vdash n : \mathtt{nat} : \mathtt{type}}{\Sigma \vdash \mathtt{type}^n : \mathtt{kind}}\ seqKind$$

$$\frac{\vdash \Sigma\ Theory}{\Sigma \vdash\ \cdot : \mathtt{type}^0}\ emptyType$$

$$\frac{\Sigma \vdash U : \mathtt{type}^m \quad \Sigma \vdash V : \mathtt{type}^n}{\Sigma \vdash\ U, V\ : \mathtt{type}^{m+n}}\ typeConcat$$

$$\frac{\vdash \Sigma\ Theory}{\Sigma \vdash \cdot : \cdot : \mathtt{type}^0}\ emptyTerm$$

$$\frac{\Sigma \vdash S : U : \mathtt{type}^m \quad \Sigma \vdash T : V : \mathtt{type}^n}{\Sigma \vdash\ S, T\ :\ U, V\ : \mathtt{type}^{m+n}}\ termConcat$$

Figure 6.5: Sequence Introduction Rules

Sequence Introduction Rules In figure 6.5 we give the typing rules sequence introduction.

The rule *seqKind* makes $type^n$ a valid kind. Then the empty type sequence $\cdot$ is kinded by $\mathtt{type}^0$, and the empty term sequence $\cdot$ is typed by the empty type sequence.

Since $\mathtt{type}^n$ is a kind for every $n : \mathtt{nat}$, type sequences $U : \mathtt{type}^n$ represent a sequence of types of length n. Therefore, in the case of type sequences concatenation, the kind $\mathtt{type}^{m+n}$ of the concatenation U, V respects the sum of the lengths of U and V. Note that V does not depend on U. Similarly, the type U, V of the term sequence concatenation S, T respects the sum of the lengths of S and V.

Example 6.8. We have $0, 1 \ :\ \mathtt{nat}, \mathtt{nat} \ :\ \mathtt{type}^2$. Recall notation 4.12 for the use of meta-level natural numbers such as 2 to denote the natural number expressions.

Sequence Elimination Rules In figure 6.6 we introduce the rules for sequence elimination.

The intuition behind the rule *termIndex* is that term sequences S are typed by type sequences U index-wise: Each element S_x of S at index x is typed by the element U_x of U at the same index. Moreover, both for term and type sequences, E_x is only well-formed if x ranges from 1 to the length of E. Therefore, in both rules *termIndex* and *typeIndex* we add the necessary premises to implement the index-within-bounds check.

Note that for type sequences $U : \mathtt{type}^n$, U_x is always a type.

$$\frac{\Sigma \vdash S : U : \mathtt{type}^{|S|} \quad \Sigma \vdash \_ : 1 \leq x : \mathtt{type} \quad \Sigma \vdash \_ : x \leq |S| : \mathtt{type}}{\Sigma \vdash S_x : U_x : \mathtt{type}}\ \mathit{termIndex}$$

$$\frac{\Sigma \vdash U : \mathtt{type}^n \quad \Sigma \vdash \_ : 1 \leq x : \mathtt{type} \quad \Sigma \vdash \_ : x \leq n : \mathtt{type}}{\Sigma \vdash U_x : \mathtt{type}}\ \mathit{typeIndex}$$

Figure 6.6: Sequence Elimination Rules

$$\frac{\Sigma \vdash n : \mathtt{nat} : \mathtt{type} \quad \Sigma, x : \mathtt{nat}, x^* : 1 \leq x, x_* : x \leq n \vdash S : U : \mathtt{type} \quad \Sigma \vdash U : \mathtt{type}}{\Sigma \vdash [S]_{x=1}^n : [U]_{x=1}^n : \mathtt{type}^n}\ \mathit{termEll}$$

$$\frac{\Sigma \vdash n : \mathtt{nat} : \mathtt{type} \quad \Sigma, x : \mathtt{nat}, x^* : 1 \leq x, x_* : x \leq n \vdash U : \mathtt{type}}{\Sigma \vdash [U]_{x=1}^n : \mathtt{type}^n}\ \mathit{typeEll}$$

Figure 6.7: Ellipsis Introduction Rules

Ellipsis Introduction Rules In figure 6.7 we give the rules for ellipsis introduction. The intuition here is that term sequence ellipses $[S]_{x=1}^n$ are typed by type sequence ellipses $[U]_{x=1}^n$ of the same length, and both rules *termEll* and *typeEll* restrict the body of the ellipsis constructor to length 1.

Note that $[E]_{x=1}^n$ actually binds three variables in E: The index x and two assumptions x_* and x^* that guarantee that x is within 1 and n. Here the framework reserves the fresh names x_* and x^* for every name x to avoid the capturing of the names in Σ that E is formed over. These assumptions can be used later on to discharge the proof obligations posited by the rules *termIndex* and *typeIndex* and by the subtraction of natural numbers, e.g., the derivation of the judgment

$$\Sigma, x : \mathtt{nat}, x^* : 1 \leq x, x_* : x \leq n \vdash S_x : U_x : \mathtt{type}$$

as a premise in the rule *termIndex* for deriving $\Sigma \vdash [S_x]_{x=1}^n : [U_x]_{x=1}^n : \mathtt{type}^n$.

$$\frac{\Sigma \vdash S : U}{\Sigma \vdash |S| : \mathtt{nat}}\ termLength \qquad \frac{\Sigma \vdash U : K}{\Sigma \vdash |U| : \mathtt{nat}}\ typeLength \qquad \frac{\Sigma \vdash K : \mathtt{kind}}{\Sigma \vdash |K| : \mathtt{nat}}\ kindLength$$

Figure 6.8: Introduction Rules for the Length Operator

The Length Operator We give the introduction rules for the length operator in figure 6.8 and give its definition in figure 6.9. The length of a sequence expression is a natural number expression in *Seq*:

In particular, the empty sequence has length 0 and the length of the concatenation of two sequences $\mid E, F \mid$ is equal to the sum of the length of each sequence E and F. The length of term sequence name $x : U$ is determined by the length of the sequence U it is typed by. The length of a type sequence name is determined similarly. The length of an ellipses $[E]_{x=1}^{n}$ is n. And the length of any element E_n of a sequence expression E is 1. Finally, every natural number expression $n : \mathtt{nat}$ has length 1.

$$\frac{}{\Sigma \vdash |\cdot| \doteq 0}\ lenEmpty \qquad \frac{\Sigma \vdash x : U}{\Sigma \vdash |x| \doteq |U|}\ lenTermSym \qquad \frac{\Sigma \vdash a : K}{\Sigma \vdash |a| \doteq |K|}\ lenTypeSym$$

$$\frac{\Sigma \vdash S : U \quad \Sigma \vdash n : \mathtt{nat}}{\Sigma \vdash |[S]_{x=1}^{n}| \doteq n}\ lenTermEll \qquad \frac{\Sigma \vdash U : K \quad \Sigma \vdash n : \mathtt{nat}}{\Sigma \vdash |[U]_{x=1}^{n}| \doteq n}\ lenTypeEll$$

$$\frac{\Sigma \vdash S : U \quad \Sigma \vdash n : \mathtt{nat}}{\Sigma \vdash |S_n| \doteq 1}\ lenTermInd \qquad \frac{\Sigma \vdash U : K \quad \Sigma \vdash n : \mathtt{nat}}{\Sigma \vdash |U_n| \doteq 1}\ lenTypeInd$$

$$\frac{\Sigma \vdash S : U \quad \Sigma \vdash T : V}{\Sigma \vdash \mid S, T \mid \doteq |S| + |T|}\ lenTermConcat \qquad \frac{\Sigma \vdash U : K \quad \Sigma \vdash V : L}{\Sigma \vdash \mid U, V \mid \doteq |U| + |V|}\ lenTypeConcat$$

$$\frac{\Sigma \vdash S : \mathtt{nat}}{\Sigma \vdash |S| \doteq 1}\ lenNat \qquad \frac{\Sigma \vdash n : \mathtt{nat}}{\Sigma \vdash |\mathtt{type}^n| \doteq n}\ lenKind$$

Figure 6.9: Equality Rules for the Length Operator

Now we can state the following properties using the length operator: Term sequences are typed by type sequence of the same length, and type family sequences are kinded by kinds of the same length.

Lemma 6.9. *The following statements hold:*

- *If* $\Sigma \vdash S : U$, *then* $\Sigma \vdash |S| \doteq |U|$.

- *If* $\Sigma \vdash U : K$, *then* $\Sigma \vdash |U| \doteq |K|$.

Proof. The proof proceeds by a straightforward induction on derivations. □

6.1.3 Conversions

Now we present the conversion rules for *Seq*-expressions in figure 6.10. These rules capture the intended semantics of the two operations $[E]_{x=1}^{n}$ and E_x.

$$\frac{\Sigma, x : \texttt{nat} \vdash x : \texttt{nat}}{\Sigma \vdash E \doteq [E_x]_{x=1}^{|E|}}\ \textit{extensionality}$$

$$\frac{\Sigma \vdash n \doteq 1 + \ldots + 1}{\Sigma \vdash [E]_{x=1}^{n} \doteq [1/x]E, \ldots, [n/x]E}\ \textit{seqBindElim}$$

$$\frac{\Sigma \vdash E \doteq E^{(1)}, \ldots, E^{(n)} \quad \Sigma \vdash |E^{(y)}| \doteq 1 \text{ for } y = 1, \ldots, n \quad \Sigma \vdash x \doteq 1 + \ldots + 1}{\Sigma \vdash E_x \doteq E^{(x)}}\ \textit{seqIndexElim}$$

Figure 6.10: Conversion Rules

The rule *extensionality* is an expansion rule and corresponds to η-expansion in simple type theory. *seqBindElim* and *seqIndexElim* are rule schemes that denote a family of rules consisting of one rule for every meta-level natural number n.

For instance, *seqBindElim* denotes the following rules:

- Case $n = 0$:
$$\frac{\Sigma \vdash n \doteq 0}{\Sigma \vdash [E]_{x=1}^{0} \doteq \cdot}$$
- Case $n = 1$:
$$\frac{\Sigma \vdash n \doteq 1}{\Sigma \vdash [E]_{x=1}^{1} \doteq [1/x]E}$$
- Case $n = 2$:
$$\frac{\Sigma \vdash n \doteq 1 + 1}{\Sigma \vdash [E]_{x=1}^{2} \doteq [1/x]E, [1 + 1/x]E}$$

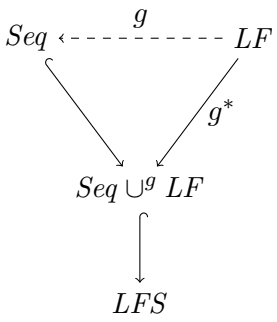
  and so on.

This means that ellipses $[E]_{x=1}^{n}$ are expanded if n is normal. Note that the case $n = 0$ of *seqBindElim* means that $[E]_{x=1}^{0}$ normalizes to the empty sequence.

In *seqIndexElim*, $E^{(1)}, \ldots, E^{(n)}$ are meta-variables for expressions, x is normal and E is a sequence expression whose elements have length 1. In that case, sequence elements can be projected.

6.2 LF with Sequences

In this section we take the generative union of the foundations *Seq* and *LF* along the partial foundation morphism $g : Sym^{LF} \to Exp^{Seq}$, where g maps the symbol $\texttt{type} \in Sym^{LF}$ to the *Seq*-expression $\texttt{type}^1$. By definition 4.19 of generative unions, the foundation morphism $g^* : Sym^{LF} \to Exp^{Seq \cup^g LF}$ is defined as follows:



$$g^*(x) = \begin{cases} \texttt{type}^1 & \text{if } x = \texttt{type} \\ x & \text{otherwise} \end{cases} \tag{6.2}$$

This modular construction allows combining the foundations *Seq* and *LF* via an instantiation that maps all occurrences of the symbol `type` in the rules of *LF* with the expression $\texttt{type}^1$.

Then we extend the generative union $Seq \cup^g LF$ with one more primitive symbol: `compose`, a flexary symbol for the composition of a sequence of n functions. We will call the resulting foundation *LFS*.

`compose` adds new expressions of the form $@(\texttt{compose}; E)$ to the syntax of *LFS* and is denoted as $;E$.

6.2.1 Syntax

As a result of modular construction in the diagram above, Sym^{LFS} consists of the symbols of Sym^{Seq}, the symbols λ, Π, `app` of Sym^{LF} and the symbol `compose`. Note that the symbol `kind` of Sym^{LF} does not occur twice in Sym^{LFS} due to set union.

We give the full grammar of *LFS* in figure 6.11. All productions for *LF* are retained but generalized to sequences. Here we mention the additional formations only:

Kinds	K	$::=$	$\texttt{type}^S \mid \Pi x{:}A.\, K$
Type Seq. Families	$U, V \ldots$	$::=$	$a \mid \Pi x{:}U.\, V \mid \lambda x : U.\, V \mid U\, S$
			$\mid \cdot \mid U, V \mid U_S \mid [U]_{x=1}^{S}$
			$\mid \texttt{nat} \mid S \leq T$
Term Sequences	$S, T \ldots$	$::=$	$x \mid \lambda x : U.\, S \mid S\, T$
			$\mid \cdot \mid S, T \mid S_T \mid [S]_{x=1}^{T} \mid ;S$
			$\mid 0 \mid 1 \mid S + T \mid S - T$
			$\mid \lvert S \rvert \mid \lvert U \rvert \mid \lvert K \rvert$

Figure 6.11: *LFS* Grammar

More specifically, **term sequences** in *LFS* are formed from, in addition to term sequences in *Seq*,

- **term sequence abstractions** $\lambda x : U.\, S$ that bind symbols x in term sequences S,
- **term sequence applications** $S\, T$ of term sequences S to term sequences T,
- the **flexary composition** $;S$ of elements of S.

Type sequence families are formed from, in addition to type sequences in *Seq*,

- **dependent type sequences** $\Pi x{:}U.\, V$ that bind sequence variables $x : U$ in V,
- **type sequence abstractions** $\lambda x : U.\, V$ that bind sequence variables $x : U$ in V,

- **type sequence applications** $U\,S$ of type family sequences U to term sequences S.

The intuition behind the primitives of LF that are now flexary in LFS is the following: Both λ and Π binders for sequences are **flexary variable binders**, i.e., bindings $x : U$ formalize variable sequences. For example, the binding $x : U^{(1)}, \ldots, U^{(n)}$ corresponds to binding $x^{(1)} : U^{(1)}, \ldots, x^{(n)} : U^{(n)}$. Then both binders can return a sequence expression. Accordingly, sequence applications $f\,T$ are **flexary applications** that apply a function f to an argument sequence T. Likewise, the function f can be a sequence expression: the application $(f^{(1)}, \ldots, f^{(n)})\,T$ returns the sequence of applications $(f^{(1)}\,T), \ldots, (f^{(n)}\,T)$.

Moreover, the intended meaning of **flexary composition** $;S$ is that it takes a sequence of functions of length 1 and returns their composition. Thus, $;(f^{(1)}, \ldots, f^{(n)})\,s$ reduces to $f^{(n)}\,(\ldots(f^{(1)}\,s)\ldots)$. Notably, this is more general than a fold operator because the type of each $f^{(i)}$ may depend on i.

Kinds K are formed, in addition to the base kind in figure 6.2, from $\Pi x{:}U.\,K$.

Examples Now, we give more operations on sequences.

We define the map operator that maps each element x of a sequence to $f(x)$ for a unary function f.

Example 6.10 (Map). The operator `map` that takes a function $f : \Pi x : U.\,E'$, a sequence expression $E : E'$, and returns the sequence $f(E^{(1)}), \ldots, f(E^{(n)})$ whenever E reduces to $E^{(1)}, \ldots, E^{(n)}$:

$$\texttt{map}\,f\,E \;=\; [f\,E_x]_{x=1}^{|E|}$$

Next, we can use our flexary composition and the sequence ellipsis constructor to define the fold operator:

Example 6.11 (Fold). The left folding operator is defined as:

$$\texttt{foldl}\,f\,E\,s \;=\; (;[\lambda x : A.\,f\,x\,E_i]_{i=1}^{n})\,s$$

Thus $\texttt{foldl}\,f\,E\,s$ reduces to $(f\ldots(f\,(f\,a\,E_1)\,E_2)\ldots E_n)$ for a folding function $f : A \to B \to A$, a start element $s : A$ and a sequence $E : B^n$.

Similarly, we define the right folding operator as:

$$\texttt{foldr}\,f\,E\,s \;=\; \texttt{foldl}\,f\,(\texttt{reverse}\,E)\,s$$

Using sequence operations, we can define flexary versions of mathematical operators. In particular, we can define the flexary version of any associative binary operator using the fold operation:

Example 6.12 (Flexary Operators). Consider a monoid with carrier M : `type`, binary operation $\circ : M \to M \to M$ and unit element $e : M$. Then we define the flexary operation $\circ^*$ as follows:

$$\circ^* \;:\; \Pi n : \texttt{nat}.\,M^n \to M \;=\; \lambda n : \texttt{nat}.\,\lambda x : M^n.\,\texttt{foldl}\,\circ\,x\,e$$

This immediately yields the power operator:

$$\mathit{power} \;:\; M \to \texttt{nat} \to M \;=\; \lambda x : M.\,\lambda n : \texttt{nat}.\,\circ^*\,x^n$$

6.2.2 Type System

In addition to the judgments in figure 6.3, *LFS* uses the judgment $\Sigma \vdash U\ \mathit{Quantifiable}$.

The type system of *LFS* inherits all the rules from Rules^{Seq}, the typing rules *depType*, *abstr*, *app* of *LF* in figure 4.6 for Π, λ and application, respectively, the conversion rules of *LF* in figure 4.7 via the generative union, and introduces the rule *typeQuan* in figure 6.12, the rule *composition* in figure 6.13, and the conversion rules in figure 6.14.

The foundation morphism $g' : \mathit{Sym}^{LF} \to \mathit{Exp}^{Seq \cup^g LF}$ maps the rules *type*, *typeInh* in figure 4.5 and *typeQuan* in figure 4.6, respectively, to the derivation cases of the rules *seqKind* from figure 6.5, *typeInhab* from figure 6.4 and *seqTypeQuan* in figure 6.12 for $n = 1$. Note that g' maps the rule *kindInh* of *LF* in figure 4.5 to itself.

The rule *seqTypeQuan* permits the λ- and Π-binders to bind term sequences $x : U$.

$$\frac{\Sigma \vdash U : \mathtt{type}^n}{\Sigma \vdash U\ \mathit{Quantifiable}}\ \mathit{seqTypeQuan}$$

Figure 6.12: Quantifiable Expressions

The rule *composition* handles $;S$ by checking that the functions in S are actually composable. This is easiest if we restrict attention to the composition of simple functions. In the case where $n = 0$, we have $U : \mathtt{type}$ and the empty sequence $S : \cdot$ of functions. Applying the flexary composition to the empty sequence gives the identity function on U (note that the element U_1 of $U : \mathtt{type}$ at index 1 is equal to U).

$$\frac{\Sigma \vdash U : \mathtt{type}^{n+1} \quad \Sigma \vdash S : [U_i \to U_{i+1}]_{i=1}^{n}}{\Sigma \vdash ; S : U_1 \to U_{n+1}}\ \mathit{composition}$$

Figure 6.13: Typing Rule for Flexary Function Composition

6.2.3 Conversions

The rules in figure 6.14 define the length operator for the new symbols in *LFS*:

$$\frac{}{\Sigma \vdash |\lambda x : U.\, E| \doteq |E|}\ \mathit{length}\lambda \qquad \frac{}{\Sigma \vdash |\Pi x : U.\, E| \doteq |E|}\ \mathit{length}\Pi$$

$$\frac{}{\Sigma \vdash |f\ S| \doteq |f|}\ \mathit{lengthApp} \qquad \frac{}{\Sigma \vdash |; S| \doteq 1}\ \mathit{lengthComp}$$

Figure 6.14: Equality Rules for the Length Operator

Functions $\lambda x : U.\, E$ and Π-expressions are allowed to return sequences. In both expressions, their length is determined by the length of their bodies. Functions f an be sequences themselves. In that case, the length of an application $f\ S$ is determined by the

length of f. Function composition is restricted to a sequence of functions of length 1 and the resulting function $;S$ has length 1.

Now we give the conversion rules for *LFS*-expressions that mix expressions from *Seq* and *LF*. Binders distribute over sequences (figure 6.15 and figure 6.16):

$$\frac{}{\Sigma \vdash \Pi x : \cdot.\, E \doteq [\cdot/x]E}\ emtpy\Pi$$

$$\frac{}{\Sigma \vdash \Pi x : (U, V).\, E \doteq \Pi y : U.\, \Pi z : V.\, [(y, z)/x]E}\ seq\Pi$$

$$\frac{}{\Sigma \vdash \lambda x : \cdot.\, E \doteq [\cdot/x]E}\ emtpy\lambda$$

$$\frac{}{\Sigma \vdash \lambda x : (U, V).\, E \doteq \lambda y : U.\, \lambda z : V.\, [(y, z)/x]E}\ seq\lambda$$

Figure 6.15: Conversion Rules for *LFS*-Binders — the Binding

$$\frac{}{\Sigma \vdash \Pi x : U.\, (E, F) \doteq (\Pi x : U.\, E), (\Pi x : U.\, F)}\ seqBody\Pi$$

$$\frac{}{\Sigma \vdash \Pi x : U.\, \cdot \doteq \cdot}\ emtpyBody\Pi$$

$$\frac{}{\Sigma \vdash \lambda x : U.\, (E, F) \doteq (\lambda x : U.\, E), (\lambda x : U.\, F)}\ seqBody\lambda$$

$$\frac{}{\Sigma \vdash \lambda x : U.\, \cdot \doteq \cdot}\ emtpyBody\lambda$$

Figure 6.16: Conversion Rules for *LFS*-Binders — the Scope

We have the following rules for sequence application in figure 6.17. Functions f take a sequence of arguments S, T by curring. In particular, the application of a function f to the empty sequence returns f itself. The application of a sequence of functions (f, g) to an expression E is a sequence of applications $(f\,E), (g\,E)$. The application of the empty sequence to an expression returns the empty sequence.

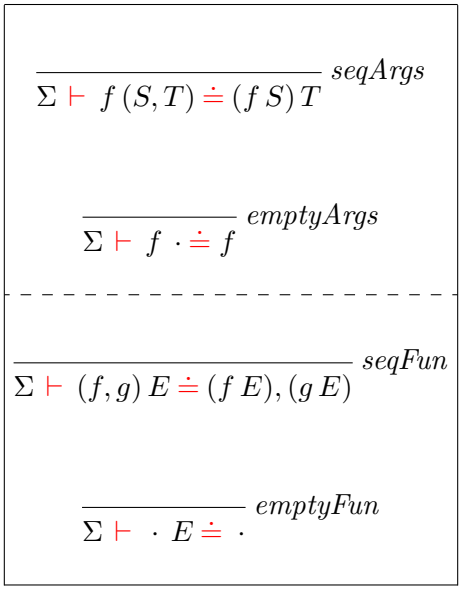


Figure 6.17: Conversion Rules for Function Applications

Figure 6.18 gives the conversion rules for that combine the index operator with binders and application:

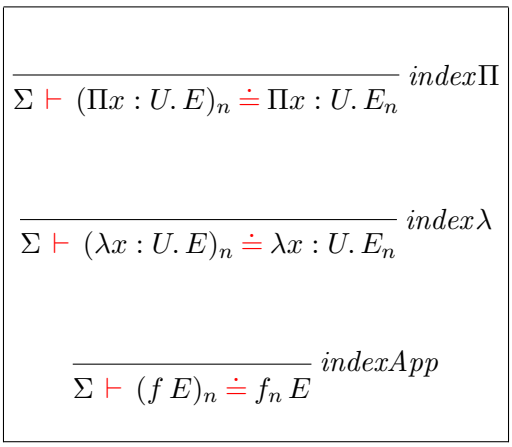


Figure 6.18: Conversion Rules for Sequence Projection

Last but not least, the conversion rules for the flexary composition are given in figure 6.19.

Meta-Properties These conversions have the effect that *LFS* is conservative over *LF* in the following sense: If we have $\Sigma \vdash S : U : \mathtt{type}^n$ and all terms of type `nat` reduce to $1 + \ldots + 1$, then n reduces to m, and S and U reduce to $S^{(1)}, \ldots S^{(m)}$ and $U^{(1)}, \ldots, U^{(m)}$, and $S^{(i)} : U^{(i)}$ for $i = 1, \ldots, m$ in the *LF* type system.

This means that if the involved natural number expressions normalize, then *LFS*-expressions reduce to sequences of *LF*-expressions, and *LFS*-judgments reduce to sequences of *LF*-judgments. Under this condition, the normal *LFS* expressions are sequences of normal *LF* expressions. Consequently, an adequate encoding of objects as *LF*-expressions, yields an adequate encoding of sequences of objects as *LFS*-expressions.

Additionally, reducing full *LFS* to *LF* would require a formalization of sequences in *LF* already, which is doable, but also very costly.

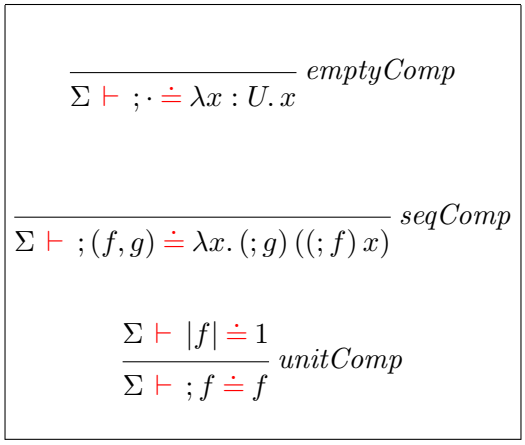


Figure 6.19: Conversion Rules for Flexary Composition

6.2.4 Stand-Alone Version

For the purpose of documentation, we give a version of the *LFS*-rules in figure 6.20 and figure 6.21 that is not modular. These rules together with the rules of *Seq* constitute the *LFS* as a standalone foundation.

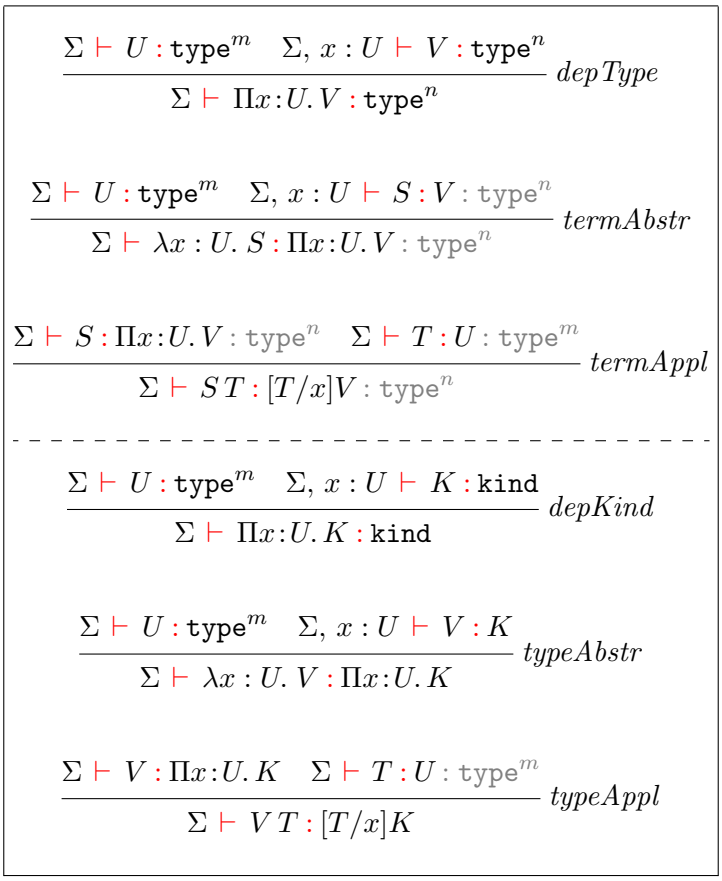


Figure 6.20: Typing Rules for Abstraction, Π-Formation and Application

$$\frac{\Sigma, x : U \vdash S : V : \mathtt{type}^m \quad \Sigma \vdash T : U : \mathtt{type}^n}{\Sigma \vdash (\lambda x : U.\, S)\, T \doteq [T/x]S}\ \beta\text{-}Conv_1$$

$$\frac{\Sigma, x : U \vdash V : \mathtt{type}^m \quad \Sigma \vdash T : U : \mathtt{type}^n}{\Sigma \vdash (\lambda x : U.\, V)\, T \doteq [T/x]V}\ \beta\text{-}Conv_2$$

$$\frac{\Sigma, x : U \vdash x : U \quad \Sigma \vdash S : \Pi x : U.\, V}{\Sigma \vdash \lambda x : U.\, (S\, x) \doteq S}\ \eta\text{-}Conv_1$$

$$\frac{\Sigma, x : U \vdash x : U \quad \Sigma \vdash V : \Pi x : U.\, K}{\Sigma \vdash \lambda x : U.\, (V\, x) \doteq V}\ \eta\text{-}Conv_2$$

Figure 6.21: β- and η-Conversions

6.3 Discussion

In this chapter we have introduced a new foundation *LFS* for sequences and ellipses that supports flexary functions and sequence arguments. *LFS* is novel in the development of sequences compared to existing approaches: We use term sequences $a_1, \ldots, a_n$ that are typed component-wise by a type sequence $A_1, \ldots, A_n$. Importantly, type sequences $A_1, \ldots, A_n$ are not types themselves – they are simply sequences of types.

Like sequences and contrary to indexed types and list types, this has the advantage that we do not change the underlying type theory. No representational artifacts are needed to flatten sequences or to apply a function to a sequence of arguments. And like for indexed types, the length of a sequence is statically known.

Moreover, the sequence constructs in *LFS* are expressive enough to define a number of interesting operations on sequences including the fold operator and the map operator. An overview of which sequence operators can be expressed in *LFS* is given in the table below:

Operation	Availability
length	object-level
map	defined
fold	defined
revert	defined
prefix	defined
suffix	defined
concatenation	primitive
empty	primitive

Not surprisingly, introducing sequences as a language primitive required natural numbers: We used a fragment of natural numbers with addition and a special subtraction that does not introduce negative numbers, which proved very useful for defining operators for sequences.

Furthermore, it is an important aspect of *LFS* that the length of sequence expressions is embedded into their types. This has the consequence that we never quantify over

sequences of arbitrary length. This is not a limitation, because we can always quantify over the length of a sequence.

Meta-Properties The foundation *Seq* is a relatively simple language with carefully chosen primitives for natural numbers and sequence expressions.

Type checking in *Seq* is harder than type checking e.g., lists, because the length of a sequence is part of the type system. Therefore, type checking in *Seq* requires reasoning about the length of a sequence. In particular, we need to check that the index i in S_i is actually within the boundaries of the length of S.

This yields a collection of constraints on natural number variables involved during type checking. This reduces type checking to provability in Presburger arithmetic. Therefore, we conjecture that type checking for *Seq* is decidable. We conjecture that type checking for *LFS*, on the other hand, is not decidable because of flexary composition, because it permits defining multiplication.

Alternative Approaches The approaches used by frameworks that support sequences can be categorized as we discuss below:

Lists We can represent flexary operators as unary operators that take a list as an argument. In that case, we represent $a_1 + a_2 + a_3$ as $+(List(a_1, a_2, a_3))$. Actually, this tacitly assumes that we have at least a flexary list constructor. In a pure fixary language, we would have to represent it as $+(cons(a_1, cons(a_2, cons(a_3, nil))))$, which is quite different from the informal mathematical object.

This approach permits using variables that quantify over sequences, and – using map and fold – it is easy to represent ellipses. This is widely used in programming languages, where lists are an accepted foundational data type.

In mathematics however, it is artificial to use lists since any formal mathematical theory for flexary operators would depend on the theory of lists, which itself is rarely used in informal mathematics. Another drawback is that all arguments must have the same domain. To permit different argument domains, we must allow lists whose elements have different types (or use sufficiently imprecise types).

Sequences Sequence types use a monadic type constructor $Seq : type \to type$ like lists and enjoy the same advantages. The difference is that sequences are always flattened, i.e., the canonical functions $Seq(Seq(A)) \to SeqA$ and $A \to Seq(A)$ are inclusions. For example, $Seq(a, b, Seq(c, d), e, f) = Seq(a, b, c, d, e, f)$. This makes sequence types closer to informal mathematics because they need less representational artifacts. Variants of sequence types occur in some programming languages but are rare in typed languages for formalized mathematics.

Sequences are more common in untyped languages. In the absence of a type system and in the presence of flattening, there is no need to write $f(Seq(a, b, c))$ at all. Instead, we can simply write $f(a, b, c)$ (even if one of the arguments is another sequence).

This approach is used in Common Logic (CL [Com07]), an untyped flexary variant of first-order logic. There, every non-logical symbol is flexary and variables may quantify over sequences. This substantially complicates the semantics because models must interpret every function symbol as a function that takes an arbitrary sequence of arguments; incidentally a proof theoretical semantics is not defined in CL.

[KB04] defines a flexary first-order logic and studies its semantics. The signature defines the arity of each non-logical symbol, and the arity can either be fixed or flexible. Similarly, variables are divided into individual and sequence variables.

Mathematica [Wol12] also uses untyped sequences, including sequence variables. Functions are fixary, but flexary functions can be defined by matching arguments against sequence patterns. Because Mathematica focuses on computation rather than logic, this is less problematic than for CL.

The untyped approaches to sequences usually cannot represent ellipses well because they tend to lack higher-order functions.

Indexed types Mixed-type lists can be represented concisely in Martin-Löf type theory [ML74], calculus of constructions [CH88] and related languages. Example implementations are Agda [Nor05] and Coq [Coq14]. If we write $[n]$ for the type containing $0, \ldots, n-1$, we call objects of type $T : [n] \to \mathit{type}$ indexed types. Mixed-type lists can be defined as indexed terms $l : \Pi i : [n]. T(i)$. Then flexary functions can be declared concisely as binary functions that take a natural number n and term indexed by $[n]$.

Ellipses can be represented very elegantly now, e.g., $a_1, \ldots, a_n$ is simply $\lambda i. a_i$. Moreover, contrary to all of the above, the length of a sequence is statically known, which permits static index-within-bounds checking when accessing an element of a sequence. Quantification only affects sequences of a certain length, e.g., $\forall x : [n] \to A. F$; to quantify over all sequences, we can use $\forall n. \forall x : [n] \to A. F$.

A disadvantage is the substantial commitment at the language level, which goes far beyond simply adding sequences: The language must be able to express the types $[n]$ and $[n] \to \mathit{type}$ (e.g. via inductive constructions and a universe hierarchy in Coq).

Chapter 7

Declarative Languages and their Translations in TFI

Our work in chapters 4, 5 and 6 has prepared us to present a solution to the research problems we discussed in chapter 1: In this chapter, we show how we overcome the over-generation problem of declarative logical frameworks we discussed in section 3 and define how to adequately represent declarative languages and their translations.

In particular, we use theory families from chapter 5 to give an adequate representation of declarative languages and specific language translations.

To simplify notation, we will fix an arbitrary foundation $\mathcal{F}$ throughout the whole chapter and give our definitions and results relative to $\mathcal{F}$.

For concrete examples of declarative languages, we will instantiate $\mathcal{F}$ with our foundation *LFS* from chapter 6. We choose to work within *LFS*, because specifying the declaration patterns of many well-established declarative languages like first-order logic requires the expressivity of sequences. In fact, we designed *LFS* specifically for this task.

7.1 Representing Declarative Languages

We represent declarative languages as theories in TFI. More specifically, for an arbitrarily fixed foundation $\mathcal{F}$, we define $\mathcal{F}$-languages in TFI:

Definition 7.1 ($\mathcal{F}$-Languages)**.** An **$\mathcal{F}$-language** is a well-formed theory in TFI/$\mathcal{F}$.

The novelty in our representation is that we can use theory families of TFI to represent the declaration patterns of a language $\mathcal{L}$. This provides a means to specify the syntactic shape of an arbitrary theory of $\mathcal{L}$. Then, we can define the theories of $\mathcal{L}$ as theories in TFI/$\mathcal{F}$ that only contain declarations that conform to a declaration pattern of $\mathcal{L}$.

Terminology 7.2. We will refer to a declaration pattern of $\mathcal{L}$ as an **$\mathcal{L}$-pattern**, and to a specific instantiation of an $\mathcal{L}$-pattern φ as a **φ-instance**. We will also say **$\mathcal{L}$-instance** to a φ-instance when φ is not of specific relevance.

As an example, we give the syntax of first-order logic and its declaration patterns from section 1.1.1 in TFI/*LFS* below. The examples that we will present in this chapter are not modular intentionally. In chapter 9 we give our examples modularly for the sake of better readability.

Example 7.3 (FOL Syntax in TFI/*LFS*). The TFI/*LFS* theory *FOL* represents the syntax of first-order logic:

$$
\begin{array}{l}
\mathit{FOL} \;=\; \{ \\
\quad \mathit{term} \;:\; \texttt{type} \\
\quad \mathit{form} \;:\; \texttt{type} \\
\quad \mathit{ded} \;:\; \mathit{form} \to \texttt{type} \\
\quad \mathit{Fun} \;=\; (n : \texttt{nat})\,\{ \\
\qquad f \;:\; \mathit{term}^n \to \mathit{term} \\
\quad \} \\
\quad \mathit{Pred} \;=\; (n : \texttt{nat})\,\{ \\
\qquad p \;:\; \mathit{term}^n \to \mathit{form} \\
\quad \} \\
\quad \mathit{Axiom} \;=\; (F : \mathit{form})\,\{ \\
\qquad a \;:\; \mathit{ded}\, F \\
\quad \} \\
\quad \mathit{false} \;:\; \mathit{form} \\
\quad \Rightarrow \;:\; \mathit{form} \to \mathit{form} \to \mathit{form} \\
\quad \forall \;:\; (\mathit{term} \to \mathit{form}) \to \mathit{form} \\
\quad \doteq \;:\; \mathit{term} \to \mathit{term} \to \mathit{form} \\
\}
\end{array}
$$

Here *term* and *form* are the *LFS*-types of first-order terms and formulas, respectively. $U \to V$ is a notation for the *LFS*-type constructor $\Pi x : U.\, V$ where x does not occur in V. Note that $\to$ is a flexary operator in *LFS*: $U_1, \ldots, U_n \to V$ normalizes to $U_1 \to \ldots \to U_n \to V$.

The declaration pattern *Fun* allows for the declaration of n-ary function symbols of the form $f : \mathit{term}^n \to \mathit{term}$ that take a sequence of first-order terms of length n and return a first-order term for any natural number n. Similarly, the declaration pattern *Pred* allows for the declaration of n-ary predicate symbols of the form $p : \mathit{term}^n \to \mathit{form}$ for any natural number n. Recall that term^n abbreviates the sequence expression $[\mathit{term}]_{x=1}^n$. This includes the case $n = 0$ of contants and propositional variables.

Axiom formalizes the shape of axiom declarations. Each axiom declaration must be of the form $a : \mathit{ded}\, F$ for some first-order formula $F : \mathit{form}$. These declaration patterns precisely describe what kind of symbol declarations are allowed in the theories of *FOL*.

Next, we add first-order logical symbols: For simplicity, we only consider the logical connectives *false* for falsehood and $\Rightarrow$ for implication. Moreover, *FOL* declares the first-order universal quantifier $\forall$ using higher-order abstract syntax, and the binary predicate symbol $\doteq$ for the equality of first-order terms. The remaining connectives and the existential quantifier are defined in terms of the given ones in the usual way.

LFS is expressive enough to define logics with flexary logical operators. For that reason, we choose to formalize our example languages the standard way using fixary primitives, and define their corresponding flexary versions using the fixary ones:

$$
\begin{array}{lcl}
\wedge^* & : & \mathit{form}^n \to \mathit{form} = \lambda F : \mathit{form}^n.\, \texttt{foldl}\ \wedge\ \mathit{F\ true} \\
\vee^* & : & \mathit{form}^n \to \mathit{form} = \lambda F : \mathit{form}^n.\, \texttt{foldl}\ \vee\ \mathit{F\ false} \\
\Rightarrow^* & : & \mathit{form}^n \to \mathit{form} \to \mathit{form} = \lambda F : \mathit{form}^n.\, \lambda G : \mathit{form}.\, \texttt{foldr}\ \Rightarrow\ F\, G \\
\forall^* & : & (\mathit{term}^n \to \mathit{form}) \to \mathit{form} \\
 & = & \lambda F.\ ; [\lambda f : \mathit{term}^i \to \mathit{form}.\, \lambda y : \mathit{term}^{i-1}.\, \forall \lambda x : \mathit{term}.\ f\,(y, x)]_{i=n}^1\, F
\end{array}
$$

The flexary conjunction $\wedge^*$ takes a natural number n and then a sequence of n conjuncts. We have $\wedge^* F_1 \ldots, F_n = (\ldots(\mathit{true} \wedge F_1)\ldots) \wedge F_n$ and $\wedge^* \cdot = \mathit{true}$. For disjunction,

we use $\vee^* F = \texttt{foldl} \vee F\, \mathit{false}$ accordingly. For implication, which is not associative, we define $\Rightarrow^* : \mathit{form}^n \to \mathit{form} \to \mathit{form}$ and $\Rightarrow^* F\, G = \texttt{foldr} \Rightarrow F\, G$.

The definition of flexary quantifiers is more involved. Intuitively, we want $\forall^* F = \forall \lambda x^1 : \mathit{term}. \ldots \forall \lambda x^n : \mathit{term}. F(x^1, \ldots, x^n)$. Let $[; G(i)]_{i=n}^1$ be the ellipsis in the definiens of $\forall^*$. Then the type of $G(i)$ is $(\mathit{term}^i \to \mathit{form}) \to (\mathit{term}^{i-1} \to \mathit{form})$, and when constructing $G(n)\,(\ldots(G(1)\,F)\ldots)$, each $G(i)$ introduces $\forall x^i$. Note that all variables are called x, the names x^i are introduced when *LFS* α-renames x during capture-avoiding substitution.

The alternative approach would be to introduce all language primitives as flexary operators without initially introducing fixary ones.

Now we define the theories of a declarative language L in TFI/$\mathcal{F}$:

Definition 7.4 (Theories of $\mathcal{F}$-Languages)**.** Let L be an $\mathcal{F}$-language, and L, Σ be a well-formed TFI/$\mathcal{F}$-theory.

We say that Σ is a **strict L-theory** if Σ is strict in the sense of definition 5.3.

Moreover, we say that Σ is an **L-theory** if the elaboration $\mathcal{E}(\Sigma)$ of Σ is equal to the elaboration $\mathcal{E}(\Sigma')$ of a strict L-theory Σ' up to the renaming of symbols.

Note that the instance declarations $\iota : \Phi$ in a strict L-theory Σ have the form either

$$\iota : \varphi\, E_1 \ldots E_n \qquad \text{or} \qquad \iota : \{\Psi\}$$

for an L-pattern φ and a strict theory fragment Ψ. Moreover, symbol declarations $x : E = E'$ are only allowed with definiens to restrict the possibility of introducing new symbol declarations that do not respect L-patterns. This guarantees that only those TFI/$\mathcal{F}$-theories that comply with the L-patterns are actually L-theories.

Now, we give the first-order theory of monoids as an example of a (strict) *FOL*-theory:

Example 7.5 (FOL-Theory of Monoid). Consider the following formalization of the theory of monoids as a strict *FOL*-theory:

$$\begin{array}{l}
\mathit{Monoid} \;=\; \{ \\
\quad \texttt{include}\ \mathit{FOL} \\
\quad e \quad : \quad \mathit{Fun}\, 0 \\
\quad \circ \quad : \quad \mathit{Fun}\, 2 \\
\quad \mathit{neut}_l \quad : \quad \mathit{Axiom}\, \forall\, \lambda x : \mathit{term}.\ e.f \circ .f\ x \;==\; x \\
\quad \vdots \\
\}
\end{array}$$

Furthermore, the following theory in TFI/*LFS* is a *FOL*-theory:

$$\begin{array}{l}
\mathit{Monoid}' \;=\; \{ \\
\quad \texttt{include}\ \mathit{FOL} \\
\quad e \quad : \quad \mathit{term} \\
\quad \circ \quad : \quad \mathit{term} \to \mathit{term} \to \mathit{term} \\
\quad \mathit{neut}_l \quad : \quad \mathit{ded}\, \forall\, \lambda x : \mathit{term}.\ e \circ x \;==\; x \\
\quad \vdots \\
\}
\end{array}$$

The elaboration of *Monoid* is the following TFI/*LFS*-theory:

$$
\begin{array}{l}
\mathcal{E}(\mathit{Monoid}) \;=\; \{ \\
\quad \texttt{include}\ \mathit{FOL} \\
\quad e.f \quad : \quad \mathit{term} \\
\quad \circ.f \quad : \quad \mathit{term} \to \mathit{term} \to \mathit{term} \\
\quad \mathit{neut}_l.a \quad : \quad \mathit{ded}\ \forall\, \lambda x : \mathit{term}.\ e.f\ \circ .f\ x \;==\; x \\
\quad \vdots \\
\}
\end{array}
$$

Note that $\mathcal{E}(\mathit{Monoid}')$ is equal to $\mathcal{E}(\mathit{Monoid})$ up to renaming of the declared symbols. And the elaboration of FOL is equal to itself.

Technically, our definition of strict L-theories allows nested instance declarations of the form

$$\iota : \{\iota_1 : \{\ldots\{\iota_n : \{x : E = E'\}\}\ldots\}\}$$

which is equivalent to

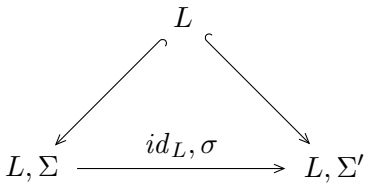
$$\iota.\iota_1.\ \ldots\ .\iota_n.x : E = E'$$

after elaboration. Nested instance declarations are rarely necessary to declare in a theory. They will usually appear as a result of translating strict theories, typically for $n = 1$ for the depth of the nesting.

Now that we have defined L-theories, we will group together those theory morphisms that are between L-theories:

Definition 7.6 (L-Morphisms)**.** Let L be an $\mathcal{F}$-language, Σ and Σ' be strict L-theories. We call the TFI/$\mathcal{F}$-morphism fragment σ in $\mathit{id}_L, \sigma : L, \Sigma \to L, \Sigma'$ an **L-morphism** from Σ to Σ'.

This definition is intuitively captured by the following diagram:



The intuition behind definition 7.6 is that the strict L-theories and the L-morphisms form a category, namely the theory category of L. It is worthwhile to remark that if the theories of a declarative language are represented adequately by strict L-theories, then usually the theory morphisms are represented adequately by L-morphisms as well.

Now we give a second example of a $\mathcal{F}$-language and its theories in order to exemplify language translations in section 7.2.

Example 7.7 (SFOL Syntax in TFI/*LFS*). The TFI/*LFS*-theory *SFOL* below is similar to *FOL* except that we use an *LFS*-type *sort* to encode the set of sorts and an *LFS* type family *tm* indexed by *sort* that provides the type $\mathit{tm}\, S$ of terms of sort S, i.e., $t : \mathit{tm}\, S$ means that t is of sort S. Universal quantification $\forall$ is sorted, i.e., it first takes a sort argument S and then binds a variable of type $\mathit{tm}\, S$.

The declaration pattern *Sort* allows for the declaration of sorts $s : \mathit{sort}$. The declaration patterns *SortedFun* and *SortedPred* formalize the shape of declarations of sorted n-ary function and predicate symbols of the form $f : \mathit{tm}\, s_1 \to \ldots \to \mathit{tm}\, s_n \to \mathit{tm}\, t$ and $q : \mathit{tm}\, s_1 \to \ldots \to \mathit{tm}\, s_n \to \mathit{form}$, respectively, for sorts $s_1 : \mathit{sort}, \ldots, s_n : \mathit{sort}$ and $t : \mathit{sort}$. Recall that the sequence $[\mathit{tm}\, s_i]_{i=1}^n$ normalizes to $\mathit{tm}\, s_1, \ldots, \mathit{tm}\, s_n$ and the type $[\mathit{tm}\, s_i]_{i=1}^n \to \mathit{tm}\, t$ normalizes to $\mathit{tm}\, s_1 \to \ldots \to \mathit{tm}\, s_n \to \mathit{tm}\, t$.

$$
\begin{array}{lcl}
\mathit{SFOL} & = & \{ \\
\quad \mathit{sort} & : & \mathtt{type} \\
\quad \mathit{tm} & : & \mathit{sort} \to \mathtt{type} \\
\quad \mathit{form} & : & \mathtt{type} \\
\quad \mathit{ded} & : & \mathit{form} \to \mathtt{type} \\
\quad \mathit{Sort} & = & \{ \\
\qquad s & : & \mathit{sort} \\
\quad \} & & \\
\quad \mathit{SortedFun} & = & (n : \mathtt{nat}, s : \mathit{sort}^n, t : \mathit{sort}) \{ \\
\qquad f & : & [\mathit{tm}\, s_i]_{i=1}^n \to \mathit{tm}\, t \\
\quad \} & & \\
\quad \mathit{SortedPred} & = & (n : \mathtt{nat}, s : \mathit{sort}^n) \{ \\
\qquad q & : & [\mathit{tm}\, s_i]_{i=1}^n \to \mathit{form} \\
\quad \} & & \\
\quad \mathit{Axiom} & = & (F : \mathit{form}) \{ \\
\qquad m & : & \mathit{ded}\, F \\
\quad \} & & \\
\quad \mathit{false} & : & \mathit{form} \\
\quad \Rightarrow & : & \mathit{form} \to \mathit{form} \to \mathit{form} \\
\quad \forall & : & \Pi S : \mathit{sort}.\, (\mathit{tm}\, S \to \mathit{form}) \to \mathit{form} \\
\} & &
\end{array}
$$

Like in *FOL*, we define the flexary version of the logical symbols in *SFOL*. The definitions for the flexary logical connectives are as in example 7.3. Here we only need to define the flexary quantifiers for *SFOL*:

$$
\begin{array}{l}
\forall^* : \Pi S : \mathit{sort}^n.\, (S_1^n \to \mathit{form}) \to \mathit{form} \\
\quad = \lambda S.\, \lambda F.\ ; [\lambda f : S_1^i \to \mathit{form}.\ \lambda y : S_1^{i-1}.\ \forall \lambda x : \mathit{tm}\, S_i.\ f\,(y, x)]_{i=n}^1\, F
\end{array}
$$

where S_a^b is an abbreviation for $[\mathit{tm}\, S_j]_{j=a}^b$. The definition of the flexary existential quantifier follows analogously.

Now we introduce the theory of vector spaces as an *SFOL*-theory:

Example 7.8 (SFOL-Theory of Vector Spaces). *VectorSpaces* is a strict *SFOL*-theory that formalizes the theory of vector spaces.

$$
\begin{array}{lcl}
\mathit{VectorSpaces} & = & \{ \\
\quad \mathtt{include}\ \mathit{SFOL} & & \\
\quad \mathit{vec} & : & \mathit{Sort} \\
\quad \mathit{sca} & : & \mathit{Sort} \\
\quad 0 & : & \mathit{SortedFun}\, 0\ \cdot\ \mathit{sca}.s \\
\quad + & : & \mathit{SortedFun}\, 2\ \ \mathit{vec}.s\,, \mathit{vec}.s\ \ \ \mathit{vec}.s \\
\} & &
\end{array}
$$

Here *vec* and *sca* instantiate the declaration pattern *Sort* to introduce sorts for vectors and scalars, respectively. 0 instantiates the declaration pattern *SortedFun* with arity 0, the empty sequence $\cdot$ for the domain and the sort $\mathit{sca}.s$ for the codomain. $+$ instantiates *SortedFun* with arity 2, the sequence $\mathit{vec}.s\,, \mathit{vec}.s$ for the domain sort $\mathit{vec}.s$ for both arguments and the sort $\mathit{vec}.s$ for the codomain.

7.2 Representing Language Translations

In section 3.2 we have discussed the main problems with the representation of language translations in declarative logical frameworks and exemplified those problems via a list of important language translations. The three main problems are *i)* whether an existing translation is adequate for expressions, *ii)* whether the representation of a translation is well-defined for L_2-theories, *iii)* whether the representation of a translation is adequate for theories.

In the following we give a new definition of a language translation based on declaration patterns which addresses the last two problems above in particular.

Definition 7.9 ($\mathcal{F}$-Translations)**.** Let L and L' be two $\mathcal{F}$-languages. We say that a TFI/$\mathcal{F}$-morphism $l : L \to L'$ is a **strict $\mathcal{F}$-translation** from L to L' if for every $\varphi := \Phi$ in l, we have that Φ is strict (in the sense of definition 5.3).

In the following we will fortify our intuition by defining a translation from SFOL to FOL using declaration patterns. Recall that we have classified this translation as inadequate in LF, because it is not possible to give a translation from SFOL to FOL in LF that maps sorted function symbols of *SFOL* to a *FOL* function symbol that is coupled with the axiom that results from the relativization of the sorts. Strict $\mathcal{F}$-translations overcome this problem:

Example 7.10 (Translation from *SFOL* to *FOL*). We give a strict *LFS*-translation from *SFOL* to *FOL* below:

$$
\begin{array}{l}
\mathit{S2F} \;:\; \mathit{SFOL} \;\to\; \mathit{FOL} \;=\; \{ \\
\quad \begin{array}{lcl}
\mathit{sort} & := & \mathit{term} \to \mathit{form} \\
\mathit{tm} & := & \lambda x : \mathit{term} \to \mathit{form}.\ \mathit{term} \\
\mathit{form} & := & \mathit{form} \\
\mathit{ded} & := & \lambda F : \mathit{form}.\ \mathit{ded}\,\mathrm{F}
\end{array} \\
\\
\quad \begin{array}{lcl}
\mathit{Sort} & := & \mathit{Pred}\,1
\end{array} \\
\\
\quad \begin{array}{lcl}
\mathit{SortedFun} & := & (n : \mathtt{nat},\ s : (\mathit{term} \to \mathit{form})^n,\ t : \mathit{term} \to \mathit{form})\,\{ \\
\qquad \begin{array}{lcl}
\iota & : & \mathit{Fun}\,n \\
f & : & \mathit{term}^n \to \mathit{term} = \iota.f \\
\mathit{wellsorted} & : & \mathit{Axiom}\,\forall^* \lambda x : \mathit{term}^n.\ (\wedge^*([s_i\,x_i]_{i=1}^n)) \;\Rightarrow\; t\,(f\,x)
\end{array} \\
\quad \}
\end{array} \\
\\
\quad \begin{array}{lcl}
\mathit{SortedPred} & := & (n : \mathtt{nat},\ s : (\mathit{term} \to \mathit{form})^n)\ \mathit{Pred}\,n \\
\mathit{Axiom} & := & (F : \mathit{form})\ \mathit{Axiom}\,F \\
\mathit{false} & := & \mathit{false} \\
\Rightarrow & := & \lambda F : \mathit{form}.\ \lambda G : \mathit{form}.\ F \Rightarrow G \\
\forall & := & \lambda s : \mathit{term} \to \mathit{form}.\ \lambda p : \mathit{term} \to \mathit{form}.\ \forall \lambda x : \mathit{term}.\ (s\,x) \;\Rightarrow\; (p\,x)
\end{array} \\
\}
\end{array}
$$

The translation of the logical primitives of *SFOL* is straightforward.

Our main contribution in this example is the translation of the *SFOL*-patterns: *Sort* is mapped to *Pred* 1 specifying that for each sort declaration $s : \mathit{sort}$, there is one declaration of a corresponding unary predicate symbol.

SortedFun is mapped to a theory family expression that *i)* takes arguments *LFS* natural number $n : \mathtt{nat}$ for the arity, an *LFS* sequence $s : (\mathit{term} \to \mathit{form})^n$ of n first-order unary predicates and a first-order unary predicate $t : \mathit{term} \to \mathit{form}$, and *ii)* returns an instance

f' of an n-ary first-order function symbol, a function f identical to the function introduced by f', and a corresponding axiom for the function introduced by f'.

SortedPred is mapped to the theory family abstraction with arguments *LFS* natural number $n : \texttt{nat}$ and an *LFS* sequence s of n first-order unary predicates, and returns the *FOL*-pattern *Pred* with argument n.

This specifies that for every n-ary *SFOL* predicate symbol declaration, there must be a corresponding n-ary *FOL* predicate symbol declaration.

The translation of *Axiom* is straightforward: Every axiom declaration in an SFOL theory is mapped to an axioms declaration in the corresponding FOL theory.

Note that the mapping of each four declaration patterns comply with definition 7.9: *SortedFun* is mapped to theory family expression whose body respects *FOL*-patterns. *Sort*, *SortedPred* and *Axiom* are mapped to theory family expressions in terms of *FOL*-patterns.

Moreover, the symbol declaration $f : term^n \to term = \iota.f$ in the translation of *SortedFun* is required by the rule *mapTheoryFamily* in figure 5.5: we have the requirement that the judgment $L' \vdash \bar{l}(\varphi) \leq \Phi$ holds for the assignment $\varphi := \Phi$ in a $TFI/\mathcal{F}$-morphism l.

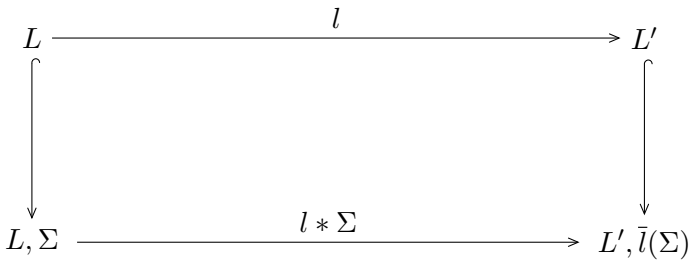
The above translation becomes non-strict if we have the following pattern assignments instead:

$$
\begin{array}{lcl}
\mathit{Sort} & := & \{s : \mathit{term} \to \mathit{form}\} \\
\mathit{SortedFun} & := & (n : \texttt{nat},\ s : (\mathit{term} \to \mathit{form})^n,\ t : \mathit{term} \to \mathit{form})\,\{ \\
\quad f & : & \mathit{term}^n \to \mathit{term} \\
\quad \mathit{wellsorted} & : & \mathit{Axiom}\ \forall^*[x : \mathit{term}^n]\,(\wedge^*([s_i\ x_i]_{i=1}^n)) \Rightarrow t\,(f\ x) \\
\}
\end{array}
$$

Definition 7.11 (Theory Translation)**.** Let L and L' be two $\mathcal{F}$-languages and $l : L \to L'$ be a strict $\mathcal{F}$-translation. The **translation of an L-theory Σ along l** is $\bar{l}(\Sigma)$.

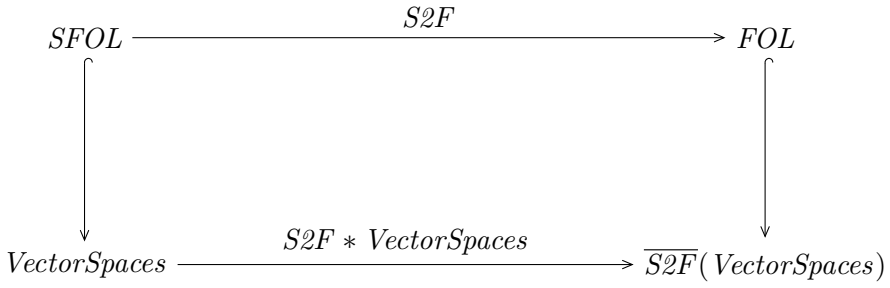
Moreover, the **translation of an expression E over an L-theory Σ** is $\overline{l * \Sigma}(E)$.

This definition is captured by the following diagram:



Note that we still translate theories along morphisms using homomorphic extensions. However, our definition of homomorphic extension is different as it is applied to theory families and their instances.

Example 7.12. We give the translation of the *SFOL*-theory *VectorSpaces* along the strict *LFS*-translation *S2F* from example 7.10.



$\overline{S2F}(\mathit{VectorSpaces})$ consists of the following declarations[1]:

$$
\begin{array}{l}
\overline{S2F}(\mathit{VectorSpaces}) \;=\; \{ \\
\quad \begin{array}{lcl}
\mathit{vec} & : & \overline{S2F * \mathit{VectorSpaces}}(\mathit{Sort}) \\
\mathit{sca} & : & \overline{S2F * \mathit{VectorSpaces}}(\mathit{Sort}) \\
0 & : & \overline{S2F * \mathit{VectorSpaces}}(\mathit{SortedFun})\, 0 \cdot \overline{S2F * \mathit{VectorSpaces}}(\mathit{sca}.s)
\end{array} \\
\quad \vdots \\
\}
\end{array}
$$

This is equal to the theory:

$$
\begin{array}{l}
\overline{S2F}(\mathit{VectorSpaces}) \;=\; \{ \\
\quad \begin{array}{lcl}
\mathit{vec} & : & \mathit{Pred}\, 1 \\
\mathit{sca} & : & \mathit{Pred}\, 1 \\
0 & : & ((n : \mathtt{nat}, s : (\mathit{term} \to \mathit{form})^n, t : \mathit{term} \to \mathit{form}) \\
 & & \quad \{ \\
 & & \qquad \begin{array}{lcl}
\iota & : & \mathit{Fun}\, n \\
f & : & \mathit{term}^n \to \mathit{term} = \iota.f \\
\mathit{wellsorted} & : & \mathit{Axiom}\, \forall^* \lambda x : \mathit{term}^n.\ (\wedge^*([s_i\, x_i]_{i=1}^n)) \;\Rightarrow\; t\,(f\, x)
\end{array} \\
 & & \quad \}\,)\, 0 \cdot \mathit{sca}.s
\end{array} \\
\quad \vdots \\
\}
\end{array}
$$

We get the following theory after β-normalization:

$$
\begin{array}{l}
\overline{S2F}(\mathit{VectorSpaces}) \;=\; \{ \\
\quad \begin{array}{lcl}
\mathit{vec} & : & \mathit{Pred}\, 1 \\
\mathit{sca} & : & \mathit{Pred}\, 1 \\
0 & : & \left\{ \begin{array}{ll}
\iota & : \mathit{Fun}\, 0 \\
f & : \mathit{term} = \iota.f \\
\mathit{wellsorted} & : \mathit{Axiom}\, \forall^* \lambda x : \mathit{term}^0.\ (\wedge^*([\cdot_i\, x_i]_{i=1}^0)) \;\Rightarrow\; \mathit{sca}.s\ (f\, x)
\end{array} \right\}
\end{array} \\
\quad \vdots \\
\}
\end{array}
$$

[1] Here (and in the next couple of following theories) we omit the respective symbol declaration for the translation of + for the sake of readability, and put vertical dots to notify the reader of this omission.

Elaborating $\overline{S2F}(\mathit{VectorSpaces})$ results with the following theory:

$$
\begin{array}{lcl}
\mathcal{E}(\overline{S2F}(\mathit{VectorSpaces})) & = & \{ \\
\quad \mathit{vec}.p & : & \mathit{term} \to \mathit{form} \\
\quad \mathit{sca}.p & : & \mathit{term} \to \mathit{form} \\
\quad 0.\iota.f & : & \mathit{term} \\
\quad 0.f & : & \mathit{term} = 0.\iota.f \\
\quad 0.\mathit{wellsorted} & : & \mathit{ded}\ \mathit{true} \Rightarrow \mathit{sca}.s\ (0.f\ x) \\
\quad \vdots & & \\
\}
\end{array}
$$

Using the non-strict translation in example 7.10, we get the following theory:

$$
\begin{array}{lcl}
\overline{S2F}(\mathit{VectorSpaces}) & = & \{ \\
\quad \mathit{vec} & : & \{s : \mathit{term} \to \mathit{form}\} \\
\quad \mathit{sca} & : & \{s : \mathit{term} \to \mathit{form}\} \\
\quad 0 & : & \left\{ \begin{array}{ll} f & : \mathit{term} \\ \mathit{wellsorted} & : \mathit{Axiom}\ \forall^*[x : \mathit{term}^0]\ (\wedge^*([\cdot_i\ x_i]_{i=1}^0)) \Rightarrow \mathit{sca}.s\ (f\ x) \end{array} \right\} \\
\quad + & : & \left\{ \begin{array}{ll} f & : \mathit{term} \to \mathit{term} \to \mathit{term} \\ \mathit{wellsorted} & : \mathit{Axiom}\ \forall^*[x : \mathit{term}^2]\ (\wedge^*([(\ \mathit{vec}.s\ ,\mathit{vec}.s\ )_i\ x_i]_{i=1}^2)) \Rightarrow \mathit{vec}.s\ (f\ x) \end{array} \right\} \\
\}
\end{array}
$$

After normalizing the sequence expressions and elaborating the declarations in $\overline{S2F}(\mathit{VectorSpaces})$ we get:

$$
\begin{array}{lcl}
\mathcal{E}(\overline{S2F}(\mathit{VectorSpaces})) & = & \{ \\
\quad \mathit{vec}.s & : & \mathit{term} \to \mathit{form} \\
\quad \mathit{sca}.s & : & \mathit{term} \to \mathit{form} \\
\quad 0.f & : & \mathit{term} \\
\quad 0.\mathit{wellsorted}\ .a & : & \mathit{ded}\ \mathit{true} \Rightarrow \mathit{sca}.s\ (f\ x) \\
\quad +.f & : & \mathit{term} \to \mathit{term} \to \mathit{term} \\
\quad +.\mathit{wellsorted}\ .a & : & \mathit{ded}\ \mathit{vec}.s\ x_1 \wedge \mathit{vec}.s\ x_2 \Rightarrow \mathit{vec}.s\ (f\ x) \\
\}
\end{array}
$$

Main Theorem Finally, we have the following main result in this chapter:

Theorem 7.13. *Assume two $\mathcal{F}$-languages L, L' and a strict $\mathcal{F}$-translation $l : L \to L'$. Then for every strict L-theory Σ, the translation $\bar{l}(\Sigma)$ of Σ is a strict L'-theory.*

Proof. By mutual induction on the formation of Σ and Φ, we prove (1) our claim and (2) if Φ is strict, then $\bar{l}(\Phi)$ is strict.

We have the following cases for Σ:

- **Case $\cdot$:** We have $\bar{l}(\cdot) = \cdot$, which is trivially strict.

- **Case** $\Sigma_0, x : E = E'$**:**

  We have $\overline{l}(\Sigma_0, x : E = E') = \overline{l}(\Sigma_0), x : \overline{l * \Sigma_0}(E) = \overline{l * \Sigma_0}(E')$, which is strict by induction hypothesis (1).

- **Case** $\Sigma_0, i : \Phi$**:**

  We have $\overline{l}(\Sigma_0, i : \Phi) = \overline{l}(\Sigma_0), i : \overline{l * \Sigma_0}(\Phi)$. By induction hypothesis (1), we have that $\overline{l}(\Sigma_0)$ is a strict L'-theory. Since l is a strict $\mathcal{F}$-translation, it follows that $l * \Sigma_0$ is a strict $\mathcal{F}$-translation. Then, by induction hypothesis (2) we get that $\overline{l * \Sigma_0}(\Phi)$ is strict.

We have the following cases for Φ:

- **Case** φ**:** Since l is a strict $\mathcal{F}$-translation, it follows immediately that $\overline{l}(\varphi)$ is strict.
- **Case** $(x : E)\,\Phi_0$**:** We have $\overline{l}((x : E)\,\Phi_0) = (x : \overline{l}(E))\,\overline{l, x := x}(\Phi_0)$. Since l is a strict $\mathcal{F}$-translation, it follows that $l, x := x$ is a strict $\mathcal{F}$-translation. Hence, $\overline{l}((x : E)\,\Phi_0)$ is strict by induction hypothesis (2).
- **Case** $\Phi_0(E)$**:** We have $\overline{l}(\Phi_0(E)) = \overline{l}(\Phi_0)(\overline{l}(E))$, which is strict by induction hypothesis (2).
- **Case** $\{\Psi\}$**:** We have $\overline{l}(\{\Psi\}) = \{\overline{l}(\Psi)\}$, which is strict by induction hypothesis (1).

□

With this theorem, we have that strict theories are translated to strict theories along a strict language translation. This is a very crucial result that is required for adequate representations of language translations.

7.3 Induced Languages and Translations

Recall the LF-induced languages $\mathcal{D}^{LF}(L)$ in definition 2.18. We can now define the corresponding operation $\mathcal{D}^{\mathcal{F}}(-)$ for an arbitrary $\mathcal{F}$ in a way that does not over-generate. We will omit the superscript $\mathcal{F}$ for the sake of readability.

Definition 7.14 ($\mathcal{F}$-Induced Languages)**.** Let L be an $\mathcal{F}$-language. L induces a declarative language $\mathcal{D}^{\mathcal{F}}(L)$, where

- the $\mathcal{D}(L)$-theories are the strict L-theories,
- the expressions over a $\mathcal{D}(L)$-theory Σ are the TFI/$\mathcal{F}$-expressions over L, Σ.

Here, the main difference to the definition of LF-induced languages in definition 2.18 is that the notion of L-theory used in our definition excludes all symbol declarations that are not L-instances.

Definition 7.15 ($\mathcal{F}$-Induced Translations)**.** Let T be a strict $\mathcal{F}$-translation from L to L'. T induces a language translation $\mathcal{D}(T) : \mathcal{D}(L) \to \mathcal{D}(L')$, where

- $\mathbf{th} : \mathbf{Th}^{\mathcal{D}(L)} \to \mathbf{Th}^{\mathcal{D}(L')}$ maps a $\mathcal{D}(L)$-theory Σ to $\overline{T}(\Sigma)$,
- $\mathbf{exp}$ is a family of mappings $\mathbf{exp}_\Sigma : \mathbf{Exp}^{\mathcal{D}(L)}(\Sigma) \to \mathbf{Exp}^{\mathcal{D}(L')}(\mathbf{th}(\Sigma))$ that map an expression F over Σ to $\overline{T * \Sigma}(F)$.

Note that definition 7.15 is well-defined due to theorem 7.13. Definition 7.15 is a significant achievement of this thesis.

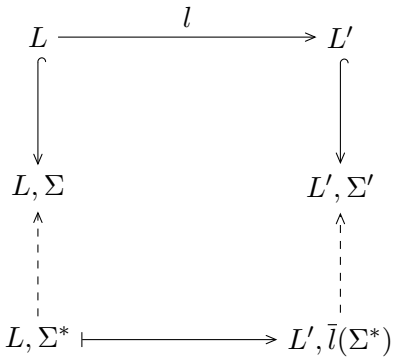
7.4 Discussion

In our meta-framework, we use theory families to give adequate representations of declarative languages. More specifically, we use theory families to represent the declaration patterns of $\mathcal{F}$-languages. This allows us to talk about the shape of an arbitrary theory of an $\mathcal{F}$-language.

Moreover, we use the instantiations of theory families, i.e., $\mathcal{L}$-instances, as a crucial syntactic means to build individual theories of declarative languages $\mathcal{L}$. In particular, our definition of *strict* theories of a declarative language overcomes the over-generation problem by restricting a theory to declarations that can only be expressed using a $\mathcal{L}$-pattern.

Furthermore, declaration patterns play a significant role in specifying translations between declarative languages. Similar to the definition of the theories of a language, our definition of *strict* language translations makes sure that theories are translated adequately between languages. In particular, strict theories of a declarative language are guaranteed to be translated to strict theories.

This result is a crucial step towards obtaining a similar result for non-strict theories. In an implementation of our framework, it would be possible to implement a procedure that computes for every symbol declaration $x : E$, a corresponding declaration pattern that x instantiates. Using such a pattern-checking procedure, it would be possible to define a pattern-based language translation that translates a non-strict L-theory Σ to a non-strict L'-theory by *i)* computing the corresponding strict L-theory Σ^* for Σ, assuming that Σ^* can be uniquely determined, *ii)* translating Σ^* along a strict language translation $l : L \to L'$, and *iii)* elaboration the resulting theory $\bar{l}(\Sigma^*)$ to a non-strict L'-theory.



Note that TFI cannot guarantee that there is a unique strict L-theory that corresponds to a particular non-strict L-theory, because the declaration patterns formalized in L may overlap: If, for example, in *FOL*, one declares both $\{c : \mathit{term}\}$ and $(n : \mathtt{nat})\,\{f : \mathit{term}^n \to \mathit{term}\}$ as *FOL*-patterns, then a declaration $t : \mathit{term}$ in a *FOL*-theory Σ would match both patterns, and thus we could give two strict-theories for Σ.

Finally, our approach proves to cover a broad range different declarative languages: In chapter 9 we evaluate our approach to create a graph of adequately represented languages and their translations.

Chapter 8

Extension Principles in TFI

In section 1.3 we discussed the extension principles that declarative languages typically employ for theory formation. In this chapter, we describe our methodology to represent extension principles in TFI/$\mathcal{F}$ and to translate them along $\mathcal{F}$-translations. As before, we fix an arbitrary foundation $\mathcal{F}$.

In section 8.1 we develop our representation of extension principles, and in section 8.1 we use TFI/$\mathcal{F}$ theory morphisms to translate extension principles. For running examples, we instantiate $\mathcal{F}$ with our foundation *LFS* from chapter 6.

8.1 Representing Extension Principles

Recall our definition of $\mathcal{F}$-languages from definition 7.1. In an $\mathcal{F}$-language L, the extension principles of L can be intuitively captured by theory families:

Definition 8.1 (Extension Principles of L)**.** Let L be an $\mathcal{F}$-language. An extension principle of L is a theory family declaration of the form $\varphi : \tau = \Phi$.

Then, the instance declarations $\iota : \varphi\, E_1 \dots E_n$ introduce new declarations that make use of the extension principle φ.

Definition 8.2. An extension principle $\varphi : \tau = \Phi$, for $\Phi = (x_1 : E_1, \dots, x_n : E_n)\,\{\Psi\}$, is called **derived** if all symbol declarations in Ψ have a definiens; otherwise, it is called **primitive**. Moreover, it is called **strict** if Φ is strict.

In the case of representing *conservative* extension principles, the proof that the corresponding primitive L-extension principle is conservative is a meta-argument that must be carried out as a part of the proof that L is an adequate TFI/$\mathcal{F}$ representation of the represented language. However, we can show generally that the conservativity of a derived L-extension principle follows from that of the primitive ones.

In the following, we will represent the extension principles we discussed in section 1.3. We have the option to give strict or non-strict extension principles. The former is necessary to translate L-extension principles along strict $\mathcal{F}$-translations. The latter is more general. In particular, we will give the *SFOL*-extension principles in strict form as we will translate them later in section 8.2 and 9.3. We will give the extension principles in *FOL* and *HOL* in non-strict form.

We will state every extension principle in the smallest theory in which it is meaningful. This also documents the (often implicit) foundational assumptions of each extension principle.

FOL-Style Explicit Definitions In figure 8.1 we define FOL-style explicit definitions for *FOL* in TFI: The extension principle *explicitDef* takes the arity $n : \texttt{nat}$ and the definiens $t : term$ as arguments and declares an n-ary function symbol $f : term^n \to term$ and the corresponding axiom $f(x_1, \ldots, x_n) = t$.

$$
\begin{array}{l}
ExplicitDefinition \;=\; \{ \\
\quad \texttt{include}\ FOL \\
\quad explicitDef \;=\; (n : \texttt{nat}, t : term)\,\{ \\
\qquad f \;:\; term^n \to term \\
\qquad ax \;:\; ded\, \forall^* x : term^n.\; f\, x \doteq t \\
\quad \} \\
\}
\end{array}
$$

Figure 8.1: Extension Principle for FOL Explicit Definitions in TFI/*LFS*

Implicit Definitions Recall the special case of implicit definitions in section 1.3 for nullary function symbols in (sorted) first-order logic. We give a representation for it in *FOL* in figure 8.2.

$$
\begin{array}{l}
ImplicitDefinition \;=\; \{ \\
\quad \texttt{include}\ FOL \\
\quad \exists^! \;:\; (term \to form) \to form = \ldots \\
\quad impldef \;=\; (P : term \to form, m : ded\, \exists^! x : term.\, P\, x)\,\{ \\
\qquad c \;:\; term \\
\qquad ax \;:\; ded\, P\, c \\
\quad \} \\
\}
\end{array}
$$

Figure 8.2: Extension Principle for FOL Implicit Definitions in TFI/*LFS*

The theory *ImplicitDefinition* includes *FOL* and defines the unique existential quantifier $\exists^!$ and a theory family *impldef*, which specifies implicit definitions of unary functions in first-order logic. *impldef* takes a first-order unary predicate $P : term \to form$ and a proof m that there exists a unique term x for which P holds. It returns the declarations $c : term$ for a nullary *FOL*-term and an axiom $ax : ded\, P\, c$ stating the property P holds for c.

We can give a strict version of *impldef* by giving the following declarations in its body instead:

$$
\begin{array}{lcl}
c & : & Fun\, 0 \\
ax & : & Axiom\, P\, c.f
\end{array}
$$

Example 8.3 (Monoids). We define the theory of monoids as a theory of *FOL* with implicit definitions below, and introduce the unit element e using the *FOL*-extension principle *impldef*:

$$
\begin{array}{l}
\mathit{Monoid}^* \ = \ \{ \\
\quad \texttt{include}\ \mathit{ImplicitDefinition} \\
\quad \circ \quad : \quad \mathit{Fun}\, 2 \\
\quad \mathit{neut}_r \quad : \quad \mathit{Axiom}\, \exists x : \mathit{term}.\, \forall y : \mathit{term}.\, y \circ .f\ x \doteq y \\
\quad e \quad : \quad \mathit{impldef}\ (\lambda x : \mathit{term}.\, \forall y : \mathit{term}.\, (x \circ .f\ y \doteq x \ \wedge \ y \circ .f\ x \doteq x))\, M \\
\}
\end{array}
$$

$\circ$ is the binary operation in monoids and M denotes the required *LFS* proof term for the unique existence of e. Here we omit the associativity axiom for the sake of simplicity.

Implicit Definitions in Sorted First-Order Logic The theory *FunctionDefinition* in figure 8.3 introduces the extension principle of function definitions in *SFOL* for any n-ary function f.

$$
\begin{array}{l}
\mathit{FunctionDefinition} \ = \ \{ \\
\quad \texttt{include}\ \mathit{SFOL} \\
\quad \mathit{function} \ = \\
\qquad (\\
\qquad\quad n : \texttt{nat}, D : \mathit{sort}^n, C : \mathit{sort}, \\
\qquad\quad \mathit{means} : [\mathit{tm}\, D_i]_{i=1}^n \to \mathit{tm}\, C \to \mathit{form}, \\
\qquad\quad \mathit{existence} : \mathit{ded}\, \forall^* x : [\mathit{tm}\, D_i]_{i=1}^n.\, \exists y : \mathit{tm}\, C.\, \mathit{means}\, x\, y, \\
\qquad\quad \mathit{uniqueness} : \mathit{ded}\, \forall^* x : [\mathit{tm}\, D_i]_{i=1}^n.\, \forall y : \mathit{tm}\, C.\, \forall y' : \mathit{tm}\, C \\
\qquad\qquad\qquad\qquad \mathit{means}\, x\, y \wedge \mathit{means}\, x\, y' \Rightarrow y \doteq y' \\
\qquad) \\
\qquad \{ \\
\qquad\quad f \quad : \quad \mathit{SortedFun}\, n\, D\, C \\
\qquad\quad \mathit{ax} \quad : \quad \mathit{Axiom}\, \forall^* x : [\mathit{tm}\, D_i]_{i=1}^n.\, \mathit{means}\, x\, (f\, x) \\
\qquad \} \\
\}
\end{array}
$$

Figure 8.3: SFOL-Style Function Definitions in TFI/*LFS*

This is an example of a very commonly used extension principle in the Mizar language [TB85]. In Mizar, the axiom ax is called the definitional theorem induced by the implicit definition.

Using the extension principle *function*, we can introduce an instance of the form c : $\mathit{function}\, n\, D\, C\, P\, E\, U$ that introduces n-ary function symbols f over the sorts in D that is defined by the property P where E and U discharge the induced proof obligations for existence and uniqueness. Recall that $\forall^*$ is the flexary universal quantifier in *SFOL* defined in example 7.7.

We can give a non-strict version of *FunctionDefinition* by giving the following declarations in its body:

$$
\begin{array}{lcl}
f & : & [\mathit{tm}\, D_i]_{i=1}^n \to \mathit{tm}\, C \\
\mathit{ax} & : & \mathit{ded}\, \forall^* x : [\mathit{tm}\, D_i]_{i=1}^n.\, \mathit{means}\, x\, (f\, x)
\end{array}
$$

Case-Based Function Definitions In Figure 8.4, the theory *CaseBasedFunction* introduces the extension principle of case-based definitions for unary functions in *SFOL*. Functions f from sort A to sort B are defined using n different cases where each case is guarded by the predicate c_i together with the respective definiens d_i. Such a definition is well-defined if for all $x \in A$ exactly one out of the $c_i\, x$ is true. Here $\vee^!$ is the flexary exclusive or: $\vee^!(F_1, \ldots, F_n)$ holds if exactly one of its arguments holds. We define it as follows:

$$\vee^! : form^n \to form = \vee^* [\wedge^* [\neg A_j]_{j=1}^{i-1}, A_i, [\neg A_j]_{j=1}^{i+1}]_{i=1}^{n}$$

where $\vee^*$ and $\wedge^*$ are flexary disjunction and conjunction, respectively, from example 7.3.

$$\begin{array}{l}
\mathit{CaseBasedFunction} \;=\; \{ \\
\quad \texttt{include}\ \mathit{SFOL} \\
\quad \mathit{casedef} \;= \\
\qquad (\\
\qquad\quad n : \texttt{nat}, A : \mathit{sort}, B : \mathit{sort}, c : (\mathit{tm}\, A \to \mathit{form})^n, d : (\mathit{tm}\, A \to \mathit{tm}\, B)^n, \\
\qquad\quad \rho : \mathit{ded}\, \forall x : \mathit{tm}\, A.\ \vee^! [c_i\, x]_{i=1}^{n} \\
\qquad) \\
\qquad \{ \\
\qquad\quad f \quad : \quad \mathit{SortedFun}\, 1\, A\, B \\
\qquad\quad \mathit{ax} \quad : \quad \mathit{Axiom}\, \forall x : \mathit{tm}\, A.\ \wedge\, [c_i\, x \;\Rightarrow\; (f\, x) = (d_i\, x)]_{i=1}^{n} \\
\qquad \} \\
\}
\end{array}$$

Figure 8.4: Case-Based Definitions in TFI/*LFS*

Then the instance $f : \mathit{casedef}\, n\, A\, B\, c_1 \ldots c_n\, d_1 \ldots d_n\, \rho$ corresponds to the following function definition:

$$f(x) = \begin{cases} d_1(x) & \text{if} \quad c_1(x) \\ \vdots & \quad\ \vdots \\ d_n(x) & \text{if} \quad c_n(x) \end{cases}$$

We can give a non-strict version of *FunctionDefinition* by giving the following declarations in its body:

$$\begin{array}{lcl} f & : & \mathit{tm}\, A \to \mathit{tm}\, B \\ f_{ax} & : & \mathit{ded}\, \forall x : \mathit{tm}\, A.\ \wedge\, [c_i\, x \;\Rightarrow\; (f\, x) = (d_i\, x)]_{i=1}^{n} \end{array}$$

HOL-Style Type Definitions The theory *TypeDefinition* in figure 8.5 introduces the extension principles of type definition in *HOL* (we introduce *HOL* in figure 9.10 in section 9.1.3). Our symbol names follow the implementation of this definition principle in Isabelle/HOL [NPW02].

Instances of the form $t : \mathit{typedef}\, A\, P\, \rho$ introduce a new non-empty type T that is isomorphic to the subtype of A defined by the predicate P. Since all HOL-types must be non-empty, a proof ρ of the non-emptiness of this subtype must be supplied. It is elaborated to the following constant declarations:

- $t.T : tp$ is the new type,
- $t.Rep : \mathit{tm}\, t.T \to \mathit{tm}\, A$ is an injection from the new type $t.T$ to A,

$$
\begin{array}{l}
\mathit{TypeDefinition} \;=\; \{ \\
\quad \texttt{include}\ \mathit{HOL} \\
\quad \mathit{typedef} \;=\; (A : \mathit{tp}, P : \mathit{tm}\, A \to \mathit{tm}\, o, \mathit{nonempty} : \mathit{ded}\, \exists x : \mathit{tm}\, A.\, P\, x)\, \{ \\
\qquad \begin{array}{lcl}
T & : & \mathit{tp} \\
\mathit{Rep} & : & \mathit{tm}\, T \to \mathit{tm}\, A \\
\mathit{Abs} & : & \mathit{tm}\, A \to \mathit{tm}\, T \\
\mathit{Rep}' & : & \mathit{ded}\, \forall x : \mathit{tm}\, T.\, P\, (\mathit{Rep}\, x) \\
\mathit{Rep\_inverse} & : & \mathit{ded}\, \forall x : \mathit{tm}\, T.\, \mathit{Abs}\, (\mathit{Rep}\, x) \doteq x \\
\mathit{Abs\_inverse} & : & \mathit{ded}\, \forall x : \mathit{tm}\, A.\, P\, x \Rightarrow \mathit{Rep}\, (\mathit{Abs}\, x) \doteq x
\end{array} \\
\quad \} \\
\}
\end{array}
$$

Figure 8.5: HOL-Style Type Definitions

- $t.\mathit{Abs} : \mathit{tm}\, A \to \mathit{tm}\, t.T$ is the partial inverse of $t.\mathit{Rep}$ from A to the new type $t.T$,
- $t.\mathit{Rep}'$ states that the property P holds for any term of type $t.T$,
- $t.\mathit{Rep\_inverse}$ states that the injection of any element of type $t.T$ to A and back is equal to itself,
- $t.\mathit{Abs\_inverse}$ states that if an element satisfies P, then injecting it to $t.T$ and back is equal to itself.

8.2 Translating Extension Principles

Since L-extension principles are theory families, we can translate them along TFI-theory morphisms. In particular, we can translate extension principles between $\mathcal{F}$-languages. Here we will give one example using the strict LFS-translation $S2F$ from example 7.10.

Example 8.4 (Translation of *function* in *FOL*). The translation of the *SFOL*-extension principle *function* from figure 8.3 along $S2F$ is as follows:

$$
\begin{array}{l}
\overline{S2F}(\mathit{function}) \;= \\
\quad (\\
\qquad \begin{array}{lcl}
n & : & \texttt{nat}, \\
D & : & (\mathit{term} \to \mathit{form})^n, \\
C & : & \mathit{term} \to \mathit{form}, \\
\mathit{means} & : & \mathit{term}^n \to \mathit{term} \to \mathit{form}, \\
\mathit{existence} & : & \mathit{ded}\, \forall^* \lambda x : \mathit{term}^n.\, \wedge^* [D_i\, x_i]_{i=1}^n \;\Rightarrow \\
 & & \qquad (\exists \lambda y : \mathit{term}. \wedge^* [D_i\, x_i]_{i=1}^n \;\wedge\; (\mathit{means}\, x\, y)), \\
\mathit{uniqueness} & : & \mathit{ded}\, \forall^* \lambda x : \mathit{term}^n.\, (\wedge^* [D_i\, x_i]_{i=1}^n) \;\Rightarrow \\
 & & \qquad (\forall \lambda y : \mathit{term}.\, (C\, y) \;\Rightarrow\; (\forall \lambda y' : \mathit{term}.\, (C\, y') \;\Rightarrow \\
 & & \qquad\quad (\mathit{means}\, x\, y \wedge \mathit{means}\, x\, y' \Rightarrow y \doteq y')))
\end{array} \\
\quad) \\
\quad \{ \\
\qquad \begin{array}{lcl}
f' & : & \mathit{Fun}\, n \\
f & : & \mathit{term}^n \to \mathit{term} = f'.f \\
\mathit{ax} & : & \mathit{Axiom}\, \forall^* \lambda x : \mathit{term}^n.\, (\wedge^* ([D_i\, x_i]_{i=1}^n)) \;\Rightarrow\; C\, (f\, x) \\
\mathit{wellsorted} & : & \mathit{Axiom}\, \forall^* \lambda x : \mathit{term}^n.\, \mathit{means}\, x\, (f\, x)
\end{array} \\
\quad \}
\end{array}
$$

Then, we can recover the implicit definitions of nullary functions in FOL using the translation of *function*:

$$\mathit{ImplicitDefinition} = \overline{\mathit{S2F}}(\mathit{function})\,0 \cdot \lambda x : \mathit{term}.\,\mathit{true}$$

We get the following definiens for *ImplicitDefinition* after β-normalizing $\overline{\mathit{S2F}}(\mathit{function})\,0 \cdot \lambda x : \mathit{term}.\,\mathit{true}$:

$$\begin{array}{l}\mathit{ImplicitDefinition} \;=\\ \quad\{\\ \qquad \begin{array}{lcl} f & : & \mathit{term} \\ \mathit{ax} & : & \mathit{ded}\,(\mathit{true} \Rightarrow \mathit{true}) \\ \mathit{wellsorted} & : & \mathit{ded}\,\mathit{means} \cdot (f \cdot) \end{array}\\ \quad\}\end{array}$$

8.3 Discussion

Our novel development of theory families in TFI/$\mathcal{F}$ enables representing, besides the declaration patterns, the extension principles of a language in declarative logical frameworks. We regard each extension principle as one language feature that permits extending the theories of a declarative language in a specific way. Theory families allow us to represent individual extension principles, and furthermore to add language extensions to any declarative language in a generic way.

This further facilitates the automation of translating language extensions along language translations. This is very important: Given a declarative language $\mathcal{L}$, its extension principles represented as theory families in an $\mathcal{L}$-theory *Ext* and a translation μ from $\mathcal{L}$ to $\mathcal{L}'$, our methodology permits translating the extension principles of $\mathcal{L}$ along l and obtains the respective extension principles for $\mathcal{L}'$ (as the diagram below illustrates):

$$\begin{array}{ccc} \mathcal{L} & \xrightarrow{\quad l \quad} & \mathcal{L}' \\ \big\downarrow & & \big\downarrow \\ \mathcal{L}, \mathit{Ext} & \longrightarrow & \mathcal{L}', \bar{l}(\mathit{Ext}) \end{array}$$

One important open question is how certain properties of extension principles, such as conservativity, behave under language translations in TFI.

Chapter 9

An Atlas of Declarative Languages

In this chapter we present a graph of adequately represented declarative languages and their translations, which we obtained using declaration patterns.

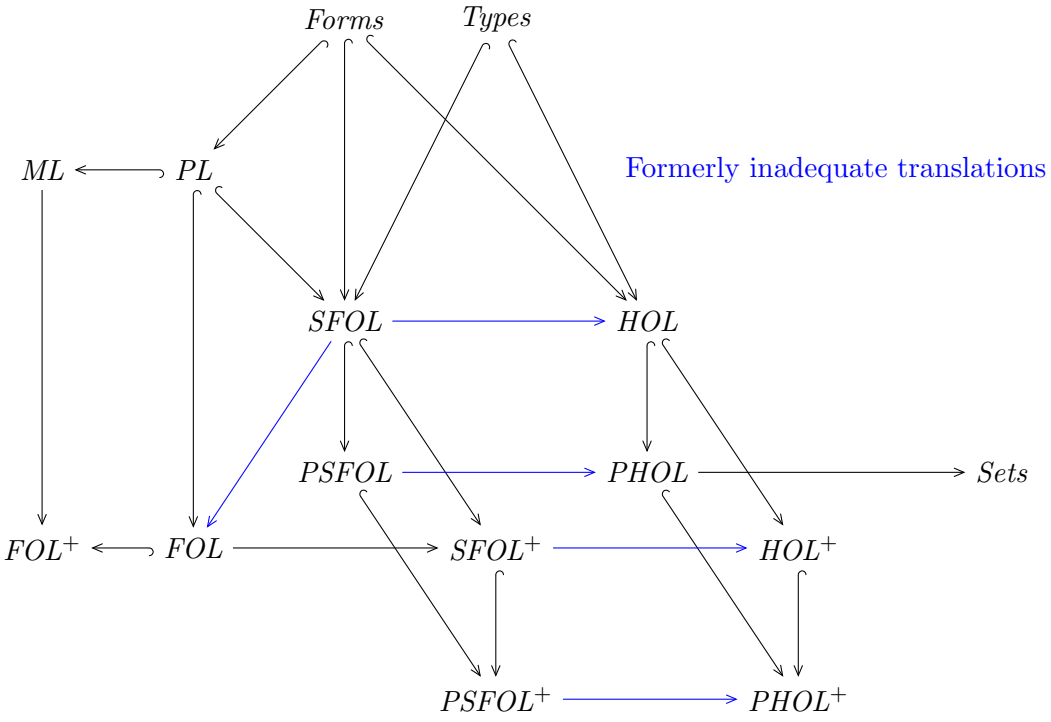


Figure 9.1: Language Atlas

Modularity We will present our language graph in a modular way that facilitates the reuse of shared content amongst the languages we consider. For this purpose, we will use the following notations: Firstly, we will write

$$\begin{array}{l} T \ = \ \{ \\ \quad \texttt{include}\ S \\ \quad \Psi \\ \} \end{array}$$

as in notation 5.8 for named theories $S = \{\Sigma\}$ and $T = \{\Sigma, \Psi\}$, and say that T *includes (the declarations in)* S or simply that T *extends* S *with* Ψ.

Secondly, we will write

$$\begin{array}{l} T \ = \{ \\ \quad \texttt{include}\ S\ \texttt{with}\ \{\varphi := \Phi\} \\ \quad \Psi \\ \} \end{array}$$

for $S = \{\Sigma, \varphi : \tau = \Phi', \Sigma'\}$ and $T = \{\Sigma, \varphi : \tau = \Phi, \Sigma', \Psi\}$, where the definiens Φ' of φ is replaced by Φ.[1]. We will use this notation for multiple assignments $\varphi_1 := \Phi_1, \ldots, \varphi_n := \Phi_n$ as well in the obvious way.

Similarly, we will write

$$\begin{array}{l} T \ = \{ \\ \quad \texttt{include}\ S\ \texttt{with}\ \{x := E\} \\ \quad \Psi \\ \} \end{array}$$

for $S = \{\Sigma, x : E', \Sigma'\}$ and $T = \{\Sigma, x : E' = E, \Sigma', \Psi\}$.

9.1 Declarative Languages

In this section, we present the languages in our atlas. We group them into *propositional languages* in section 9.1.1, *single-typed languages* in section 9.1.2, *many-typed languages* in section 9.1.3 and *polymorphic languages* in section 9.1.4.

While we choose *LFS* as the specific foundation to define these languages, we will actually use sequences and natural numbers to define the declaration patterns of each language L in our atlas, and restrict the definition of L-expressions to the LF subset of *LFS*.

9.1.1 Propositional Languages

Logical Formulas In figure 9.2, we give an auxiliary TFI/LFS-theory *Forms* of logical formulas.

$$\begin{array}{l} \mathit{Forms} \ = \{ \\ \quad \mathit{form} \ : \ \texttt{type} \\ \quad \mathit{ded} \ : \ \mathit{form} \to \texttt{type} \\ \quad \mathit{Axiom} \ = \ (F : \mathit{form})\,\{ \\ \qquad a \ : \ \mathit{ded}\, F \\ \quad \} \\ \} \end{array}$$

Figure 9.2: Representation of Logical Formulas

[1]Since the assignment $\varphi := \Phi$ maps φ to Φ it must respect the judgment $\Sigma \vdash \Phi' \leq \Phi$.

Forms introduces a type *form* for formulas and a type family *ded* that assigns for every formula $F : form$ the type of the proof of F. *Forms* also introduces one declaration pattern named *Axiom* for axiom declarations of the form $a : ded\, F$ for some formula F.

Note that we do not need to add a separate declaration pattern for theorems since we write theorems F as symbol declarations $t : ded\, F = D$ with definiens D, where D is an LFS-term for the proof of F.

Propositional Logic We can build our first language on top of *Forms*: Propositional logic.

$$
\begin{array}{l}
PL \;=\; \{ \\
\quad \texttt{include}\; Forms \\
\quad PropSym \;=\; \{ \\
\qquad p \;:\; form \\
\quad \} \\
\quad false \;:\; form \\
\quad \Rightarrow \;:\; form \to form \to form \\
\}
\end{array}
$$

Figure 9.3: Representation of Propositional Logic in *LFS*

The theory *PL* in figure 9.3 represents the propositional logic syntax by including the declarations of *Forms* and adding two symbols *false* and $\Rightarrow$ for falsehood and implication, respectively. Here, we only give *false* and $\Rightarrow$ as the primitive logical connectives and define the remaining connectives in terms of the primitive ones in the usual way.

Note that *PL* inherits the symbols *form* and *ded* as well as the pattern axiom declarations from *Forms*.

Modal Logic The theory *ML* in figure 9.4 represents the syntax of modal logic. It includes the declarations in *PL* and adds the symbols $\Box$ and $\Diamond$ for the modal operators of necessity and possibility, respectively.

$$
\begin{array}{l}
ML \;=\; \{ \\
\quad \texttt{include}\; PL \\
\quad \Box \;:\; form \to form \\
\quad \Diamond \;:\; form \to form \\
\}
\end{array}
$$

Figure 9.4: Representation of Modal Logic

9.1.2 Single-Typed Languages

The single-typed languages we consider in this section are *i)* first-order logic, *ii)* the proof theory of first-order logic, and *iii)* set theory.

$$
\begin{array}{l}
FOL \;=\; \{ \\
\quad \texttt{include}\ Forms \\
\quad term \;:\; \texttt{type} \\
\quad Fun \;=\; (n : \texttt{nat})\,\{ \\
\qquad f \;:\; term^n \to term \\
\quad \} \\
\quad Pred \;=\; (n : \texttt{nat})\,\{ \\
\qquad p \;:\; term^n \to form \\
\quad \} \\
\quad \texttt{include}\ PL\ \texttt{with}\ \{ \\
\qquad PropSym \;:=\; Pred\,0 \\
\quad \} \\
\quad \forall \;:\; (term \to form) \to form \\
\quad \exists \;:\; (term \to form) \to form \\
\quad \doteq \;:\; term \to term \to form \\
\}
\end{array}
$$

Figure 9.5: Representation of FOL

First-Order Logic In example 7.3, we gave a representation of FOL syntax in TFI/*LFS*. In figure 9.5 we integrate our representation of FOL into our language atlas.

The novelty here is that we can explicitly represent the relation between *PL*-theories and *FOL*-theories in our framework: A PL-theory consists of propositional variables, which are nullary predicates in first-order logic. We develop the representation of FOL on top of PL by including *PL* in *FOL* and mapping the *PL*-pattern *PropSym* for propositional variables to the pattern expression $Pred\,0$ for nullary predicate symbols. Note that our notation $\texttt{include}\ PL\ \texttt{with}\ PropSym := Pred\,0$ introduces the declaration $PropSym = Pred\,0$ in *FOL*.

FOL^+ extends *FOL* with the declaration $acc : term \to term \to form$. In section 9.2.2, we will use the binary predicate *acc* as the accessibility relation between two ML worlds in FOL.

Proof Theory Besides the syntax of logics, declarative logical frameworks like LF have been used for representing the proof theory. In our work in [HR11], we presented a case study of representing FOL syntax, proof theory and (for the first time) model theory in LF. Representations of the proof theory have been given in LF in, for example, [PSK+03, Soj10, CHK+11].

The proof theory of a logic is typically specified as a set of inference rules. In logical frameworks based on the Curry-Howard isomorphism, such as LF and LFS, inference rules of a logic can be represented as symbol declarations.

In the following we only consider FOL proof theory as an example and will represent it in TFI/*LFS* in terms of natural deduction rules. We only give the introduction and the elimination rules for the universal quantifier $\forall$ as examples in figure 9.6.

Occasionally, the proof theory may add declaration patterns. An example is the congruence of n-ary function symbols in FOL-theories. For any n-ary FOL function symbol f, we want a proof in the respective theory of the FOL proof theory that f is congruent.

$$
\begin{array}{l}
FOLPF \;=\; \{ \\
\quad \texttt{include}\; FOL\; \texttt{with}\; \{Fun := congruence\} \\
\quad \forall_I \;:\; \Pi x : tm\, S.\, ded\, F\, x \to ded\, \forall\, F \\
\quad \forall_E \;:\; ded\, \forall\, F \to \Pi x : tm\, S.\, ded\, F\, x \\
\}
\end{array}
$$

Figure 9.6: FOL Proof Theory in TFI/*LFS*

For that purpose, we may add a declaration pattern in the representation of FOL proof theory, which allows declarations of n-ary function symbols f together with a proof term that proves congruency of f:

$$
\begin{array}{l}
congruence \;=\; (n : \texttt{nat})\, \{ \\
\quad f \;:\; term^n \to term \\
\quad f_{\cong} \;:\; \Pi x : term^n.\, \Pi x' : term^n.\, ded\, [x_i \doteq x'_i]_{i=1}^n \to ded\, (f\, x \doteq f\, x') \\
\}
\end{array}
$$

Set Theory We give parts of our representation of set theory in figure 9.7. The full version is quite involved and can be found in [HR11]. The version in [HR11] uses LF as the underlying framework, and here we present new developments that use declaration patterns and extension principles, and therefore require TFI/*LFS*.

$$
\begin{array}{l}
Sets \;=\; \{ \\
\quad \texttt{include}\; FOL \\
\quad \texttt{include}\; FOLPF \\
\quad \texttt{include}\; ImplicitDefinition \\
\quad set \;:\; \texttt{type} = term \\
\quad \in \;:\; set \to set \to form \\
\quad //\text{axioms} \\
\quad Element \;=\; (A : set, nonempty : ded\, \exists x : set.\, x \in A)\, \{ \\
\qquad x \;:\; Fun\, 0 \\
\qquad x_{in} \;:\; Axiom\, (x.f \;\in A) \\
\quad \} \\
\quad \emptyset \;:\; impldef\, (\lambda x : set.\, \forall y : set.\, \neg\, (y \in x))M \\
\}
\end{array}
$$

Figure 9.7: Representation of Axiomatic Set Theory in TFI/LFS

Our representation of set theory is based on *FOL* and its proof theory *FOLPF*. Moreover, we use the extension principle of implicit definitions to define set operators.

First, we declare the symbol $\in$: $set \to set \to form$ for the membership relation between two sets (i.e., *FOL*-terms). Here we omit the LFS-encodings of the usual axioms of set theory and refer the reader to [HR11].

Then, using the extension principle of implicit definitions, we define the standard set operators in a way that is close to how they are defined in mathematical practice. Here we only give the empty set as an example:

$$\emptyset : impldef\, (\lambda x : set.\, \forall y : set.\, \neg\, (y \in x))M$$

where M denotes the existence proof for the empty set and uses the proof rules in *FOLPF* and the axioms of set theory. Set union, unordered pairs and power set can be defined in a similar way. This is different than our version in [HR11], where set operators are defined using a primitive description operator.

Furthermore, we introduce a declaration pattern *Element* that permits declaring elements of a non-empty set $A : set$. It expands into a new set x, and an axiom x_{in} that states the membership relation between x and A. Note that *Element* requires a proof term for the non-emptiness of A as an argument.

Note that we do not need to declare a separate pattern for set declarations as they are a special case of the pattern *Fun* in *FOL* with arity 0.

9.1.3 Many-Typed Languages

Types The second base theory in our atlas is *Types*, which consists of the shared components of the typed languages. We will use *Types* for all typed or sorted declarative languages.

$$\begin{array}{lcl} Types & = & \{ \\ \quad tp & : & \texttt{type} \\ \quad tm & : & tp \to \texttt{type} \\ \} & & \end{array}$$

Figure 9.8: Base Theory for Typed Languages

In *Types*, we first introduce a TFI/*LFS* type $tp : \texttt{type}$. We will use tp to represent the universe of all L-types in a language L that classify L-expressions into multiple disjoint sets. Then we introduce a type family tm that assigns for every L-type $S : tp$, the type $tm\, S$ of L-terms x of L-type S.

Sorted First-Order Logic Recall our representation of SFOL from example 7.7. Here we integrate that representation into our language atlas.

SFOL includes the declarations from *Forms* and *Types*, and adds declaration patterns for sort declarations $s : tp$, and n-ary SFOL function and predicate symbol declarations in SFOL-theories. Moreover, it includes all logical connectives from *PL*. Note that *PL* only allows declaration of propositional variables $q : form$, which is a special case of predicate declarations where the arity is 0. Therefore, we map the declaration pattern *PropSym* to $SortedPred\, 0\, \cdot$ when we include *PL* in *SFOL*. *SFOL* includes the declaration pattern *Axiom* of *PL*.

The symbols $\forall$ and $\exists$ represent the universal and the existential quantification of sorted terms, respectively. The symbol $\doteq$ represents the equality of *SFOL*-terms of the same sort.

The theory $SFOL^+$ in our language atlas is an extension of *SFOL* with the distinguished sort $i : tp$.

Higher-Order Logic The next language with multiple types in our atlas is higher-order logic. The theory *HOL* in figure 9.10 represents the HOL syntax by including the theory *Types* and adding the following declarations: The declaration pattern *BaseType* allows the declaration of base types $a : tp$, and *TypedSym* allows the declaration of typed symbols $c : tm\, A$ of any HOL-type $A : tp$. Then *HOL* declares a distinguished base type $o : tp$ for

$$
\begin{array}{l}
\mathit{SFOL} \;=\; \{ \\
\quad \texttt{include}\ \mathit{Forms} \\
\quad \texttt{include}\ \mathit{Types} \\
\quad \mathit{Sort} \;=\; \{ \\
\qquad s \;:\; \mathit{tp} \\
\quad \} \\
\quad \mathit{SortedFun} \;=\; (n : \texttt{nat}, s : \mathit{tp}^n, t : \mathit{tp})\,\{ \\
\qquad f \;:\; [\mathit{tm}\, s_i]_{i=1}^n \to \mathit{tm}\, t \\
\quad \} \\
\quad \mathit{SortedPred} \;=\; (n : \texttt{nat}, s : \mathit{tp}^n)\,\{ \\
\qquad p \;:\; [\mathit{tm}\, s_i]_{i=1}^n \to \mathit{form} \\
\quad \} \\
\quad \texttt{include}\ \mathit{PL}\ \texttt{with}\ \{ \\
\qquad \mathit{PropSym} \;:=\; \mathit{SortedPred}\, 0\cdot \\
\quad \} \\
\quad \forall \;:\; (\mathit{tm}\, S \to \mathit{form}) \to \mathit{form} \\
\quad \exists \;:\; (\mathit{tm}\, S \to \mathit{form}) \to \mathit{form} \\
\quad \doteq \;:\; \mathit{tm}\, S \to \mathit{tm}\, S \to \mathit{form} \\
\}
\end{array}
$$

Figure 9.9: Representation of SFOL in TFI/*LFS*

higher-order formulas. Note that this is the intrinsic way of representing object-level HOL formulas, in contrast to the extrinsic way, which uses $o : \texttt{type}$ in *LFS*.

Next, *HOL* includes *Forms* by mapping the *LFS*-type *form* of formulas to the *LFS*-type $\mathit{tm}\, o$ of HOL formulas.

Then, we introduce the HOL type constructor, binder and the application operator: $\Longrightarrow$ is a type constructor that constructs a HOL type $A \Longrightarrow B$ given two HOL types $A : \mathit{tp}$ and $B : \mathit{tp}$. *lam* constructs a HOL abstraction $\mathit{lam}\, f$ given an *LFS* abstraction $f : \mathit{tm}\, A \to \mathit{tm}\, B$. Given a HOL abstraction $f : \mathit{tm}\,(A \Longrightarrow B)$ and a HOL term $x : \mathit{tm}\, A$, @ constructs the HOL application $f \mathbin{@} x$ of f to x. Then we add higher-order connectives, quantifiers and equality.

We also define a flexary function type constructor, flexary λ-abstraction, and flexary application in terms of the fixary ones.

$$
\begin{array}{lcl}
\Longrightarrow^* & : & \mathit{tp}^n \to \mathit{tp} \to \mathit{tp} \\
 & = & \lambda A : \mathit{tp}^n.\ \lambda B : \mathit{tp}.\ \texttt{foldr}\ \Longrightarrow\ A\, B \\
\mathit{lam}^* & : & ([\mathit{tm}\, A_i]_{i=1}^n \to \mathit{tm}\, B) \to \mathit{tm}\,(A \Longrightarrow^* B) \\
 & = & \lambda F.\ ;[\lambda f : [\mathit{tm}\, A_j]_{j=1}^i \to \mathit{tm}\,\big([A_j]_{j=i+1}^n \Longrightarrow^* B\big). \\
 & & \qquad \lambda y : [\mathit{tm}\, A_j]_{j=1}^{i-1}.\ \mathit{lam}\,\lambda x : \mathit{tm}\, A_i.\ f\,(y, x) \\
 & & \quad]_{i=n}^1\, F \\
@^* & : & \mathit{tm}\,(A \Longrightarrow^* B) \to [\mathit{tm}\, A_i]_{i=1}^n \to \mathit{tm}\, B \\
 & = & \lambda F.\,\lambda a.\ ;[\lambda f : A_i^n \Longrightarrow B.\ @\, f\, x_i]_{i=1}^n\, F
\end{array}
$$

The theory HOL^+ in our language atlas is an extension of *HOL* with a distinguished HOL type $i : \mathit{tp}$.

$$
\begin{array}{l}
HOL \;=\; \{ \\
\quad \texttt{include}\ \mathit{Types} \\
\quad \mathit{BaseType} \;=\; \{ \\
\qquad a \;:\; tp \\
\quad \} \\
\quad \mathit{TypedSym} \;=\; (A : tp)\,\{ \\
\qquad c \;:\; tm\,A \\
\quad \} \\
\quad o \;:\; tp \\
\quad \texttt{include}\ \mathit{Forms}\ \texttt{with}\ \{ \\
\qquad \mathit{form} \;:=\; tm\,o \\
\quad \} \\
\quad \Longrightarrow \;:\; tp \to tp \to tp \\
\quad \mathit{lam} \;:\; (tm\,A \to tm\,B) \to tm\,(A \Longrightarrow B) \\
\quad @ \;:\; tm\,(A \Longrightarrow B) \to tm\,A \to tm\,B \\
\quad \mathit{false} \;:\; tm\,o \\
\quad \Rightarrow \;:\; tm\,(o \Longrightarrow o \Longrightarrow o) \\
\quad \forall \;:\; tm\,((A \Longrightarrow o) \Longrightarrow o) \\
\quad \doteq \;:\; tm\,(A \Longrightarrow A \Longrightarrow o) \\
\}
\end{array}
$$

Figure 9.10: Representation of HOL in TFI/*LFS*

9.1.4 Polymorphic Languages

In this section we represent polymorphic languages by adding shallow polymorphism on top of SFOL and HOL as a new feature. Our main intuition is that non-polymorphic languages are special cases of the polymorphic ones, and we can explicitly show this relation in our representations by exploiting declaration patterns. In fact, the crucial difference between the representations of polymorphic languages and their non-polymorphic versions is in the legal declarations, e.g., polymorphic sorted first-order logic theories may declare n-ary type operators and polymorphic function and predicate symbols. This shows the importance of declaration patterns in our framework as this difference could not be captured in frameworks like LF.

In the following we give our representation of polymorphic SFOL and polymorphic HOL.

Polymorphic SFOL Figure 9.11 illustrates the representation of sorted first-order logic syntax with polymorphic type constructors in theory *PSFOL*. *PSFOL* includes the base theories *Forms* and *Types* for formulas and types, respectively. Then we specify the declaration patterns:

- *TypeOp* permits n-ary type operators $t : tp^n \to tp$ in *PSFOL*-theories.
- *PolyFun* permits polymorphic function symbols f in *PSFOL*-theories, which take m type arguments $a_1, \ldots, a_m$, then n term arguments of types $A_1(a_1, \ldots, a_m), \ldots, A_n(a_1, \ldots, a_m)$ and return an expression of type $B(a_1, \ldots, a_m)$. Note that we use

higher-order abstract syntax to represent types A with m free variables $a_1 : tp, \ldots, a_m : tp$ as LFS expressions $A : tp^m \to tp$.

- *PolyPred* permits polymorphic predicate symbols p in *PSFOL*-theories, which take m type arguments $a_1, \ldots, a_m$, then n term arguments of types $A_1(a_1, \ldots, a_m), \ldots, A_n(a_1, \ldots, a_m)$.
- *PolyAxiom* permits polymorphic axioms F in *PSFOL*-theories, which take m type arguments $a : tp^m$.

$$
\begin{array}{l}
PSFOL \;=\; \{ \\
\quad \texttt{include}\ Forms \\
\quad \texttt{include}\ Types \\
\quad TypeOp \;=\; (n : \texttt{nat})\,\{ \\
\qquad t \;:\; tp^n \to tp \\
\quad \} \\
\quad PolyFun \;=\; (m : \texttt{nat},\, n : \texttt{nat},\, A : (tp^m \to tp)^n,\, B : tp^m \to tp)\,\{ \\
\qquad f \;:\; \Pi\, a : tp^m.\, [tm\,(A_i\; a)]_{i=1}^n \to tm\,(B\; a) \\
\quad \} \\
\quad PolyPred \;=\; (m : \texttt{nat},\, n : \texttt{nat},\, A : (tp^m \to tp)^n)\,\{ \\
\qquad p \;:\; \Pi\, a : tp^m.\, [tm\,(A_i\; a)]_{i=1}^n \to form \\
\quad \} \\
\quad PolyAxiom \;=\; (m : \texttt{nat},\, F : tp^m \to form)\,\{ \\
\qquad q \;:\; \Pi\, a : tp^m.\, ded\,(F\; a) \\
\quad \} \\
\quad \texttt{include}\ SFOL\ \texttt{with}\ \{ \\
\qquad Sort \;:=\; TypeOp\; 0 \\
\qquad SortedFun \;:=\; (n : \texttt{nat},\, A : tp^n,\, B : tp)\; PolyFun\; 0\; n\; A\; B \\
\qquad SortedPred \;:=\; (n : \texttt{nat},\, A : tp^n)\; PolyPred\; 0\; n\; A \\
\qquad Axiom \;:=\; (F : form)\; PolyAxiom\; 0\; F \\
\quad \} \\
\}
\end{array}
$$

Figure 9.11: Representation of Polymorphic SFOL in TFI/*LFS*

Finally, *PSFOL* inherits its logical symbols from *SFOL* and interprets the *SFOL*-patterns as special cases of the *PSFOL*-patterns. For example, the *SFOL*-pattern *Sort* for sort declarations is a special case of the *PSFOL*-pattern *TypeOp* with arity 0.

$PSFOL^+$ extends *PSFOL* with the distinguished sort $i : tp$ from $SFOL^+$.

Polymorphic HOL We give our representation of polymorphic higher-order logic in the theory *PHOL* in figure 9.12. Our representation follows a similar structure as in *PSFOL*.

The main feature of the PHOL syntax is that PHOL theories are allowed to declare type operators, higher-order function symbols that may take type arguments and axioms with type variables. For each one of these, we have a declaration pattern in *PHOL*. Then, *PHOL* includes the logical symbols of *HOL* and interprets the *HOL*-patterns as special cases of the *PHOL*-patterns.

$PHOL^+$ extends *PHOL* with the distinguished HOL type $i : tp$ from HOL^+.

$$
\begin{array}{l}
PHOL \;=\; \{ \\
\quad \texttt{include}\ \mathit{Types} \\
\quad \mathit{TypeOp} \;=\; (n : \texttt{nat})\,\{ \\
\qquad \tau \;\;:\;\; tp^n \to tp \\
\quad \} \\
\quad \mathit{TypedPolySym} \;=\; (m : \texttt{nat},\, \tau : tp^m \to tp)\,\{ \\
\qquad c \;\;:\;\; \Pi a : tp^m.\, tm\,(\tau\, a) \\
\quad \} \\
\quad \mathit{PolyAxiom} \;=\; (m : \texttt{nat},\, F : tp^m \to tm\, o))\,\{ \\
\qquad q \;\;:\;\; \Pi\, a : tp^m.\, ded\,(F\, a) \\
\quad \} \\
\quad \texttt{include}\ \mathit{HOL}\ \texttt{with}\ \{ \\
\qquad \mathit{BaseType} \;\;:=\;\; \mathit{TypeOp}\, 0 \\
\qquad \mathit{TypedSym} \;\;:=\;\; (\tau : tp)\ \mathit{TypedPolySym}\, 0\, \tau \\
\qquad \mathit{Axiom} \;\;:=\;\; (F : tm\, o)\ \mathit{PolyAxiom}\, 0\, F \\
\quad \} \\
\}
\end{array}
$$

Figure 9.12: Representation of Polymorphic HOL in TFI/*LFS*

9.2 Language Translations

In this section, we present our novel translations in our language atlas. The main characteristic of these translations is that they exploit declaration patterns to describe the embeddings.

Notation 9.1. In the following we will write, for the sake of readability,

$$
\begin{array}{l}
\mu \;:\; S \;\to\; T \;=\; \{ \\
\quad \texttt{include}\ \mu' \\
\quad \vdots \\
\}
\end{array}
$$

whenever S includes S' and, $\mu' : S' \to T$ is a translation from S' to T.

9.2.1 Embeddings of Weaker Languages

In this section we introduce language translations that embed less expressive languages, into more expressive ones. These translations are *i)* from PL to HOL, *ii)* from FOL to HOL, *iii)* from SFOL to HOL, *iv)* from PSFOL to PHOL,and *v)* from FOL to SFOL.

Translating PL to HOL A translation from PL to HOL must map every propositional formula of type *form* to a higher-order formula of type $tm\, o$.

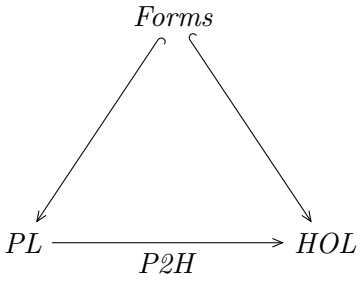
The theory morphism *P2H* in figure 9.13 is a strict language translation from *PL* to *HOL*. Recall that both *PL* and *HOL* include the theory *Forms*: *PL* includes *Forms* with no modifications, whereas *HOL* includes *Forms* by mapping the type *form* to the type $tm\, o$ of HOL formulas, and therefore adding the declaration $form : tp = tm\, o$ in *HOL*. Then, in *P2H*, it suffices to map the type *form* in *PL* to the type *form* in *HOL*.

$$
\begin{array}{l}
P2H \ : \ PL \ \to \ HOL \ = \ \{ \\
\quad form \quad := \quad tm\, o \\
\quad PropSym \ := \ \{ \\
\qquad p' \quad : \quad TypedSym\, o \\
\qquad p \quad : \quad tm\, o = p'.c \\
\quad \} \\
\quad false \quad := \quad false \\
\quad \Rightarrow \quad := \quad \lambda F : o.\, \lambda G : o.\, (\Rightarrow \ @\, F) \, @\, G \\
\}
\end{array}
$$

Figure 9.13: A Strict Translation from *PL* to *HOL*.

Next, for every declaration $q : form$ in a *PL*-theory, a corresponding HOL formula $q : tm\, o$ must be declared. Therefore, every propositional variable must be mapped to a HOL-symbol of HOL-type o. In a strict translation from *PL* to *HOL*, this is specified by mapping the *PL*-pattern *PropSym* to the instantiation of the *HOL*-pattern *TypedSym* with the type $tm\, o$. We add the second declaration $q : tm\, o = \iota.c$ in the mapping of *PropSym* to fulfill the inclusion requirement in *mapTheoryFamily* of the translation of the body $\{q : form\}$ of *PropSym*.

Moreover, the below diagram commutes if our translation contains the following assignments:



We map each PL connective to an *LFS*-expression of the same arity in terms of the respective HOL connective. For example, implication $\Rightarrow$ in *PL* is mapped to an *LFS*-expression that takes two HOL-formulas $F : tm\, o$ and $G : tm\, o$ as arguments and applies the HOL implication to them $(\Rightarrow \ @\, F) \, @\, G$.

A non-strict variant of *P2H* would map *PropSym* to $\{q : tm\, o\}$.

Translating FOL to HOL We embed first-order logic in higher-order logic by giving a strict *LFS*-translation from *FOL* to HOL^+ in figure 9.14. We map the type *term* of FOL-terms to the distinguished base type $tm\, i$ for HOL individuals. Then n-ary first-order function and predicate symbols are mapped to n-ary HOL functions over the distinguished base type: We instantiate the HOL^+-pattern *TypedSym* with the HOL^+-types $i^n \Longrightarrow^* i$ and $i^n \Longrightarrow^* o$, respectively. Logical symbols of *FOL* are mapped to their respective counterparts in HOL^+.

Translating SFOL to HOL We give a strict *LFS*-translation from *SFOL* to *HOL* in figure 9.15.

$$
\begin{array}{l}
F2H \;:\; FOL \to HOL^{+} \;=\; \{ \\
\quad \texttt{include}\ P2H \\
\quad term \;:=\; tm\, i \\
\quad Fun \;:=\; (n : \texttt{nat})\,\{ \\
\qquad f' \;:\; TypedSym\,(i^n \Longrightarrow^{*} i) \\
\qquad f \;:\; [tm\, i]_{i=1}^{n} \to tm\, i = \lambda\, x : [tm\, i]_{i=1}^{n}.\, f'.c\ @^{*}\ x \\
\quad \} \\
\quad Pred \;:=\; (n : \texttt{nat})\,\{ \\
\qquad p' \;:\; TypedSym\,(i^n \Longrightarrow^{*} o) \\
\qquad f \;:\; [tm\, i]_{i=1}^{n} \to tm\, i = \lambda\, x : [tm\, i]_{i=1}^{n}.\, p'.c\ @^{*}\ x \\
\quad \} \\
\quad Axiom \;:=\; (F : tm\, o)\ Axiom\ F \\
\quad \forall \;:=\; \lambda F : tm\, i \to tm\, o.\ \forall\, @\, (lam\, F) \\
\quad \exists \;:=\; \lambda F : tm\, i \to tm\, o.\ \exists\, @\, (lam\, F) \\
\quad \doteq \;:=\; \lambda s : tm\, i.\, \lambda t : tm\, i.\ (\doteq\, @\, s) @\, t \\
\}
\end{array}
$$

Figure 9.14: A Strict *LFS*-Translation from *FOL* to HOL^{+}

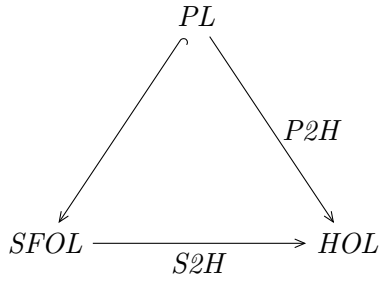
Since *SFOL* includes *PL*, we include the translation *P2H* in *S2H* using our notation in 9.1.

Then we give the following assignments for the *SFOL*-patterns:

- For any sort declaration $s : sort$ in an *SFOL*-theory, we need a *HOL* base type $s : tp$. Therefore, we map the *SFOL*-pattern *Sort* to the *HOL*-pattern *BaseType*.
- For any n-ary function symbol $f : tm\, s_1 \to \ldots \to tm\, s_n \to tm\, t$ in an *SFOL*-theory, we need an n-ary *HOL*-function, and therefore give *i)* an instance f of the *HOL*-pattern *TypedSym* with the respective *HOL*-type $tm\,(s_1 \Longrightarrow \ldots \Longrightarrow s_n \Longrightarrow t)$ in terms of the flexary *HOL* type constructor $\Longrightarrow^{*}$, and *ii)* an LFS-term f' defined in terms of f.
- For any n-ary predicate symbol $p : tm\, s_1 \to \ldots \to tm\, s_n \to form$ in an *SFOL*-theory, we give *i)* an instance p of the an *HOL*-pattern *TypedSym* with the respective *HOL*-type $tm\,(s_1 \Longrightarrow \ldots \Longrightarrow s_n \Longrightarrow o)$ in terms of the flexary *HOL* type constructor $\Longrightarrow^{*}$, and *ii)* an LFS-term p' defined in terms of p.

Then we map $\forall$, $\exists$ and $\doteq$ to their counterparts in *HOL*.

It is important to mention that our translation makes the following diagram to commute:



$$
\begin{array}{l}
\mathit{S2H} \;:\; \mathit{SFOL} \to \mathit{HOL} = \{ \\
\quad \texttt{include}\ \mathit{P2H} \\
\quad \mathit{Sort} \;:=\; \mathit{BaseType} \\
\quad \mathit{SortedFun} \;:=\; (n : \texttt{nat}, s : \mathit{tp}^n, t : \mathit{tp})\,\{ \\
\qquad f' \;:\; \mathit{TypedSym}\,(s \Longrightarrow^* t) \\
\qquad f \;:\; [\mathit{tm}\,s_i]_{i=1}^n \to \mathit{tm}\,t = \lambda x : [\mathit{tm}\,s_i]_{i=1}^n.\ f'.c\ @^*\ x \\
\quad \} \\
\quad \mathit{SortedPred} \;:=\; (n : \texttt{nat}, s : \mathit{tp}^n)\,\{ \\
\qquad p' \;:\; \mathit{TypedSym}\,(s \Longrightarrow^* o) \\
\qquad p \;:\; [\mathit{tm}\,s_i]_{i=1}^n \to \mathit{tm}\,o = \lambda x : [\mathit{tm}\,s_i]_{i=1}^n.\ p'.c\ @^*\ x \\
\quad \} \\
\quad \forall \;:=\; \lambda s : \mathit{tp}.\ \lambda p : \mathit{tm}\,s \to \mathit{tm}\,o.\ \forall\,@\,(\mathit{lam}\,p) \\
\quad \exists \;:=\; \lambda s : \mathit{tp}.\ \lambda p : \mathit{tm}\,s \to \mathit{tm}\,o.\ \exists\,@\,(\mathit{lam}\,p) \\
\quad \doteq \;:=\; \lambda s : \mathit{tp}.\ \lambda x : \mathit{tm}\,s.\ \lambda y : \mathit{tm}\,s.\ \doteq\,@\,x\,@\,y \\
\}
\end{array}
$$

Figure 9.15: A Translation from SFOL to HOL

The only non-trivial case to check is the translation of the *PL*-pattern *PropSym* along the arrows in the diagram. Recall that in figure 9.9, *SFOL* includes *PL* by mapping *PropSym* to $\mathit{SortedPred}\,0\,\cdot$. Then we want to show that $\overline{\mathit{P2H}}(\mathit{PropSym}) = \overline{\mathit{S2H}}(\mathit{SortedPred}\,0\,\cdot)$.

Recall that in figure 9.13, we have

$$\overline{\mathit{P2H}}(\mathit{PropSym}) = \left\{\begin{array}{l} p' : \mathit{TypedSym}\,o \\ p \;: \mathit{tm}\,o = p'.c \end{array}\right\}$$

Now we compute $\overline{\mathit{S2H}}(\mathit{SortedPred}\,0\,\cdot)$:

$$
\begin{array}{ll}
& \overline{\mathit{S2H}}(\mathit{SortedPred}\,0\,\cdot) \\
= & \left((n : \texttt{nat},\ s : \mathit{tp}^n)\left\{\begin{array}{l} p' : \mathit{TypedSym}\,(s \Longrightarrow^* o) \\ p \;: [\mathit{tm}\,s_i]_{i=1}^n \to \mathit{tm}\,o = \lambda x : [\mathit{tm}\,s_i]_{i=1}^n.\ p'.c\ @^*\ x \end{array}\right\}\right)\ 0\,\cdot \\
= & \left\{\begin{array}{l} p' : \mathit{TypedSym}\,(\cdot \Longrightarrow^* o) \\ p \;: [\mathit{tm}\,\cdot_i]_{i=1}^0 \to \mathit{tm}\,o = \lambda x : [\mathit{tm}\,\cdot_i]_{i=1}^0.\ p'.c\ @^*\ x \end{array}\right\} \\
= & \left\{\begin{array}{l} p' : \mathit{TypedSym}\,o \\ p \;: \mathit{tm}\,o = p'.c \end{array}\right\}
\end{array}
$$

as desired.

Furthermore, we give a strict *LFS*-translation from SFOL^+ to HOL^+ by including the translation from *SFOL* to *HOL* and mapping the SFOL^+-sort $i : \mathit{tp}$ to the HOL^+-type $i : \mathit{tp}$.

Translating PSFOL to PHOL We give a strict *LFS*-translation from *PSFOL* to *PHOL* in figure 9.16:

Since *PSFOL* and *PHOL* include *SFOL* and *HOL*, respectively, we first include the translation *S2H* from *SFOL* to *HOL*. The pattern assignments in *PS2PH* is similar to those in *S2H*: We map the *PSFOL*-pattern *TypeOp* for n-ary type operators to the respective *PHOL*-pattern *TypeOp* for n-ary type operators. Then or every polymorphic sorted function and predicate symbol, we give an instantiation of the *PHOL*-pattern

$$
\begin{array}{l}
\mathit{PS2PH} \;:\; \mathit{PSFOL} \to \mathit{PHOL} = \{ \\
\quad \texttt{include}\ \mathit{S2H} \\
\quad \mathit{TypeOp} \quad := \quad (n : \texttt{nat})\ \mathit{TypeOp}\, n \\
\quad \mathit{PolyFun} \;:=\; (m : \texttt{nat}, n : \texttt{nat}, A : (tp^m \to tp)^n, B : (tp^m \to tp))\,\{ \\
\qquad f' \;:\; \mathit{TypedPolySym}\, m\, (\lambda a : tp^m.\, [A_i\, a]_{i=1}^n \Longrightarrow^* (B\, a)) \\
\qquad f \;:\; \Pi a : tp^m.\, [tm\, (A_i\, a)]_{i=1}^n \to tm\, (B\, a) \\
\qquad\quad = \; \lambda a : tp^m.\, \lambda x : [tm\, (A_i\, a)]_{i=1}^n.\; f'.c\ @^*\, a\, @^*\, x \\
\quad \} \\
\quad \mathit{PolyPred} \;:=\; (m : \texttt{nat}, n : \texttt{nat}, A : (tp^m \to tp)^n)\,\{ \\
\qquad p' \;:\; \mathit{TypedPolySym}\, m\, (\lambda a : tp^m.\, [A_i\, a]_{i=1}^n \Longrightarrow^* o) \\
\qquad p \;:\; \Pi a : tp^m.\, [tm\, (A_i\, a)]_{i=1}^n \to tm\, o \\
\qquad\quad = \; \lambda a : tp^m.\, \lambda x : [tm\, (A_i\, a)]_{i=1}^n.\; p'.c\ @^*\, a\, @^*\, x \\
\quad \} \\
\quad \mathit{PolyAxiom} \quad := \quad (m : \texttt{nat}, F : tp^m \to tm\, o)\ \mathit{PolyAxiom}\, m\, F \\
\}
\end{array}
$$

Figure 9.16: A Strict *LFS*-Translation from Polymorphic SFOL to Polymorphic HOL

TypedPolySym for polymorphic symbols in *PHOL*-theories. Finally, we map the *PSFOL*-pattern *PolyAxiom* to the respective *PHOL*-pattern *PolyAxiom*.

We have the following commuting diagram:

$$
\begin{array}{ccc}
\mathit{SFOL} & \xrightarrow{\quad \mathit{S2H} \quad} & \mathit{HOL} \\
\big\downarrow & & \big\downarrow \\
\mathit{PSFOL} & \xrightarrow{\quad \mathit{PS2PH} \quad} & \mathit{PHOL}
\end{array}
$$

The non-trivial cases for showing the commutativity of the above diagram are the *SFOL*-patterns *SortedFun* and *SortedPred*. For example, *SortedPred* is mapped to

$$
(n : \texttt{nat}, s : tp^n) \begin{Bmatrix} p' \;:\; \mathit{TypedPolySym}\, 0\, (s \Longrightarrow^* o) \\ p \;:\; [tm\, s_i]_{i=1}^n \to tm\, o \\ \quad = \; \lambda x : [tm\, s_i]_{i=1}^n.\, p'.c\, @^* x \end{Bmatrix}
$$

in *PHOL* along both paths through the diagram.

Translating FOL to SFOL We give an embedding of first-order logic in sorted first-order logic by fixing a distinguished sort for the universe of first-order terms. For that reason, we give a translation from *FOL* to SFOL^+ by mapping the type *term* of *FOL*-terms to the distinguished SFOL^+-sort $tm\, i$ and the type *form* to itself. Then *FOL*-patterns for n-ary function and predicate symbols are mapped to the respective *SFOL*-patterns using the distinguished short i:

$$
\begin{array}{lcl}
\mathit{Fun} & := & (n : \texttt{nat})\ \mathit{SortedFun}\, n\, i^n\, i \\
\mathit{Pred} & := & (n : \texttt{nat})\ \mathit{SortedPred}\, n\, i^n
\end{array}
$$

The *FOL*-pattern *Axiom* is mapped to the *SFOL*-pattern *Axiom*.

9.2.2 Semantics

In this section we give translations of *i)* HOL and its polymorphic version PHOL into set theory, and *ii)* modal logic into Kripke models formalized in first-order logic.

Set-Theoretical Semantics of HOL We give a strict *LFS*-translation *H2Z* from *HOL* to *Sets* in figure 9.17. *H2Z* represents the set-theoretic semantics of HOL. The key idea of this translation is that HOL types are interpreted as sets.

$$
\begin{array}{l}
T2Z \ : \ Types \ \to \ Sets \ = \ \{ \\
\quad tp \quad := \quad set \\
\quad tm \quad := \quad \lambda s : set.\, set \\
\}
\end{array}
$$

$$
\begin{array}{l}
H2Z \ : \ HOL \ \to \ Sets \ = \ \{ \\
\quad \texttt{include}\ T2Z \\
\quad BaseType \quad := \quad Fun\, 0 \\
\quad TypedSym \ := \ (\tau : set)\, \{ \\
\qquad c \quad : \quad Fun\, 0 \\
\qquad ax \quad : \quad ded\, c \in \tau \\
\quad \} \\
\}
\end{array}
$$

Figure 9.17: A Strict *LFS*-Translation from *HOL* to *Sets*

In particular, the *LFS*-type *tp* for HOL terms is mapped to the *LFS*-type *set* for sets. Then the *LFS*-type $o : tp$ of HOL formulas is mapped to the set $\{0, 1\}$ of the boolean truth values. Consequently, all true formulas are interpreted as terms provably equal to the element 1, and false ones as 0.

For the details of this translation, we refer the readers to our case-study in [HR11]. In that case study, we used LF as the underlying logical framework. Our translation in *LFS* enriches that case-study with declaration patterns.

Translating *PHOL* to *Sets* We give a translation from *PHOL* to *Sets* in figure 9.18.

Note that we get the equality $\doteq$ on sets from *FOL*.

Kripke-Semantics of Modal Logic We give a strict *LFS*-translation from *ML* to FOL^+ in figure 9.19.

The key idea in this translation is that ML-formulas are interpreted as unary FOL-predicates by mapping the LFS-type *form* of formulas in *ML* to the LFS-type $term \to form$ of unary predicates in *FOL*. Then ML axioms are interpreted as FOL axioms that hold for every FOL term: For any axiom declaration $m : ded\, F$ in a *ML*-theory, we have a corresponding declaration $m : ded\, \forall\, x : term.\, F\, x$ in *FOL*. This is captured by the mapping of the *ML*-pattern *Axiom*. The mapping of the connectives for falsehood, negation and implication are straightforward. The modal operators $\Box$ and $\diamond$ of *ML* are interpreted using the FOL^+-symbol *acc* for the accessibility relation between the interpretations of two ML worlds in FOL.

$$
\begin{array}{l}
PH2Z \ : \ PHOL \ \to \ Sets \ = \ \{ \\
\quad \texttt{include} \ T2Z \\
\quad TypeOp \ := \ (n : \texttt{nat}) \, Fun \, n \\
\quad TypedPolySym \ := \ (m : \texttt{nat}, \tau : set^m \to set) \, \{ \\
\qquad c' \ : \ Fun \, m \\
\qquad c \ : \ \Pi a : set^m . \, set = \lambda a : set^m . \, c' . f \ a \\
\qquad ax \ : \ Axiom \, \forall^* \lambda x : set^m . (c \, x) \in (\tau \, x) \\
\quad \} \\
\quad PolyAxiom \ := \ (m : \texttt{nat}, F : set^m \to set) \, \{ \\
\qquad q \ : \ Axiom \, \forall^* \lambda x : set^m . \, (F \, x) \doteq 1 \\
\quad \} \\
\quad \texttt{include} \ H2Z \\
\}
\end{array}
$$

Figure 9.18: A Translation from *PHOL* to *Sets*

$$
\begin{array}{lcl}
M2F \ : \ ML \ \to \ FOL^+ \ = \ \{ & & \\
\quad form & := & term \to form \\
\quad ded & := & \lambda F : term \to form . \, ded \, \forall x : term . \, F \, x \\
\quad Axiom & := & (F : term \to form) \, Axiom \, \forall x : term . \, F \, x \\
\quad false & := & \lambda x : term . \, false \\
\quad \Rightarrow & := & \lambda F : term \to form . \, \lambda G : term \to form . \, \lambda x : term . \, (F \, x) \ \Rightarrow \ (G \, x) \\
\quad Prop & := & Pred \, 1 \\
\quad \Box & := & \lambda F : term \to form . \, \lambda x : term . \, \forall \lambda y : term . \, (acc \, x \, y) \Rightarrow (F \, y) \\
\quad \diamond & := & \lambda F : term \to form . \, \lambda x : term . \, \exists \lambda y : term . \, (acc \, x \, y) \wedge (F \, y) \\
\} & &
\end{array}
$$

Figure 9.19: Semantics of ML

9.3 Extension Principles

In this section, we add the individual extension principles we represented in section 8.1 into our language atlas from figure 9.1.

If we take only those nodes of our atlas that we need for the extension principles we selected, we have the diagram in figure 9.20.

The strength of our approach of representing extension principles in a logical framework is that every individual principle can be specified in a separate theory that extends a certain declarative language. In particular, we can translate extension principle from one declarative language to another.

The ability to add extension principles to declarative languages and to translate them between languages is also very powerful. For example, function definitions in SFOL can now be translated to FOL or HOL using the translations we have given in section 9.2.

Translating Extension Principles of SFOL to FOL

In section 8.2 we have given the translation of the *SFOL*-extension principle *function* in *FOL*.

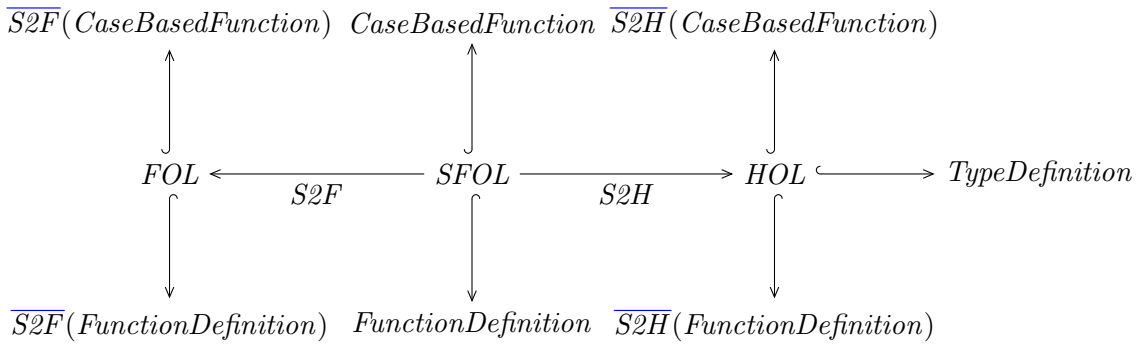


Figure 9.20: A Graph of Extension Principles

Now we translate the *SFOL*-extension principle *casedef* from figure 8.4 along the strict *LFS*-translation *S2F*:

$$
\begin{array}{l}
\overline{S2F}(casedef) \;=\; \\
\quad (\\
\qquad n : \mathtt{nat}, A : term \to form, B : term \to form, \\
\qquad c : (term \to form)^n, \\
\qquad d : (term \to term)^n, \\
\qquad \rho : ded\, \forall x : term.\, (A\, x) \Rightarrow \vee^* [c_i\, x]_{i=1}^n \\
\quad) \\
\quad \{ \\
\qquad \begin{array}{lcl}
f & : & Fun\, 1 \\
ax & : & Axiom\, \forall^* \lambda x : term.\, (A\, x) \;\Rightarrow\; B\, (f\, x) \\
wellsorted & : & Axiom\, \forall x : term.\, \wedge^* [c_i\, x \;\Rightarrow\; (f\, x) = (d_i\, x)]_{i=1}^n
\end{array} \\
\quad \}
\end{array}
$$

Translating Extension Principles from SFOL to HOL

Now we translate the *SFOL*-extension principles along the strict *LFS*-translation *S2H*:

The translation of the *SFOL*-extension principle *function* from figure 8.3 along the strict *LFS*-translation *S2H* from figure 8.3 is as follows:

$$
\begin{array}{l}
\overline{S2H}(function) \;=\; \\
\quad (\\
\qquad \begin{array}{lcl}
n & : & \mathtt{nat},\, D : tp^n,\, C : tp, \\
means & : & [tm\, D_i]_{i=1}^n \to tm\, C \to o, \\
existence & : & ded\, \forall^* \lambda x : [tm\, D_i]_{i=1}^n.\, \exists \lambda y : tm\, C.\, means\, x\, y, \\
uniqueness & : & ded\, \forall^* \lambda x : [tm\, D_i]_{i=1}^n.\, \forall \lambda y : tm\, C.\, \forall \lambda y' : tm\, C. \\
 & & \qquad means\, x\, y \wedge means\, x\, y' \Rightarrow y \doteq y'
\end{array} \\
\quad) \\
\quad \{ \\
\qquad \begin{array}{lcl}
f & : & TypedSym(D \Longrightarrow^* C) \\
f' & : & tm\, [D_i]_{i=1}^n \to tm\, C \;=\; \lambda x.\, f\, @^*\, x \\
ax & : & Axiom\, \forall^* \lambda x : [tm\, D_i]_{i=1}^n.\, means\, x\, (f\, x)
\end{array} \\
\quad \}
\end{array}
$$

The translation of the *SFOL*-extension principle *casedef* from figure 8.4 along the strict *LFS*-translation *S2H* from figure 8.3 is as follows:

$$
\begin{array}{l}
\overline{S2H}(casedef) \;=\; \\
\quad (\\
\qquad n : \mathtt{nat},\ A : tp, B : tp, c : (tm\ A \to tm\ o)^n, \\
\qquad d : (tm\ A \to tm\ B)^n, \rho : ded\ \forall x : tm\ A.\ \vee^{!}[c_i\ x]_{i=1}^{n} \\
\quad) \\
\quad \{ \\
\qquad f \;:\; TypedSym\ (A \Longrightarrow B) \\
\qquad f' \;:\; TypedSym\ A \to tm\ B = \lambda x.\ f @ x \\
\qquad ax \;:\; Axiom\ \forall x : tm\ A.\ \wedge^{*}\ [c_i\ x\ \Rightarrow\ (f\ x) = (d_i\ x)]_{i=1}^{n} \\
\quad \}
\end{array}
$$

9.4 TPTP Languages

9.4.1 Overview

TPTP [SS98] was introduced as a simple representation format for benchmark problems for automated theorem provers (ATPs) in first-order logic (FOL). It has been very successful at combining [PS07a], applying [DFS04], and evaluating [PSS02] ATPs. In fact, it has become the standard input language for first-order theorem provers with most provers supporting it natively. The TPTP problem library comprises about 20000 problems from about 50 domains and comes with extensive tool support [Sut10]. TPTP has also been used as a knowledge representation format for other large libraries [Urb06, PS07b].

While TPTP originally handled only classical unsorted FOL, it has gradually expanded to a variety of logics. These include in particular typing [BRS08, SSSB12], polymorphism [BP12], higher-order types [BRS08], and arithmetic [SSSB12]; a modal version was proposed in [RO09]. These extensions have been defined by extending the TPTP syntax. We expect the future interest in additional extensions to rise further as theorem provers are tackling more and more complex languages. Extensions that have been suggested or are already under development include, for example, product types, dependent types, and description and choice operators.

Thus, TPTP is a very promising candidate for a universal interface language for ATP developers and users. Such a language would permit the smooth integration of ATP (problem-solving) systems and (problem-generating) applications. As long as we only work with classical untyped first-order logic, this is already possible. But in general, the successful communication between ATP systems crucially depends on a common understanding of the semantics. Because different ATP systems make different logical assumptions, which are often implicit in the implementation or only documented informally, the growing number of logics that are of interest to ATP systems present new challenges for interface languages.

It is not always obvious what the relations between different TPTP logics are. For example, there are intuitive sublanguage relations between untyped FOL, typed FOL, and higher-order logic. But these can be difficult to specify precisely, especially when the larger language introduces new concepts and then recovers the smaller language as special cases.

We can solve this problem by representing TPTP languages within TFI/*LFS*. Actually, our language atlas in figure 9.1 already covers the existing TPTP languages.

Thus, using our framework, we can give formalizations of the TPTP logics and translations between them. Concretely, we get the representations of the syntax and the declarations patterns of the first-order, typed first-order, polymorphic first-order and higher-order TPTP logics from our language atlas, and translations between them.

9.4.2 TPTP in a Logical Framework

Our language atlas in figure 9.1 covers the existing TPTP logics. Figure 9.21 illustrates the nodes in our language atlas that correspond to the specific TPTP logics.

Language Atlas	TPTP
FOL	*FOF*
$SFOL^+$	*TFF0*
HOL^+	*THF0*
$PSFOL^+$	*TFF1*

Figure 9.21: TPTP Logics in the Language Atlas

The syntax we used in the atlas differs from the TPTP syntax via a simple renaming of the primitive symbols, which we show in figure 9.22.

	FOL	*FOF*
First-order formulas	*form*	`$o`
First-order terms	*term*	`$i`
Universal quantifier	$\forall$	!
Existential quantifier	$\exists$	?
	$SFOL^+/PSFOL^+$	*TFF0*/*TFF1*
Sorts	*tp*	`$tType`
Terms of a sort	*tm*	`$tm`
Distinguished base type	ι	`$i`
Universal quantifier	$\forall$	!
Existential quantifier	$\exists$	?
	$HOL^+/PHOL^+$	*THF0*/*THF1*
Higher-order terms	i	`$i`
Higher-order formulas	o	`$o`
Provability judgment	$\vdash$	$\vdash$
Type constructor	$\Longrightarrow$	$>$
Func. abstraction	*lam*	^
Func. application	@	@
Univ. quantifier	$\forall$	!
Exist. quantifier	$\exists$	?

Figure 9.22: TPTP Logical Symbols

The only technical difference is that $SFOL^+$ uses an LFS-type *form* : `type` in order to distinguish formulas from terms, whereas the description of *TFF0* in [SSSB12] uses a TPTP-type `$o` : `$tType` for the formulas. Our representation has the advantage that we do not need case distinctions in order to avoid `$o` as an argument of a function or predicate symbol or of a quantifier.

The TPTP syntax supports polymorphism for higher-order logic. However, that part of the TPTP syntax has not been yet specified. By representing the TPTP languages in a logical framework with modularity, we are able to specify polymorphic higher-order logic for TPTP, called *THF1*, by combining *TFF1* and *THF0* (as illustrated in figure 9.23).

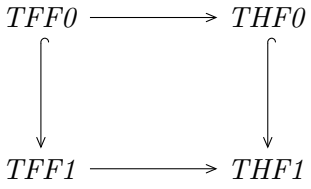


Figure 9.23: Polymorphic Higher-Order Logic in TPTP

Translating TPTP Languages

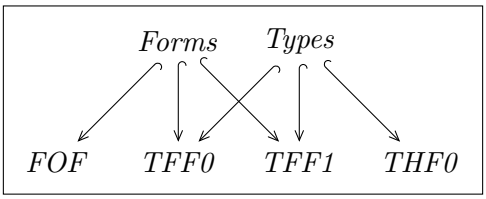


Figure 9.24: TPTP Logics

We are able to relate the TPTP languages by the language translations we have given in our language atlas. An overview is given in figure 9.25.

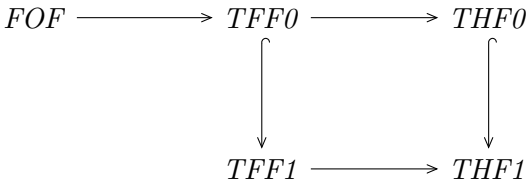


Figure 9.25: Translations between TPTP Languages

Chapter 10

Conclusion

10.1 Summary

This thesis is centered around a new concept we have identified within the context of declarative languages: *declaration patterns* — patterns that specify the shape of the declarations in the theories of a declarative language $\mathcal{L}$. We use declaration patterns as the defining characteristics of the theories of $\mathcal{L}$ by regarding every $\mathcal{L}$-theory as a list of symbol declarations that conform to a declaration pattern of $\mathcal{L}$.

Even though declaration patterns are a significant part of the informal definitions of declarative languages, they were not recognized as an essential primitive concept in the formalizations of these languages in logical frameworks prior to this thesis. Early formalizations, in particular in declarative logical frameworks, focused on representing the expressions and the proof systems. The theories were known but not specified explicitly. The later development of tool support for logical frameworks enabled large-scale formalizations, such as the LATIN logic atlas, where it was not feasible to leave declaration patterns implicit any longer.

Following this motivation, we have developed TFI — a calculus-based, foundation-independent meta-framework for representing declarative languages. The key novelty of TFI is the novel primitive notions of theory families and their instantiations, which we invented for defining declaration patterns.

Declaration patterns proved crucial in addressing the research problems we discussed in section 3: TFI permits adequate representations of declarative languages and their translations, and enables to give representations of extension principles.

More specifically, the main problem with declarative logical frameworks like LF has been that they over-generate the theories of the represented language, leading to an inadequate representation in the framework. Using declaration patterns, we were able overcome the over-generation problem by formally defining the theories of a language within TFI itself.

Secondly, we identified that declarative logical frameworks like LF are not expressive enough to formalize many interesting language translations adequately. TFI improves on this problem by defining pattern-based language translations that are adequate.

Thirdly, declaration patterns proved to be a very suitable concept for representing extension principles in declarative logical frameworks. In particular, TFI provides a systematic approach to represent individual extension principles as well as to build theories by appealing to them. This feature has been completely absent in previous declarative frameworks.

In particular, we have given the representations of a collection of widely used extension

principles from mathematics and theorem proving.

We have evaluated TFI by developing an atlas of adequately represented declarative languages including the representation of several commonly used extension principles, which are interrelated by adequate language translations. Our atlas demonstrates that our design choices for representing declaration patterns cover a broad range of different languages including the TPTP logics.

Our main contribution to the field of logical frameworks is the novel approach of defining declarative languages and translations using declaration patterns. We believe that declaration patterns will be a standard part of future logical frameworks.

10.2 Applications

We have integrated the notion of declaration patterns within the MMT API [Rab13b] — the implementation of the MMT system.

The Mizar Mathematical Library in OMDoc In [IKRU13] declaration patterns served as a major representation tool to translate the Mizar mathematical library into the OMDoc format [Koh06]. The representation of Mizar in LF [IR11] was enriched over 30 declaration patterns (directly given within the MMT API) for the various extension principles of Mizar. The translation was implemented within the MMT API and parsed every Mizar article to produce a corresponding OMDoc representation as a list of instances of the Mizar declaration patterns. The produced OMDoc representations were used for querying the Mizar library, which was integrated into the Mizar interface.

Compiling Logics The work [CHMR12] introduced an architecture implemented in the MMT API that permits generating logic implementations in the Heterogeneous Tool Set (Hets) [MML07] from their representations in *LFS*. The architecture used the declaration patterns of a logic L in *LFS* for the automatic generation of datatypes and functions for parsing and static analysis of L in Hets.

Flexary Operators for Formalized Mathematics *LFS* proved to be a very suitable foundation for defining flexary languages and to give representations of flexary mathematical operations. In [HRK14] we used *LFS* and flexary versions of SFOL and HOL to define various mathematical concepts such as polynomials.

10.3 Future Work and Directions

Tool Support The natural next step towards systematic practical applications is a full implementation of TFI and in particular of TFI/*LFS*.

Regarding TFI, the existing infrastructure for declaration patterns in the MMT API already provides the initial steps. The main open problem is how to integrate pattern-based translations into the MTT API: MMT assumes that the equality judgment is always preserved along theory morphisms whereas TFI relaxes this assumption for the case of declaration patterns. This difference in the treatment of equality would need to be addressed.

Regarding TFI/*LFS*, the MMT API can already reflect some modular composition of foundations to a certain degree: In an implementation of a specific foundation within the MMT API, the rules of the foundation are implemented as special MMT constants, which are given as Scala functions. The main open problem will be the integration of

foundational morphisms into the MMT API to allow for modular implementation of, in particular, the rules of a foundation. Another open problem is to formally investigate the decidability of *LFS* type-checking, which we conjectured to be undecidable, and to find a proper heuristic for the implementation.

Meta-Properties of *LFS* Besides type-checking, other meta-properties of *LFS* should be investigated in full detail. In particular, conservativity of *LFS* over LF and canonical forms of *LFS* are two important meta-properties that remain as open problems at the moment.

Functorial Translations Most declarative languages naturally form a category. Indeed, most abstract logical frameworks define a declarative language using a category of theories and the a translation between two declarative languages using a functor between the respective two theory categories.

Using declaration patterns, we are able to define theory categories, which are usually adequate for the declarative languages represented in TFI.

However, pattern-based translations in TFI cannot always induce a functor between these theory categories. In particular, translating theory morphisms $\sigma : \Sigma_1 \to \Sigma_2$ along a pattern-based translation is only possible for the simple case where σ is a renaming. We cannot translate morphisms σ that map instance symbols ι to instance expressions I. This is a major open problem for the future.

Bibliography

[ACTZ06] A. Asperti, C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli. Crafting a Proof Assistant. In T. Altenkirch and C. McBride, editors, *TYPES*, pages 18–32. Springer, 2006.

[Bar92] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.

[BCC+04] S. Buswell, O. Caprotti, D. Carlisle, M. Dewar, M. Gaetano, and M. Kohlhase. The Open Math Standard, Version 2.0. Technical report, The Open Math Society, 2004. See http://www.openmath.org/standard/om20.

[Ber90] S. Berardi. *Type dependence and constructive mathematics*. PhD thesis, Dipartimento di Matematica, Università di Torino, 1990.

[BP12] J. Blanchette and A. Paskevich. TFF1: The TPTP Typed First-Order Form with Rank-1 Polymorphism. 2012. in preparation.

[BRS08] C. Benzmüller, F. Rabe, and G. Sutcliffe. THF0 – The core of the TPTP Language for Higher-Order Logic. In A. Armando, P. Baumgartner, and G. Dowek, editors, *4th International Joint Conference on Automated Reasoning*, pages 491–506. Springer, 2008.

[CELM96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4, pages 65–89, 1996.

[CF58] H. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.

[CH88] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[CHJ+13] M. Codescu, F. Horozal, A. Jakubauskas, T. Mossakowski, and F. Rabe. Compiling Logics. In N. Martí-Oliet and M. Palomino, editors, *Recent Trends in Algebraic Development Techniques 2012*, pages 111–126. Springer, 2013.

[CHK+11] M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, and F. Rabe. Project Abstract: Logic Atlas and Integrator (LATIN). In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 289–291. Springer, 2011.

[CHK+12] M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, F. Rabe, and K. Sojakova. Towards the Heterogeneous Tool Set Hets. In T. Mossakowski and

H. Kreowski, editors, *Recent Trends in Algebraic Development Techniques 2010*, pages 139–159. Springer, 2012.

[CHMR12] M. Codescu, F. Horozal, T. Mossakowski, and F. Rabe. Compiling Logics. In *Workshop on Algebraic Development Techniques*, 2012.

[Chu40] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(1):56–68, 1940.

[CMM13] Mihai Codescu, Till Mossakowski, and Christian Maeder. Checking conservativity with hets. In Reiko Heckel and Stefan Milius, editors, *Algebra and Coalgebra in Computer Science - 5th International Conference, CALCO 2013, Warsaw, Poland, September 3-6, 2013. Proceedings*, volume 8089 of *Lecture Notes in Computer Science*, pages 315–321. Springer, 2013.

[Com07] Information technology — Common Logic (CL): a framework for a family of logic-based languages. Technical Report 24707:2007, ISO/IEC, 2007.

[Coq14] Coq Development Team. The Coq Proof Assistant: Reference Manual. Technical report, INRIA, 2014.

[dB70] N. de Bruijn. The Mathematical Language AUTOMATH. In M. Laudet, editor, *Proceedings of the Symposium on Automated Demonstration*, volume 25 of *Lecture Notes in Mathematics*, pages 29–61. Springer, 1970.

[DFS04] E. Denney, B. Fischer, and J. Schumann. Using Automated Theorem Provers to Certify Auto-generated Aerospace Software. In D. Basin and M. Rusinowitch, editors, *Automated Reasoning - Second International Joint Conference*, pages 198–212. Springer, 2004.

[FS87] J. Fiadeiro and A. Sernadas. Structuring Theories on Consequence. In D. Sannella and A. Tarlecki, editors, *Recent Trends in Data Type Specification*, volume 332, pages 44–72. Springer, 1987.

[GB86] J. Goguen and R. Burstall. A study in the foundations of programming methodology: specifications, institutions, charters and parchments. In D. Pitt, S. Abramsky, A. Poigné, and D. Rydeheard, editors, *Workshop on Category Theory and Computer Programming*, pages 313–333. Springer, 1986.

[GB92] J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.

[Gor88] M. Gordon. HOL: A Proof Generating System for Higher-Order Logic. In G. Birtwistle and P. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer-Academic Publishers, 1988.

[GR02] J. Goguen and G. Rosu. Institution morphisms. *Formal Aspects of Computing*, 13:274–307, 2002.

[HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

[HKR12] F. Horozal, M. Kohlhase, and F. Rabe. Extending MKM Formats at the Statement Level. In J. Campbell, J. Carette, G. Dos Reis, J. Jeuring, P. Sojka, V. Sorge, and M. Wenzel, editors, *Intelligent Computer Mathematics*, pages 64–79. Springer, 2012.

[How80] W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[HR11] F. Horozal and F. Rabe. Representing Model Theory in a Type-Theoretical Logical Framework. *Theoretical Computer Science*, 412(37):4919–4945, 2011.

[HR12] F. Horozal and F. Rabe. Representing Logics of Theorem Provers. see `http://kwarc.info/frabe/Research/HR_tptp_12.pdf`, 2012.

[HRK14] F. Horozal, F. Rabe, and M. Kohlhase. Flexary Operators for Formalized Mathematics. In S. Watt, J. Davenport, A. Sexton, P. Sojka, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 312–327. Springer, 2014.

[HST94] R. Harper, D. Sannella, and A. Tarlecki. Structured presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994.

[IKRU13] M. Iancu, M. Kohlhase, F. Rabe, and J. Urban. The Mizar Mathematical Library in OMDoc: Translation and Applications. *Journal of Automated Reasoning*, 50(2):191–202, 2013.

[IR11] M. Iancu and F. Rabe. Formalizing Foundations of Mathematics. *Mathematical Structures in Computer Science*, 21(4):883–911, 2011.

[KB04] Temur Kutsia and Bruno Buchberger. Predicate logic with sequence variables and sequence function symbols. In Andrea Asperti, Grzegorz Bancerek, and Andrej Trybulec, editors, *Mathematical Knowledge Management, MKM'04*, number 3119 in LNAI, pages 205–219. Springer Verlag, 2004.

[KMR09] M. Kohlhase, T. Mossakowski, and F. Rabe. The LATIN Project, 2009. see `https://trac.omdoc.org/LATIN/`.

[Koh06] M. Kohlhase. *OMDoc: An Open Markup Format for Mathematical Documents (Version 1.2)*. Number 4180 in Lecture Notes in Artificial Intelligence. Springer, 2006.

[MAH06] T. Mossakowski, S. Autexier, and D. Hutter. Development graphs – proof management for structured specifications. *Journal of Logic and Algebraic Programming*, 67(1-2):114–145, 2006.

[Mes89] J. Meseguer. General logics. In H.-D. Ebbinghaus et al., editors, *Proceedings, Logic Colloquium, 1987*, pages 275–329. North-Holland, 1989.

[ML74] P. Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In *Proceedings of the '73 Logic Colloquium*, pages 73–118. North-Holland, 1974.

[MML07] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editor, *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522, 2007.

[MOM94] Narciso Martí-Oliet and José Meseguer. What is a logical system? chapter General logics and logical frameworks, pages 355–391. Oxford University Press, Inc., New York, NY, USA, 1994.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML,* Revised edition. MIT Press, 1997.

[Nor05] U. Norell. The Agda WiKi, 2005. http://wiki.portal.chalmers.se/agda.

[NPW02] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[Pau94] L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[Paw95] Wieslaw Pawlowski. Context institutions. In M. Haveraaen, O. Owe, and O. J. Dahl, editors, *Recent Trends in Data Type Specification*, Lecture Notes in Computer Science, pages 436–457. Springer-Verlag, 1995.

[PS99] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, 1632:202–206, 1999.

[PS07a] L. Paulson and K. Susanto. Source-Level Proof Reconstruction for Interactive Theorem Proving. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics*, pages 232–245. Springer, 2007.

[PS07b] A. Pease and G. Sutcliffe. First Order Reasoning on a Large Ontology. In J. Urban, G. Sutcliffe, and S. Schulz, editors, *Empirically Successful Automated Reasoning in Large Theories*, number 257 in CEUR Workshop Proceedings, pages 59–69, 2007.

[PSK+03] F. Pfenning, C. Schürmann, M. Kohlhase, N. Shankar, and S. Owre. The Logosphere Project, 2003. http://www.logosphere.org/.

[PSS02] F. Pelletier, G. Sutcliffe, and C. Suttner. The Development of CASC. *AI Communications*, 15(2-3):79–90, 2002.

[Rab06] F. Rabe. First-Order Logic with Dependent Types. In N. Shankar and U. Furbach, editors, *Automated Reasoning*, pages 377–391. Springer, 2006.

[Rab13a] F. Rabe. A Logical Framework Combining Model and Proof Theory. *Mathematical Structures in Computer Science*, 23(5):945–1001, 2013.

[Rab13b] F. Rabe. The MMT API: A Generic MKM System. In J. Carette, D. Aspinall, C. Lange, P. Sojka, and W. Windsteiger, editors, *Intelligent Computer Mathematics*, pages 339–343. Springer, 2013.

[RK13] F. Rabe and M. Kohlhase. A Scalable Module System. *Information and Computation*, 230(1):1–54, 2013.

[RO09] T. Raths and J. Otten. Building a Problem Library for First-Order Modal Logics. In *TABLEAUX 2009 Position Papers and Workshop Proceedings*, 2009.

[RS09] F. Rabe and C. Schürmann. A Practical Module System for LF. In J. Cheney and A. Felty, editors, *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*, pages 40–48. ACM Press, 2009.

[Soj10] K. Sojakova. Mechanically Verifying Logic Translations. Master's thesis, Jacobs University Bremen, 2010.

[SS98] G. Sutcliffe and C. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[SSSB12] G. Sutcliffe, K. Claessen S. Schulz, and P. Baumgartner. The TPTP Typed First-order Form with Arithmetic. In *Logic for Programming, Artificial Intelligence, and Reasoning*, 2012. to appear.

[Sut10] G. Sutcliffe. The TPTP World - Infrastructure for Automated Reasoning. In E. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 1–12. Springer, 2010.

[SW83] D. Sannella and M. Wirsing. A Kernel Language for Algebraic Specification and Implementation. In M. Karpinski, editor, *Fundamentals of Computation Theory*, pages 413–427. Springer, 1983.

[TB85] A. Trybulec and H. Blair. Computer Assisted Reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28, 1985.

[Urb06] J. Urban. MPTP 0.2: Design, Implementation, and Initial Experiments. *Journal of Automated Reasoning*, 37(1-2):21–43, 2006.

[Wol12] Wolfram Research, Inc. Mathematica 9.0, 2012.